Universidad de Almería
Departamento de Informática

Tesis Doctoral

Tecnología Software aplicada al Diseño de Sistemas Basados en el Conocimiento

José Joaquín Cañadas Martínez

Directores:

Dr. D. José Tomás Palma Méndez
Dr. D. Samuel Túnez Rodríguez
Dra. Dña. Isabel María del Águila Cano

Almería, Julio de 2015

A Sonia, Joaquín y Lola

A Antoñita y Joaquín

Agradecimientos

El trabajo que se presenta en esta Memoria de Tesis Doctoral ha sido logrado gracias al esfuerzo y apoyo que numerosas personas me han brindado a lo largo de estos años, la mayoría de ellas vinculadas directa o indirectamente a la Universidad de Almería, donde se ha desarrollado esta investigación, y también a la Universidad de Murcia.

En primer lugar, quiero dar mi más sincero agradecimiento a mis directores de Tesis, D. José Tomás Palma Méndez, D. Samuel Túnez Rodríguez, y Dña. Isabel María del Águila Cano. Sin su confianza, apoyo, dirección, soporte, orientación y, sobre todo paciencia, no hubiese llevado a término este trabajo.

Agradecer a todos los compañeros del grupo de investigación "Ingeniería de Datos, del Conocimiento y del Software (DKSE)" de la Universidad de Almería por su colaboración y apoyo en las temas de trabajo que hemos realizado conjuntamente durante estos años. También quiero agradecer a los miembros del grupo de investigación "Inteligencia Artificial e Ingeniería del Conocimiento (AIKE)" de la Universidad de Murcia, al profesor D. Roque Marín, a D. José M. Juarez y al resto de chicos del Laboratorio por su excelente acogida, apoyo y el buen ambiente de trabajo que disfruté con ellos cuando inicié este trabajo de investigación.

Mi agradecimiento al profesor D. Grzegorz J. Nalepa de la AGH University of Science and Technology de Krakow, por su confianza, colaboración y los buenos consejos que me ha ofrecido, así como la oportunidad de colaborar con él junto al profesor D. Joachim Baumeister, de la Universidad de Würzburg, en la organización del Workshop KESE, y en cuyas sesiones se discutieron parte de los trabajos de investigación incluidos en esta Tesis Doctoral.

Y a todos mis compañeros del Departamento de Informática de la Universidad de Almería por hacer que el trabajo día a día sea tan satisfactorio. Mención especial a D. Francisco Javier Orellana, Zubi, con el

que he coincidido en el tiempo en el desarrollo de nuestros respectivos trabajos de investigación, por los buenos años de compañero compartidos y la excelente colaboración que hubo entre nosotros.

Agradecer también a las entidades que han otorgado la financiación necesaria para la consecución de este trabajo, mediante diversos proyectos: Ministerio de Ciencia y Tecnología, proyecto TIN2004-05694, y Junta de Andalucía, proyecto de Excelencia P06-TIC-02411.02.

Por último quiero agradecer a mi familia, que ha tenido la paciencia de apoyarme durante los años de esfuerzo que ha supuesto la realización de este trabajo, y que son los que verdaderamente han sufrido las consecuencias.

A todos ellos... ¡Gracias!

Índice general

Resumen xvii

Summary xix

1 Introducción 1

1.1 Ingeniería del Conocimiento e Ingeniería del Software . 1

1.1.1 Metodologías de la Ingeniería del Conocimiento 2

1.1.2 Deficiencias en la transición al diseño de la IC . 3

1.1.3 Desarrollo de Software Dirigido por Modelos . . 4

1.1.4 Sistemas basados en reglas 8

1.1.5 Aplicaciones web 8

1.1.6 Aplicaciones web basadas en reglas 9

1.2 Hipótesis de partida y objetivos 11

1.3 Alcance de la propuesta 12

1.3.1 Desarrollo dirigido por modelos de la base de reglas 15

1.3.2 Desarrollo dirigido por modelos de arquitectura web 16

1.3.3 Caso de estudio en el dominio agrícola 18

1.4 Contexto del trabajo de investigación 19

1.5 Estructura de la memoria 21

2 Estado de la cuestión 23

2.1 La Ingeniería del Conocimiento 23

2.1.1 Características de los Sistemas Basados en Conocimiento 25

2.1.2 Ventajas y limitaciones de los sistemas basados en conocimiento 28

2.1.3 Representación del Conocimiento 30

2.1.4 Metodologías de primera generación de SBC . . 33
2.1.5 Metodologías de segunda generación de SBC . . 36
2.1.6 Ontologías. Ingeniería Ontológica 40
2.2 Modelado y formalización de sistemas basados en reglas 43
2.2.1 Modelado de sistemas basados en reglas 48
2.2.2 Formalización de sistemas basados en reglas . . 49
2.3 Desarrollo de Software Dirigido por Modelos 51
2.3.1 Modelos y metamodelos 52
2.3.2 Transformaciones de modelos 54
2.3.3 Eclipse Modeling Framework 55
2.3.4 ATL. Atlas Transformation Language 56
2.3.5 JET. Java Emitter Templates 56
2.4 Ingeniería Web 57
2.4.1 Tecnología de implementación Aplicaciones Web Ricas (RIAs) 57
2.4.2 Frameworks para RIAs en JavaEE 58
2.4.3 Librería de componentes RichFaces de JBoss . . 58
2.5 Conclusiones del capítulo 59

3 Desarrollo dirigido por modelos de la base de conocimiento 61
3.1 Perspectiva general 62
3.2 Modelado de aplicaciones basadas en reglas con CML . 63
3.2.1 Ejemplo ilustrativo CML 64
3.2.2 Modelado conceptual basado en reglas 66
3.2.3 Editor de modelos CML basado en Eclipse . . . 69
3.3 Modelado específico de base de reglas en Jess 71
3.4 Transformaciones de modelos: desde CML a Jess 72
3.4.1 Transformación Modelo a Modelo 73
3.4.2 Transformación Modelo a Texto 75
3.5 Modelado de aplicaciones basadas en reglas con OWL y SWRL 78
3.5.1 Modelado con OWL y SWRL 78
3.5.2 Ejemplo ilustrativo en OWL y SWRL 81
3.6 Transformaciones de modelos: desde OWL y SWRL a Jess 83
3.6.1 Clases e individuos OWL 85
3.6.2 Reglas SWRL 86
3.7 Conclusiones del capítulo 88

4 Desarrollo dirigido por modelos de la arquitectura web 91
4.1 Perspectiva general 92
4.2 Modelado de la interfaz de usuario sobre CML 96
4.2.1 Modelado de la interacción 96
4.2.2 Modelado de la presentación 99
4.3 Modelado de la arquitectura web para Java JSF 102
4.4 Transformaciones de modelos desde CML a la arquitectura web 103
4.4.1 Transformación Modelo a Modelo 103
4.4.2 Transformación Modelo a Texto 104
4.5 Modelado de la interfaz de usuario para OWL 105
4.5.1 Metamodelo de presentación web para OWL . . 106
4.6 Transformaciones de modelos desde OWL a la arquitectura web 107
4.6.1 Clases y propiedades OWL 110
4.7 Conclusiones del capítulo 112

5 Caso de estudio y evaluación 115
5.1 Un caso de estudio en el dominio agrícola 115
5.1.1 Modelo conceptual de SAVIA 117
5.1.2 Reglas de decisión para control fitosanitario . . 120
5.1.3 Transformación a modelo especifico de base de reglas y código Jess 127
5.1.4 Modelado de la aplicación Web 132
5.1.5 Transformación a modelo de la arquitectura web 134
5.1.6 Uso de SAVIA 137
5.2 Evaluación de la aplicación web del ejemplo *Familia* . . 141
5.3 Discusión 144
5.4 Conclusiones del capítulo 145

6 Conclusiones 147
6.1 Conclusiones 147
6.2 Trabajos futuros 150
6.3 Contribuciones 152
6.3.1 Revistas Internacionales 152
6.3.2 Congresos Internacionales 153
6.3.3 Congresos Nacionales 154

Bibliografía 157

A Metamodelo CML **173**
A.1 Introducción 173
A.2 Paquete *simpleCML* 176
A.2.1 Metaclases 176
A.2.2 Enumeraciones 188
A.3 Resumen 190

B Metamodelo para Reglas JESS **191**
B.1 Introducción 191
B.2 Paquete *ruleMetaModel* 193
B.2.1 Metaclases 193
B.2.2 Enumeraciones 197
B.3 Resumen 201

C Metamodelo Arquitectura Web JSF **203**
C.1 Introducción 203
C.2 Paquete *JavaBeans* 203
C.2.1 Metaclases 203
C.2.2 Enumeraciones 209
C.3 Resumen 210

D Metamodelo OWL2Web **211**
D.1 Introducción 211
D.2 Paquete *owl2web* 212
D.2.1 Metaclases 212
D.2.2 Enumeraciones 215
D.3 Resumen 215

Índice de figuras

1.1 Elementos básicos del proceso MDA 6
1.2 Metamodelos, modelos y transformaciones en MDA . . . 7
1.3 Arquitectura de una aplicación web basada en reglas . . 11
1.4 Esquema general: MDA para aplicaciones web basadas en reglas 13
1.5 Esquema general partiendo de CML 14
1.6 Esquema general partiendo de OWL y SWRL 14
1.7 Proceso para obtener la base de reglas *Jess* desde CML 16
1.8 Proceso para generar la aplicación web Java JSF desde CML 17
1.9 Proceso para generar la aplicación web Java JSF desde OWL 18
1.10 *InSCo* Suite 20

2.1 Arquitectura general de un SBC 27
2.2 Ciclo de vida de Buchanan 35
2.3 Conjunto de modelos CommonKADS 39
2.4 Methontology: actividades principales 44
2.5 Metamodelos, modelos y transformaciones en MDA . . . 54
2.6 Conceptos de transformación de modelos 55

3.1 Diagrama de clases UML del ejemplo Familia 65
3.2 Estructura del modelado de conocimiento del dominio . 66
3.3 Fragmento del metamodelo de CML: Conceptos y relaciones 67
3.4 Fragmento del metamodelo CML: Tipo de regla 68
3.5 Fragmento del metamodelo CML: Base de conocimiento e Instancia de Tipo de Regla 70
3.6 Fragmento del modelo del ejemplo Familia 70
3.7 Metamodelo para Jess (*ruleMetaModel)* 71

3.8 Resumen de la transformación ATL *CML2RuleMetamodel* 76
3.9 Ejemplo de regla ATL 77
3.10 Fragmento del modelo de reglas Jess del ejemplo Familia 77
3.11 Fragmento de plantilla Jet del ejemplo Familia 78
3.12 Fragmento del metamodelo OWL: *Clases OWL* 80
3.13 Fragmento del metamodelo OWL: *Propiedades OWL* . . 81
3.14 Clases del ejemplo Familia en OWL 82
3.15 Propiedades de objeto y aserciones sobre individuos de Familia 83
3.16 Resumen de la transformación ATL *Owl2ruleMM* 84
3.17 a) Modelo Jess de *templates* y b) código Jess obtenido . 86
3.18 a) Modelo Jess de *facts* y b) código Jess obtenido 87
3.19 Regla SWRL que define la relación tío, *uncle*, en Familia 87

4.1 Modelado de la interacción con la propiedad *NotifiesTo* 98
4.2 Aplicación web de Familia: edición de instancia de Persona 100
4.3 Aplicación web de Familia: listado de Personas 101
4.4 *JSFWebmm*: Metamodelo de la arquitectura Web JSF . 103
4.5 Resumen de la transformación ATL *CML2JavaBeans* . . 104
4.6 Fragmento del modelo basado en *JSFWebmm* del ejemplo Familia 105
4.7 Metamodelo de Presentación *OWL2Web* 108
4.8 MDD para generar la arquitetura web desde OWL . . . 109
4.9 Tabla con paginación (3 filas seleccionadas, fondo gris) . 111

5.1 Jerarquía de niveles en el proceso seguido por los expertos 118
5.2 Esquema de dominio de *SAVIA* (listado general) 119
5.3 Propiedades de *Muestreo* y detalles de la propiedad *fenologia* 119
5.4 Detalles de la propiedad derivada *criterioIntervencionTrips* de *Muestreo* 120
5.5 Tipos de reglas 121
5.6 Tipos valor (enumerados) 121
5.7 Relaciones binarias entre conceptos 122
5.8 Caso de la araña roja 122
5.9 Instancias de los conceptos *VariedadUva* y *EstadoFenológico* para *parral* 123
5.10 Instancias de conceptos de *Plaga*, *Enfermedad* y *Muestreo* 124
5.11 Doce instancias de tipo de regla *Fenologia-abstracción* . 125
5.12 Detalles de una instancia de regla *Fenologia-abstracción* 125

5.13 Lista de reglas de la base de conocimiento del *trips* . . . 126
5.14 Detalles de regla que evalúa la infestación por *trips* . . . 126
5.15 Detalles de una regla que evalúa el criterio de intervención para los *trips* 127
5.16 Listado de Fact Templates 128
5.17 Código Jess del listado de Fact Templates 128
5.18 Regla que establece las posibles fenologias en el mes de enero 129
5.19 Código Jess de la regla que establece las posibles fenologias en el mes de enero 130
5.20 Detalles de una regla que evalúa el criterio de intervención para los *trips* 130
5.21 Código Jess de una regla que evalúa el criterio de intervención para los *trips* 131
5.22 Lista de bases de hechos y detalles de hechos con recomendaciones para los *trips* 131
5.23 Código Jess de detalles de hechos con recomendaciones para los *trips* 132
5.24 Lista de *Queries* necesarias para consultar la base de hechos de Jess 132
5.25 Código Jess de una *Query* necesaria para consultar la base de hechos de Jess 133
5.26 Atributos por los que es notificado el atributo *criterioInterveciónTrips* de *Muestreo* 134
5.27 Elementos del modelo de aplicación web basado en JSF: Tipos, Paquete y JSF Project 135
5.28 JavaBeans y Enumeraciones del paquete Java 135
5.29 Detalles del atributo multivaluado *agentesPosibles* del JavaBean *Muestreo* 136
5.30 Componentes del proyecto JSF: archivo de configuracion, páginas de edición y de listado 136
5.31 Detalles del archivo de configuración *faces-config.xml*: beans gestionados y reglas de navegación 137
5.32 Lista de muestreos de un cultivo 138
5.33 Vista de la edición de un muestreo. Datos generales . . . 139
5.34 Datos de muestreo para Trips: resultado *Tratar* 140
5.35 Datos de muestreo para Trips: resultado *No Tratar* . . . 140
5.36 Tiempos de inicializacion para la aplicación web *Familia* 142
5.37 Aplicación web de Familia: listado de personas 143

5.38 Aplicación web de Familia: edición de persona 143

A.1 Metamodelo de CML: Conocimiento del Dominio, Conceptos y Relaciones 173
A.2 Metamodelo de CML: Conocimiento del Dominio, Tipos de regla 174
A.3 Metamodelo de CML: Base de conocimiento e Instancias de Tipo de Regla 174
A.4 Metamodelo de CML: Relaciones de herencia (metamodelo completo) 175

B.1 Jess Rules Metamodel 192

D.1 Metamodelo de Presentación Web para complementar OWL 211

Índice de tablas

1.1 Jerarquía de metamodelado: ejemplos UML y CommonKADS 6

2.1 Métodos de primera generación para el desarrollo de SBC 36
2.2 Métodos de segunda generación para el desarrollo de SBC 40
2.3 Metodologías en Ingeniería Ontológica 43

3.1 Operadores binarios en CML 69

4.1 Metamodelo de Presentación *OWL2Web*: Class 109
4.2 Metamodelo de Presentación *OWL2Web*: Property . . . 114

A.1 Propiedades de la clase: `Argument` 176
A.2 Propiedades de la clase: `Attribute` 177
A.3 Propiedades de la clase: `BinaryRelation` 178
A.4 Propiedades de la clase: `ComposedLogicalEquation` 178
A.5 Propiedades de la clase: `Concept` 178
A.6 Propiedades de la clase: `DomainKnowledge` 179
A.7 Propiedades de la clase: `DomainSchema` 179
A.8 Propiedades de la clase: `Equation` 180
A.9 Propiedades de la clase: `ImplicationRule` 180
A.10 Propiedades de la clase: `Instance` 181
A.11 Propiedades de la clase: `InstanciatedAttribute` . . 181
A.12 Propiedades de la clase: `KnowledgeBase` 182
A.13 Propiedades de la clase: `KnowledgeModel` 182
A.14 Propiedades de la clase: *`NamedElt`* 182
A.15 Propiedades de la clase: `RuleElement` 183
A.16 Propiedades de la clase: *`RuleType`* 183
A.17 Propiedades de la clase: `RuleTypeExpression` 183
A.18 Propiedades de la clase: `RuleTypeInstance` 184

A.19 Propiedades de la clase: `Symbol` 184
A.20 Propiedades de la clase: `SymbolList` 185
A.21 Propiedades de la clase: `Terminology` 185
A.22 Propiedades de la clase: *`TerminologyElement`* 185
A.23 Propiedades de la clase: `Tuple` 186
A.24 Propiedades de la clase: *`TypedElement`* 186
A.25 Propiedades de la clase: `Value` 187
A.26 Propiedades de la clase: `ValueSpecification` 187
A.27 Propiedades de la clase: `ValueType` 187
A.28 Enum literals: *BinaryOperator* 188
A.29 Enum literals: *Cardinality* 188
A.30 Enum literals: *LogicalOperator* 189
A.31 Enum literals: *RelationType* 189
A.32 Enum literals: *RuleElementType* 189
A.33 Enum literals: *ValueSpecificationType* 190
A.34 Enum literals: *ValueTypeType* 190

B.1 Propiedades de la clase: `RCondElem` 193
B.2 Propiedades de la clase: `RDefFact` 193
B.3 Propiedades de la clase: `RDefFacts` 194
B.4 Propiedades de la clase: `RDefSlot` 194
B.5 Propiedades de la clase: `RDefSlotValue` 194
B.6 Propiedades de la clase: `RFactSlot` 195
B.7 Propiedades de la clase: `RFactTemplate` 195
B.8 Propiedades de la clase: `RFunction` 195
B.9 Propiedades de la clase: `RModule` 196
B.10 Propiedades de la clase: `RPattern` 196
B.11 Propiedades de la clase: `RQuery` 197
B.12 Propiedades de la clase: `RRule` 197
B.13 Propiedades de la clase: `RSlotMatchExp` 198
B.14 Propiedades de la clase: `RVariable` 198
B.15 Enum literals: *RCondElemEnum* 198
B.16 Enum literals: *RFunctionType* 198
B.17 Enum literals: *RMatchOperators* 199
B.18 Enum literals: *RQueryType* 199
B.19 Enum literals: *RRuleType* 200
B.20 Enum literals: *RSlotCardinality* 200
B.21 Enum literals: *RSlotType* 200

C.1 Propiedades de la clase: `ConfigurationFile` 203

C.2 Propiedades de la clase: `Enum` 204
C.3 Propiedades de la clase: `File` 204
C.4 Propiedades de la clase: `JSFProject` 204
C.5 Propiedades de la clase: *`JSPPage`* 205
C.6 Propiedades de la clase: `JavaBean` 205
C.7 Propiedades de la clase: *`JavaBeanEntity`* 205
C.8 Propiedades de la clase: `Literal` 206
C.9 Propiedades de la clase: `ManagedBean` 206
C.10 Propiedades de la clase: `MessageEntry` 206
C.11 Propiedades de la clase: `MessagesFile` 207
C.12 Propiedades de la clase: `NavigationCase` 207
C.13 Propiedades de la clase: `NavigationRule` 207
C.14 Propiedades de la clase: `Package` 208
C.15 Propiedades de la clase: `Property` 208
C.16 Propiedades de la clase: *`Type`* 209
C.17 Enum literals: *JSPPageType* 209
C.18 Enum literals: *ScopeBean* 209
C.19 Enum literals: *SubTypes* 210

D.1 Propiedades de la clase: `Class` 212
D.2 Propiedades de la clase: `Enum` 212
D.3 Propiedades de la clase: `FormField` 213
D.4 Propiedades de la clase: `ListColumn` 213
D.5 Propiedades de la clase: `MenuItem` 213
D.6 Propiedades de la clase: `Package` 213
D.7 Propiedades de la clase: `Project` 214
D.8 Propiedades de la clase: `Property` 214
D.9 Propiedades de la clase: `TableList` 214
D.10 Propiedades de la clase: `namedElement` 214
D.11 Propiedades de la clase: `namespacedElement` 215
D.12 Enum literals: *SelectorType* 215

Resumen

Esta tesis doctoral presenta una nueva propuesta para hacer frente al desarrollo de sistemas basados en reglas integrados en aplicaciones web. La integración de motores de reglas aumenta la funcionalidad de los sistemas de información, proporcionando capacidades de razonamiento e inferencia para deducir nueva información y alcanzar conclusiones "inteligentes" a partir de la información mantenida en el sistema. Para lograr este objetivo, se ha propuesto la aplicación del desarrollo de software dirigido por modelos para sistematizar y automatizar el proceso de desarrollo de aplicaciones web basadas en reglas. Como modelos origen se emplean los modelos conceptuales propios de la Ingeniería del Conocimiento que permiten especificar la estructura de la información del dominio mediante ontologías, que combinados con las reglas, definen la semántica de una aplicación web basada en reglas. El enfoque de desarrollo dirigido por modelos aplica un conjunto de transformaciones de modelos para convertir los modelos en código. Una importante ventaja de este enfoque es que permite la definición y ejecución automática de transformaciones entre modelos y de modelos a código, reduciendo sustancialmente el tiempo de desarrollo. Por el contrario, puesto que los modelos no permiten representar todos los aspectos de una aplicación software, esto implica que muchos elementos de la aplicación deben estar predefinidos. Por tanto, la funcionalidad de una aplicación web basada en reglas queda predefinida para permitir al usuario final crear, recuperar, actualizar y eliminar instancias de las clases definidas en el dominio. A diferencia de diversas propuestas y herramientas actuales que permiten la generación automática de aplicaciones que realizan esas funciones sobre información almacenada en bases de datos relacionales, la contribución de esta tesis es que esas operaciones se ejecutan sobre la memoria de trabajo del motor de reglas, permitiendo así que este ejecute un mecanismo de inferencia (suele ser encadenamiento hacia delante o

fordward-chaining) que dirige el proceso de razonamiento deductivo.

La propuesta ha sido implementada usando las herramientas de soporte al desarrollo de software dirigido por modelos proporcionadas en Eclipse Modeling Project. Las aplicaciones web resultantes están basadas en la arquitectura Modelo-Vista-Controlador y el framework JavaServer Faces, e integran el motor de reglas *Jess* que es el encargado de proporcionar las características de inferencia. El enfoque propuesto materializa *InSCo*, una metodología que entremezcla Ingeniería del Conocimiento e Ingeniería del Software para el desarrollo de sistemas de información inteligentes híbridos.

Summary

This PhD dissertation presents an approach for specification and development of web applications performing usual functionalities of data management and incorporating a rule engine for reasoning capabilities. The integration of rule engines enhances information systems functionality, providing reasoning and inference capabilities to deduce new information and reach intelligent conclusions from managed data. To reach this goal, a model-driven software development approach has been applied to automate the development of rule-based web applications. The proposed approach is based on Knowledge Engineering formalisms for conceptual modeling combining lightweight ontologies and production rules. These models are used as the source for a model-driven method that applies several model transformations to conceptual models generating the rule-based web application code in an automatic process. An important advantage of Model-Driven Development is that it enables the definition and automatic execution of transformations between models and from models to code, reducing the development time substantially. On the contrary, since models do not represent all aspects of a rule-based web application, many elements must be predefined. Hence, the functionality for the rule-based web application is predefined to enable end-users to create, retrieve, update and delete instances. In contrast to current approaches for automatic generation of web applications that perform those functions on relational databases, our contribution executes those operations on the rule engine working memory, enabling the inference engine to execute a forward-chaining inference mechanism to drive the reasoning process.

To put our proposal into practice, it has been implemented using the tools for model-driven development provided by the Eclipse Modeling Project. The resulting web application, based on the Model-View-Controller architectural pattern and the JavaServer Faces framework,

embeds the Jess rule engine to provide inference features. The proposed approach materializes *InSCo*, a methodology intertwining knowledge engineering and software engineering approaches for hybrid intelligent information systems development.

Capítulo 1
Introducción

Esta memoria recoge el trabajo de tesis doctoral donde se presenta una propuesta de aplicación del desarrollo de software dirigido por modelos a la construcción de sistemas software basados en conocimiento, en concreto a aplicaciones web basadas en reglas. En este capítulo se introduce el contexto metodológico y tecnológico, la motivación de partida para la realización de esta tesis doctoral, se plantea el alcance de la propuesta y los objetivos a alcanzar, y se enmarca el trabajo de investigación dentro de su contexto.

1.1 Ingeniería del Conocimiento e Ingeniería del Software

La Ingeniería del Conocimiento (IC) es la parte de la Inteligencia Artificial que tiene como objetivo transformar el proceso de desarrollo de Sistemas Basados en Conocimiento (SBC) en una disciplina de ingeniería, estableciendo metodologías que permitan abordar el desarrollo de SBC de una forma sistemática (Alonso~Betanzos y cols., 2004; Palma y cols., 2000). Un SBC se define como "un sistema software capaz de soportar la representación explícita del conocimiento de un dominio específico y de explotarlo a través de los mecanismos apropiados de razonamiento para proporcionar un comportamiento de alto nivel en la resolución de problemas" (Guida y Tasso, 1994). Este tipo de software trata con problemas poco estructurados en los que podemos encontrar requisitos subjetivos, entradas inconsistentes, incompletas o con incertidumbre, y que no pueden ser resueltos aplicando algoritmos clásicos o la investigación operativa (Alonso y cols., 1995). Entre las distintas técnicas de construcción de SBC, los sistemas basados en reglas son los

más utilizados, ya que fueron los primeros en aparecer desde los inicios de los sistemas expertos en Inteligencia Artificial como técnica para representar y emular el razonamiento del humano en la toma de decisiones (Buchanan y Shortliffe, 1984; Newell y Simon, 1973).

Al igual que la Ingeniería del Software (IS) se encarga de la aplicación de la ingeniería al desarrollo del software (IEEE, 1999), la IC aplica métodos de ingeniería al desarrollo de software basado en conocimiento. En realidad IC e IS son disciplinas paralelas y complementarias, aunque presenten un desfase temporal de unos años en su evolución histórica (Águila y cols., 2014). Muchas de las técnicas que se han desarrollado dentro de la IS han sido adaptadas a la IC con relativo éxito, y al revés, técnicas que evolucionaron antes en la IC se han aplicado más tarde a la IS, aunque aún queda mucho camino por recorrer.

1.1.1 Metodologías de la Ingeniería del Conocimiento

En la actualidad las metodologías de la IC están suficientemente maduras y ofrecen un conjunto de actividades y técnicas que permiten cubrir el análisis y modelado conceptual del sistema a ser desarrollado de una manera adecuada. Entre ellas podemos destacar CommonKADS (Schreiber y cols., 2000) y la Ingeniería Ontológica (Gómez-Pérez y cols., 2004).

Sin embargo, estas metodologías sufren en general una carencia en la especificación de la transición desde los modelos conceptuales hasta los modelos computacionales, es decir, del análisis y diseño a la implementación del sistema, tal y como se entiende en Ingeniería del Software, que pondremos de manifiesto tras una breve descripción de estos dos enfoques metodológicos.

La metodología CommonKADS (CK) abarca todos los aspectos del desarrollo de un SBC, desde los análisis iniciales, que sirven para establecer la idoneidad de la solución basada en conocimiento, hasta la implementación del SBC, proporcionando un marco de trabajo donde llevar a cabo la gestión del proyecto. CK ofrece un conjunto de seis modelos organizados jerárquicamente. Los tres primeros modelos, *organización*, *tareas* y *agentes*, describen el contexto en el cual se va a desenvolver el SBC. Los modelos de *conocimiento* y *comunicación* proporcionan los requisitos de entrada que guiarán la implementación del sistema a través del último modelo, el modelo de *diseño*. El modelo central de esta metodología es el *modelo de conocimiento*, que describe el conocimiento relevante para la consecución de una determinada tarea, y

se realiza en la fase de adquisición del conocimiento. Este modelo permite describir el sistema a ser desarrollado en el nivel de conocimiento, tal y como lo definió Newell (Newell, 1982), sin hacer referencia a aspectos de implementación.

Partiendo del modelo de conocimiento, el *modelo de diseño* sirve de puente entre los modelos del nivel conceptuales y la implementación. En este último modelo se define la arquitectura del sistema a implementar a partir de una arquitectura de referencia, se identifica la plataforma de implementación, se especifican los componentes de la arquitectura y se proyectan los elementos del modelo de conocimiento en componentes de la arquitectura.

Desde el punto de vista de la Ingeniería Ontológica, se han producido numerosos avances en los últimos años principalmente en los formalismos de modelado de conocimiento e intercambio de conocimiento para la Web Semántica (Berners-Lee y cols., 2001). El lenguaje de ontologías para la web, Web Ontology Language (OWL) (Dean y cols., 2004) ha evolucionado hasta convertirse en estándar en este ámbito. Cualquier propuesta que pretenda dar soporte metodológico para la implementación de SBC a partir de modelos de conocimiento debe considerar este estándar y ser compatible con él.

1.1.2 Deficiencias en la transición al diseño de la IC

La complejidad del diseño de SBC estriba en que, en la mayor parte de los casos, la plataforma de implementación es una herramienta específica para el desarrollo de SBC que proporciona características no incluidas en el software tradicional, como técnicas de representación del conocimiento (reglas, árboles de decisión, ...), representación de incertidumbre (lógica difusa, redes bayesianas, ...), ontologías, métodos de razonamiento, aprendizaje, etc., y que constituye una máquina virtual a la que hay que traducir el modelo de conocimiento para su ejecución.

Trabajos previos de desarrollo de SBCs utilizando CK (Águila y cols., 2003; Cañadas y cols., 2002) han constatado que el modelo de diseño de CK posee importantes carencias:

- la arquitectura de referencia que plantea la metodología no se ajusta al tipo de arquitecturas distribuidas y basadas en la web utilizadas en la mayoría de los desarrollos actuales;

- la relación entre los elementos del modelo de conocimiento (reglas, tareas, inferencias, etc.) y los componentes del modelo de diseño

se establece de manera imprecisa;

- no se dispone de herramientas que faciliten la proyección de los elementos del modelo de conocimiento a clases y objetos que constituyan un modelo de diseño;

- la mayoría de las herramientas de implementación de SBC no incorporan facilidades para el desarrollo en CK, con lo que la implementación final depende de las características del entorno de implementación, y no de los modelos conceptuales y de diseño de CK.

Además, durante el mantenimiento y evolución de un SBC, la adquisición de conocimiento se prolonga en el tiempo, realizando actualizaciones conforme se tiene más conocimiento sobre el dominio. Es por ello que una mejora metodológica en la transformación de los modelos conceptuales a modelos computacionales puede suponer una ventaja sustancial en este tipo de sistemas.

La Ingeniería Ontológica se centra fundamentalmente en establecer los aspectos metodológicos y buenas prácticas para el desarrollo de ontologías. La estandarización de OWL ha favorecido la evolución de las herramientas de soporte al desarrollo de ontologías y de razonadores semánticos que permiten interpretar directamente ontologías OWL e inferir consecuencias lógicas a partir de los axiomas y hechos definidos en ellas (Carroll y cols., 2004; Sirin y cols., 2007). Sin embargo, en general sigue quedando fuera del soporte metodológico la integración estos razonadores en arquitecturas distribuidas y basadas en la web, así como los aspectos de diseño de arquitecturas software que puedan obtenerse a partir de las ontologías.

Todo ello hace necesario enriquecer las metodologías de IC mediante la aplicación de técnicas que faciliten la transformación de los modelos conceptuales a modelos de diseño y finalmente a código.

1.1.3 Desarrollo de Software Dirigido por Modelos

En el campo de la IS el problema del paso de modelos conceptuales a modelos de diseño e implementación está resuelto: existen procesos, técnicas y elementos que juegan un papel importante en la fase de diseño del sistema final y que están suficientemente probados y validados. Se trata del Desarrollo de Software Dirigido por Modelos.

El Desarrollo de Software Dirigido por Modelos (DSDM), nombre adaptado del original Model-Driven Architecture (MDA) (Object Management Group, 2003a) acuñado por el Object Management Group (OMG), aunque otros nombres como Model-driven Engineering (MDE) (Schmidt, D.C., 2006) o Model-driven Development (MDD) (Mellor y cols., 2003) también se utilizan comúnmente para denominar este enfoque de desarrollo de software basado en el modelado de software y en la definición de transformaciones entre modelos y de modelos a código, permitiendo el desarrollo de generadores de código a partir de modelos.

En los últimos años el enfoque MDA/MDE está siendo aplicado en multitud de dominios, entre otros la Ingeniería Web (Moreno y cols., 2008b), el desarrollo de ontologías (Gašević y cols., 2006) y el desarrollo de interfaces de usuario (Hussmann y cols., 2011). En este trabajo se plantea extrapolar estos resultados al campo de la IC facilitando las transiciones entre modelos que evitarán las deficiencias descritas previamente en esta sección.

La figura 1.1 muestra los elementos básicos del proceso de desarrollo MDA. Lo forman dos elementos fundamentales: los modelos y las transformaciones de modelos. Se definen tres tipos de modelos. El primer tipo es un modelo independiente de plataforma (PIM, Platform Independent Model) con un alto nivel de abstracción que es independiente de cualquier tecnología de implementación. Un modelo PIM representa el problema utilizando los conceptos de la realidad del dominio de aplicación, sus propiedades y relaciones. Por ejemplo, si el dominio del problema es el sector bancario, ejemplos de conceptos serían “cliente” y “cuenta bancaria”; propiedades de los conceptos serían “número de teléfono” y “saldo”; y una relación entre conceptos sería “la cuenta bancaria es de un cliente”. El proceso de desarrollo establece la transformación del PIM en uno o más modelos específicos de plataforma (PSM, Platform Specific Model). Un PSM representa el sistema en términos de los constructores que están disponibles en una tecnología de implementación concreta. Así por ejemplo, un modelo específico de plataforma para bases de datos relacionales utiliza elementos como “tabla”, “columna”, “clave externa”, etc., para ofrecer una especificación del sistema basada en esa tecnología. El concepto “cliente” ahora se representa como la tabla “cliente”, y la propiedad “saldo” como la columna “saldo” de la tabla “cuenta bancaria”. El paso final en un desarrollo según MDA es la transformación del PSM a código. Esta transformación es relativamente directa porque un PSM es un modelo directamente relacionado con la

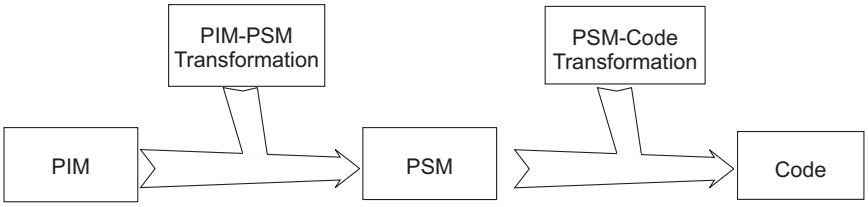


Figura 1.1: Elementos básicos del proceso MDA

Tabla 1.1: Jerarquía de metamodelado: ejemplos UML y CommonKADS

Nivel	*Descripción*	*Modelado UML*	*Modelado CommonKADS*
M3	**MOF**	MOF Class	MOF Class
M2	**Metamodelos**, instancias de elementos de MOF	Clase, Atributo, Asociación, ...	Tarea, Inferencia, Regla, ...
M1	**Modelos**, instancias de elementos de M2	Cliente, Nombre, Cuenta, saldo, ...	Diagnosis, seleccionar, hipótesis, ...
M0	**Instancias**: objetos y datos de elementos de M1	C1: Cliente nombre = "Jose" dirección="C/Paseo."	aDiagnosis: Diagnosis hallazgos = "fiebre", "sudor" hipotesis= "infección viral"

tecnología de implementación.

MDA organiza los modelos jerárquicamente en cuatro niveles, representados en la tabla 1.1. El nivel superior es el meta-metamodelado, M3, que tiene la responsabilidad de definir el lenguaje para la especificación de lenguajes de modelado. Meta Object Facility (MOF) (Object Management Group, 2003b) es el lenguaje abstracto común de meta-metamodelado en MDA. En el nivel M2 se definen los metamodelos. Un metamodelo es el modelo de un lenguaje de modelado. Por ejemplo, Unified Modeling Language (UML) y otros lenguajes de modelado están definidos a partir de su metamodelo (Object Management Group, 2003c). El nivel M1 contiene los modelos: especificaciones de sistemas concretos en cualquier ámbito o dominio de aplicación creados en un lenguaje de modelado específico. Los modelos se definen de acuerdo con el metamodelo del lenguaje de modelado utilizado. Por último, el nivel más bajo de la jerarquía es M0 que representa las instancias en tiempo de ejecución de los elementos definidos en un modelo. En la tabla 1.1 aparecen ejemplos de cada uno de los niveles para dos lenguajes de modelado: UML y CommonKADS.

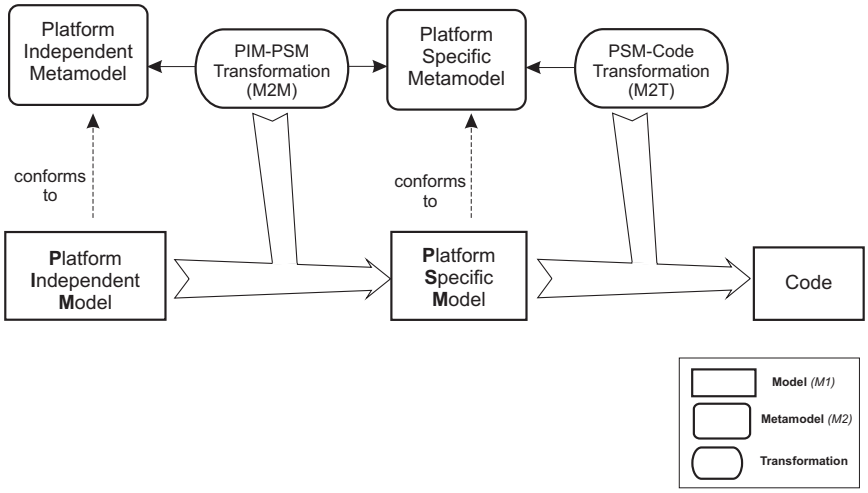


Figura 1.2: Metamodelos, modelos y transformaciones en MDA

El otro elemento fundamental en MDA son las transformaciones de modelos (ver figura 2.5). Una transformación de modelos consiste en definir cómo convertir un modelo en otro modelo, que puede ser del mismo nivel de abstración (de PIM a PIM) o de distinto nivel de abstracción (de PIM a PSM y de PSM a código). Las transformaciones entre modelos reciben el nombre de transformaciones modelo a modelo (M2M, Model to Model), mientras que las transformaciones de modelo a código reciben el nombre de transformaciones modelo a texto (M2T, Model to Text) o también modelo a código (M2C, Model to Code). Las herramientas de soporte a MDA permiten diseñar y ejecutar estas transformaciones utilizando lenguajes específicos para ello.

Uno de los parámetros que hay que establecer cuando se pretende aplicar un enfoque de desarrollo de software dirigido por modelos es la plataforma del software destino, es decir, la tecnología de implementación y la arquitectura de la aplicación a desarrollar. Esto determina los modelos y lenguajes de modelado específicos de plataforma, así como las transformaciones de modelos, necesarios para la puesta en práctica del proceso. En el caso de construir software basado en conocimiento se mantiene la necesidad de establecer la tecnología de implementación y la arquitectura software, pero además, el abanico de posibilidades de multiplica puesto que existen numerosos métodos de representación del conocimiento, complicando así el proceso.

1.1.4 Sistemas basados en reglas

Las reglas de producción (IF-condición-THEN-acción) son el esquema de representación de conocimiento más popular. Las reglas han sido aplicadas satisfactoriamente en sistemas expertos y SBC, donde han demostrado su validez para la representación declarativa del conocimiento del experto y de la lógica de negocio sobre numerosos dominios de aplicación (Durkin, 1993). Los motores de reglas trabajan con conjuntos de reglas (bases de reglas) y ejecutan métodos de inferencia para disparar las reglas adecuadas cuyas condiciones son verdaderas en base a la información conocida por el sistema, consiguiendo alcanzar nuevos resultados, que se deduzca nueva información y llegar a conclusiones aplicando un proceso deductivo (Brachman y Levesque, 2004).

El concepto de *regla* también se utiliza actualmente en Ingeniería del Software, ya que las reglas de negocio han demostrado su utilidad para el modelado de la lógica de negocio como parte de la especificación de sistemas software (IIBA, 2015). Hoy en día, ambos puntos de vista han convergido, favoreciendo una adopción extensiva de los sistemas basados en reglas en la implementación de sistemas software para procesos de toma de decisiones complejos (Object Management Group, 2008c), lo que hace atractivo trasladar el enfoque MDA a sistemas basados en reglas.

1.1.5 Aplicaciones web

En IS se denomina aplicación web a aquellos productos software que los usuarios pueden utilizar accediendo a un servidor web a través de Internet o de una intranet mediante un navegador. El concepto inicial de sitio Web, que inicialmente era un conjunto de documentos estáticos accesibles desde cualquier parte del mundo (Berners-Lee, 1992), evolucionó hacía el concepto de las aplicaciones Web actuales, que proveen páginas dinámicas que se alimentan con información almacenada en bases de datos, y ejecutan programas con una serie de funcionalidades, actualizando los contenidos de manera rápida y sencilla (Baresi y cols., 2000). Formalmente se denominan aplicaciones web intensivas en datos aquellos sistemas de información basados en la web que permiten el acceso y la gestión de grandes cantidades de datos estructurados, almacenados normalmente en bases de datos relacionales (Ceri y cols., 2002b). Para su desarrollo es necesario la definición de modelos de datos que representan la estructura de la información, así como el diseño de

interfaces gráficas de usuario que permitan interactuar adecuadamente con el sistema para el acceso, visualización, introducción, modificación y consulta de la información mantenida en el sistema.

Las aplicaciones web presentan una serie de características que las convierten en productos software de gran valor, de entre ellas podemos destacar algunas como la capacidad de ser accesibles y operables desde cualquier plataforma y ubicación, y la capacidad de distribuirse con mayor facilidad puesto que no es necesario su descarga para una posterior instalación o configuración. Además el usuario dispone siempre de una versión actualizada de la aplicación, sin necesidad en tareas de actualización. Todas estas características han propiciado que las aplicaciones web hayan sido adoptadas actualmente como un elemento esencial e indispensable en toda actividad empresarial, sistemas de información y redes sociales. Algunos ejemplos de aplicaciones web ampliamente conocidos son los webmails, wikis, weblogs, tiendas en línea, sistemas de reservas de viajes, gestores de contenidos, y un largo etcétera.

1.1.6 Aplicaciones web basadas en reglas

La evolución de las tecnologías web propició que el interés en desarrollar SBC en forma de aplicaciones web creciera rápidamente. El objetivo era ofrecer en Internet sistemas que tecnológicamente ya llevaban un recorrido de décadas, como eran los sistemas expertos y los SBC, pero ahora aprovechando las ventajas de las aplicaciones web. Algunos primeros trabajos establecían las bases del potencial que ofrecía Internet y sus tecnologías asociadas como entorno de desarrollo y despliegue de sistemas expertos (Duan y cols., 2005; Grove, 2000; Huntington, 2000; Simpson y cols., 1999; Yao y Yao, 2003).

Un caso de esta evolución de SBC en la web son los sistemas basados en reglas. La especificación y diseño de lenguajes de reglas y motores de inferencia que proporcionen capacidades de razonamiento en las aplicaciones basadas en la web es un tema de investigación dentro del ámbito de la Web Semántica (Berners-Lee y cols., 2001). Se han definido varias propuestas para la representación de modelos de reglas (Eiter y cols., 2008), una de las más representativas es el lenguaje de reglas para la web semántica, Semantic Web Rule Language (SWRL) (Horrocks y cols., 2004), que permite la especificación de reglas que complementan a las ontologías definidas en OWL.

Con el objetivo de desarrollar aplicaciones basadas en reglas para la web, entendemos por **aplicación web basada en reglas** una aplicación

web intensiva en datos que integra un motor de inferencia para reglas de producción, lo que aumenta la funcionalidad del sistema proporcionando capacidades de razonamiento e inferencia para deducir nueva información y alcanzar conclusiones "inteligentes" a partir de la información mantenida en el sistema.

La figura 1.3 muestra la arquitectura de una aplicación web basada en reglas que constituye el tipo de aplicación que queremos obtener aplicando el proceso dirigido por modelos propuesto en este trabajo de tesis doctoral. La arquitectura está basada en el patrón arquitectónico modelo-vista-controlador (MVC), la plataforma Java Enterprise Edition (JavaEE) (Oracle, 2009) y el framework JavaServer Faces (JSF) (Geary y Horstmann, 2007) como plataforma de implementación que permite el desarrollo de aplicaciones web basadas en Java, y componentes gráficos definidos en la librería JBoss Richfaces (JBoss, 2009) que ofrecen propiedades de aplicaciones web ricas (RIAs, Rich Internet Applications) (Driver y cols., 2005).

Dentro de la arquitectura destaca el motor de inferencia, en este caso *Jess*, Java Expert System Shell (Friedman-Hill, 2003), embebido en la arquitectura, que gestiona la base de reglas que es el archivo de texto que contiene el código fuente de las reglas *Jess* y las instancias de conceptos del dominio, llamados *hechos (facts)*. El motor de inferencia consta de tres componentes principales: la memoria de trabajo, que mantiene toda la información o *hechos*, la base de reglas que contiene todas las reglas de producción definidas, y el motor de inferencia propiamente dicho, que es el proceso que se encarga de comprobar si las condiciones de las reglas concuerdan con los hechos de la memoria de trabajo, y en tal caso ejecuta (*dispara*) las reglas. La arquitectura de la aplicación web se completa con un conjunto de clases Java que implementan la lógica de negocio de la aplicación, que corresponden a las capas del modelo y del controlador en la arquitectura MVC, y una serie de páginas JSF que componen la interfaz para el usuario final y permiten la interacción con el sistema, que corresponden a la vista.

La funcionalidad básica de una aplicación web basada en reglas debe permitir al usuario final realizar las operaciones de crear, recuperar, actualizar y eliminar instancias de las clases definidas en el dominio, funcionalidad conocida como CRUD (Create, Read, Update, Delete). Estudios como (Albert y cols., 2010) muestran que estas operaciones suponen un 80 % de la funcionalidad en aplicaciones intensivas en datos convencionales. A diferencia de diversas propuestas y herramientas de

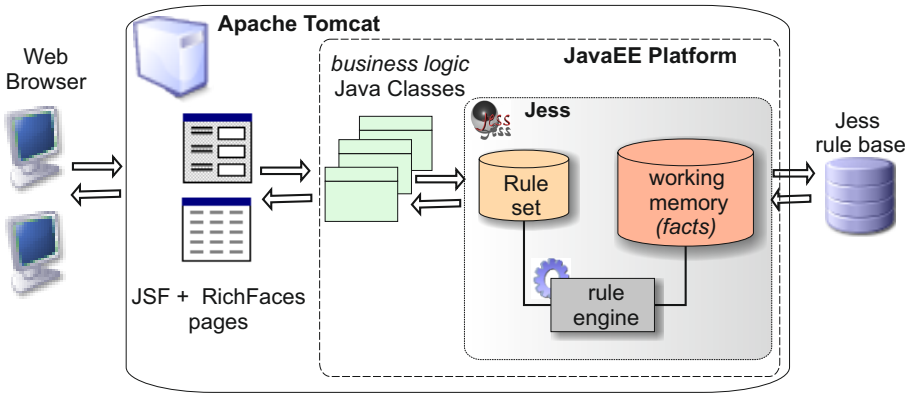


Figura 1.3: Arquitectura de una aplicación web basada en reglas

desarrollo de software actuales que permiten la generación automática de aplicaciones CRUD sobre información almacenada en bases de datos relacionales, nuestra contribución es que esas operaciones son ejecutadas sobre la memoria de trabajo del motor de inferencia, permitiendo así que este ejecute un mecanismo de inferencia de encadenamiento hacia delante (*fordward-chaining*) que dirige el proceso de razonamiento deductivo disparando las reglas cuyas condiciones son evaluadas como verdadero, y ejecutando las acciones definidas en las mismas para modificar la información existente y derivar nueva información.

Esta arquitectura se beneficia de tener la lógica de decisión separada del código de la aplicación, es decir, las reglas se encuentran definidas en la base de reglas y son independientes del código fuente de la aplicación. Este desacoplamiento incrementa la escalabilidad y mantenibilidad de las aplicaciones web basadas en reglas (Frankel y cols., 2004).

1.2 Hipótesis de partida y objetivos

Esta tesis doctoral trata de hacer frente al desarrollo de aplicaciones web basadas en reglas a partir de los modelos conceptuales definidos en lenguajes propios de la IC. Para lograrlo se parte de la hipótesis de que el desarrollo de software dirigido por modelos debe permitir la formalización y automatización del proceso de desarrollo de aplicaciones web basadas en reglas a partir de los modelos conceptuales de la IC, los cuales especifican la estructura de la información del dominio, y combinados con las reglas, definen la lógica de decisión de la aplicación. Y para

obtener una aplicación web potente y personalizada, los modelos conceptuales deben ser enriquecidos mediante la incorporación del modelado de características propias de aplicaciones web.

Así, se han definido los siguientes objetivos generales:

1. Definir una propuesta de aplicación del desarrollo de software dirigido por modelos para sistematizar el proceso de construcción de sistemas web basados en reglas a partir de los modelos conceptuales de la IC.

2. Desarrollar los metamodelos de los lenguajes a utilizar en el proceso, los editores de modelos y las trasformaciones de modelos.

3. Hacer que el proceso definido dé soporte al desarrollo de la base de reglas en una tecnología de sistemas basados en reglas a partir de los modelos conceptuales de la IC.

4. Además debe dar soporte al desarrollo de la aplicación web intensiva en datos que integra un motor de inferencia basado en reglas y proporciona la funcionalidad de razonamiento característica de los SBC.

5. Aplicar el proceso propuesto al desarrollo de en un caso de estudio real, en concreto en el dominio agrícola.

1.3 Alcance de la propuesta

Una vez definido el contexto tecnológico y metodológico, y expuesta la hipótesis de partida, a continuación se describe el alcance de esta tesis doctoral, refinando los objetivos generales expuestos anteriormente.

La figura 1.4 muestra un esquema general del proceso dirigido por modelos propuesto para el desarrollo de aplicaciones web basadas en reglas. Está dividido en dos subprocesos: el primero (el flujo *inferior* en la figura 1.4) genera la implementación de la base de reglas a partir de un modelo conceptual a nivel de conocimiento que contiene la ontología del dominio y la lógica de decisión en forma de reglas; y el segundo proceso (el flujo *superior* en la figura 1.4) produce el código de los componentes de la arquitectura web a partir del mismo modelo original enriquecido con características para modelado de la interfaz de usuario de la aplicación web.

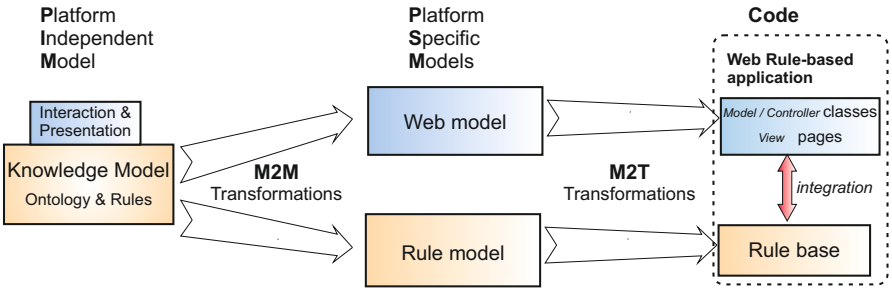


Figura 1.4: Esquema general: MDA para aplicaciones web basadas en reglas

La aplicación del proceso comienza con la especificación de un modelo independiente de plataforma (PIM), en nuestro caso un modelo conceptual de la ontología del dominio de aplicación junto con el conjunto de reglas con la lógica de decisión del problema. En este trabajo se han considerado dos alternativas como formalismos para representar los modelos de ontologías y reglas independientes de plataforma: Conceptual Modeling Language (CML), propuesto en la metodología CommonKADS, y Web Ontology Language (OWL) (Dean y cols., 2004) junto con Semantic Web Rule Language (SWRL) (Horrocks y cols., 2004), propios de la Ingeniería Ontológica y la Web Semántica.

Como plataforma de implementación de reglas se ha utilizardo el motor de inferencia *Jess* (Friedman-Hill, 2003), uno de los motores de reglas para Java más populares, que puede ser embebido en aplicaciones Java proporcionando una comunicación bidireccional flexible en tiempo de ejecución.

Y respecto a la aplicación web, la arquitectura está basada en el patrón arquitectónico modelo-vista-controlador (MVC), la plataforma Java Enterprise Edition (JavaEE) (Oracle, 2009) y el framework JavaServer Faces (JSF) (Geary y Horstmann, 2007), que permite el desarrollo de aplicaciones web intensivas en datos basadas en Java.

Para poner nuestra propuesta en práctica, se han diseñado e implementado los metamodelos de los lenguajes de modelado y las transformaciones de modelos implicadas en el proceso empleando el entorno en Eclipse Modeling Framework (EMF) (Steinberg y cols., 2008), que es el núcleo de las herramientas de MDD proporcionadas en el Eclipse Modeling Project[1]. Así, para poner en práctica las propuestas de esta

[1]http://www.eclipse.org/modeling/

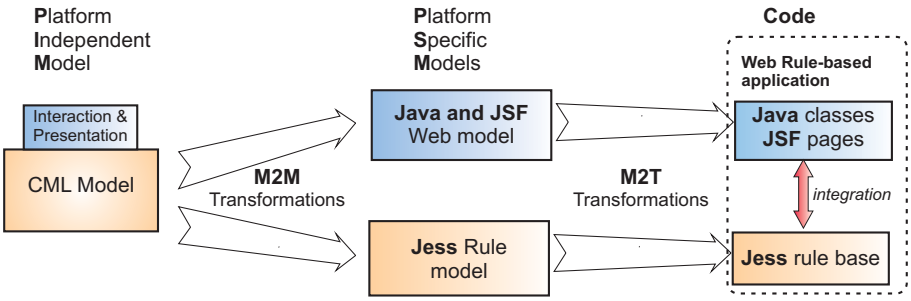


Figura 1.5: Esquema general partiendo de CML

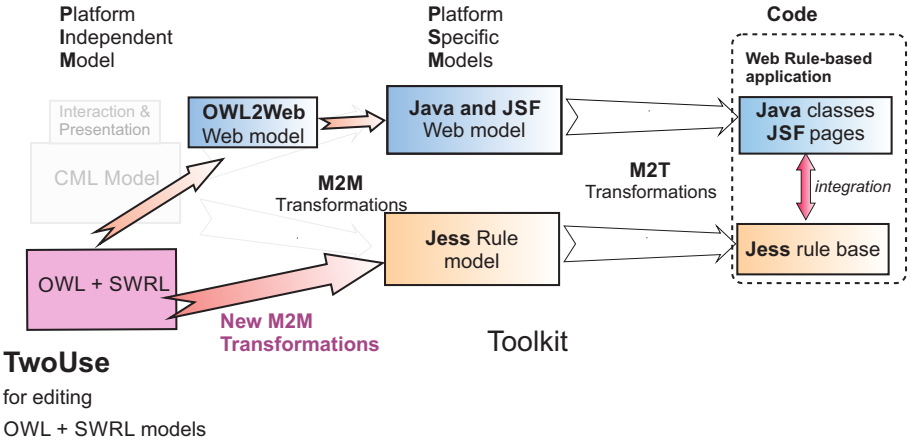


Figura 1.6: Esquema general partiendo de OWL y SWRL

tesis doctoral, se han diseñado e implementado los metamodelos de los lenguajes de modelado utilizando EMF y su formato Ecore; para personalizar los editores de modelos basados en los metamodelos Ecore se ha utilizado Exeed (EXtended Emf EDitor) (Kolovos, 2007), que ofrece una versión mejorada del editor reflexivo basado en árbol de EMF; por su parte, las transformaciones modelo a modelo se han diseñado con ATL (Atlas Transformation Language) (Jouault y cols., 2008) y las transformaciones modelo a texto con JET (Java Emitter Templates) (Eclipse Modeling Project, 2007).

Como se ha comentado al inicio de esta sección, el enfoque dirigido por modelos propuesto está dividido en dos subprocesos, el que genera la base de reglas *Jess* y el que genera la aplicación web, por tanto ambos producen resultados diferenciados. Sin embargo, aunque estos dos subprocesos son ejecutados independientemente uno de otro, el resultado

final debe integrar la base de reglas en la aplicación web. Esto se consigue produciendo un código de la aplicación web que hace llamadas a métodos de la librería Java de *Jess*, la cual proporciona una API (Application Programming Interface) de componentes que son importados en la aplicación Java y referenciados desde el código Java de la aplicación web generada, consiguiendo así la integración del motor de inferencia *Jess*.

Esta configuración hace posible que los dos procesos MDD sean ejecutados separadamente, y por tanto, cualquier cambio en el modelo de reglas que afecte unicamente a la lógica de decisión representada en las reglas pero que no afecte a la estructura de la información (conceptos, atributos, relaciones, etc), puede ser trasladado en una nueva base de reglas sin tener que modificar o regenerar los componentes de la arquitectura web. Con este enfoque se consiguen aplicaciones web más fáciles de mantener y evolucionar.

A continuación se describen estos dos procesos, considerando en ambos las dos alternativas utilizadas como modelos origen del proceso: CML (ver figura 1.5) y OWL (ver figura 1.6).

1.3.1 Desarrollo dirigido por modelos de la base de reglas

Este primer proceso se encarga de obtener la base de reglas a partir de los modelos conceptuales basados en conocimiento.

El modelo conceptual con la ontología del dominio y las reglas es transformado a una base de reglas *Jess*, tecnología que permite la implementación de sistemas basados en reglas fuertemente acoplados a aplicaciones Java. La base de reglas *Jess* contiene, entre otros elementos, el conjunto de reglas en la sintaxis de *Jess*. La figura 1.7 describe este proceso para el caso de modelado CML que ha implicado la realización de las siguientes tareas: se ha diseñado el metamodelo de CML utilizando EMF, así como una herramienta de modelado basada en dicho metamodelo; igualmente, se ha diseñado el metamodelo específico de plataforma para reglas *Jess* y la herramienta de modelado correspondiente; para completar el proceso, se han diseñado la transformaciones M2M desde CML a modelos de reglas *Jess* utilizando el lenguaje ATL, y la transformación M2T desde modelo de reglas *Jess* a código *Jess* utilizando el lenguaje JET basado en plantillas.

En el caso alternativo de utilizar modelos OWL y SWRL como modelos de conocimiento origen para el proceso, se sustituye CML por OWL y SWRL como modelos independientes de plataforma (PIM), manteniendo

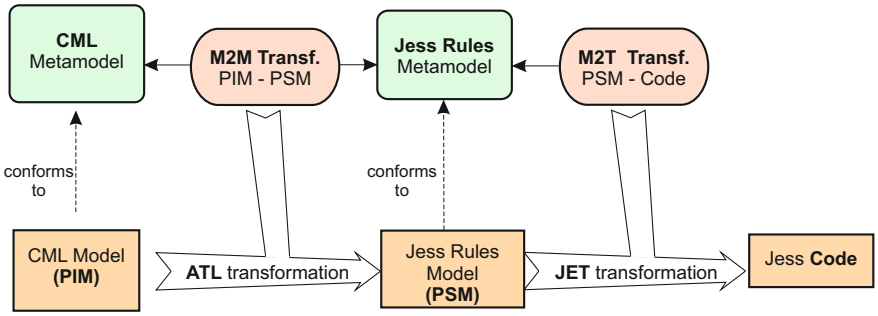


Figura 1.7: Proceso para obtener la base de reglas *Jess* desde CML

el modelado específico de plataforma *Jess*. Para la creación y edición de los modelos OWL y SWRL se ha usado de la herramienta *TwoUse Toolkit* (Parreiras y Staab, 2010; Parreiras y cols., 2007) que ya implementa el metamodelo OWL estándar, Ontology Definition Metamodel (ODM) (Object Management Group, 2009a), y además proporciona un entorno gráfico para la especificación de ontologías OWL y SWRL (ver figura 1.6). Esto ha servido como base para implementar una nueva transformación desde OWL y SRWL al metamodelo de reglas *Jess* sobre el que se vuelve a aplicar la transformación M2T ya definida.

1.3.2 Desarrollo dirigido por modelos de arquitectura web

El segundo proceso dirigido por modelos se encarga de obtener arquitectura de la aplicación web partir de los modelos conceptuales a nivel de conocimiento. Esta Tesis doctoral no tiene como objetivo proponer un formalismo para modelado Web, ya que ese problema entraría dentro del ámbito de la Ingeniería Web donde hay diversas propuestas en ese sentido, aunque sí aplica conceptos e ideas de Ingeniería Web para lograr el objetivo de integrar una base de reglas en una arquitectura Web aplicando el desarrollo dirigido por modelos. La arquitectura de aplicación web es generada a partir del modelo conceptual el cual se enriquece con el modelado de la interacción y presentación de la interfaz de usuario web, para producir así una aplicación web más potente y personalizada.

Se han aplicado dos aproximaciones para el modelado de las características de diseño de la aplicación web, según se utilice como modelo conceptual origen del proceso un modelo CML o bien un modelo OWL y SWRL. La primera aproximación ha consistido en anotar el modelo

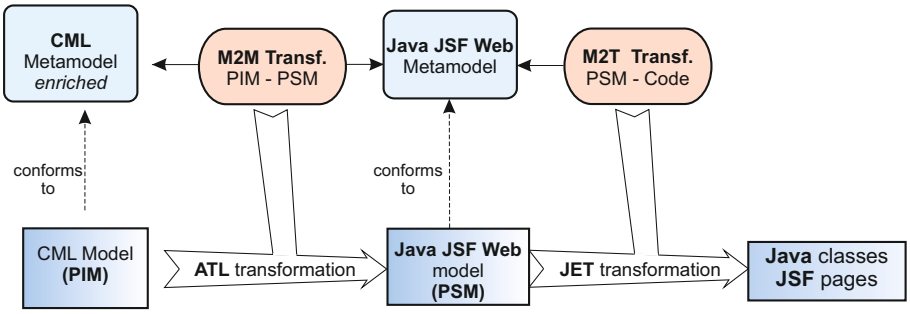


Figura 1.8: Proceso para generar la aplicación web Java JSF desde CML

conceptual con propiedades específicas de la aplicación web, y se ha aplicado al modelado CML, ya que se trata de un metamodelo diseñado en este trabajo de tesis a partir de su definición original, y por tanto, se ha podido modificar el metamodelo para añadirle las propiedades que permiten modelar características de diseño de aplicación web.

La figura 1.8 muestra el esquema general de este proceso para el caso de CML, que ha implicado añadir las características de interacción y presentación sobre los modelos de conocimiento, el diseño del metamodelo para aplicaciones web basadas en Java y JSF y su herramienta de modelado correspondiente, así como las transformaciones M2M y M2T necesarias. La aplicación web generada está basada en la arquitectura descrita en la sección 1.1.6.

Y la segunda aproximación, aplicada a la propuesta de usar OWL, ha consistido en mantener la separación de aspectos (Ernst, 2003), propia de la Ingeniería Web, y usar diferentes modelos, uno para cada aspecto. Así, los modelos conceptuales OWL y SWRL sin alteración, definidos mediante el estándar ODM, han sido usados para el modelado conceptual que define la estructura de la información, y se ha añadido un nuevo modelo web específico para modelar las peculiaridades de la interfaz de usuario web, lo que ha llevado a la definición de este nuevo metamodelo, denominado *OWL2Web*, para complementar OWL en este aspecto. Además, ha sido necesario definir las transformaciones de modelos desde OWL y el nuevo modelo *OWL2Web* hasta el modelo Java JSF, sobre el que se vuelve a aplicar la transformación M2T ya definida (ver figura 1.9).

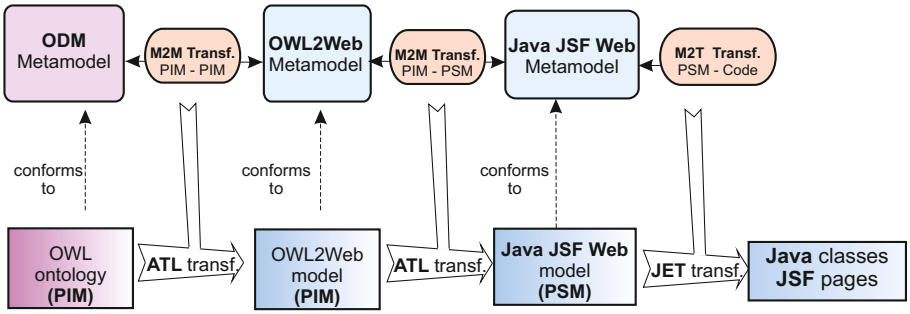


Figura 1.9: Proceso para generar la aplicación web Java JSF desde OWL

1.3.3 Caso de estudio en el dominio agrícola

El proceso de desarrollo propuesto ha sido aplicado en SAVIA, un sistema de información web basado en reglas para la gestión de información y el control de plagas en cultivos agrícolas.

El impacto de las plagas y enfermedades sobre las producciones agrícolas es un factor de elevada relevancia en la industria agrícola actual. Métodos como la Producción Integrada (Norris y cols., 2003) tratan de reducir ese impacto manteniendo las plagas por debajo de un umbral de acción, maximizando el uso de los recursos de producción y mecanismos naturales. El sistema de ayuda a la decisión SAVIA aplica razonamiento basado en reglas para ayudar a agricultores y técnicos agrícolas en el cumplimiento de la regulación de calidad Producción Integrada respecto la control de plagas, informando si es necesario o no realizar un control sobre la plaga a partir de los valores observados en el muestreo del cultivo. El criterio de intervención ofrecido por SAVIA es orientativo, ya que en última instancia el ingeniero agrícola responsable de la explotación es quien decide qué acción tomar, realizando ajustes o modificaciones sobre el criterio ofrecido por el sistema.

La sanidad vegetal constituye un dominio con mucha variabilidad respecto a los cultivos, las plagas y enfermedades que afectan a la producción agrícola. En nuestro área de estudio, geográficamente localizada en el sureste español, los cultivos tenidos en cuenta son el parral, el olivo y los vegetales que se cultivan en invernadero: tomate, berenjena, calabacín, pimiento, pepino, judía verde, melón y sandía. Respecto a las plagas y enfermedades que afectan a estas especies vegetales, existen siete plagas para el parral y nueve para el olivo. Cada pareja cultivo/paga constituye una pieza o módulo de conocimiento que tiene que ser adqui-

rido y representado, así que estos números se traducen en la necesidad de adquirir siete y nueve módulos de conocimiento diferentes respectivamente, es decir, siete y nueve bases de conocimiento independientes, una para cada combinación de cultivo/plaga. Este número se multiplica de manera extrema en el caso de los vegetales que se producen en invernadero, puesto que a cada especie vegetal le afectan entre 17 y 23 plagas y enfermedades distintas, dando lugar a más de 160 combinaciones diferentes. Por si fuera poco, año tras año las plagas evolucionan genéticamente ofreciendo mecanismos de resistencia a los métodos de tratamiento establecidos. Es por ello que el desarrollo de un sistema de ayuda a la decisión en este dominio conlleva una necesidad de mantenimiento frecuente y evolución del conocimiento a lo largo el tiempo. El enfoque MDA es muy apropiado en este ámbito, por las razones expuestas en las secciones anteriores.

1.4 Contexto del trabajo de investigación

El trabajo de investigación se ha realizado dentro del grupo de investigación DKSE: Ingeniería de Datos, del Conocimiento y del Software (TIC-181) de la Universidad de Almería, y en colaboración con el grupo de investigación AIKE: Inteligencia Artificial e Ingeniería del Conocimiento, de la Universidad de Murcia. El grupo DKSE actualmente trabaja en distintos campos de la informática, centrando sus líneas de investigación en Bases de Datos e Ingeniería del Conocimiento e Ingeniería del Software, trasladando los resultados a diversos campos de aplicación como la gestión de información basada en la Web, sistemas de apoyo a la toma de decisiones, en especial en el dominio agrícola.

El origen de esta tesis se gestó en el marco del proyecto "Metodología para el análisis y diseño de sistemas híbridos: Integración de técnicas de modelado de software y de conocimiento" (TIN2004-05694). Este proyecto tenía un doble objetivo. Por una parte, la definición de una metodología que permitiera el desarrollo de sistemas software híbridos que combina componentes basados en conocimiento y no basados en conocimiento. Por otra, la validación de la metodología propuesta aplicándola en el desarrollo de un producto software para ayuda en la toma de decisiones en el seguimiento de cultivos bajo la norma de calidad de la Producción Integrada.

El proyecto se ha planteado paralelamente en el nivel conceptual y en el computacional. En el primero de ellos, desarrollado por parte de

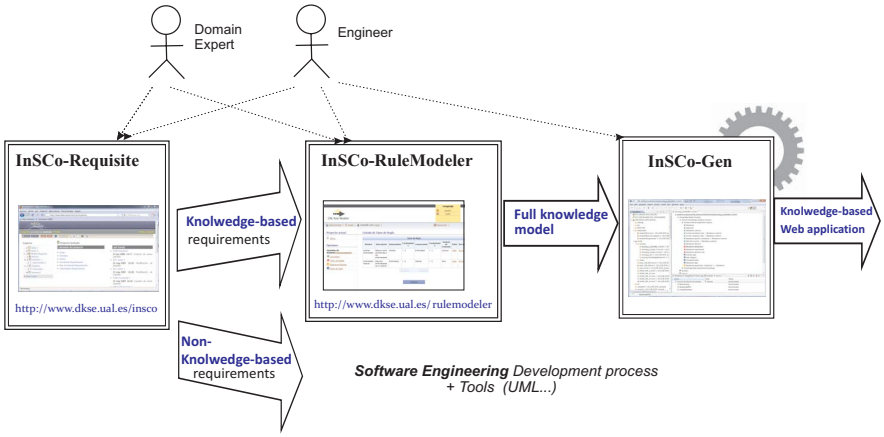


Figura 1.10: *InSCo* Suite

otros miembros del grupo de investigación, se realizó la definición del modelo de proceso InSCo (Águila, 2010); y en el segundo, en el que se enmarca este trabajo de tesis, se propone la transformación de los modelos conceptuales a modelos computacionales utilizando tecnologías propias de MDA.

En la definición de la propuesta InSCo en el marco del proyecto de investigación TIN2004-05694 se estableció la necesidad de desarrollar una suite de herramientas para la implantación del modelo de proceso InSCo (Cañadas, Orellana, Águila, Palma, Túnez, 2009). Este conjunto de tres herramientas software interoperables dan soporte al modelo de proceso propuesto, constituyendo la denominada *InSCo-Suite* que se muestra en la figura 1.10.

InSCo-Requisite, desarrollada por otros miembros del grupo de investigación en el ámbito del proyecto, permite la especificación de los aspectos conceptuales (Orellana y cols., 2008) y se utiliza como entrada a *Rule-Modeler* (Cañadas y cols., 2010), una aplicacion web que permite editar y especificar las partes conceptuales basadas en conocimiento. Finalmente, estos modelos conceptuales basados en conocimiento son la entrada a *InSCo-Gen* que aplica MDA para trasladar estos modelos conceptuales a modelos computacionales (Cañadas y cols., 2009).

El marco propuesto materializa *InSCo* (Águila y cols., 2006), una metodología que entremezcla Ingeniería del Conocimiento e Ingeniería del Software para el desarrollo de sistemas de información inteligentes híbridos.

1.5 Estructura de la memoria

Esta memoria de tesis doctoral está estructurada de la siguiente forma:

- En el "Capítulo 2 - Estado de la cuestión" se presenta el estado del arte.
- Tras esto el "Capítulo 3 - Desarrollo dirigido por modelos de la base de conocimiento" introduce el proceso de desarrollo dirigido por modelos para obtener una base de reglas a partir de un modelo conceptual basado en reglas. El proceso se ilustra utilizando el caso de estudio de las relaciones familiares (Horrocks y cols., 2004), un problema basado en reglas sencillo que es usado frecuentemente en la literatura.
- A continuación, el "Capítulo 4 - Desarrollo dirigido por modelos de la arquitectura web" describe el proceso de desarrollo dirigido por modelos para la obtención de la aplicación web que integra el motor de inferencia y es capaz de ejecutar la base de reglas.
- El siguiente "Capítulo 5 - Caso de estudio y evaluación", presenta una evaluación del proceso propuesto y el caso de estudio SAVIA en dominio agrícola.
- Por último el "Capítulo 6 - Conclusiones" resume las principales conclusiones y trabajos futuros obtenidos de la realización de esta tesis doctoral.
- Tras la Bibliografia, varios Apéndices detallan los metamodelos diseñados para dar soporte al proceso propuesto.

Capítulo 2

Estado de la cuestión

En este capítulo se hace una revisión del estado del arte de las técnicas y métodos en los que se basa el desarrollo de esta tesis doctoral.

En primer lugar se introduce la Ingeniería del Conocimiento como disciplina y la problemática que aparece en la construcción de SBC trasladada a las etapas de diseño. Entre las tecnologías de desarrollo de SBC, los sistemas basados en reglas destacan por su amplio uso, por lo que se hace una revisión de los métodos de modelado de sistemas basados en reglas así como de las tecnologías de implementación de estos sistemas.

A continuación se introduce el desarrollo de software dirigido por modelos como solución que aporta la Ingeniería del Software al problema de la transición de modelos conceptuales a modelos de diseño y a código.

Para terminar, tal como se ha dicho en el capítulo 1, centramos la problemática de desarrollar SBC para la web, por lo que se introducen los métodos de Ingeniería Web y la aplicación de MDA al desarrollo de aplicaciones web. Finalmente se concluye con una revisión de las aplicaciones de MDA a sistemas basados en reglas existentes.

2.1 La Ingeniería del Conocimiento

El Diccionario de la Real Academia Española define "ingeniería" como "el estudio y aplicación, por especialistas, de las diversas ramas de la tecnología", y "conocimiento" como "acción y efecto de conocer; entendimiento, inteligencia, razón natural".

Atendiendo a estas definiciones podríamos establecer como definición de Ingeniería del Conocimiento: *"Conjunto de conocimientos y técnicas que permiten aplicar el saber científico a la utilización del conocimiento (entendimiento, inteligencia o razón natural)"*.

Por otra parte, en la bibliografía relativa a la informática ya se ha utilizado el concepto de ingenieria junto con términos del campo de los computadores, definiendo así el concepto de Ingeniería del Software como:

La aplicación de una aproximación sistemática, disciplinada y cuantificable al desarrollo, funcionamiento y mantenimiento del software; en otras palabras, la aplicación de la ingeniería al software (IEEE, 1999).

Esta definición nos aporta la visión de que el concepto de ingeniería aplicado a las ciencias de computación (o más ampliamente la visión de la informática como ingeniería) va encaminada hacia la sistematización del desarrollo de soluciones del dominio de la computación, en este caso software. Si establecemos un paralelismo entre la IS y la IC, llegamos a deducir que si en la IS el elemento central es el software, en la IC el elemento central debe ser el "software con conocimiento" (Águila y cols., 2014).

Por "software con conocimiento" se entienden varios términos que siempre se han entrado es esta categoría como son sistema experto y sistema basado en conocimiento, aunque es preferible esta última: "un sistema software capaz de soportar la representación explícita del conocimiento de un dominio específico y de explotarlo a través de los mecanismos apropiados de razonamiento para proporcionar un comportamiento de nivel alto en la resolución de problemas" (Guida y Tasso, 1994). En otras palabras, los SBC son aplicaciones software que tratan con problemas poco estructurados en los que nos podemos encontrar requisitos subjetivos, entradas inconsistentes, incompletas o con incertidumbre y que no pueden ser resueltos aplicando los algoritmos clásicos o la investigación operativa (Alonso y cols., 1995).

De esta forma podemos establecer que el objetivo general de la IC es el mismo que el de la IS y que no es más que el de transformar el proceso de desarrollo de SBC de un arte a un disciplina de ingeniería. A partir de estas consideraciones ya somos capaces de dar definición más ajustada: *la Ingeniería del Conocimiento es la disciplina tecnológica que se centra en la aplicación de una aproximación sistemática, disciplinada y cuantificable al desarrollo, funcionamiento y mantenimiento de Sistemas Basados en Conocimiento.* Es decir, el objetivo último de la IC es el establecimiento de metodologías que permitan abordar el desarrollo de SBC de una forma más sistemática.

La IC y la IS son disciplinas con similitudes pero que también presentan grandes diferencias. Lo que es claro que que han seguido caminos

paralelos, aunque presenten un desfase temporal de unos 10 años (Águila y cols., 2014). Puesto que la IS suele ir adelantada, muchas de las técnicas de IS han sido adaptadas a la IC con relativo éxito, como la planificación basada en hitos o puntos de anclaje (Águila y cols., 2001), aunque queda todavía mucho camino por recorrer. Debemos considerar la IC como una herramienta que nos permitirá aislar y delimitar las partes del problema que necesiten utilizar técnicas no numéricas o heurísticas y determinar cómo abordar su desarrollo.

2.1.1 Características de los Sistemas Basados en Conocimiento

Los sistemas basados en conocimiento son sistemas software que mantienen una gran cantidad de conocimiento y que incluyen métodos adecuados para explotarlo. Una de las ventajas que aportan estos sistemas es su capacidad para capturar experiencia y comportamientos escasos y específicos, y hacerla más ampliamente disponible, y alcanzar decisiones consistentes y repetibles.

Un SBC incorpora el conocimiento del sistema en diversos aspectos, una estructura de datos y relaciones de carácter declarativo y procedimental, así como representaciones heurísticas. Además, poseen métodos para realizar inferencias sobre el conocimiento almacenado, procesos de búsqueda, toma de decisiones en problemas complejos, y a menudo incorporan mecanismos de aprendizaje y comunicación en lenguaje natural. No obstante, estos elementos aparecen cada vez con mayor frecuencia en los sistemas de información (SI) que se desarrolla con técnicas convencionales, por lo en ocasiones es difícil diferenciar un SBC de un SI. Las siguientes propiedades que permiten distinguir los SBC:

- Los SBC se aplican a dominios y problemas más complejos que los que tratan en proyectos de desarrollo de software tradicional. Las salidas que proporcionan los SBC necesitan hacer uso de procesos elaborados donde intervienen métodos para deducir nueva información y también técnicas heurísticas para reducir los espacios de búsqueda de solución.

- Los SI son sistemáticos y procedimentales, es decir, utilizan como entrada datos organizados y emplean algoritmos deterministas para obtener una salida. En cambio, los SBC son declarativos y

heurísticos, utilizan bases de conocimientos y métodos de resolución de problemas que se adaptan al estado del problema, sobre los que se pueden modificar los objetivos de la solución, y que pueden ir optimizando su comportamiento a lo largo del tiempo.

- Los SI convencionales, por regla general, tienen un proceso de especificación de requisitos establecido y limitado en el tiempo mientras que en los SBC existe una fase de adquisición de conocimiento muy extensa y costosa, que no acaba al finalizar el desarrollo del proyecto, sino que debe mantenerse mientras esté en uso ese software puesto que se va obteniendo más conocimiento sobre el dominio.

- En los SBC, las estructuras de representación son declarativas, de forma que se permite separar el conocimiento del dominio de los mecanismos de deducción utilizados, que pueden ser aplicados a otros dominios. Esta separación permitirá la reutilización tanto de la base de conocimiento como de los métodos de resolución para otros proyectos, lo que es muy importante si tenemos en cuenta el esfuerzo que requiere la fase de adquisición de conocimiento. Esto no es habitual en los SI convencionales, ya que los programas están diseñados para utilizar específicamente los datos que tiene almacenados el sistema.

- Algunos SBC describen y justifican los pasos de razonamiento que se han seguido para alcanzar una solución o tomar una decisión. La existencia de esta capacidad de dar explicaciones puede ser fundamental en dominios donde la toma de decisiones sea muy sensible o importante. Bastaría pensar en SBC que controlan centrales nucleares, diagnostiquen enfermedades o dispongan la situación de una plantilla de empleados.

- Finalmente, los SBC se desarrollan para realizar una tarea específica sobre un dominio determinado. El término tarea se refiere a alguna actividad orientada hacia una meta o a la resolución de problemas. El dominio se refiere al área en que se ejecutará la tarea. Las tareas de los SBC se suelen agrupar en tareas de síntesis (como clasificación, asesoramiento, diagnóstico, etc.) y tareas de análisis (como diseño, planificación, modelado, etc.).

Cualquier SBC consta de dos partes principales: la base de conocimiento, y el motor de inferencia o proceso de razonamiento. Otros componentes importantes son la base de hechos, la interfaz y el módulo

de explicación. En la Figura 2.1 se muestra la estructura general de un SBC.

La *base de conocimiento* es el elemento clave en este tipo de sistemas. Contiene tanto conocimiento factual como heurístico. El conocimiento factual es el conocimiento del dominio que es compartido ampliamente, suele encontrarse en los libros y está aceptado por todos los expertos en ese campo. Por otro lado, el conocimiento heurístico es menos riguroso, basado en la experiencia y, en gran parte, individual. Es el conocimiento de la buena práctica, el buen juicio, y el razonamiento plausible en ese campo. Es el conocimiento que subyace al "arte de formular buenas hipótesis". De manera informal podría decirse que una base de conocimiento es un conjunto de representaciones de ciertos hechos del mundo. Estos hechos son representados en algún formalismo para la representación del conocimiento. En la sección 2.1.3 se describen algunas de las técnicas más usadas en IA para la representación del conocimiento.

Por otro lado, el paradigma o modelo de resolución de problemas organiza y controla los pasos que se siguen hacia la resolución del problema. Estos métodos de resolución de problemas son utilizados por programas que se conocen como *motores de inferencia*, que manipulan y usan la base de conocimiento para crear una línea de razonamiento. Por lo tanto, vemos que hay dos cuestiones estudiadas en IA que son de especial relevancia en el desarrollo de los SBC: la representación de conocimiento y los métodos de razonamiento.

Aunque los componentes principales de un SBC son la base de conocimiento y el motor de inferencia, existen otras características importantes

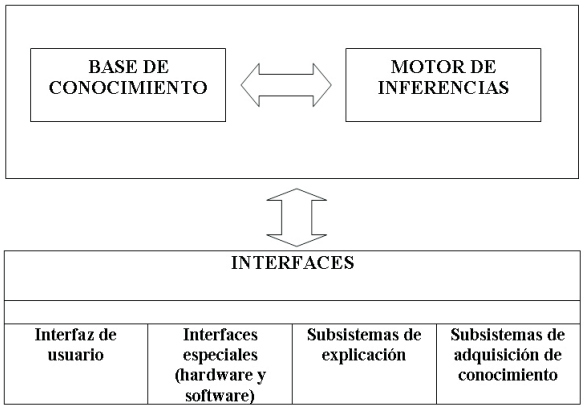


Figura 2.1: Arquitectura general de un SBC

como la capacidad para el tratamiento de la incertidumbre o para generar explicaciones de su línea de razonamiento. Este componente debe, idealmente, poder responder a preguntas como las siguientes: ¿Por qué se tomó o no una cierta decisión? ¿Cuándo, cómo y por qué se utiliza una información determinada? ¿Qué decisión se tomó ante un subproblema? Sería deseable que el sistema proporcionara este tipo de respuestas de forma comprensible y completa.

2.1.2 Ventajas y limitaciones de los sistemas basados en conocimiento

El desarrollo de SBC para ciertos dominios es muy aconsejable y, en algunos casos, imprescindible. Las ventajas que nos ofrece un SBC se pueden resumir en las siguientes:

- **Mantenimiento del conocimiento**: Extraer el conocimiento de expertos tiene un alto coste, pero puede ser reutilizable y no se pierde.
- **Resolución de problemas complejos**: Algunos problemas no tienen soluciones algorítmicas tratables en tiempo real. Los SBC reducen el espacio de búsqueda aplicando alguna heurística.
- **Ajuste de objetivos**: Muchos SBC buscan una solución dependiendo de las indicaciones dadas sobre la importancia de cada uno de los parámetros que pueden ser considerados, como costes, riesgos, rapidez, calidad, etc.
- **Tratamiento de la incertidumbre**: A menudo se necesita tomar decisiones u ofrecer una respuesta sobre problemas con conocimiento que tienen asignado un grado de certeza. Sistemas de dominios como los bursátiles, meteorológicos o médicos son algunos de los sistemas que utilizan la teoría de las probabilidades y otros modelos para resolver consultas e indicar la seguridad sobre la solución hallada.
- **Explicación del razonamiento**: Permite justificar las salidas en el caso de dominios problemáticos o críticos. Esta cualidad puede utilizarse, además, para formar a personal no experto en el ámbito de aplicación.
- **Reducción de costes**: Los expertos suelen ser caros y difíciles de encontrar. Una vez extraído este conocimiento e implementado

en un SBC se puede utilizar personal no especializado para ayudar a tomar decisiones. A pesar de que es costoso desarrollar un SBC, frente a usar expertos en una materia, el mantenimiento es muy reducido, se pueden hacer tantas copias como se desee, y las técnicas adquiridas se pueden emplear para otros dominios.

- **Aumento de la fiabilidad**: En comparación con el empleo de expertos, las soluciones que dan los SBC son más rápidas, objetivas y evitan operaciones incómodas y monótonas. Las decisiones tomadas por el SBC van a depender de todos los casos y parámetros que conozca el sistema, y no van a prevalecer más las últimas experiencias como ocurre con los expertos humanos.

- **Modularidad**: La característica de separar la base de conocimiento de los métodos de razonamiento ofrece una enorme facilidad para incorporar nuevo conocimiento y aprovecharlo en el razonamiento sin tener que modificar el resto de la información introducida.

Por otro lado, aunque su uso es cada vez mayor, no existe todavía un empleo generalizado de SBC para todos los dominios. Existen varios factores de peso que inhiben la comercialización de SBC, entre los que podemos destacar:

- **Dificultad en la adquisición de conocimiento**: La principal barrera en el desarrollo de SBC está en que el proceso de adquisición de conocimiento es lento y arduo desde cualquier fuente de conocimiento, pero sobre todo desde expertos en un área determinada. La experiencia es difícil de extraer de las personas, ya que las respuestas de cada experto en una misma situación pueden ser diferentes, aunque pueden también ser válidas en la mayoría de los casos. Es difícil, para un experto altamente práctico, abstraer buenas referencias de situación cuando está bajo observación y suele ser complicado obtener unas cuantas reglas que engloben un comportamiento general. Por otro lado, no debemos olvidar que al transferir el conocimiento desde un experto a un sistema, el primero pierde su situación de privilegio, por lo que suele ser reacio a la construcción de SBC sobre su campo.

- **Reutilización del conocimiento**: No siempre un SBC creado sobre un dominio es fácil reutilizarlo en otro entorno. Se debe realizar

un considerable esfuerzo de representación para que el conocimiento adquirido pueda ser aprovechado en otros sistemas con diferentes metas. Se debe decir que ésta es la tendencia de los últimos sistemas desarrollados: favorecer la reutilización del conocimiento estático y de los métodos de resolución de problemas.

- **Falta de creatividad y sentido común**: Cuando un experto humano se encuentra con un caso distinto, pero que se diferencia poco de otro tratado anteriormente, da una solución sin problemas. En cambio esto no suele suceder con los SBC. Es muy costoso dotar a los SBC de mecanismos capaces de tratar estos casos porque es difícil implementar el "sentido común", no suelen extrapolar fácilmente casos parecidos y carecen de creatividad para responder a situaciones inusuales.

- **Obstáculos para el aprendizaje y adaptación**: No es fácil para un SBC aprender de sus salidas o incorporar nuevos ejemplos para afinar sus respuestas futuras. Adaptarse a entornos cambiantes, poder detectar cuándo temporalmente están ocurriendo casos parecidos o dar importancia a unos ejemplos frente a otros son cualidades que se esperan de los humanos, pero no es simple proporcionárselas a los SBC.

2.1.3 Representación del Conocimiento

El conocimiento adquirido sobre una materia debe ser organizado y preparado para que pueda ser utilizado computacionalmente en un sistema software. Al desarrollar un SBC, primero se intenta ver el problema desde el punto de vista del usuario y después se cambia la perspectiva para verlo desde el punto de vista del sistema, es decir, modelar el dominio de la aplicación y traducirlo al dominio de la implementación. Para realizar esta transformación se debe llevar a cabo una actividad de modelado del problema (Gómez y cols., 1997) que se divide en dos partes. La primera es una especificación de lo que debe hacer el sistema; determina si el sistema es válido y se adecúa a las necesidades del usuario. Esta primera parte es llamada **modelado conceptual** y creará los modelos conceptuales del sistema que representan la estructura de los conceptos identificados y los métodos de resolución empleados. La segunda parte de la actividad de modelado del sistema determina que el producto software sea correcto, no tenga errores, permita representar

todo el conocimiento necesitado y tenga capacidad para realizar los procesos de inferencia para dar soluciones; esta parte es llamada **modelado formal** y crea modelos formales.

Un **modelo conceptual**, es una representación del conocimiento del dominio, "externa" al computador, en estructuras no computables que modelan su conducta en la solución de problemas. Un **modelo formal**, por su parte, es una representación "semiinterna" o "semicomputable" del conocimiento. Para que un modelo formal sea totalmente operativo necesita de un modelo computable, formado por bases de conocimiento, motores de inferencias y estrategias de control de acuerdo con la arquitectura de un SBC. Terminado el modelado desde el punto de vista del usuario, se pasa al modelado desde el punto de vista del sistema. A esta fase se le llama fase de **formalización** y consiste en transformar los modelos conceptuales obtenidos en modelos formales, utilizando los formalismos de representación del conocimiento. Por lo tanto, formalizar consiste en representar simbólicamente el conocimiento mediante alguno de los formalismos existentes, organizarlos de acuerdo con algún modelo de diseño y determinar los métodos de inferencia adecuados para manejar eficientemente dichos conocimientos, a esta fase se le suele llamar representación del conocimiento.

El conocimiento puede estructurarse y representarse de diversas formas, de acuerdo con el problema que se intenta resolver. Es decir, en la IC existen varios formalismos o esquemas de representación del conocimiento adecuados para diferentes tipos de problemas. Por lo tanto, una vez que se ha realizado el laborioso proceso de adquisición del conocimiento, desde expertos u otras fuentes y que se ha creado el modelo conceptual que representa a los componentes del dominio concreto, debe buscarse el modelo semicomputable que tenga la suficiente capacidad expresiva para contener al conocimiento necesitado por el sistema, y que permita realizar los procesos de inferencia para deducir nueva información o resolver el problema que tiene como meta el sistema. En este punto, el ingeniero del conocimiento deberá escoger cómo representar simbólicamente los conceptos y relaciones obtenidos, y estudiar la capacidad que tendrán estos esquemas para emplear los métodos de razonamiento necesarios para explotar de forma efectiva los conocimientos representados.

En los SBC la representación de los datos y los procedimientos de resolución se diseñan por separado, al contrario que en los sistemas software tradicionales. De este modo se representan de forma independiente

el conocimiento "estático" de un dominio y el conocimiento "dinámico" para realizar inferencias, aunque ambos, claro está, deberán ser compatibles en el SBC. Así se consigue que estos elementos (la representación de las estructuras y los procedimientos de resolución) puedan ser reutilizados en otros sistemas. Con esta idea, en 1991 Neches (Neches y cols., 1991) propusieron la construcción de SBC uniendo o "ensamblando" componentes reutilizables, de manera que los desarrolladores de los nuevos sistemas sólo tuvieran que preocuparse de la creación de conocimientos especializados y de desarrollar nuevos razonadores para la tarea específica de su sistema. Los nuevos sistemas interoperarían con los sistemas existentes, compartiendo así sus conocimientos y métodos de razonamiento. El conocimiento declarativo, las técnicas de resolución de problemas y los métodos de razonamiento podrían estar en los sistemas compartidos, que proporcionan sistemas mejores, mayores y más económicos de construir. El objetivo era construir sistemas ensamblando conocimientos y métodos de razonamiento, en vez de desarrollarlos desde la nada.

La elección de los esquemas de representación más adecuados para el sistema que se va a desarrollar no es trivial. Debido al enorme esfuerzo que conllevan los procesos de adquisición de conocimiento de un dominio, la construcción de su modelo conceptual y la formalización de tales conocimientos, la tendencia actual es representar el conocimiento de un dominio de forma que sea fácilmente reutilizable. A continuación, se describen brevemente algunos de los paradigmas de representación del conocimiento clásicos más importantes, escogidos por su importancia histórica y por ser los más utilizados en IA.

- **Esquemas lógicos.** Los hechos se representan como fórmulas lógicas en algún formalismo matemático, siendo la más utilizada la lógica de primer orden, aunque también se emplean otras, como lógicas multivaluadas, lógicas difusas, lógicas temporales, etc. Una fórmula lógica es una combinación de predicados, variables, constantes, conectivas, cuantificadores y funciones. En este caso, una base de conocimiento es una colección de fórmulas lógicas bien formadas y verdaderas que describen un dominio.

- **Redes semánticas.** En las redes semánticas o grafos conceptuales la información se presenta en grafos dirigidos, donde nodos y arcos deben estar etiquetados. Los nodos representan conceptos, instancias, eventos o características de esos conceptos, y los arcos

representan distintos tipos de relaciones binarias entre ellos. La base de representación consiste en modelar los conocimientos relativos a un objeto o concepto mediante pares atributo-valor. El mayor potencial que poseen las redes semánticas es el empleo de las relaciones estructurales para formar jerarquías de conceptos. .

- **Sistemas de producción.** Las reglas de producción son el esquema de representación procedimental más popular. En los sistemas de producción, desarrollados por Newell y Simon (Newell y Simon, 1973), los conocimientos sobre las acciones o reglas de deducción son más importantes que los conocimientos estáticos del dominio. La representación del conocimiento se identifica más con heurísticas o formas de proceder de expertos, en contraposición con la representación declarativa de los dos esquemas anteriores.

- **Marcos.** Los marcos o *frames* es la técnica de representación que posee una gran capacidad de representar aspectos semánticos sobre el conocimiento *declarativo* del dominio. Pero, además, permiten representar conocimiento *procedimental* que se refiere a cómo rellenar los valores de los atributos, cómo realizar comprobaciones sobre la información introducida, qué acciones realizar cuando se cumplan o fallen los procesos de búsqueda, etc. La representación basada en marcos posee una gran capacidad expresiva para representar el conocimiento declarativo y posibilita la incorporación de conocimiento procedimental. Los lenguajes para representar ontologías se basan en este formalismo.

- **Ontologías.** Las ontologías proporcionan la conceptualización explícita de los términos de un dominio, que sirve como soporte para la implementación de bases de conocimiento preparadas para ser utilizadas por aplicaciones, y que resuelven diferentes tareas. No obstante, además de ser la base para los sistemas basados en conocimiento, actualmente el ámbito e aplicación de las ontologías abarca otras áreas, como procesamiento del lenguaje natural, web semántica, comercio electrónico, gestión del conocimiento, etc

2.1.4 Metodologías de primera generación de SBC

En esta sección se realiza una revisión general de cómo se han desarrollado los SBC hasta ahora, es decir, cuales son las metodologías para el desarrollo de un proyecto software donde el resultado es un SBC. La

Ingeniería del Conocimiento aparece con la necesidad de aplicar metodologías para el desarrollo de un SBC a finales de los años setenta. En su origen, las primeras metodologías presentan el desarrollo de un SBC como un proceso de adquisición de conocimiento, entendido éste como la transferencia y transformación de la experiencia en la resolución de problemas desde alguna fuente de conocimiento o experto a un programa de ordenador, siendo necesaria la intervención de un ingeniero del conocimiento que realiza el proceso (Buchanan y cols., 1983; Hayes-Roth y cols., 1984).

La aportación más importante del trabajo de Buchanan fue la identificación del cuello de botella que aparece como consecuencia de un problema de comunicación con el experto. Para ayudar al ingeniero del conocimiento a solventar este problema, se propone un ciclo de vida para el proceso de adquisición de conocimiento que abarca desde la concepción del sistema hasta su madurez. La metodología aplica el ciclo de vida de en cascada mostrado en la figura 2.2 y consta de varias etapas. En la etapa de identificación se caracterizan los aspectos más importantes del problema. En la etapa de conceptualización se hacen explícitos los conceptos y relaciones construyendo un diagrama que se utiliza como base conceptual para la elaboración de un prototipo. La etapa de formalización consiste en llevar los conceptos claves, los problemas y las características de los flujos de información a una representación más formal. La etapa de implementación implica la traducción del resultado de la formalización a otro marco de representación que tiene que estar asociado con la herramienta utilizada. Por último, en la etapa de validación se evalúa el prototipo obtenido y la representación elegida para implementarlo. La importancia de esta metodología radica en que es el primer intento que permite abordar el desarrollo de un SBC desde un punto de vista industrial. Durante los años siguientes se promueven numerosas modificaciones y mejoras generando metodologías que plantean el desarrollo de SBC con una aproximación más cercana a las técnicas de Ingeniería del Software, definiendo y organizando las diferentes actividades que conducen a la implementación y uso rutinario de los SBC. Entre estas metodologías, que podemos llamar de primera generación, destacamos KLIC (Guida y Tasso, 1994), e IDEAL (Hernández y Sierra, 1988).

La primera incluye la necesidad de realizar un estudio de viabilidad y análisis de alternativas previos al desarrollo propiamente dicho, además define las fases y los artefactos generados en el ciclo de vida.

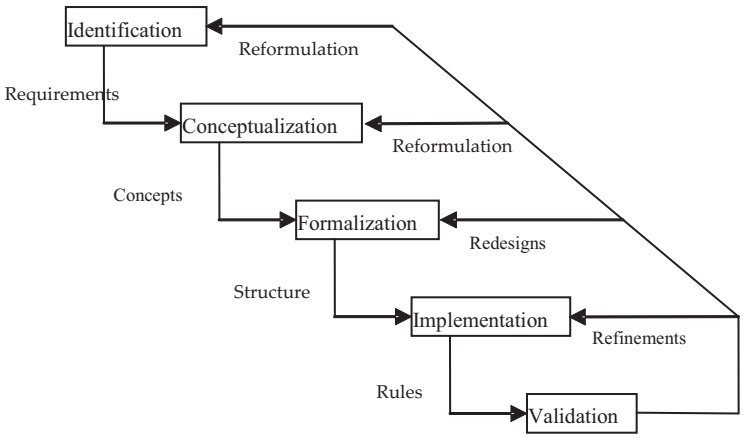


Figura 2.2: Ciclo de vida de Buchanan

Básicamente, el nivel más alto de abstracción el ciclo de vida está compuesto por: análisis de posibilidades, análisis de viabilidad, construcción de un demostrador, desarrollo del prototipo, implementación, instalación y entrega del producto final, mantenimiento y extensión. Por su parte, la metodología IDEAL recibe su nombre de las iniciales de cada una de las fases que conlleva: Identificación de la tarea, Desarrollo de los prototipos, Ejecución de la construcción del sistema integrado, Actuación para conseguir el mantenimiento perfectivo, Adecuada transferencia tecnológica. IDEAL incorpora un ciclo de vida en espiral cónico en tres dimensiones (Alonso y cols., 1996). Este modelo solventa el problema del modelo en espiral plano, que no tiene en cuenta el mantenimiento perfectivo, es decir, no recoge la necesidad de la incorporación sistemática de nuevos conocimientos que se producen por el propio uso del SBC.

Esta familia de metodologías, mostrada de forma resumida en la tabla 2.1, fue utilizada con relativo éxito hasta principios de la década de los noventa, fecha en la que se replantea la Ingeniería del Conocimiento desde la perspectiva de la transferencia hacía la perspectiva del modelado. La aproximación basada en la transferencia del conocimiento sólo es posible para pequeños sistemas, y falla a la hora de producir sistemas grandes con aspiraciones comerciales, fiables y con un mantenimiento asequible.

Tabla 2.1: Métodos de primera generación para el desarrollo de SBC

Artefactos	*Notación*	*Etapas*
Buchanan		
Knowledge Bases Inference Methods	Rules Frames	Identification Conceptualization Formalization Implementation Validation
IDEAL		
Plan Use Case Model Static conceptual Model Process and control Model Formal Model Computational Model	Rule languages LISP	Identification of the tasks Development of prototypes Execution of integrates system Perfective Maintenance Tecnology Transfer

2.1.5 Metodologías de segunda generación de SBC

Las metodologías de segunda generación vienen a permitir realizar el análisis del sistema en el nivel de conocimiento, ofreciendo la posibilidad de especificar el problema a diferentes niveles de granularidad, con lo cual se potencia la idea de la reutilización de componentes de conocimiento. Por otro lado, ofrecen un ciclo de vida completo para el proceso de desarrollo, proporcionado pautas a seguir desde el análisis hasta la implementación final del sistema, al estilo de las propuestas de IDEAL. El origen de estas nuevas metodologías de desarrollo de SBC se produce el replanteamiento hacia el modelado de la Ingeniería del Conocimiento. Su origen está en las ideas propuestas en trabajos anteriores que están todavía muy lejos de ser considerados como metodologías; de entre ellos desatacamos la aportación de Chandrasekaran (Chandrasekharan, 1986) que propone la existencia de tareas genéricas como componentes combinables que pueden reutilizarse para la construcción de diferentes SBC. Posteriormente, el mismo equipo de Chandrasekaran propone las estructuras de tareas para distinguir entre el concepto de tarea que se asocia con el tipo de problema a resolver, y el de Problem Solving Method que es la forma de llevar a cabo una tarea. KADS (Knowledge Acquisition and Design Structuring) y Protégé son los primeros trabajos que resolvían el problema de la adquisición del conocimiento desde el enfoque del modelado y que ya se pueden considerar de segunda generación. A partir de estos proyectos se han desarrollado metodologías más completas de entre las que destacamos CommonKADS (Schreiber y cols., 2000),

MIKE (Angele y cols., 1998) y PROTÉGÉ-II, (Eriksson y cols., 1995). A continuación se describen brevemente todas ellas prestando especial atención a CommonKADS que es una de las metodologías importante para este trabajo de tesis, puesto que en ella se define el lenguaje de modelado CML. Un resumen de las mismas se muestra en la tabla 2.2.

PROTÉGÉ-II fue el resultado de un proyecto cuyo principal objetivo era resolver las limitaciones de las primeras versiones. Es mas una herramienta de adquisición del conocimiento que de desarrollo. PROTÉGÉ-II proporciona un entorno en el que el desarrollador puede especificar tareas y puede seleccionar métodos de resolución de una librería de métodos reusables. Para permitir el modelado, se distingue entre Tareas, Métodos y Mecanismos. Las tareas representan un problema en el mundo real, localizado dentro de una organización (tarea aplicación), o una especificación abstracta de objetivos independientes del dominio que tienen que ser alcanzados (clases de tareas, o tarea). Un método es un modelo independiente del dominio, que indica cómo se resuelve el problema especificado por la tarea. Los métodos pueden delegar problemas a subtareas que, a su vez, pueden ser resueltas por otros métodos, formando un árbol de tareas-métodos-subtareas. Por último, los mecanismos se pueden considerar como métodos primitivos que no pueden ser descompuestos en otras subtareas. Por lo tanto, pueden ser considerados como cajas negras que no admiten descomposición. La descomposición termina al alcanzar los mecanismos. Esta estrategia de descomposición ayuda a la reutilización de componentes de dos formas: por un lado, los mecanismos, al desarrollar funciones muy definidas, pueden ser aplicables a varios dominios; y por otro, la existencia de métodos descomponibles hace más flexible a la hora de la reutilización, ya que permiten la utilización de métodos y mecanismos alternativos para la resolución de sus subtareas.

MIKE es una metodología para el desarrollo de SBC que aplica un ciclo de vida en espiral, aunando el prototipado con un proceso de desarrollo incremental y el paradigma del modelado. El proceso de desarrollo se plantea con fases cíclicas: adquisición, interpretación, formalización, diseño, implementación y evaluación. La fase de adquisición empieza con la extracción de conocimiento, con la que se obtienen descripciones informales sobre el conocimiento del dominio y del proceso de resolución. Se genera el modelo de adquisición, donde la información se almacena en lenguaje natural. Durante la interpretación se redescriben las estructuras identificadas en la fase anterior en un lenguaje más restringido, es el

llamado modelo de estructura, que recoge todos los aspectos funcionales y no funcionales del dominio. Este modelo, junto con el de adquisición, son la base para establecer la comunicación entre el experto y el ingeniero del conocimiento. En la fase de formalización, se crea un modelo más formal que el de estructura mediante la utilización de un lenguaje formal, KARL (Knowledge Acquisition Representation Language) (Fensel, 1995), que se denomina modelo KARL. La característica principal de este lenguaje es que combina la descripción conceptual de un SBC con una especificación formal y ejecutable. De esta forma, se pueden especificar formalmente, eliminado cualquier vaguedad e imprecisión, las inferencias detectadas en la fase anterior. En la fase de diseño se genera un nuevo modelo más detallado, el modelo del diseño, especificado en el lenguaje DesignKARL, que es una extensión de KARL a la que se le añade la capacidad de expresar algoritmos y estructuras de datos. Una vez obtenido el modelo del diseño, se pasa a la fase de implementación, en la que esté se materializa en la plataforma hardware y software elegida. La última fase del proceso de desarrollo termina con la evaluación, en la que se vuelve a requerir la intervención del experto. MIKE es una potente metodología que integra formalismos de representación informales y formales con un desarrollo por prototipado. Pero como ya se ha indicado, la utilización de métodos formales conlleva alto consumo de recursos que desde un punto de vista comercial no son asumibles en la mayor parte de los proyectos de desarrollo de software.

CommonKADS, una de las metodologías de IC más extendidas en Europa, es la evolución de KADS y cubre la gestión del proyecto, el análisis organizacional y los aspectos relativos al desarrollo. Esta metodología recoge ideas que han sido objetivo de IC e IS: modelado, gestión de riesgos y reusabilidad. El desarrollo de un SBC se plantea como la construcción de distintos modelos sobre el comportamiento en la resolución de problemas en un contexto organizacional concreto. Estos modelos forman el conjunto de modelos de CommonKADS, que como se muestra en la figura 2.3 está formado por seis modelos: modelo de organización, modelo de tareas, modelo de experiencia, modelo de comunicación, modelo de agentes y modelo de diseño. A nivel de contexto, el modelo de organización tiene por objetivo analizar las principales características de la organización para descubrir los problemas y las oportunidades de uso de los SBC, además de establecer su viabilidad y valorar su impacto en la organización. El modelo de tareas muestra las tareas como partes relevantes del proceso de negocio. Analiza la organización global

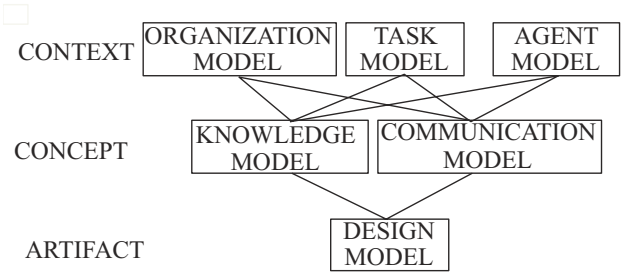


Figura 2.3: Conjunto de modelos CommonKADS

de las tareas, sus entradas, salidas, precondiciones, criterios de rendimiento, así como la competencia y recursos necesarios. El modelo de agentes muestra los agentes como ejecutores de tareas. Un agente puede ser una persona, un sistema software o cualquier cosa capaz de realizar una tarea. Se describen las características de los agentes, así como las conexiones entre agentes.

A nivel de concepto, el modelo de conocimiento o modelo de experiencia, como fue llamado en versiones anteriores de la metodología, describe el conocimiento que tiene un determinado agente y que es relevante para la consecución de una determinada tarea, además de describir la estructura de este conocimiento en función de su uso. Este modelo se define en el nivel de conocimiento, sin hacer referencia a aspectos de implementación. Para poder llevar a cabo el modelado de los distintos papeles que puede jugar el conocimiento, éste se descompone en tres categorías disjuntas: el conocimiento de las tareas, del dominio y de las inferencias. El conocimiento de tareas describe de una forma recursiva la descomposición de una tarea de alto nivel en varias subtareas. El conocimiento del dominio especifica los hechos y asunciones que necesita el proceso de razonamiento para llevar a cabo su cometido en el dominio de la aplicación. Por una parte se va a especificar la forma y estructura del conocimiento que es lo que se denomina ontología del dominio, y por otra, el contenido relevante para la aplicación. El último tipo es el conocimiento sobre inferencias que describe los procesos primitivos de razonamiento que tienen lugar en una aplicación, así como los roles de conocimiento que son usados por las inferencias. El modelo de comunicación es el que se encarga de describir las transacciones que tienen lugar entre los agentes del sistema, así como de especificar los procesos de intercambio de información que materializan el intercambio de

Tabla 2.2: Métodos de segunda generación para el desarrollo de SBC

Artefact	*Notation*	*Stages/workflows*
CommonKADS		
Organization Model Task Model Agent Model Knowledge Model Communication Model Design Model	Inference Diagrams Task-Methods Diagrams Class Diagrams State Diagrams Use Case Diagrams Templates	Revision Risks study Monitoring
MIKE		
Elicitation Model KARL Model Design Model	KARL Design KARL	Acquisition Interpretation Formalization Design Implementation Evolution
Protégé		
Knowledge Model	Knowledge elicitation tool Knowlegge Base	—-

conocimiento entre agentes.

Por último, a nivel de artefacto, el modelo de diseño se ocupa de la transición del nivel de conocimiento (modelado conceptual) al nivel simbólico (implementación). Esta transición se realiza a través de diferentes puntos de vistas: el diseño de la aplicación, el diseño de la arquitectura y el diseño de la plataforma, y se ocupa de la transición del nivel conceptual al nivel de implementación. Cada una de las visiones del diseño recoge un aspecto concreto del proceso de desarrollo. El diseño de la aplicación trabaja en el nivel conceptual de las entidades y tareas del mundo real que se tienen que llevar a la aplicación. El diseño de la arquitectura hace referencia a las estructuras computacionales abstractas en las que se tiene que materializar el diseño de la aplicación. El diseño de la plataforma indica el lenguaje de programación que se va a utilizar así como la plataforma hardware y software que va a ser utilizada.

2.1.6 Ontologías. Ingeniería Ontológica

El tránsito de las metodologías de IC hacia el modelado de conocimiento ha evolucionado en el uso de ontologías como medio de representar y compartir conocimiento. En Inteligencia Artificial y representación del conocimiento, la definición declarativa más consolidada de ontología es la propuesta pro Gruber (Gruber, 1995), y extendida por Studer y co-

legas (Studer y cols., 1998), como "una especificación explícita y formal sobre de una conceptualización consensuada". La interpretación de esta definición es que las ontologías definen sus conceptos, propiedades, relaciones, funciones, restricciones y axiomas de forma "explícita" en algún lenguaje capaz de contener este conocimiento. El término conceptualización se refiere a un modelo abstracto de algún fenómeno del mundo. "Formal" indica que la especificación debe representarse por medio de un lenguaje de representación formalizado, y "compartida" refleja que una ontología debe dar cuenta de conocimiento aceptado.

Las ontologías se componen de:

- Conceptos: son las ideas básicas que se intentan formalizar. Los conceptos pueden ser clases de objetos, métodos, planes, estrategias, procesos de razonamiento, etc.

- Relaciones: representan la interacción y enlace entre los conceptos de un dominio. Suelen formar la taxonomía del dominio. Por ejemplo: subclase-de, parte-de, parte-exhaustiva-de, conectado-a, etc.

- Funciones: son un tipo concreto de relación donde se identifica un elemento mediante el cálculo de una función que considera varios elementos de la ontología. Por ejemplo, pueden aparecer funciones como: asignar-fecha, categorizar-clase, etc.

- Instancias: se utilizan para representar objetos determinados de un concepto.

- Reglas de restricción o axiomas: son teoremas que se declaran sobre relaciones que deben cumplir los elementos de la ontología. Por ejemplo: "Si A y B son de la clase C, entonces A no es subclase de B", "Para todo A que cumpla la condición B1, A es C", etc. Los axiomas, junto con la herencia de conceptos, permiten inferir conocimiento que no esté indicado explícitamente en la taxonomía de conceptos.

En general, se distinguen tres tipos fundamentales de ontologías:

- Ontologías de un dominio, en las que se representa el conocimiento especializado pertinente de un dominio o subdominio, como la medicina, las aplicaciones militares, la cardiología, etc.

- Ontologías genéricas, en las que se representan conceptos generales y fundacionales del conocimiento como las estructuras parte/todo, la cuantificación, los procesos o los tipos de objetos.

- Ontologías representacionales, en las que se especifican las conceptualizaciones que subyacen a los formalismos de representación del conocimiento, por lo que también se denominan meta-ontologías u ontologías de alto nivel.

Las ontologías proporcionan la conceptualización explícita de los términos de un dominio, que sirve como soporte para la implementación de bases de conocimiento preparadas para ser utilizadas por aplicaciones, y que resuelven diferentes tareas. No obstante, además de ser la base para los sistemas basados en conocimiento, actualmente el ámbito de aplicación de las ontologías abarca otras áreas, como procesamiento del lenguaje natural, web semántica, comercio electrónico, gestión del conocimiento, etc.

La Ingeniería Ontológica (Gómez-Pérez y cols., 2004) es la disciplina que se ocupa de los métodos y metodologías para la construcción de ontologías. Este área incluye todo el conjunto de actividades que trata el proceso completo, el ciclo de vida el una ontología, así como las metodologías, herramientas y lenguajes necesarios para construir una ontología.

Una buena revisión de los temas relacionados con esta disciplina aparece en (Corcho y cols., 2003; Fernández-López y Gómez-Pérez, 2003) (ver resumen en tabla 2.3). A continuación enumeramos algunos de los métodos mas conocidos. CYC puede considerarse como el trabajo mas antiguo en este tema (Lenat, 1995). El proyecto de desarrollo de ontología TOVE (Toronto Virtual Enterprise) fue una aproximación con lógica de primer orden para representar ontologías en una arquitectura de integración a nivel de empresa (Gruninger y cols., 1995). El proyecto COINS (Context Interchange System) es un trabajo extenso de esfuerzos relacionados con el desarrollo de ontologías (Bressan y cols., 2000). Methontology (Corcho y cols., 2003), es una metodología que permite construir ontologías en el nivel de conocimiento, y se utiliza como referencia en el IEEE standard 1074-1995. Una solución unificada de trabajos anteriores se recoge en OTKM (On-To-Knowledge Methodology) (Sure y cols., 2004). Finalmente. UPON (United Process for ONtologies) es una aproximación basada en UML y centrada en casos de uso que es incrementa e iterativa (Nicola y cols., 2009).

Tabla 2.3: Metodologías en Ingeniería Ontológica

Artefact	*Notation*	*Stages/workflows*
Methontology		
Ontologies: concepts relationships	Ontology languages: OIL DAML + OIL OWL	Pre-development Development Post-development Management and support
OTKM		
Ontology Requisite Specification Ontology states: Semi-formal described Target Evaluated Evolved	Ontology languages	Feasibility Study Kickoff Refinement Evaluation Application and Evolution
UPON		
Domain and reference lexicon UML class diagrams UML activity diagrams Semantic network Ontology	OWL	Requirement Analysis Design Implementation

Methontology tiene un proceso mas detallado, OTKM presenta un mayor número de actividades y TOVE es la más formal. Methontology es la mas conocida para construir ontologías y propone un modelo de proceso, una ciclo de vida así como una especificación de todas las actividades a ejecutar siguiendo el estándar IEEE 1074. Estas actividades se clasifican en tres categorías, como muestra la figura 2.4. La primera categoría son las actividades de gestión relacionadas con la planificación y el control del proyecto. El segundo tipo de actividades son las actividades orientadas al desarrollo que se ordenan en tres etapas: predesarrollo, desarrollo y postdesarrollo. Finalmente se encuentran las actividades de soporte.

2.2 Modelado y formalización de sistemas basados en reglas

Como ya se ha indicado, cuando se utiliza un sistema basado en reglas los procesos se representan como acciones independientes, pero que pueden integrarse para resolver una tarea más general. La arquitectura de un sistema de producción está compuesta por una base de hechos, una base de reglas y un motor de inferencia. La base de hechos y la

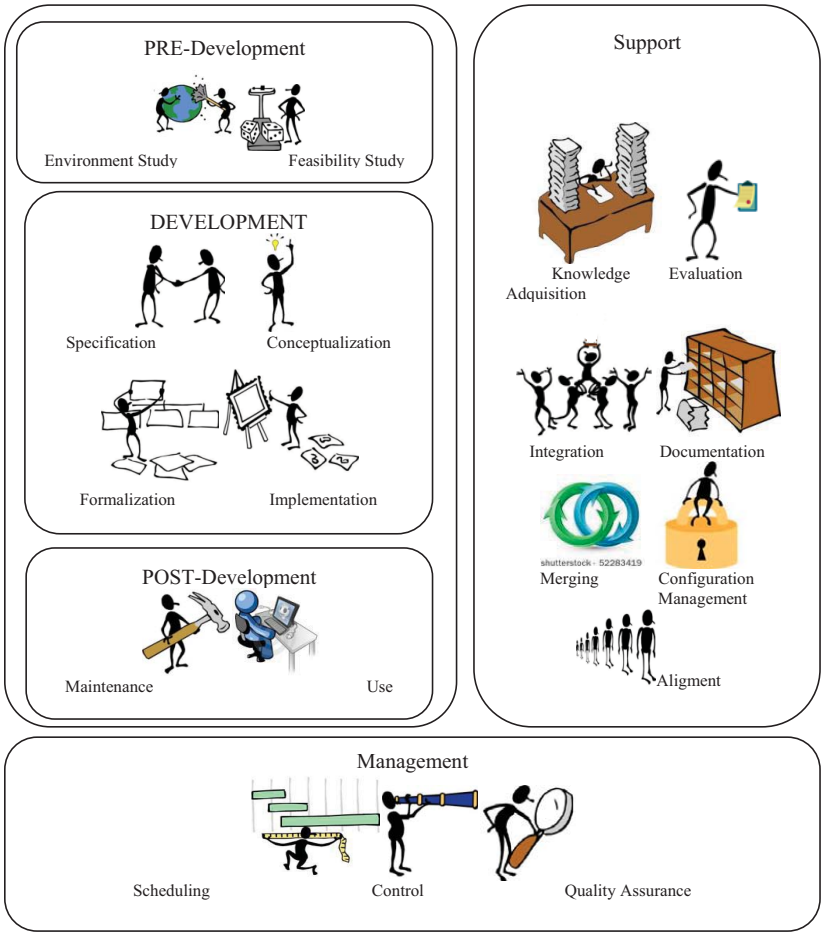


Figura 2.4: Methontology: actividades principales

base de reglas forman la base de conocimiento del sistema. Estos tres componentes representan lo siguiente:

1. La descripción del estado del dominio o del contexto del problema se realiza en la base de hechos. Se modificará conforme se vayan aplicando los procesos de inferencia. La base de hechos contiene tres elementos:

    - **Estado inicial**, que representa la situación origen del problema.

- **Estados finales**, que representan las situaciones objetivo que se pretenden alcanzar.
- **Estados intermedios** o memoria de trabajo que contienen la descripción de la situación en curso de resolución. Esta memoria de trabajo es temporal y se destruye al alcanzar un estado final. La ejecución de las reglas modifica la memoria de trabajo añadiendo o eliminando hechos.

2. La base de reglas contiene la parte fundamental de la base de conocimiento. Las reglas describen los elementos de deducción básicos que utilizará el sistema, y pueden contener información adicional sobre el grado de certeza que se tiene sobre esa acción. De esta forma, las reglas pueden recoger las apreciaciones en los métodos de actuación de expertos en un dominio. La forma más sencilla de representación es la formada por un conjunto de acciones situadas en el consecuente que se ejecutarán si se cumplen un conjunto de hechos situados en el antecedente de la regla. Suelen seguir una representación uniforme del tipo: ***SI condiciones ENTONCES acciones***.

3. El motor de inferencia (también llamado estrategia de control, intérprete de reglas o máquina deductiva) actúa sobre los dos componentes anteriores según una estrategia de deducción determinada. Decide la regla que se debe aplicar en cada caso según la situación de la memoria de trabajo, hasta llegar a un estado objetivo. Las reglas pueden tener asociada una prioridad que establecerá el orden en el que se seleccionará una frente a otras, aunque el motor de inferencia tendrá en cuenta otros factores según una política de resolución establecida.

A la hora de construir un sistema de producción, el ingeniero de conocimiento deberá definir con precisión el vocabulario y las normas semánticas que se van a utilizar para expresar las reglas y hechos de la base de conocimiento ya que, aunque no se invocan entre sí directamente, sí se comunican a través de la descripción del contexto situada en la memoria de trabajo. Una de las características más importantes de los sistemas de producción es la independencia de unas reglas frente a otras. Las acciones que las reglas tienen definidas en el consecuente se refieren a la incorporación de nuevos hechos de la memoria de trabajo o la eliminación de algunos existentes, pero nunca consisten en la invocación

de otras reglas del sistema. Con ello se consigue que la base de conocimiento tenga un carácter totalmente modular, y en cualquier momento se pueden incorporar o eliminar nuevas reglas, favoreciendo el desarrollo incremental del sistema conforme se obtiene nuevo conocimiento.

Esta forma de definir el conocimiento está muy próxima a la forma de proceder de los expertos: decidir qué hacer o qué se puede deducir cuando se cumplen una serie de condiciones. Por otro lado, la sintaxis con la que se representa el conocimiento en los sistemas de producción es muy simple y homogénea. Estas dos características de los sistemas de producción favorecen el proceso de adquisición de conocimiento, puesto que constituyen un método muy intuitivo de recoger los métodos de actuación de los expertos de un dominio. Además, los usuarios del sistema pueden seguir fácilmente cada una de las fases realizadas en el proceso de deducción, puesto que se puede comunicar qué condiciones se conocían en un momento determinado, por qué se ha escogido una regla y qué se ha deducido tras aplicarla.

En cambio, los sistemas de producción, debido a que representan conocimiento básicamente procedimental y heurístico, tienen los inconvenientes propios de estos paradigmas de representación, como la dificultad de verificación de la consistencia en la base de conocimiento. Cada vez que se añade una regla al sistema, es complicado detectar que no se está incorporando conocimiento contradictorio, más aún cuando tratamos con sistemas probabilísticos o de manejo de información incompleta o con incertidumbre. Por otro lado, el conocimiento de estos sistemas no siempre puede ser aplicado a problemas del mismo dominio pero en otro entorno, puesto que tiene una fuerte vinculación con el contexto en el que se ha definido (pensemos, por ejemplo, en un sistema meteorológico, que puede ser muy válido para un lugar geográfico determinado pero puede no funcionar al aplicarlo a otro entorno).

El control de los procesos de deducción lo realiza el motor de inferencia, que examina la base de hechos en cada ciclo de resolución y decide qué regla ejecutar. El funcionamiento básico consiste en seleccionar alguna regla de la base de reglas que pueda aplicarse a la situación actual, hasta que los hechos de la memoria de trabajo satisfagan una condición de terminación o se ejecute una regla de parada. La estrategia de control debe recoger las siguientes características:

- Debe permitir avanzar siempre hacia la resolución del problema. Si una estrategia de control repitiese indefinidamente uno o más pasos, nunca llegaría a una solución.

- Debe ser sistemática, es decir, que garantice que va a encontrar la solución (si existe) en un número finito de pasos. Un ejemplo de estrategia no sistemática sería elegir de forma aleatoria la siguiente regla a disparar, lo que podría llevar a repetir estados y a realizar búsquedas inútiles que no conducen a la solución.

- Debe ser también eficiente, es decir, debe emplear un tiempo razonable para encontrar la solución.

La estrategia de control decide qué única regla aplicar en cada ciclo. Para ello, primero se realiza la selección de la regla más adecuada y luego se activa, incorporando o eliminando hechos de la memoria de trabajo. Tiene dos alternativas: encadenamiento hacia delante y encadenamiento hacia atrás. En el encadenamiento hacia delante, como el mostrado en el ejemplo anterior, el motor de inferencia crea el conjunto conflicto seleccionando aquellas reglas cuyo antecedente satisface la memoria de trabajo actual. En el proceso de resolución, se selecciona una regla y se ejecutan las acciones especificadas en el consecuente (añadir o eliminar hechos de la memoria de trabajo). En cambio, en los sistemas de producción regresivos o dirigidos por el consecuente que utilizan el encadenamiento hacia atrás, se trata de probar una premisa partiendo de una situación final o un hecho que se debe llegar a demostrar. Lo que se va analizando es el consecuente de las reglas desde el estado objetivo. Si coincide con la memoria de trabajo se termina la búsqueda; si no, se sigue buscando reglas recursivamente intentando resolver el antecedente nuevo de las reglas dispuesto en la memoria de trabajo. La meta final se reemplaza secuencialmente por la conjunción de submetas equivalentes.

Los sistemas basados en reglas han sido la tecnología más importante para el desarrollo de sistemas basados en conocimiento desde el comienzo de la Inteligencia Artificial. Son un tipo de sistemas software en los que el conocimiento del experto humano es aplicado para resolver una tarea compleja, como por ejemplo diagnosis, monitorización, validación, etc, y dicho conocimiento es representado como un conjunto de reglas de producción declarativas. Los motores de reglas interpretan las reglas y ejecutan uno o varios métodos de razonamiento utilizando algún método de inferencia para alcanzar una conclusión, tal y como lo haría un experto humano (Durkin, 1993; Russell y Norvig, 1995).

En general, estos sistemas consisten en un conjunto de reglas (base de reglas), una memoria de trabajo y un motor de inferencia. Las reglas permiten representar el conocimiento del dominio y la lógica de nego-

cio como pares condición-acción. La memoria de trabajo representa el inicialmente las entradas al sistema, pero las acciones que se ejecutan cuando las reglas se disparan hacen que el estado de la memoria de trabajo cambie. El motor de inferencia ejecuta un método de razonamiento para disparar reglas, normalmente un mecanismo de encadenamiento hacia delante o hacia atrás, que permite inferir nuevos datos a partir de los existentes en la memoria de trabajo.

2.2.1 Modelado de sistemas basados en reglas

Existen numerosos formalismos de modelado de sistemas basados en reglas. Podemos clasificarlos en distintos tipos y categorías, dependiendo de su origen y tipo de representación utilizadas.

En primer lugar tenemos los lenguajes de modelado de reglas de que tienen su origen en la Inteligencia Artificial y la Ingeniería del Conocimiento, surgidos a partir de la necesidad de usar métodos de especificación y diseño de sistemas expertos. Algunos ejemplos son Conceptual Modeling Language (CML) (Schreiber y cols., 2000) propuesto en la metodologia CommonKADS, Unified Problem-solving Method description Language (UPML) (Angele y cols., 1998) y Operational Conceptual Modelling Language (OCML) (Motta, 1998).

Con la evolución de la Web surgió rápidamente el interés en convertir la Web en una plataforma más efectiva para los sistemas inteligentes (Grove, 2000). La necesidad de representar y intercambiar reglas en la web originó la necesidad de crear representaciones basadas en lenguajes de marcado, como son Rule Markup Initiative (RuleML) (Boley y cols., 2001), y REWERSE Rule Markup Language (R2ML) (Wagner y cols., 2006). En la misma línea, el World Wide Web Consortium (W3C) promueve el empleo de Rule Interchange Format (RIF) (Kifer, 2008) como lenguaje de intercambio de reglas para la Web.

En tercer lugar encontramos los lenguajes de la Web Semántica para modelado de ontologías y reglas: Web Ontology Language (OWL) (Dean y cols., 2004) y Semantic Web Rule Language (SWRL) (Horrocks y cols., 2004) son la referencia en este ámbito. Las reglas y las ontologías proporcionan capacidades de modelado del conocimiento y razonamiento a las aplicaciones Web semánticas (Eiter y cols., 2008). Los formalismos de modelado y representación de ontologías están bastante evolucionados y el Lenguaje de Ontologías para la Web (Web Ontology Language, OWL) se ha convertido en un estándar, sin embargo los formalismos de representación de reglas, así como el desarrollo e integración de lenguajes

de reglas y motores de inferencia que permitan añadir razonamiento a sistemas de información complejos sigue siendo un tema de investigación activo.

La comunidad de la Ingeniería del Software también ha fijado su interés en las reglas como un formalismo adecuado para la representación de la lógica de negocio de sistemas software. Por ello se puede afirmar que hoy en día los dos puntos de vista, el de la Inteligencia Artificial y los sistemas expertos por un lado, y el de la Ingeniería del Software por otro, han confluido en el uso de la tecnología de reglas, favoreciendo la amplia adopción de los sistemas basados en reglas y las reglas de negocio en la implementación de sistemas de información y de toma de decisiones complejos. Semantics of Business Vocabulary and Business Rules (SBVR) (Object Management Group, 2008c) es un formalismo de representación de reglas de negocio para sistemas de información. La OMG propuso los metamodelos para la Definición de Ontologías (Ontology Definition MetaModel (Object Management Group, 2009a)) y para la Representación de Reglas de Producción (Production Rule Representation (Object Management Group, 2009b)) como metamodelos estándares para la introducción de ambas tecnologías en el contexto de MDA. En este bloque podemos también considerar las representaciones basadas en estándares de modelado de software como son los enfoques para modelado de reglas basados en UML, como UML-Based Rule Modeling Language (URML) (Lukichev y Wagner, 2007), o el propio Object Constraint Language (OCL) (Object Management Group, 2012) que contiene el operador *implies* que permite modelar implicaciones similares a las reglas.

Entre todos ellos, en esta tesis doctoral se han utilizado CML como lenguaje de modelado que tiene su origen en la Ingeniería del Conocimiento, y OWL junto con SWRL, que igualmente tienen su origen en la IC pero que han evolucionado para su uso estandarizado actualmente en la Web Semántica.

2.2.2 Formalización de sistemas basados en reglas

Las tecnologías de implementación de sistemas basados en reglas están basadas en dos componentes principales: la base de reglas que codifica el conocimiento especifico del dominio utilizando un lenguaje de reglas, y el motor de inferencia, basado en el algoritmo Rete (Forgy, 1982), un algoritmo de reconocimiento de patrones para implementar sistemas de producción de reglas. También pueden incluir otros compo-

nentes, como son un mecanismo de explicación y una interfaz de usuario incorporada. A continuación se introducen brevemente las tecnologías de implementación de sistemas basados en reglas más importantes.

CLIPS (C Language Integrated Production System). Es un sistema experto a modo de interfaz de linea de comandos (*expert system shell*), es decir, la interacción con el usuario se realiza a través de una consola basada en texto. Usa el algoritmo Rete y proporciona su propio lenguaje de programación que soporta varios métodos de representación del conocimiento (Giarratano y Riley, 2004): *reglas*, que están dirigidas principalmente a representar el conocimiento heurístico basado en la experiencia, y *funciones* que están dirigidas principalmente al codificar conocimiento procedural. Igualmente soporta las características generales de la programación orientada a objetos. Fue escrito en lenguaje C por lo que es eficiente e independiente de plataforma. Se ha convertido en una plataforma para la implementación de aplicaciones basadas en reglas en investigación, educación y aplicaciones comerciales.

Jess (Java Expert System Shell) (Friedman-Hill, 2003). Es uno de los motores de reglas para Java más populares. Escrito en Java, Jess da soporte para desarrollar sistemas expertos basados en reglas que estén totalmente integrados con aplicaciones Java. Está inspirado en CLIPS y utiliza también el algoritmo Rete. Proporciona una librería de clases escrita en Java que sirve como un intérprete para el lenguaje Jess y que puede ser integrada en cualquier aplicación Java.

Drools (o JBoss Rules) es un sistema de gestión de reglas de negocio (BRMS, por las siglas en inglés de Business Rule Management System) que utiliza un motor de reglas basado en inferencia de encadenamiento hacia adelante (forward chaining) y de encadenamiento hacia atrás (backward chaining), y que utiliza una implementación avanzada del algoritmo Rete. Es software libre y forma parte del marco de trabajo de JBoss de RedHat. Soporta el enfoque de reglas de negocio (Ross, 2003) que pone énfasis en la integración de métodos basados en reglas en el contexto de aplicaciones de negocios complejas.

Existen otros muchos sistemas, tanto comerciales como de código abierto, que proporcionan igualmente capacidades de inferencia y representación del conocimiento usando reglas. Están orientados hacia la integración de sistemas basados en reglas principalmente con Java y .NET.

WebSphere ILOG JRules[1] es el BRMS de IBM. Este entorno

[1] http://www-01.ibm.com/software/integration/business-rule-

integrado soporta el desarrollo y la optimización de sistemas basados en reglas, así como una ejecución de las reglas robusta, escalable y precisa. Proporciona una herramienta potente y flexible para el modelado de reglas incluyendo muchas extensiones como la integración con Java y .NET.

OpenRules[2] es un entorno de reglas completo computo de un motor de reglas y un repositorio de reglas, así como de un repositorio de aprendizaje automático y una librería para la creación de interfaces de usuario. Se trata de una plataforma abierta en busca de aplicaciones comerciales.

Otros sistemas comerciales incluyen **Visual Rules**, de Bosch Software Innovations[3], **G2** de Gensym [4] y **RuleXpress** de RuleArts [5]. Todos ellos proporcionan capacidades de inferencia a partir de conocimiento modelado mediante reglas, utilizando notaciones específicas en cada caso.

Entre todos ellos, en esta tesis doctoral se han utilizado Jess, tal y como se indica en el capítulo 1.

2.3 Desarrollo de Software Dirigido por Modelos

El Desarrollo de Software Dirigido por Modelos (DSDM), nombre generalizado de la propuesta original Model-Driven Architecture (MDA) (Object Management Group, 2001) acuñado por el Object Management Group (OMG), es un enfoque del desarrollo de software basado en el modelado del software y en la definición de transformaciones entre modelos permitiendo el desarrollo de generadores de código a partir de modelos.

En la actualidad se emplea el término Ingeniería del Software Dirigida por Modelos (Model-Driven Software Engineering, MDSE o simplemente MDE) como el área de la Ingeniería del Software que se ocupa de la utilización sistemática de los modelos en las diferentes etapas de los procesos de producción de software, con el fin de mejorar la productividad a través de un incremento en los niveles de abstracción y automatización (Brambilla y cols., 2012; Cabot y cols., 2014). Existen

management/jrules-family/

[2] http://openrules.com/

[3] http://visual-rules.com

[4] http://www.gensym.com/product/G2

[5] http://www.rulearts.com/RuleXpress

cuatro escenarios principales de uso de MDE: automatizar el desarrollo de software, modernización dirigida por modelos, interoperabilidad de sistemas y uso de modelos en tiempo de ejecución (Brambilla y cols., 2012). De ellos el uso más conocido y extendido es el primero que consiste en la automatización de alguna tarea del ciclo de vida de desarrollo de software, lo que se suele denominar Desarrollo de Software Dirigido por Modelos (DSDM, o simplemente MDD en inglés).

Una solución MDA/MDE se basa en la aplicación de transformaciones sobre modelos. Estos modelos son expresados en un lenguaje de modelado que se define por medio de un metamodelo. Por lo tanto, podemos considerar que los elementos clave de MDE son: modelos, metamodelos y transformaciones de modelos.

En los últimos años el enfoque MDA/MDE está siendo aplicado en multitud de dominios, como por ejemplo la Ingeniería Web (Moreno y cols., 2008b), Ingeniería Ontológica (Gašević y cols., 2006) y al desarrollo de interfaces de usuario (Hussmann y cols., 2011).

La propuesta presentada en este trabajo de tesis doctoral consiste en la utilización de la tecnología MDA como marco en el que definir la transformación automática de modelos conceptuales a nivel de conocimiento, propios de la Ingeniería del Conocimiento, a modelos de diseño y posteriormente a código fuente del sistema software a desarrollar. Con ello conseguimos que el desarrollo de SBC se realice mediante un proceso lo más sistemático posible, automatizando parte del desarrollo mediante la utilización de herramientas.

2.3.1 Modelos y metamodelos

Un metamodelo define un lenguaje de modelado. Mediante el metamodelo se define la sintaxis abstracta del lenguaje de modelado (los conceptos del lenguaje y las relaciones entre ellos), mientras que la sintaxis concreta expresa la notación del lenguaje de modelado. Todos los modelos especificados con un lenguaje de modelado deben ser una instanciación correcta de su metamodelo. Se utiliza el término "conforma con" (*conforms to*) para expresar estas relaciones de instanciación entre modelos y metamodelos. Los lenguajes de modelado pueden ser gráficos, textuales o híbridos según la representación que puedan utilizar. Por ejemplo, UML tiene una notación gráfica para crear modelos, aunque también existe una notación textual equivalente, ambas conformes al metamodelo de UML (Cabot y cols., 2014).

Para el modelado de sistemas software, MDA define una jerarquía de

modelos en cuatro niveles. El nivel superior es el meta-metamodelado, M3, que tiene la responsabilidad de definir el lenguaje para la especificación de los metamodelos. Meta Object Facility (MOF) (Object Management Group, 2003b) es el lenguaje abstracto común de meta-metamodelado en MDA para la especificación de metamodelos. En el nivel de metamodelos, M2, se definen los lenguajes de modelado. Por ejemplo, Unified Modeling Language (UML) (Object Management Group, 2003c) y otros lenguajes de modelado están definidos a partir de su metamodelo. El nivel M1, contiene los modelos: especificaciones de sistemas concretos en cualquier ámbito o dominio de aplicación. Los modelos se definen de acuerdo con el metamodelo del lenguaje de modelado utilizado. Por último, el nivel más bajo de la jerarquía es M0 que representa las instancias en tiempo de ejecución de los elementos definidos en un modelo.

MDA plantea el diseño e implementación de una cadena de transformación de modelos como medio de automatizar la producción de código. Se suele partir de un modelo fuente que representa algún aspecto del sistema software, con frecuencia un modelo del dominio con las entidades del negocio, sus atributos y relaciones, y a partir de este modelo se generan artefactos software de la aplicación final a través de una cadena de transformaciones de modelos, como muestra la figura 1.1. El primer modelo que se define en MDA es un modelo independiente de plataforma (PIM, Platform Independent Model) con un alto nivel de abstracción que es independiente de cualquier tecnología de implementación. Un modelo PIM representa el problema utilizando los conceptos de la realidad del dominio de aplicación, sus propiedades y relaciones. Por ejemplo, si el dominio del problema es el sector bancario, ejemplos de conceptos serían “cliente” y “cuenta bancaria”, propiedades de los conceptos serían “número de teléfono” y “saldo”, y una relación entre conceptos sería “la cuenta bancaria es de un cliente”. El proceso de desarrollo establece la transformación del PIM en uno o más modelos específicos de plataforma (PSM, Platform Specific Model). Un PSM representa el sistema en términos de los constructores que están disponibles en una tecnología de implementación concreta. Así por ejemplo, un modelo específico de plataforma para bases de datos relacionales utiliza elementos como “tabla”, “columna”, “clave externa”, etc., para ofrecer una especificación del sistema basada en esa tecnología. El concepto “cliente” ahora se representa como la tabla “cliente”, y la propiedad “saldo” como la columna “saldo” de la tabla “cuenta bancaria”. El paso final en un desarrollo según MDA

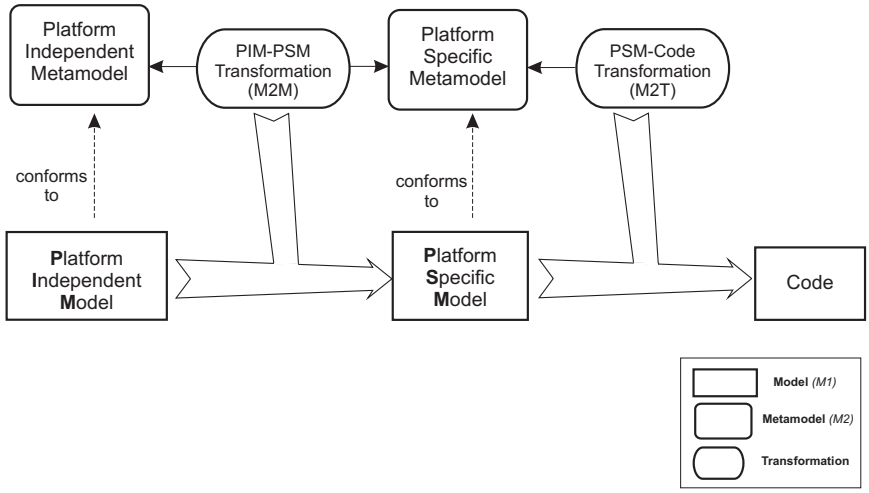


Figura 2.5: Metamodelos, modelos y transformaciones en MDA

es la transformación del PSM a código. Esta transformación es relativamente directa porque un PSM es un modelo directamente relacionado con la tecnología de implementación.

2.3.2 Transformaciones de modelos

Las transformaciones de modelos son elementos fundamentales en MDA (ver figura 2.5). Una transformación de modelos consiste en convertir un modelo en otro modelo, que puede ser del mismo nivel de abstracción (de PIM a PIM) o de distinto nivel de abstracción (de PIM a PSM y de PSM a código). Las transformaciones entre modelos reciben el nombre de transformaciones modelo a modelo (M2M, Model to Model), mientras que las transformaciones de modelo a código reciben el nombre de transformaciones modelo a texto (M2T, Model to Text) o también modelo a código (M2C, Model to Code). El objetivo de las transformaciones M2M es reducir el salto semántico desde el metamodelo del modelo fuente al código mediante el uso de modelos intermedios. Por lo tanto, cuando este salto semántico es pequeño la cadena puede reducirse a una única transformación M2T (García y cols., 2013).

Los lenguajes de transformaciones de modelo a modelo proporcionan construcciones para definir correspondencias entre los elementos de un metamodelo fuente y otro destino, de modo que la ejecución de la transformación pueda generar el modelo destino a partir del modelo fuente

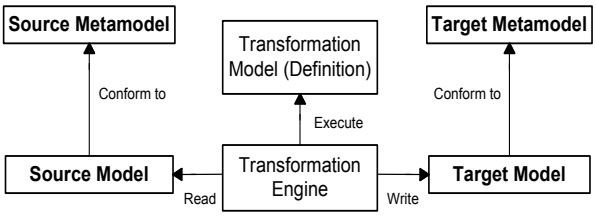


Figura 2.6: Conceptos de transformación de modelos

(ver figura 2.6). Algunos lenguajes permiten incluso definir una transformación que implique varios modelos fuente y varios modelos destino.

Una transformación modelo a texto se considera como un caso especial de transformación de modelos en el que un modelo origen es trasladado a un modelo textual destino, normalmente código en un lenguaje de programación definido por su correspondiente gramática. Los lenguajes para la definición de transformaciones modelo a texto proporcionan construcciones que recorren el modelo origen de acuerdo a la estructura definida en su metamodelo y expresan qué texto se debe generar para cada uno de los elementos del metamodelo accedidos.

Las herramientas de soporte a MDE permiten diseñar y ejecutar estas transformaciones utilizando lenguajes específicos para ello, entre los que destacan ATL (Atlas Transformation Language) (Jouault y cols., 2008) y QVT (Query/View/Transformation) (Object Management Group, 2008b), Epsilon (Epsilon, 2007) para las transformaciones M2M, y JET (Java Emitter Templates) (Eclipse Modeling Project, 2007), Acceleo (Musset y cols., 2006), Mof2Text (Object Management Group, 2008a) y XPand (Efftinge y Völter, 2006) para las transformaciones M2T.

2.3.3 Eclipse Modeling Framework

Esta tesis doctoral utiliza Eclipse Modeling Framework (EMF) (Steinberg y cols., 2008), el entorno de modelado definido para la plataforma Eclipse inspirado en MOF que constituye el núcleo del proyecto Eclipse Modeling Project[6]. En la actualidad EMF es el entorno de modelado más utilizado por la comunidad MDE, y numerosas herramientas tanto de modelado como de transformación de modelos están basadas en él. EMF está compuesto por un mema-metamodelo llamado Ecore y las he-

[6]http://www.eclipse.org/modeling/

rramientas necesarias para la creación y manipulación de (meta)modelos Ecore.

Así, para poner en práctica las propuestas de esta tesis doctoral, se han diseñado e implementado los metamodelos de los lenguajes de modelado utilizando el formato Ecore; para personalizar los editores de modelos basados en los metamodelos Ecore se ha utilizado Exeed (EXtended Emf EDitor) (Kolovos, 2007), que ofrece una versión mejorada del editor reflexivo basado en árbol de EMF; por su parte, las transformaciones modelo a modelo se han diseñado con ATL (Atlas Transformation Language) (Jouault y cols., 2008) y las transformaciones modelo a texto con JET (Java Emitter Templates) (Eclipse Modeling Project, 2007).

2.3.4 ATL. Atlas Transformation Language

El lenguaje ATL es el lenguaje oficial del proyecto Eclipse M2M. Es un lenguaje de transformación de modelos que permite especificar cómo uno o más modelos destino pueden ser producidos a partir de uno o más modelos fuente. ATL se enfoca principalmente a la transformación de modelo a modelo. Es un lenguaje un híbrido entre programación declarativa e imperativa, está especificado mediante su metamodelo y cuenta con una sintaxis concreta textual basada en OCL (Object Constraint Language) (Object Management Group, 2012). El estilo preferido de transformación escrita es la declarativa, ya que permite expresar reglas de correspondencia entre los elementos del modelo(s) fuente y los elementos del modelo(s) destino. Además de las reglas, se pueden definir *helpers* que son una estructura intermedia dentro de las transformaciones que permiten definir operaciones y ser llamados como si fueran funciones.

2.3.5 JET. Java Emitter Templates

JET (Java Emitter Templates) es un lenguaje para crear transformaciones de modelo a texto basado en plantillas. Proporciona un mecanismo de generación de código a partir de modelos EMF basado en plantillas que son creadas en una sintaxis formada por etiquetas que reutiliza parte de la sintaxis de JSP (Java Server Pages). Ofrece un editor que facilita la creación de las plantillas de generación. Forma parte del propio núcleo de EMF, ya que toda la infraestructura de generación de EMF está basada en JET.

2.4 Ingeniería Web

La Ingeniería Web (Rossi y cols., 2008) tiene como objetivo la definición de procesos, técnicas y modelos adecuados para el desarrollo de aplicaciones web. Metodologías como UML-based Web Engineering (UWE) (Koch y cols., 2008), Web Modeling Language (WebML) (Ceri y cols., 2002b) y Web Domain Specific Language (WebDSL) (Groenewegen y cols., 2008), abordan el diseño y desarrollo de aplicaciones web a partir de modelos conceptuales (Ceri y cols., 2002a), centrándose fundamentalmente en el modelado del contenido, la navegación y la presentación como los aspectos mas relevantes en el modelado de aplicaciones Web (Linaje y cols., 2007; Moreno y cols., 2008a). A partir de estos modelos, una aplicación web puede ser desarrollada aplicando el enfoque dirigido por modelos, incluyendo la capa de presentación compuesta por todos los elementos que forma la interfaz gráfica de usuario, como son páginas Web, formularios Web, enlaces, etc.

2.4.1 Tecnología de implementación Aplicaciones Web Ricas (RIAs)

Los frameworks para aplicaciones web son entornos de desarrollo diseñados para dar soporte a la construcción de sitios web dinámicos, servicios y aplicaciones web. Intentan reducir el exceso de carga asociado con actividades comunes empleadas en desarrollos web. Muchos frameworks, por ejemplo, disponen de librerías para acceder a bases de datos, librerías de componentes gráficos, estructuras para plantillas, gestión de sesiones, y en muchas ocasiones facilitan la reutilización de código. En la actualidad, existen gran cantidad de frameworks para lenguajes como, por ejemplo, Python, Ruby, PHP, Java o Perl.

El desarrollo de interfaces de usuario modernas requiere el uso de extensos frameworks y librerías de software, y el código llega a ser bastante específico de la plataforma (Hussmann y cols., 2011).

Las tecnologías actuales de la Web 2.0 incluyen las llamadas aplicaciones Web ricas (Rich Internet Applications, RIAs) (Driver y cols., 2005) que mejoran de la experiencia de usuario mediante interfaces Web más avanzadas e interactivas, similares a las de aplicaciones de escritorio, mediante la incorporación de AJAX (Asynchronous JavaScript And XML), y a su vez minimizando el tráfico de la red e incrementa extraordinariamente la usabilidad y eficiencia para el usuario final (Wright y Dietrich, 2008).

2.4.2 Frameworks para RIAs en JavaEE

Con el objetivo de poder integrar la base de reglas y el motor de inferencias Jess, basado en Java, en una aplicación Web basada en reglas, en este trabajo de tesis nos hemos centrado el la tecnología Java Enterprise Edition (JavaEE[7]), y para la parte Web en el framework JavaServerFaces (JSF) (Geary y Horstmann, 2007) como plataforma de implementación de aplicaciones Web ricas. Para conseguir una de las características más importantes de las RIAs, una interacción rica se consigue añadiendo la tecnología AJAX que proporciona una experiencia de usuario mejorada. Existe diversos frameworks de interfaces de usuario para aplicaciones JSF, como RichFaces[8] (JBoss project), ICEfaces[9] (ICEsoft project), MyFaces[10] (Apache project), ADF Faces[11] (Oracle project) and Google Web Toolkit[12] (Google project). En esta tesis hemos considerado RichFaces por ser el más estandarizado en el momento de tomar la decisión, aunque el mismo enfoque puede ser fácilmente adaptado a otros frameworks, ya que en su mayoría funcionan de forma similar y la principal diferencia radica en los nombres de los componentes que ofrecen en su librería de componentes gráficos.

2.4.3 Librería de componentes RichFaces de JBoss

La decisión de usar RichFaces como librería de componentes Java para JSF, como hemos introducido, se basó en la mayor madurez y más completo catálogo de componentes de interfaz de usuario que poseía RichFaces respecto a otros frameworks.

JBoss RichFaces es un framework avanzado basado en JSF que proporciona un completo conjunto de componentes de interfaz de usuario ricos incorporando AJAX. RichFaces está compuesto por dos librerías de etiquetas de componentes:

- *a4j:* representa la funcionalidad básica de AJAX,
- *rich:* hace referencia a componentes avanzados autocontenidos como calendarios, tablas de datos, árboles, etc. (ver el catálogo Rich-

[7] http://java.sun.com/javaee
[8] http://www.jboss.org/richfaces
[9] http://www.icefaces.org/
[10] http://myfaces.apache.org/
[11] http://www.oracle.com/technetwork/developer-tools/adf/
[12] http://code.google.com/webtoolkit/

Faces [13] para más detalle).

La última versión, en el momento de realización de este trabajo de tesis, es RichFaces 4.x, que puede se usada en cualquier contenedor compatible con JSF 2.0.

2.5 Conclusiones del capítulo

La definición de una metodología o proceso de desarrollo para la creación de sistemas basados en reglas a partir de modelos es un tema de trabajo actual.

Diouf y colaboradores (Diouf y cols., 2007) describen como mezclar modelos UML y ontologías OWL para la generación automática de reglas de negocio. Proponen el uso de ontologías para añadir semántica a los modelos UML y la aplicación del enfoque MDA para la extracción de reglas implícitas de los modelos. Sus reglas de negocio son generadas en la sintaxis de Semantics of Business Vocabulary and Business Rules (SBVR) (Object Management Group, 2008c) y aunque solamente obtienen una primera abstracción de las reglas de negocio, la propuesta incluye una arquitectura general para la aplicación de MDA y ontologías OWL para la generación de conjuntos de reglas.

Otro método relacionado es la metodologia HeKatE (Nalepa y Ligeza, 2010) que describe una aproximación similar a MDA para el diseño y desarrollo de sistemas basados en reglas complejos. Utiliza árboles tabulares extendidos (eXtended Tabular Trees) como formalismo de representación de reglas, un paradigma basado en tablas de decisión conectadas en una rede de inferencia. Las reglas son traducidas a código basado en *Prolog* e integradas como el módulo de lógica en el núcleo del sistema software final.

En relación a las aplicación de MDA en aplicaciones web que integren reglas, el trabajo de Ribarić y colaboradores (Ribarić y cols., 2007) describe los principios del desarrollo dirigido por modelos para el modelado de servicios web basados en reglas, utilizando REWERSE Rule Markup Language (R2ML) como lenguaje de representación, y propone un enfoque MDA para la generación de servicios web a partir de los modelos de reglas.

[13] http://showcase.richfaces.org/

Capítulo 3

Desarrollo dirigido por modelos de la base de conocimiento

Este capítulo describe el proceso dirigido por modelos que permite obtener una base de reglas implementada en la tecnología de sistemas basados en reglas Jess a partir de los modelos conceptuales. Se han considerado dos alternativas como formalismos para representar los modelos conceptuales de ontologías y reglas: Conceptual Modeling Language (CML), propuesto en la metodología CommonKADS, y Web Ontology Language (OWL) junto con Semantic Web Rule Language (SWRL), estándar para la definición de ontologías en Ingeniería Ontológica y la Web Semántica.

Así, este capítulo se estructura en dos bloques principales: la secciones 3.2, 3.3 y 3.4 relativas al proceso dirigido por modelos que parte de CML como lenguaje de especificación de los modelos origen del proceso; y las secciones 3.5 y 3.6, relativas al uso de OWL y SWRL como modelos independientes de plataforma origen del proceso. En ambos casos, primero describen los lenguajes implicados, sus metamodelos y herramientas de modelado, y a continuación las transformaciones de modelos. Para un mejor entendimiento del proceso se emplea un sencillo ejemplo ilustrativo.

3.1 Perspectiva general

Tal y como se comentó en la sección 1.3, el proceso dirigido por modelos propuesto para el desarrollo de aplicaciones web basadas en reglas está dividido en dos subprocesos, el primero de ellos (el flujo *inferior* en la figura 1.4) es el que se describe en este capítulo, que genera la implementación de la base de reglas a partir de un modelo conceptual a nivel de conocimiento que contiene la ontología del dominio y la lógica de decisión en forma de reglas.

Estos modelos conceptuales a nivel de conocimiento representan la especificación del dominio mediante una ontología *ligera* (Gómez-Pérez y cols., 2004), que es comparable en expresividad a los modelos conceptuales ampliamente usado en la Ingeniería del Software, como los modelos de clases UML (Kogut y cols., 2002). El enfoque propuesto se centra en la especificación del problema combinando reglas y permiten la especificación de la estructura de la información del dominio mediante conceptos, taxonomías de conceptos, relaciones entre conceptos y propiedades que los describen. Las ontologías ligeras se diferencian de las ontologías pesadas en que estas últimas permiten clarificar el significado de los términos recogidos en la ontología, proporcionando formalismos semánticamente más ricos que los modelos conceptuales (Gómez-Pérez y cols., 2004).

La base de reglas a obtener como resultado de este proceso está compuesta principalmente por reglas en una plataforma específica de sistemas basados en reglas, como es Jess. Para realizar este proceso se aplica un enfoque dirigido por modelos que permite establecer de manera sistemática la correspondencia entre los elementos del modelo concep-

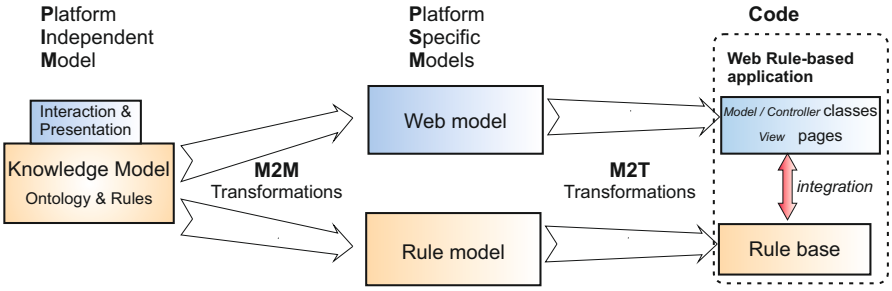


Figura 1.4: Esquema general: MDA para aplicaciones web basadas en reglas (repetición)

tual con los elementos de la plataforma de implementación y por último obtener el código fuente en Jess.

3.2 Modelado de aplicaciones basadas en reglas con CML

Existen numerosos formalismos para el modelado de ontologías y reglas, tal y como se describe en la sección 2.2.1. El primero de los formalismos seleccionados en esta tesis doctoral es el Lenguaje de Modelado Conceptual (Conceptual Modeling Language, CML) que fue propuesto en la metodología CommonKADS (Schreiber y cols., 2000). CML tiene varias características deseables para este trabajo, entre las que destacan las siguientes:

- Permite la representación en un mismo modelo y formato de ontologías ligeras y reglas íntimamente relacionadas mediante la asociación directa entre las reglas y los conceptos de la ontología.
- Cumple con los requisitos básicos de los formalismos de representación de reglas, como son el modelado del antecedente y consecuente de la regla, reglas con nombre y definición de bases de reglas, con una expresividad comparable a SWRL que tampoco es altamente expresivo ni decidible, pero se mantiene simple (Paptaxiarhis y cols., 2009).
- Los modelos escritos en este formalismo son independientes de cualquier plataforma de implementación, y por lo tanto, pueden ser usados como modelos origen en un enfoque dirigido por modelos.
- Al tratarse de un lenguaje de modelado no estándar facilita la extensión de la notación para poder representar algunas características semánticas de aplicaciones web basadas en reglas, permitiendo el objetivo de esta tesis de integración de las reglas en una aplicación web.

En Abdullah y cols. (2007) los autores presentan un perfil UML basado en CML para la especificación de modelos de conocimiento y discuten la posibilidad de definir una correspondencia entre los elementos del perfil y un modelo específico para reglas Jess. A diferencia de ese trabajo, en esta tesis se ha diseñado un metamodelo para CML que se

presenta a continuación en esta sección, siguiendo la tendencia de la comunidad MDE, en lugar de usar el mecanismo de perfiles UML por su dificultad de adaptarlo a dominios concretos (Volter, 2009). Igualmente en esta tesis se ha definido formalmente la correspondencia entre CML y Jess mediante transformaciones de modelos, tal y como se describe en las siguientes secciones en este capítulo.

3.2.1 Ejemplo ilustrativo CML

Para ilustrar la descripción de la propuesta, vamos a emplear un caso de estudio sencillo que modela las relaciones familiares (Horrocks y cols., 2004) y define reglas sencillas para derivar relaciones entre personas de una misma familia[1]. En el dominio Familia, el concepto *Persona* se describe mediante un conjunto de atributos: *nombre*, *edad*, *sexo*, *padre* y *madre*. El *nombre* y la *edad* son atributos de los tipos primitivos cadena de caracteres (*String*) y entero (*Integer*) respectivamente. El atributo *sexo* puede tomar uno de los dos siguientes valores literales, *masculino* o *femenino*, los cuales se definen como un tipo enumerado. Los atributos *padre* y *madre* son de tipo *Persona*, por lo que estos 5 atributos permiten relacionar una instancia o individuo de tipo Persona con otra instancia o individuo de tipo Persona. Cada uno de estos atributos puede tener a lo sumo un valor relacionado, es decir, su cardinalidad es simple. El resto de atributos de Persona, como son el conjunto de *hermanos y hermanas*, *hermanos masculinos*, *hermanas*, *hijos* de cualquier sexo, *hijos masculinos*, *hijas*, *tíos*, *tías*, etc., definen cada uno de ellos relaciones Persona-Persona mediante atributos de tipo Persona, pero en este caso la cardinalidad es múltiple, ya que cada instancia o individuo de tipo Persona puede estar relacionado con cero, una o varias personas mediante estas relaciones. Todos los atributos de tipo Persona, bien de cardinalidad simple o múltiple, se podían haber definido igualmente mediante relaciones binarias entre Persona, pero se ha elegido su definición como atributos ya que es equivalente y permite su representación gráfica más sencilla. La figura 3.1 muestra el ejemplo Familia en forma de diagrama de clases UML.

[1]Por claridad en el texto se han utilizado la traducción de los términos al castellano aunque en las figuras y reglas se mantiene la versión en el idioma original. La correspondencia entre ambos es la siguiente:

name	nombre	*age*	edad	*gender*	sexo	*father*	padre
mother	madre	*male*	masculino	*female*	femenino	*person*	persona
brother	hermanos	*sisters*	hermanas	*children*	hijos/as	*sons*	hijos
siblings	hermanos/as	*daugthers*	hijas	*uncles*	tios	*aunts*	tías

Una vez definido el modelo conceptual del ejemplo Familia, con las clases y atributos descritas mediante un diagrama UML para una mejor comprensión visual del modelo, el modelo se enriquece añadiendo las reglas que que permiten deducir los valores de las relaciones familiares. Estas reglas se definen ligadas a la clase Persona. Por ejemplo, si dos personas tienen el mismo padre y la misma madre entonces son hermanos. Utilizando una sintaxis próxima al lenguaje natural, considerando las variables x e y de tipo Persona, la regla anterior puede expresarse así:

```
IF (x.father==y.father) AND (x.mother==y.mother)
THEN (x.siblings = y) AND (y.siblings = x)
```

Igualmente, la regla que describe la relación tío (*uncle*) se define mediante la siguiente expresión:

```
IF (x.father==y) AND (y.brother==z)
THEN (x.uncle = z)
```

El resto de relaciones personales entre las personas de una familia se definen mediante reglas similares.

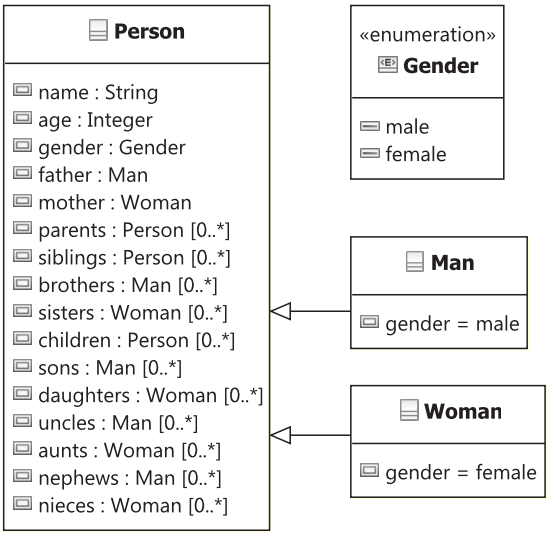


Figura 3.1: Diagrama de clases UML del ejemplo Familia

3.2.2 Modelado conceptual basado en reglas

El formalismo de modelado conceptual de conocimiento CML permite la especificación de ontologías ligeras del dominio junto con reglas de producción ligadas a los elementos de la ontología, dentro de lo que se denomina el modelo de *conocimiento del dominio*[2]. Un modelo conceptual de conocimiento del dominio en CML está compuesto básicamente por dos componentes: un *esquema de dominio*[3] y una *base de conocimiento*[4].

El esquema de dominio incluye las clases o conceptos del dominio (*Concepts*), las relaciones binarias entre conceptos (*Binary relationships*), los tipos de reglas (*Rule types*) y los tipos enumerados (*Value types*). Por otro lado, la base de conocimiento se compone de las instancias de los conceptos (individuos), instancias de las relaciones (tuplas) y las instancias de las reglas. La Figura 3.2 muestra los componentes del modelo de conocimiento del dominio en CML.

Como se muestra a continuación, el metamodelo de CML especifica las primitivas que pueden ser usadas en el lenguaje de modelado CML, mediante una notación de diagramas de clase UML simplificadas llamada MOF (Object Management Group, 2003b). La Figura 3.3 muestra un fragmento del metamodelo de CML en el que se definen las metaclases para las primitivas Concepto (*Concept*), Enumerado (*ValueType*) y Relación binaria (*BinaryRelation*) que forman parte del esquema de dominio.

La principal diferencia entre este formalismo y otras aproximaciones

[2] *Domain Knowlege*
[3] *Domain schema*
[4] *Knowledge base*

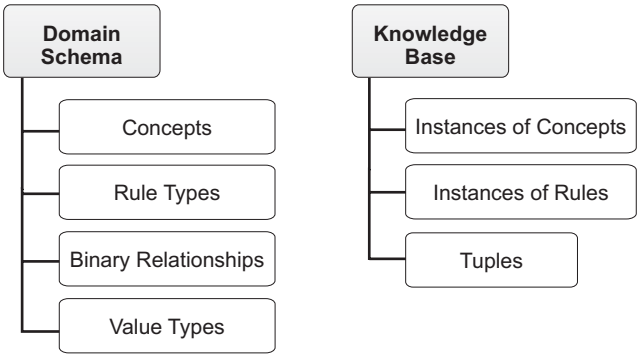


Figura 3.2: Estructura del modelado de conocimiento del dominio

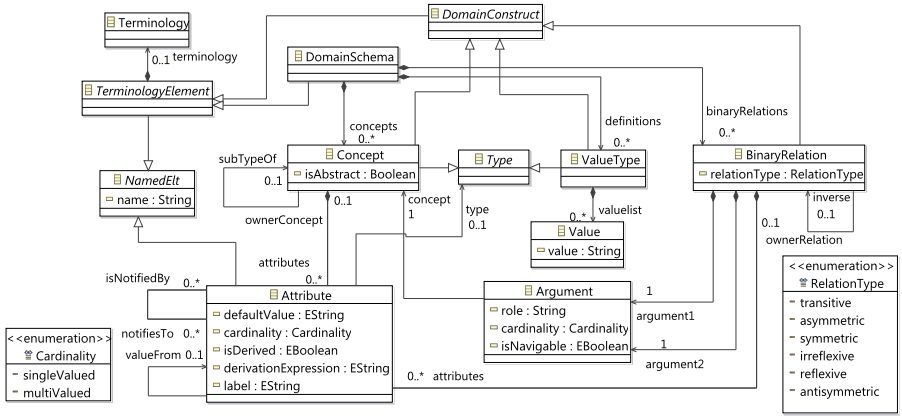


Figura 3.3: Fragmento del metamodelo de CML: Conceptos y relaciones

de modelado conceptual en Ingeniería del Software, tales como los diagramas de clase UML, es la posibilidad de modelar reglas de producción. Para ello se utilizan las primitivas Tipo de regla (*RuleType*) e Instancia de tipo de regla (*RuleTypeInstance*). Un Tipo de regla describe un conjunto de reglas similares mediante la especificación de la estructura de las mismas indicando los tipos ligados al antecedente y consecuente del Tipo de regla.

Una especialización de Tipo de regla es el Tipo de regla de producción, que se modela mediante la primitiva Regla de implicación (*ImplicationRule*), que hereda de Tipo de regla (*RuleType*). El metamodelo para *RuleType* e *ImplicationRule* se muestra en la Figura 3.4. Cada regla de implicación posee una propiedad denominada *ConnnectionSymbol* que consiste en una palabra o texto que representa el significado de la regla en lenguaje natural, así como un antecedente y un consecuente, cada uno de ellos ligados con uno o varios conceptos (*Concept*s) o relaciones binarias (*BinaryRelation*) definidas en el modelo.

Una Base de conocimiento (*knowledge base*) contiene instancias de conceptos (individuos), instancias de relaciones (tuplas) e instancias de tipos de regla. Estos elementos se definen en el metamodelo mediante las metaclases Instancia (*Instance*), Tupla (*Tuple*) e Instancia de regla (*RuleTypeInstance*), respectivamente.

Los Tipos de regla (*RuleType*s) se particularizan en Instancias de reglas (*RuleTypeInstance*s). Una Instancia de regla representa una dependencia lógica específica entre los conceptos asociados al antecedente

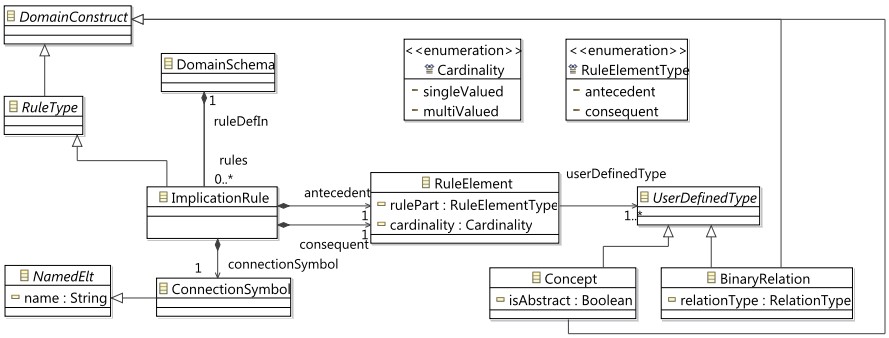


Figura 3.4: Fragmento del metamodelo CML: Tipo de regla

y al consecuente de la regla, indicando valores específicos que toman los atributos de los conceptos asociados al antecedente y consecuente.

Las reglas de producción se especifican como Instancias de tipo de Regla (*RuleTypeInstance*), que son ejemplos concretos o particularizaciones de los tipos de regla (*RuleTypes*). Una instancias de tipo de regla se define mediante las condiciones o expresiones lógicas ligadas a los conceptos ligados en el antecedente y las acciones o expresiones ligadas a los conceptos del consecuente de la regla (Fig. 3.5). Cuando se trata del antecedente de una instancia de tipo de regla, las expresiones lógicas se pueden definir mediante conjunciones, disyunciones y/o negaciones en el antecedente de la instancia de regla. Dependiendo de la cardinalidad de los atributos usados en la expresión lógica, que pueden atributos de cardinalidad sencilla (single) o múltiple, es decir, una lista de valores (list), las expresiones deberán utilizar operadores adecuados para dicha cardinalidad (single/list). La Tabla 3.1 resume los operadores binarios que se pueden usar en las expresiones. Tanto las variables como los valores literales se pueden utilizar como operandos en las condiciones del antecedente y como acciones en el consecuente de la regla.

Esta tabla contiene los operadores de la descripción original de CML ampliada con los operadores sobre colecciones (listas) para proporcionar una mayor expresividad al lenguaje. Estos operadores sobre colecciones están inspirados en la notación OCL (Object Constraint Language) (Object Management Group, 2012).

Una descripción completa de este metamodelo se puede consultar el apéndice A.

Tabla 3.1: Operadores binarios en CML

Operador	*Operandos*	*Resultado*	*Descripción*
`<`	numérico	lógico	menor
`>`	numérico	lógico	mayor
`<=`	numérico	lógico	menor o igual
`>=`	numérico	lógico	mayor o igual
`==`	numérico	lógico	igual
`!=`	numérico	lógico	distinto
`+`	numérico	numérico	suma
`-`	numérico	numérico	resta
`*`	numérico	numérico	multiplicación
`/`	numérico	numérico	división
`=`	*cualquiera*	*cualquiera*	asignación de valores de tipo compatible
`add`	lista, elemento	lista	añadir un elemento a la lista
`remove`	lista, elemento	lista	eliminar un elemento de la lista
`includes`	lista, elemento	lógico	*true*, la lista contiene al elemento; *false* en otro caso
`excludes`	lista, elemento	lógico	*true*, la lista no contiene al elemento; *false* en otro caso
`includesAll`	lista, lista	lógico	*true*, la primera lista contiene todos los elementos de la segunda lista; *false*, en otro caso
`excludesAll`	lista, lista	lógico	*true*, la primera lista no contiene ninguno de los elementos de la segunda lista; *false*, en otro caso
`union`	lista, lista	lista	unión de listas
`intersection`	lista, lista	lista	intersección de listas
`difference`	lista, lista	lista	diferencia de listas

3.2.3 Editor de modelos CML basado en Eclipse

El modelo conceptual para el ejemplo ilustrativo Familia puede ser especificado de acuerdo con el metamodelo descrito. La clase Persona se define como un concepto con un conjunto de atributos. Aquellos atributos que pueden tomar solamente un valor se definen como simples (*singleValued*), como son padre, madre, etc., mientras que la cardinalidad múltiple (*multiValued*) se aplica a los atributos que pueden tomar una lista de valores, como son *hermanos*, *hermanas*, etc. El tipo *Sexo* se define como un enumerado (*ValueType*) con dos valores literales, *masculino* y *femenino*. El tipo de regla *PersonAbstraction* define el esqueleto para las reglas que permiten deducir relaciones familiares relacionando valores de atributos de Persona en el antecedente con valores de atributos de Persona en el consecuente. Las reglas concretas que permiten derivar cada tipo de relación familiar se definen como instancias de reglas del tipo *PersonAbstraction*. La Figura 3.6 muestra un fragmento del modelo del ejemplo de familia, creado con una herramienta de edición

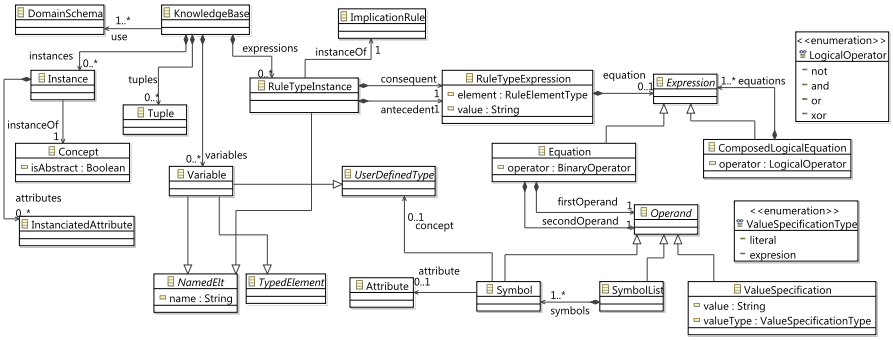


Figura 3.5: Fragmento del metamodelo CML: Base de conocimiento e Instancia de Tipo de Regla

basada en el metamodelo de CML propuesto.

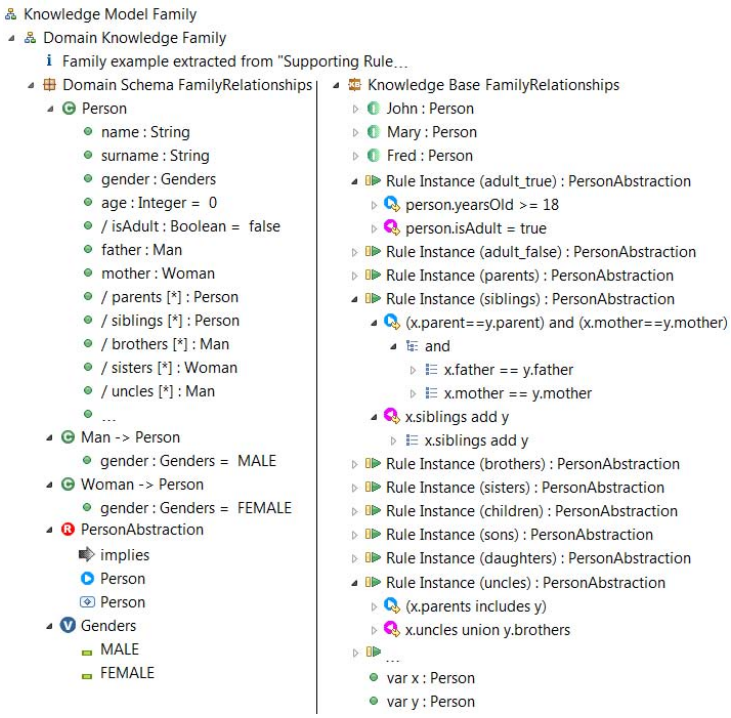


Figura 3.6: Fragmento del modelo del ejemplo Familia

3.3 Modelado específico de base de reglas en Jess

Como se ha comentado al inicio de este capítulo, el primer paso del proceso dirigido por modelos propuesto en esta tesis implica la transformación de los modelos conceptuales en CML a código en un lenguaje de implementación de sistemas de reglas, en nuestro caso Jess. La primera transformación modelo a modelo (M2M) implica transformar un modelo CML independiente de plataforma (PIM) en un modelo específico de plataforma (PSM) para reglas Jess. Para ello, se propone un metamodelo para Jess que permite representar modelos específicos de reglas Jess. Este metamodelo, mostrado en la figura 3.7, se ha diseñado ampliando y extendiendo un metamodelo básico para sistemas basados en reglas propuesto en (Chaur G. Wu, 2004).

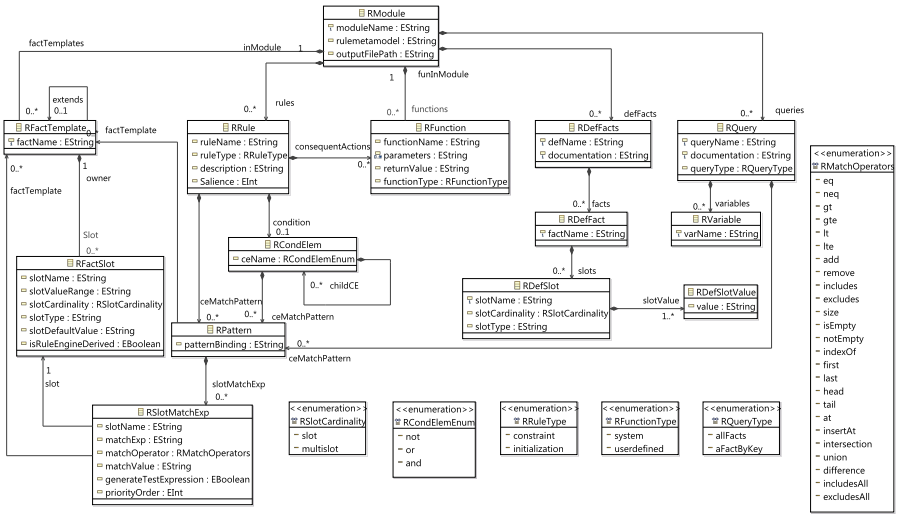


Figura 3.7: Metamodelo para Jess (*ruleMetaModel)*

El metamodelo específico para Jess define una metaclase para cada elemento del lenguaje Jess. El elemento raíz de un modelo de reglas Jess es un módulo representado por la clase *RModule*. Un módulo contiene los distintos elementos que se pueden definir en Jess, como son:

- plantillas de hechos (fact templates),
- reglas,

- funciones,
- hechos (facts)
- y consultas (queries).

La clase *RFactTemplate* permite modelar plantillas de hechos, que es el constructor que permite representar y almacenar conceptos con información en Jess. La clase *RRule* permite la representación de las reglas. Toda regla tiene las siguientes propiedades:

- un nombre (*ruleName*),
- un *salience* que indica la prioridad de la regla y determina el orden en que las reglas son disparadas por el motor de inferencia cuando hay varias reglas aplicables en un mismo instante.
- una referencia a la clase *RCondElem* llamada *condition* que representa la condición de la regla (antecedente). Las clases *RPattern* y *RSlotMatchExp* definen expresiones de *coincidencia de patrones (pattern matching)* en las condiciones de la regla.
- una referencia a la clase *RFunction* llamada *consequentActions* que representa las funciones que se ejecutan como acciones de la regla (consecuente). Las acciones se definen como instancias de *RFunction* que son llamadas a funciones que añaden nuevos hechos a la base de hechos, o bien eliminan o modifican hechos existentes en la base de hechos.

Los hechos se definen mediante la metaclase *RDefFacts*. Los hechos iniciales de la base de hechos se obtienen a partir de las instancias de conceptos del modelo origen CML. Finalmente, *RQuery* permite representar consultas que acceden a la memoria de trabajo en tiempo de ejecución.

Una descripción completa de este metamodelo se puede consultar el apéndice B.

3.4 Transformaciones de modelos: desde CML a Jess

Las transformaciones modelo a modelo (M2M) y modelo a texto (M2T) permiten diseñar la forma en que los modelos se convierten en

otros modelos de similar o distinto nivel de abstracción, transformación PIM-to-PIM o transformación PIM-to-PSM respectivamente, y en texto asociado al código fuente en la plataforma de implementación elegida.

A continuación se describe la cadena de transformaciones diseñadas para obtener el código fuente de la base de conocimiento en Jess a partir del modelo CML independiente de plataforma. El proceso, descrito en la figura 1.7, se compone de una primera transformación M2M en la que se parte de CML y se obtiene un modelo basado en el lenguaje de modelado de reglas Jess propuesto en la sección 3.3, y a continuación se aplica una segunda transformación M2T en la que a partir de los elementos del modelo de reglas Jess se genera el código fuente en Jess.

3.4.1 Transformación Modelo a Modelo

La conversión de modelos CML a modelos de reglas en Jess ha sido diseñada mediante una transformación modelo a modelo (M2M) que traslada cada primitiva del metamodelo CML (ver sección 3.2) a uno o varios elementos del metamodelo de reglas Jess (ver sección 3.3).

Esta transformación se define mediante un conjunto de reglas que describen el mapeo de elementos del metamodelo origen, CML, en elementos del metamodelo destino, en este caso el metamodelo de reglas Jess. Las principales reglas de esta transformación se resumen a continuación:

- *Esquema de dominio a base de conocimiento (Domain Schema to knowledge base).* Un esquema de dominio en CML se convierte en

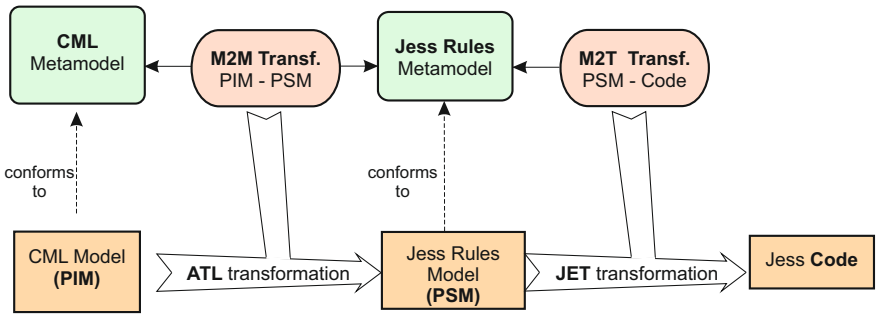


Figura 1.7: Proceso para la generación de la base de reglas *Jess* (repetición)

una base de conocimiento Jess que contendrá todos los elementos en Jess. El nombre del esquema de dominio es el mismo que el nombre de la base de conocimiento en Jess. Una base de conocimiento en Jess se convertirá en un archivo .clp de la base de conocimiento correspondiente, y este paso se realizará mediante la posterior transformación modelo a código (M2T).

- *Concepto a Plantilla de hechos (Concept to Template).* Un Concepto se transforma en una plantilla de hechos en Jess, es decir en un *RFactTemplate*. El nombre de la plantilla de hechos coincide con el nombre del concepto. Cada atributo del concepto se convierte en un *Slot* de la plantilla de hechos. El nombre del slot se corresponde con el nombre del atributo. Puesto que Jess permite slots simples que contienen un valor único, y multislots que contienen una lista de valores, un atributo con cardinalidad uno se convertirá en un slot mientras que atributos multivaluados (cardinalidad a muchos) se convertirán en multislots.

- *Relación a Slot.* Cada relación binaria es transformada in un slot de la plantilla de hechos generada a partir del concepto relacionado por el primer argumento de la relación binaria.

- *Instacia de concepto a Hecho (Concept Instance to DefFact).* Una instancia se convierte en un hecho (*RDefFact*), y los atributos instanciados se trnasforman en los valores que toman los slots en la definición del hecho. *Cuando una instancia tiene un slot de una relación, una tupla da el valor a ese slot.*

- *Tipo de regla e Instancia de regla a Regla Jess (RuleType and RuleInstance to Rule).* Cada instancia de regla en CML se transforma a una Regla Jess, con un *Antecedente* y un *Consecuente*. El *Antecedente* está compuesto de patrones y expresiones de reconocimiento de patrones que hacen referencia a las plantillas incluidas en la definición del tipo de regla. Un *Patrón* indica las plantillas que debe tener una expresión de reconocimiento de patrones en la regla. Una *Expresión de reconocimiento* de patrones representa una condición en el antecedente de la regla, por ejemplo una expresión de comparación lógica.

- *Concepto a Consultas (Concept to Queries).* Finalmente, para cada Concepto se crean dos *Consultas*: una consulta de todas las

instanicas (*all instances query*) y una consulta por identificado de concepto (*Fact byId query*). La primera es necesaria para recuperar todas las instancias de la clase, y la segunda para recuperar una instancia concreta a partir de su identificador.

La transformación ha sido implementada en ATL. En total está compuesta por un conjunto de 34 reglas de transformación que convierten cada elemento del metamodelo CML en uno o varios elementos del metamodelo de reglas Jess, y 19 *helpers*, que son funciones necesarias en la transformación. La figura 3.8 muestra el listado de componentes de la transformación *CML2RuleMetamodel* tal y como se visualizan en la vista resumen de Eclipse.

A modo de ejemplo, la figura 3.9 muestra el código de la primera regla ATL que transforma un esquema de dominio a un módulo, el elemento principal de que constituye la base de conocimiento Jess. Esta regla es sencilla, aunque es importante resaltar que en general el diseño de las reglas ATL ha sido uno de los elementos más complejos y que han supuesto un mayor esfuerzo de esta tesis doctoral, debido a la naturaleza declarativa del lenguaje ATL y a la extensión y complejidad de los metamodelos a transformar.

Un fragmento del modelo de reglas Jess resultante de la aplicación de la transformación al ejemplo de la *Familia* presentado en la figura 3.6 se muestra en la figura 3.10.

3.4.2 Transformación Modelo a Texto

El modelo de reglas Jess generado en la transformación M2M anterior es el modelo origen para una transformación modelo a texto (M2T) que se encarga de generar automáticamente el código fuente de la base de reglas en Jess, produciendo un archivo Jess con extensión .clp que contiene el código para cada elemento que forma parte del modelo de reglas Jess.

Esta transformación M2T se caracteriza por ser una transformación relativamente directa, ya que el modelo origen es un modelo específico para la plataforma Jess, con lo cual todos los elementos de ese modelo tienen una conversión directa a código fuente en Jess. Para el diseño de la trasformación, se han creado plantillas de texto basadas en la sintaxis de Jess, usando la herramienta JET, que permiten la generación del código fuente basado en esas plantillas. El ejemplo de la figura 3.11 muestra un ejemplo de plantilla JET para la generación de un *deftemplate*, y cómo

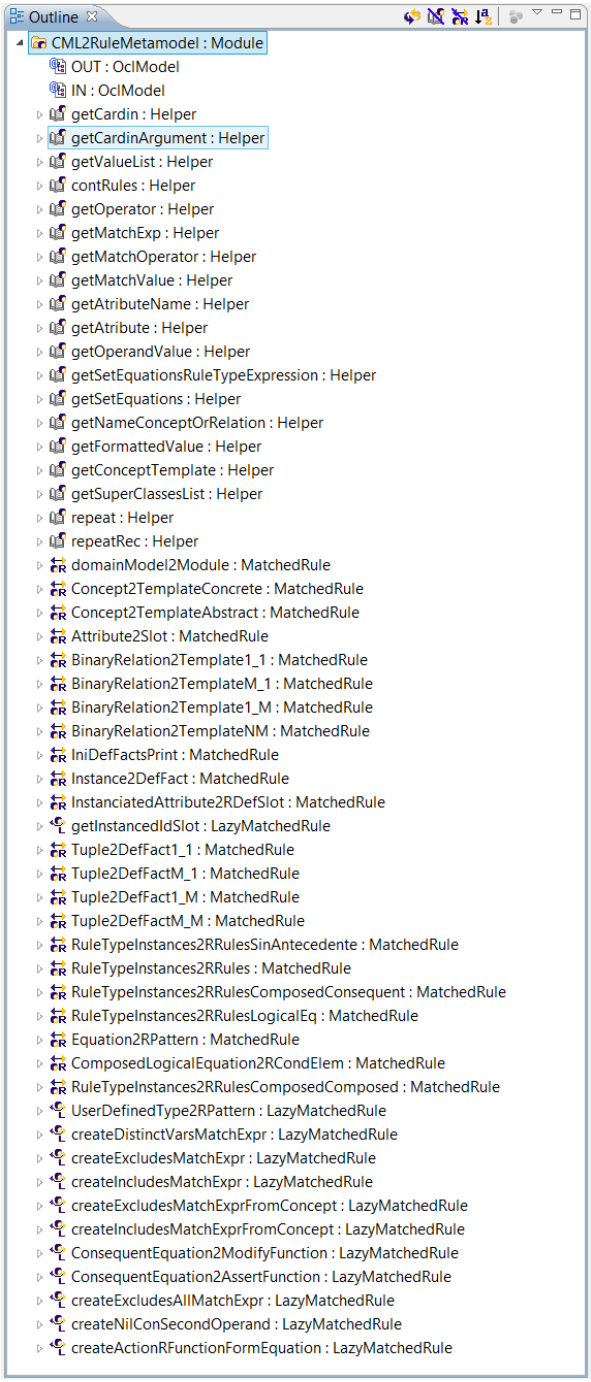


Figura 3.8: Resumen de la transformación ATL *CML2RuleMetamodel*

```
rule domainModel2Module{
    from
        d : simplecml!DomainSchema
    to
        m : simpleRuleMetaModel!RModule(
            moduleName <- d.name,
            factTemplates <- d.concepts,
            rules <- simpleRuleMetaModel!RRule.allInstances(),
            defFacts <- simpleRuleMetaModel!RDefFacts.allInstances(),
            queries <- simpleRuleMetaModel!RQuery.allInstances()
        )
}
```

Figura 3.9: Ejemplo de regla ATL

Figura 3.10: Fragmento del modelo de reglas Jess del ejemplo Familia

a partir de los elementos del modelo se obtiene el código final.

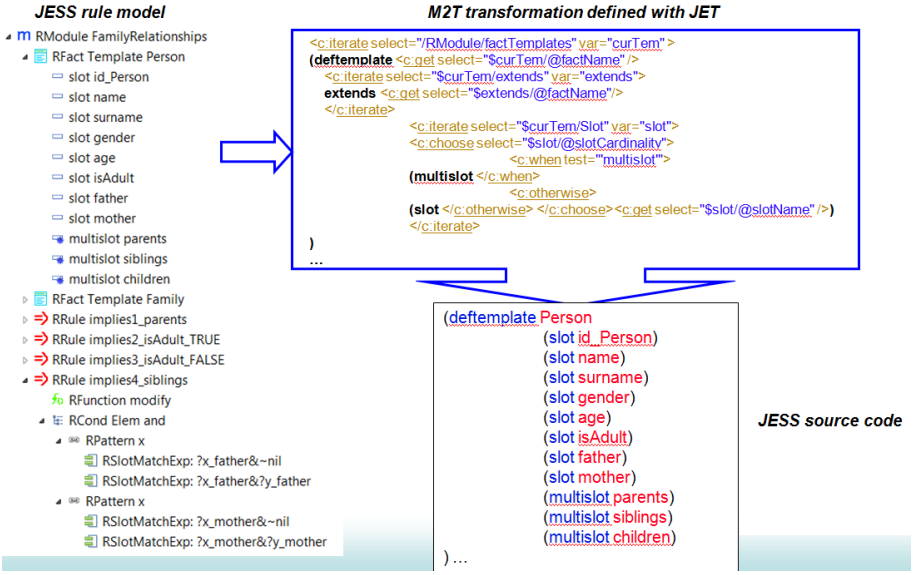


Figura 3.11: Fragmento de plantilla Jet del ejemplo Familia

3.5 Modelado de aplicaciones basadas en reglas con OWL y SWRL

En esta sección se presenta un enfoque dirigido por modelos para el desarrollo de aplicaciones basadas en reglas partiendo de especificaciones OWL y SWRL como modelos origen del proceso. Aunque existen otros formalismos para el modelado de ontologías y reglas, el uso de OWL y SWRL es una elección razonable ya que son considerados lenguajes estándar para la representación de ontologías y son usados ampliamente por la comunidad (principalmente OWL).

3.5.1 Modelado con OWL y SWRL

Una ontología se utiliza comúnmente para el modelado de un dominio de aplicación, proporcionando una conceptualización del dominio específico (Gómez-Pérez y cols., 2004). Los formalismos para la representación de ontologías son bastante maduros y entre la variedad de ellos destaca Web Ontology Language (OWL) (Dean y cols., 2004), que se ha

convertido en un estándar (Dean y cols., 2012). Semantic Web Rule Language (SWRL) (Horrocks y cols., 2004), el estándar de la World Wide Web Consortium (W3C)[5] para reglas en la web semántica, fue propuesto para enriquecer OWL proporcionando una sintaxis abstracta de alto nivel para reglas. Las reglas SWRL son implicaciones entre el antecedente (*body*) y el consecuente (*head*). Así, cuando se combinan ontologías y reglas, la ontología describe los conceptos relevantes y las relaciones entre los mismos en el dominio de aplicación, y por su parte las reglas permiten formalizar la lógica de inferencia, incrementando así la cantidad de conocimiento que puede ser representado por las ontologías.

Existen numerosas herramientas para la edición y razonamiento sobre especificaciones OWL y SWRL. Entre ellas, TwoUse Toolkit [6] (Parreiras y Staab, 2010; Parreiras y cols., 2007) permite integrar tecnologías como ontología OWL y el desarrollo dirigido por modelos (MDE) . TwoUse ha sido desarrollada sobre Eclipse Modeling Project[7], lo que permite la integración directa con todas las herramientas de MDE de Eclipse. Implementa Ontology Definition Metamodel (ODM) (Object Management Group, 2009a) y proporciona un entorno gráfico para la especificación de ontologias OWL y SWRL. Gracias a que esta herramienta se proporciona como un plugin de Eclipse, otras herramientas de MDD basadas en Eclipse pueden ser aplicadas e integradas directamente, lo que nos permite proponer el diseño de un proceso dirigido por modelos basado en ontologías OWL y SWRL creadas en TwoUse Toolkit.

Esto supone una ventaja ya que la formalización y especificación de los modelos en OWL y SWRL, a diferencia de lo que ocurria con CML, ya está resuelta, por lo que el trabajo en esta sección de la tesis doctoral consistirá en diseñar las transformaciones necesarias para la generación del código Jess de la base de reglas a partir de los modelos OWL y SWRL.

Destacar que solamente una parte de OWL puede ser representada en un sistema de reglas de producción como Jess, ya que la semántica de OWL es más rica que la semántica de los sistemas de reglas de producción. Los problemas derivados de la combinación de reglas y ontologías también tienen efecto en el enfoque propuesto. Por ejemplo, la semántica de OWL y SWRL adopta la presunción del mundo abierto, mientras

[5] http://www.w3.org/Consortium/
[6] http://code.google.com/p/twouse/
[7] http://www.eclipse.org/modeling/

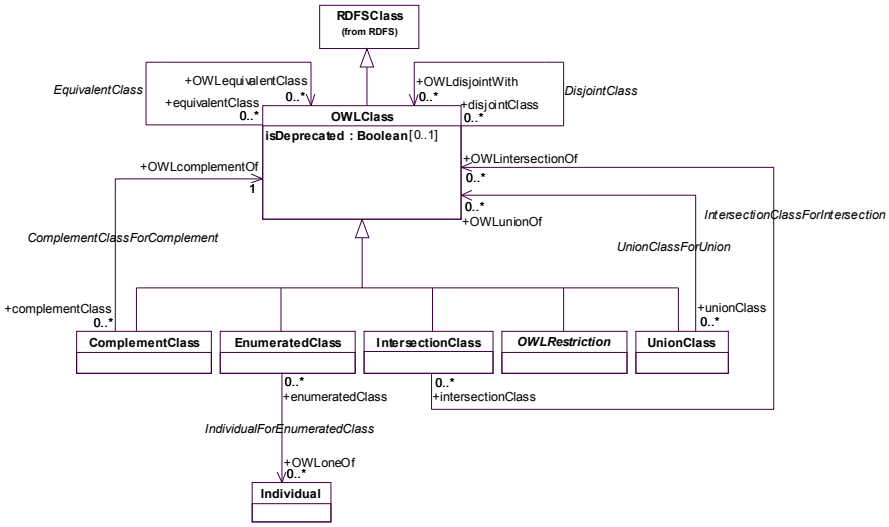


Figura 3.12: Fragmento del metamodelo OWL: *Clases OWL*

que los lenguajes de programación lógica como Jess están basados en la presunción del mundo cerrado. Como consecuencia, solo se transforman a Jess las reglas SWRL DL-Safe (Motik y cols., 2005), que son un subconjunto restringido de reglas SWRL que tiene la deseada propiedad de decibilidad, restringiendo a las reglas a operar solamente sobre individuos conocidos de la ontología OWL. Igualmente, las reglas Jess solamente operan sobre hechos de la memoria de trabajo, que contiene únicamente individuos o hechos conocidos.

La figura 3.12 muestra un fragmento del metamodelo OWL definido en ODM, en concreto la parte relacionada con las clases en OWL. Las clases OWL se representan a través de descripciones de clase. Existen seis tipos de descripciones de clase: un identificador de clase, una enumeración exhaustiva de los individuos, una restricción de propiedad, la intersección de dos o más descripciones de clase, la unión de dos o más descripciones de clase, y el complemento de una descripción de clase. Una restricción es un tipo especial de clase. Frecuentemente es anónima pero no siempre debe serlo, y representa a la clase de todos los individuos que satisfacen cierta restricción de propiedad. Existen dos tipos de restricciones de propiedad: restricciones de valor y restricciones de cardinalidad.

De forma similar, la figura 3.13 muestra el fragmento del metamodelo

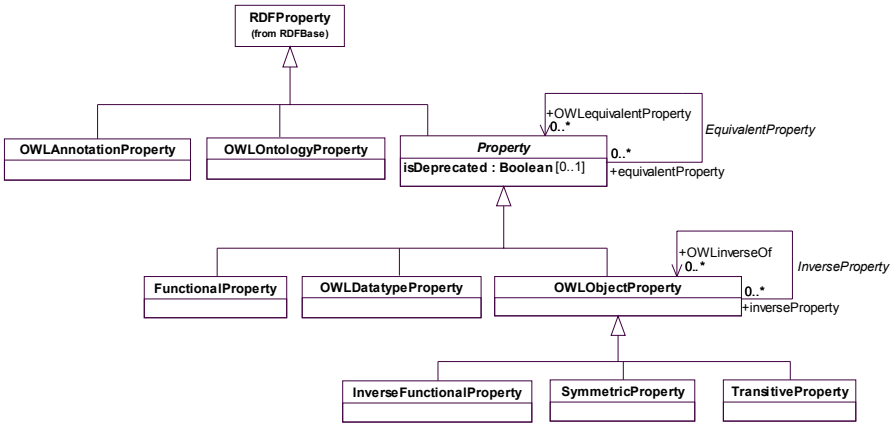


Figura 3.13: Fragmento del metamodelo OWL: *Propiedades OWL*

relacionado con las propiedades OWL. Las propiedades representan relaciones binarias en el dominio del modelado. OWL distingue dos clases de propiedades: las propiedades de objeto y las propiedades de tipo de dato. Ambos tipos de propiedades son subclases de la metaclase *Property*. Las propiedades de tipo de dato proporcionan relaciones entre instancias de clases y las instancias de los rangos de datos. Una propiedad de objeto relaciona individuos con individuos, por lo que proporcionan relaciones entre las instancias de dos clases. Así, Los dominios de las propiedades son siempre clases, mientras que el rango de una propiedad de objeto es una clase mientras que el rango de una propiedad de tipo de dato es un tipo de dato. Las propiedades pueden ser funcionales (su rango contiene como máximo un elemento). Una propiedad de objeto puede ser de tipo funcional, inversa, transitivas, simétricas o inversas a otra propiedad.

3.5.2 Ejemplo ilustrativo en OWL y SWRL

Esta sección introduce un ejemplo hecho en OWL y SWRL que sirve como ilustración para las siguientes secciones. Puesto que la semántica de OWL es más rica que la semántica de los sistemas de reglas de producción como Jess, solamente una parte de OWL puede ser representada en Jess. Para ilustrar el subconjunto de OWL soportado por la propuesta se ha empleado un ejemplo ilustrativo de relaciones familiares (Horrocks y cols., 2004). La figura 3.14 muestra las clases del ejemplo y sus relaciones, tal y como se visualizan en el editor gráfico OWL de TwoUse.

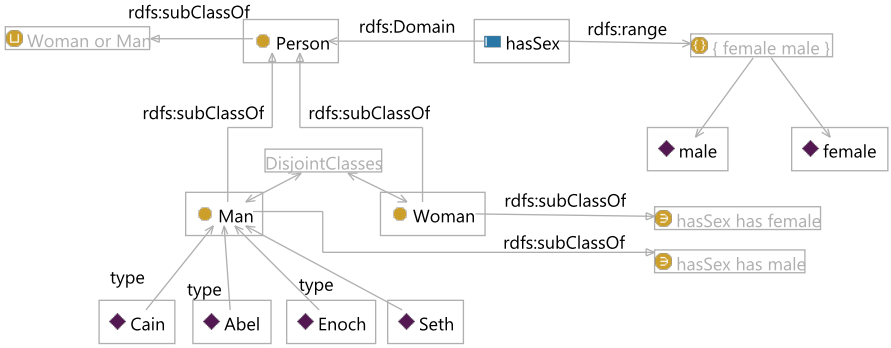


Figura 3.14: Clases del ejemplo Familia en OWL

Los conceptos del dominio se representan en OWL mediante clases con nombre (*named class*), las cuales pueden tener subclases. En el ejemplo, la clase *Person* tiene dos subclases: *Man* y *Woman*. Los individuos en OWL son instancias de una clase declarado como miembros de una clase concreta a través de la propiedad *type*. Asi, en el ejemplo, *Cain*, *Abel*, *Enoch* y *Seth* son individuos de la clase *Man*. OWL proporciona varios constructores de extensión de clases para definir clases anónimas sin nombre (*unnamed anonyomus class*). La expresión "oneOf" permite la definición de una clase enumerada mediante la lista de individuos de la clase. Un ejemplo de clase enumerada es *{female, male}*.

En OWL se distinguen dos tipos de propiedades: propiedad de tipo de dato (*datatype*), es una propiedad que relaciona una instancia de clase con un tipo de datos primitivo (por ejemplo, ingeger o string); y propiedad de objeto (*object*), es una propiedad que representa una relación entre instancias de dos clases. Cada propiedad tiene un dominio y un rango, y dependiendo del valor dado al rango, la propiedad será de un tipo u otro. En el ejemplo de la familia, la propiedad de objeto *hasSex* tiene como dominio *Person* y como rango emph{female, male}, relacionando así instancias de la clase *Person* con uno de las posibles instancias de la clase enumerada. El comportamiento predeterminado en OWL indica que una propiedad puede relacionar una instancia del dominio con múltiples instancias del rango. Para relacionar un individuo del dominio con uno y solo un individuo del rango, la propiedad debe ser definida como *functional*, como es el caso de *hasSex*.

Es posible restringir más el rango de una propiedad mediante las restricciones de propiedad: la restricción *"has-Value"* especifica clases

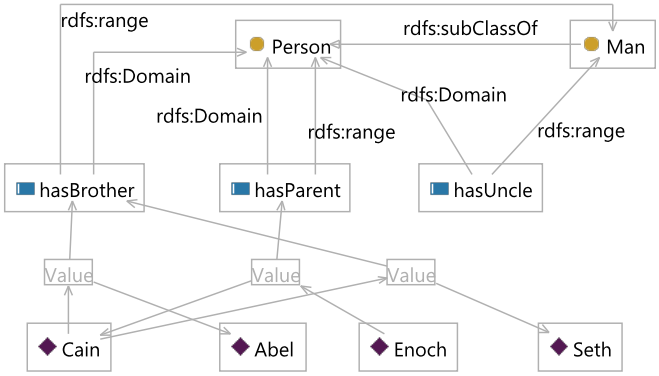


Figura 3.15: Propiedades de objeto y aserciones sobre individuos de Familia

basadas en la existencia de valores de propiedades particulares. Una restricción de propiedad puede ser tratada como una clase OWL anónima, lo que significa que es posible definir otra clase OWL como una subclase de una restricción de propiedad. En el ejemplo, *Woman* es una subclase de *"hasSex has female"*, y de forma similar, *Man* es subclase de *"hasSex has male"*.

La ontología familia define varias propiedades de objeto para representar las relaciones familiares entre individuos, como se muestra en la figura 3.15. Las propiedades de objeto *hasBrother* y *hasUncle* tienen ambas *Person* como dominio y *Man* como rango. Por su parte, la propiedad *hasParent* tiene *Person* como dominio y como rango. Estas tres propiedades son *non-functional*. Además, se definen tres aserciones de propiedad para representar las relaciones explícitas sobre los individuos concretos: *hasBrother(Cain, Abel)*, *hasBrother(Cain, Seth)* y *hasParent(Enoch, Cain)*.

3.6 Transformaciones de modelos: desde OWL y SWRL a Jess

La generación de código Jess desde ontologías OWL y SWRL se realiza mediante una cadena de transformaciones compuesta por:

- Una nueva transformación modelo a modelo que convierte elementos del metamodelo OWL y SWRL a elementos del metamodelo

Jess descrito en la sección 3.3.

- A continuación se aplica la transformación M2T ya descrita en la sección 3.4.2 que genera el código Jess a partir del modelo de reglas Jess.

La nueva transformación modelo a modelo ha sido implementada en ATL. En total está compuesta por un conjunto de 15 reglas de transformación y 10 *helpers*. La figura 3.16 muestra el listado de elementos de esta transformación, *Owl2ruleMM*, tal y como se visualizan en la vista resumen de Eclipse.

A continuación se describe la nueva transformación modelo a modelo y el resultado de su aplicación.

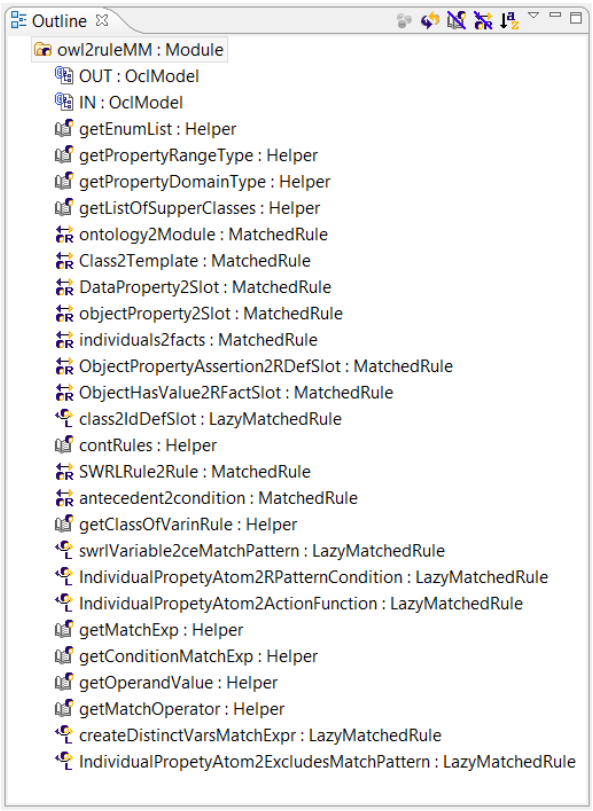


Figura 3.16: Resumen de la transformación ATL *Owl2ruleMM*

3.6.1 Clases e individuos OWL

Comenzando por la transformación de clases e individuos OWL, las reglas más importantes de esta transformación, que mapean elementos de la ontología OWL a elementos del metamodelo Jess, se describen a continuación:

- Cada clase OWL se convierte en una plantilla de hechos (template) en Jess. En Jess, una relación de subclase entre templates se representa mediante la propiedad *extends*, permitiendo únicamente herencia simple entre templates.

- Las propiedades OWL, tanto de tipo de dato como de objeto, se convierten en slots. Cada slot se añade al template correspondiente a la clase del dominio de la propiedad. Las propiedades funcionales generan slots simples (mono valuados), mientras que las propiedades no funcionales generan multislots (multi valuados).

- Cuando el rango de una propiedad es un enumerado definido a través de una clase anónima OWL de tipo *"oneOf"*, los literales de la enumeración obtenidos de los individuos asociados con la enumeracion son guardados como la lista de valores permitidos para el slot.

- una restricción de tipo *"has-Value"* se transforma al valor predeterminado de un slot.

La figura 3.17 (a) muestra el modelo Jess obtenido a partir de la ontología OWL de familia, centrándose en el mapeo desde clases y propiedades OWL a plantillas y slots de Jess. La ejecución de la transformación modelo a texto sobre este modelo Jess genera el código mostrado en la figura 3.17 (b).

Como muestra la figura 3.17, el estilo del código generado Jess atiende a un estilo orientado a objetos, que es diferente del código Jess obtenido por otras herramientas que generan código Jess a partir de OWL como Protege JessTab (Eriksson, 2003). En lugar de generar un hecho por cada propiedad en un estilo de lógica proposicional, como hace JessTab, nuestra propuesta transforma las propiedades a *slots* de *templates*, obteniendo un resultado más cercano y similar a las clases Java de la aplicación web que embeberán la base de reglas Jess, facilitando así la integración de la información entre las diferentes capas de la aplicación

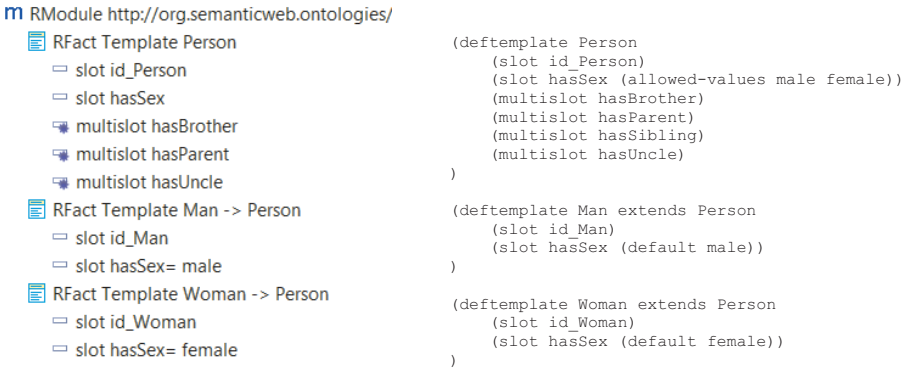


Figura 3.17: a) Modelo Jess de *templates* y b) código Jess obtenido

(Jess como motor de inferencia, clases Java que implementan la lógica de negocio y páginas web que presentan la información al usuario final).

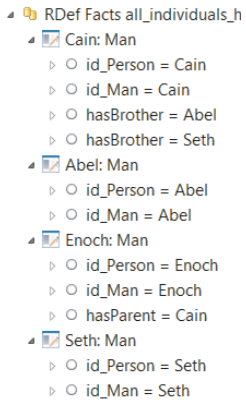
La figura 3.18 (a) muestra el modelo Jess obtenido a partir de la ontología familia, focalizándose en el mapeo desde individuos de clases con nombre OWL a definiciones de hechos en Jess (facts). Estos hechos se corresponden con todas las definiciones cargadas mediante el constructor *deffacts* que constituyen el estado inicial de la memoria de trabajo del motor de inferencia Jess:

- Cada individuo es mapeado a una definición de hecho (fact). Todos los individuos son considerados diferentes entre ellos.

- Cada aserción de propiedad genera un valor para el slot del correspondiente hecho generado a partir de la clase asociada al dominio de la propiedad. En el ejemplo, las aserciones *hasBrother(Cain, Abel)* y *hasBrother(Cain, Seth)* generan la definición del hecho *Cain* con el multislot *hasBrother* que contiene los valores *Abel* y *Seth*.

3.6.2 Reglas SWRL

Para finalizar la descripción del caso de estudio, a continuación se define un ejemplo de regla SWRL que permite inferir la relación de tío entre los miembros de familia. Esta regla se muestra en la figura 3.19 y permite derivar el valor de la propiedad *hasUncle*: un individuo de tipo persona (*x1*) que tiene otro individuo de tipo persona (*x2*) relacionado mediante la propiedad *hasParent*, y ese segundo individuo Persona (*x2*)




```
(deffacts all_individuals_http://...
"All individuals definitions

    (Man (id_Person Cain)
         (id_Man Cain)
         (hasBrother Abel Seth)
    )
    (Man (id_Person Abel)
         (id_Man Abel)
    )
    (Man (id_Person Enoch)
         (id_Man Enoch)
         (hasParent Cain)
    )
    (Man (id_Person Seth)
         (id_Man Seth)
    )
)
```

Figura 3.18: a) Modelo Jess de *facts* y b) código Jess obtenido

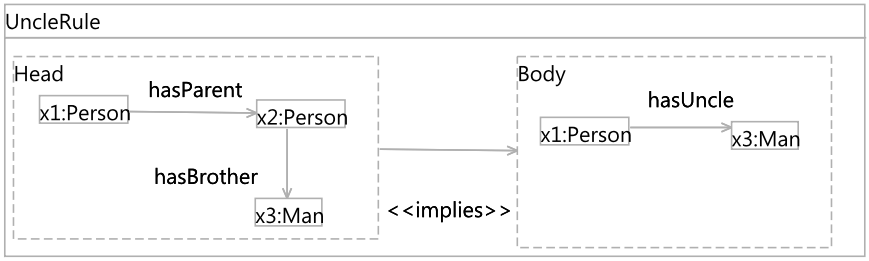


Figura 3.19: Regla SWRL que define la relación tío, *uncle*, en Familia

tiene un individuo de tipo Hombre (*x3*) relacionado mediante la propiedad hermano *hasBrother*, entonces la regla deriva una nueva aserción estableciendo la relación de tío *hasUncle* entre los individuos *x1* y *x3*.

El antecedente o cabeza de la regla, *head*, es transformado a una serie de patrones de emparejamiento, *pattern matchings*, y expresiones condicionales. Por su parte, para el consecuente o cuerpo de la regla, *body*, cada operación sobre propiedades de individuos se convierte en una función de modificación, *modify*, que actualiza slots de los hechos correspondientes a los individuos implicados.

El código resultante de la transformación de la regla SWRL a Jess es el siguiente:

```
(defrule R1_UncleRule
  (declare (salience 10))
  ?x1<-(Person (id_Person ?x1_id_Person) (hasUncle $?x1_hasUncle)
               (hasParent  $?x1_hasParent))
  ?x2<-(Person (id_Person ?x2_id_Person) (hasBrother $?x2_hasBrother))
  ?x3<-(Man    (id_Man    ?x3_id_Man))
  (test (neq  ?x1_id_Person ?x2_id_Person ?x3_id_Man ))
  (test (not (member$ ?x3_id_Man ?x1_hasUncle)))
  (test (member$ ?x2_id_Person ?x1_hasParent))
  (test (member$ ?x3_id_Man ?x2_hasBrother))
  =>
  (modify ?x1 (hasUncle (create$ ?x1_hasUncle ?x3_id_Man)))
)
```

En el ejemplo, cada variable en el antecedente de la regla, *?x1*, *?x2* y *?x3*, se ha mapeado a un patrón de emparejamiento permitiendo al motor de inferencia emparejar hechos para las variables. La tercera y cuarta expresión condicional de la regla Jess se obtiene de los *individual property atoms* del antecedente de la regla SWRL. Dos expresiones condicionales más son añadidas a la regla Jess: la primera comprueba que los tres hechos emparejados son diferentes, y la segunda es necesaria porque si ella la naturaleza monotónica de el algoritmo del motor de inferencia genera un bucle infinito cuando la acción de la regla tiene una función de modificación, como es este caso.

3.7 Conclusiones del capítulo

En este capítulo se ha presentado el proceso dirigido por modelos propuesto para la generación de una base de reglas en Jess a partir de modelos conceptuales, primero a partir de modelos CML y después a partir ontologías ligeras OWL y SWRL. Como aportaciones principales, se han diseñado los metamodelos y las transformaciones de modelos para la puesta en práctica del proceso. Para el caso de usar CML como modelos origen, se ha diseñado el metamodelo del conocimiento del dominio de CML, el cual se ha enriquecido con operadores sobre colecciones para proporcionar una mayor expresividad al lenguaje; y el metamodelo de reglas Jess. Igualmente se han diseñado e implementado las transformaciones de modelos aplicadas en el proceso: la transformación M2M desde CML al modelo de reglas Jess, y la transformación M2T desde el modelo de reglas Jess hasta código Jess. Para el caso de OWL y SWRL como modelo origen del proceso MDD, se ha descrito el uso del metamodelo de OWL y SWRL estándar ODM ya disponible en una herramienta de soporte, y la aportación principal ha sido una nueva transformación M2M desde OWL y SWRL al metamodelo de reglas Jess, lo que, reutilizando

la misma transformación M2T para el caso anterior, permite aplicar un proceso dirigido por modelos partiendo de OWL y SWRL.

Capítulo 4

Desarrollo dirigido por modelos de la arquitectura web

Este capítulo describe el proceso dirigido por modelos propuesto que, a partir de los modelos conceptuales basados en conocimiento introducidos en el capítulo 3, permite la generación de una arquitectura web en la que se integra la base de reglas. Para ello es necesario enriquecer los modelos conceptuales proporcionando mecanismos de modelado adicionales que permitan representar las características del diseño de la arquitectura web y la interfaz de usuario. Al igual que en el capítulo 3, se han propuesto dos alternativas como lenguajes de modelado independientes de plataforma: CML y OWL. Y en cada una de ellas se ha aplicado una aproximación para el modelado de las características de diseño de la aplicación web basada en reglas: la primera aproximación, aplicada al modelado CML, ha consistido en enriquecer el metamodelo con propiedades específicas de la aplicación web; y la segunda aproximación, aplicada a la propuesta de usar OWL, ha consistido en mantener la separación de aspectos y usar diferentes modelos, uno para la estructura de la información, que seria OWL, y otro para las peculiaridades de la interfaz de usuario web, lo que ha llevado a la definición de un nuevo metamodelo que complementa a OWL y que permite especificar las características de diseño de la interfaz de usuario para la arquitectura de la aplicación web basada en reglas.

Así, este capítulo se estructura en dos bloques principales: la seccio-

nes 4.2, 4.3 y 4.4 relativas al proceso dirigido por modelos que parte de CML como lenguaje de especificación de los modelos origen del proceso; y las secciones 4.5 y 4.6, relativas al uso de OWL y SWRL como modelos independientes de plataforma origen del proceso. En ambos casos, primero describen los lenguajes implicados, sus metamodelos y herramientas de modelado, y a continuación las transformaciones de modelos. Para un mejor entendimiento del proceso se emplea el mismo ejemplo ilustrativo utilizando en el capítulo 3.

4.1 Perspectiva general

En el capítulo 3 se ha descrito como generar una base de reglas en Jess a partir de los modelos conceptuales. Sin embargo, una base de reglas por si sola constituye un componente que únicamente se puede ejecutar sobre el motor de inferencia de Jess, el cual por si solo únicamente permite una interacción mediante una consola de texto. Sin embargo, utilizando la librería de clases Java que proporciona Jess, el motor de inferencia puede ser integrado en cualquier aplicación Java. Así que para que la base de reglas constituya el núcleo de una aplicación funcional y operativa, tanto la base de reglas como el motor de inferencia deben ser integrados dentro de una arquitectura de aplicación que contenga los elementos necesarios para su uso, como es una interfaz gráfica de usuario.

Tal y como se introdujo en el capítulo 1, el interés por el desarrollo de SBC en la web ha motivado la propuesta en esta tesis doctoral de una arquitectura de aplicación web. La elección de Jess como motor de inferencia condiciona a usar Java como tecnología de implementación de la aplicación web. Esta tesis doctoral no tiene como objetivo proponer

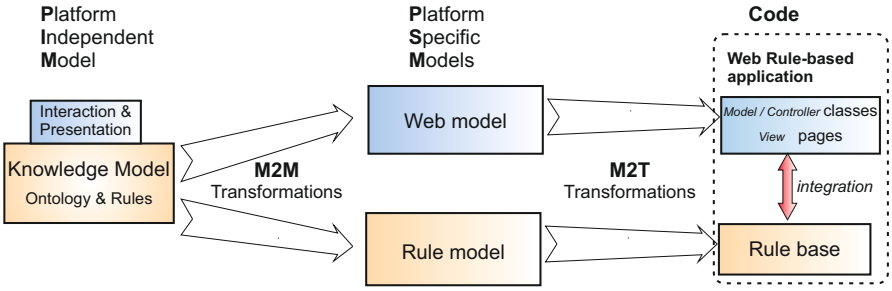


Figura 1.4: Esquema general: MDA para aplicaciones web basadas en reglas (repetición)

un formalismo para modelado web, ya que ese problema entraría dentro del ámbito de la Ingeniería Web donde hay diversas propuestas en ese sentido, aunque sí aplica ideas de Ingeniería Web dirigida por modelos para lograr el objetivo de integrar una base de reglas en una arquitectura web aplicando desarrollo dirigido por modelos.

En este capítulo se describe el segundo de los subprocesos dirigidos por modelos introducidos en la sección 1.3: el flujo *superior* en la figura 1.4, que genera la implementación de la aplicación web a partir del modelo conceptual enriquecido con el modelado de la interacción y presentación de la interfaz de usuario web.

Se han aplicado dos aproximaciones para modelar las características específicas de la aplicación web en el proceso dirigido por modelos para la generación de la aplicación web basada en reglas, según se utilice como modelo conceptual origen del proceso un modelo CML o bien un modelo OWL y SWRL.

La primera aproximación ha consistido en anotar el modelo conceptual con propiedades específicas de la aplicación web, y se ha aplicado al modelado CML, ya que se trata de un metamodelo diseñado en esta tesis a partir de su definición original, y por tanto, se ha podido modificar el metamodelo para añadirle las propiedades que permiten modelar características de diseño de aplicación web. Esto ha permitido aplicar el esquema general del proceso dirigido por modelos presentado en la figura 1.8.

Y la segunda aproximación, aplicada a la propuesta de usar OWL y SWRL como modelos conceptuales, ha consistido en aplicar la separación de aspectos (Ernst, 2003), propia de la Ingeniería Web, y usar diferentes modelos, uno para cada aspecto. Así, los modelos conceptua-

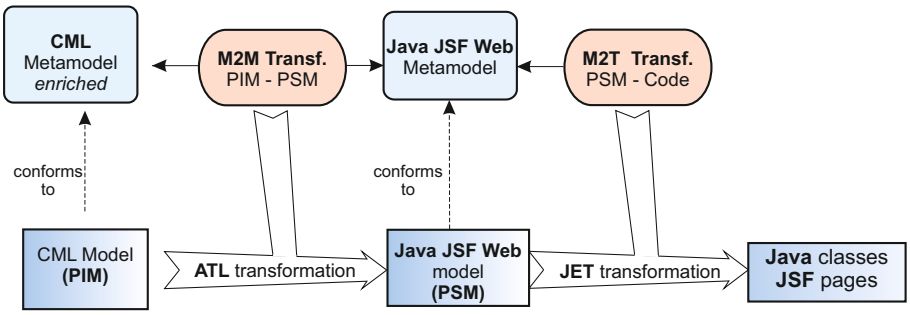


Figura 1.8: Proceso para la generación de aplicación web Java y JSF desde CML (repetición)

les OWL y SWRL sin alteración, definidos mediante es estándar ODM, han sido usados para el modelado conceptual que define la estructura de la información, y se ha añadido a la propuesta el modelado de diseño de la interfaz de usuario para aplicaciones web basadas en reglas, mediante la definición de un metamodelo específico e independiente para la representación de ese aspecto. Esto ha permitido aplicar el esquema general del proceso dirigido por modelos presentado en la figura 1.9.

Una vez aportado el modelado de características de la interfaz de usuario web de una aplicación basada en reglas, el siguiente paso de la propuesta dirigida por modelos consiste en la transformación a un modelo específico de plataforma mediante una transformación M2M. En nuestro caso la plataforma de implementación elegida ha sido una arquitectura de aplicación web basada en tecnología Java, en concreto en el framework JavaServer Faces (JSF), y para ello se ha propuesto un metamodelo para arquitectura web basada en JSF, descrito en la sección 4.3. Dependiendo del lenguaje de modelado origen del proceso, se utiliza una de las dos transformaciones M2M propuestas: una desde CML a modelado web basado en JSF; y otra desde OWL y SWRL a modelado web basado en JSF. Estas transformaciones de modelos se describen en las secciones 4.4 y 4.6, respectivamente.

Tras obtener el modelo web basado en JSF, una transformación modelo a texto (M2T) permite la obtención del código fuente para la aplicación web que integra la base de reglas.

La aplicación web obtenida mediante el proceso se describe mediante la arquitectura presentada en la sección 1.1.6, figura 1.3. Integra el motor de inferencia Jess que ejecuta la base de reglas obtenida mediante el proceso presentado en el capítulo 3. Las clases Java de lógica de negocio

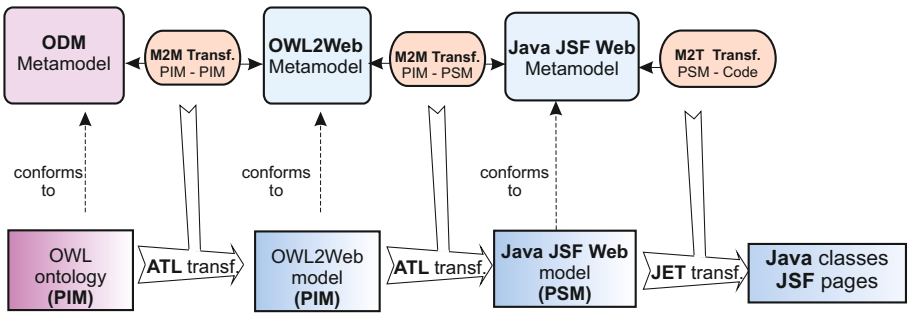


Figura 1.9: Proceso para la generación de aplicación web Java y JSF desde OWL (repetición)

implementan dicha integración. Para la visualización de la información e interacción del usuario, se utilizan páginas basadas en el framework JavaServer Faces con características de Aplicaciones de Internet Ricas (RIA). De esta manera, como una prueba de concepto de aplicación web basada en reglas, se consigue obtener un ejemplo de aplicación web funcional que implementa las operaciones básicas de creación, listado, actualización y eliminación (CRUD de sus siglas en inglés: Create, Read, Update, Delete) enriquecidas con la lógica que ofrece la base de reglas. Al contrario de las herramientas actuales para la generación automática de sistemas CRUD que realizar estas funciones sobre bases de datos relacionales, nuestra contribución es que estas funciones son ejecutadas sobre la memoria de trabajo del motor de inferencia, permitiendo la ejecución del mecanismo de inferencia de encadenamiento hacia delante (forward-chaining) que dirige el proceso de razonamiento.

Es decir, no se trata de una aplicación web CRUD convencional ya que algunos valores son derivados por el motor de inferencia, consiguiendo así ofrecer una funcionalidad propia de los SBC. Es por ello que ha sido necesario proponer un modelado de características específicas para la aplicación web basada en reglas, ya que esta tiene peculiaridades de los sistemas basados en reglas no presentes en sistemas convencionales.

Para ilustrar los procesos de este capítulo se emplea el mismo ejemplo ilustrativo de la familia, introducido en el capítulo 3.

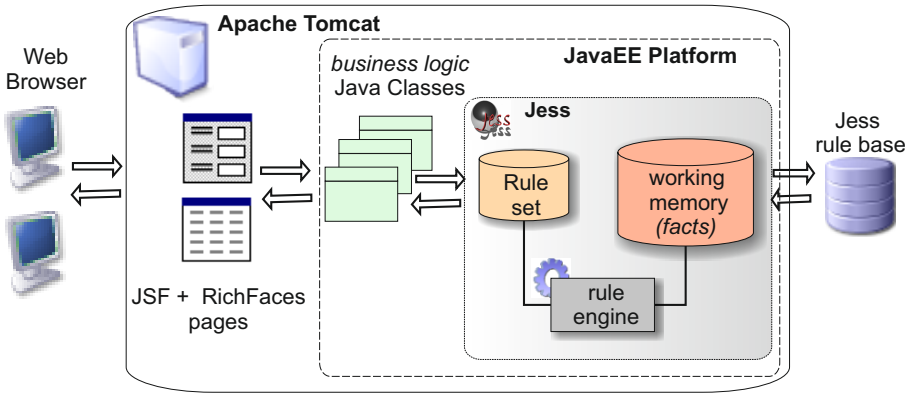


Figura 1.3: Arquitectura de una aplicación web basada en reglas (repetición)

4.2 Modelado de la interfaz de usuario sobre CML

Esta sección describe como se ha enriquecido el metamodelo de CML para permitir la especificación de características de modelado web. Nuestra propuesta consiste en anotar el modelo conceptual con propiedades específicas para el diseño web, como es la característica que permite modelar si un atributo de un concepto es editable o no por el usuario a través de la interfaz web, o si el valor de un atributo de un concepto es derivado o no por el motor de inferencia. Podemos generalizar y afirmar que estas características de diseño de interfaz de usuario son aplicables no solo a interfaces web, sino a cualquier interfaz de usuario de una aplicación basada en reglas. Por tanto, hemos enriquecido el modelado CML con estas propiedades que hacen referencia a la interfaz de usuario.

Nuestra propuesta consiste en enriquecer el modelado CML para que permita la representación de propiedades del diseño de la interfaz de usuario para aplicaciones web basadas en reglas (y en general, para aplicaciones basadas en reglas), que se describen a continuación como características agrupadas en propiedades de interacción y de presentación.

4.2.1 Modelado de la interacción

El modelado de características de interacción permite la especificación de diversas propiedades de la interactividad del usuario con la capa de presentación. Estas propiedades están asociadas a varias primitivas del metamodelo de CML, como son las metaclases *Atrituto* (ver secciones 3.2.2 y A.2.1), y *Concepto* (ver secciones 3.2.2 y A.2.1).

Las siguientes propiedades se añaden a la metaclase *Atributo* , para enriquecer el modelo con características de interacción relacionadas con un atributo de una clase:

- ***isDerived****: Boolean*: toma valor *true* cuando el valor de atributo es derivado por el motor de inferencia tras la ejecución de una o varias reglas, por lo que dicho valor no puede ser modificado por el usuario (debe aparecer desactivado para evitar su edición en la interfaz de usuario).

- ***notifiesTo****: Attribute [0..*]*: establece la lista de atributos que deben ser refrescados o actualizados en la interfaz de usuario cuando

el valor del atributo actual cambie. Esta actualización o refresco se debe a que los valores de los atributos contenidos en la lista *notifiesTo* serán derivados por el motor de inferencia cuando cambie el valor del atributo actual.

Por ejemplo, la clase *Persona* tiene los atributos *edad:Entero* y *esAdulto:booleano*. El atributo *esAdulto* va a ser un valor derivado (*isDerived = true*) a partir de la *edad* puesto que existe un par de reglas que infieren su valor a partir del valor de edad. Ese par de reglas serían las siguientes:

```
x : Persona;
IF (x.edad >= 18) THEN (x.esAdulto = true )
IF (x.edad <  18) THEN (x.esAdulto = false)
```

En este ejemplo, la propiedad *notifiesTo* del atributo *edad* nos permite especificar la interacción que debe implementar la interfaz de usuario, indicando que *edad* notifica (*notifiesTo*) a *esAdulto*, ya que al actualizar el valor de *edad*, bien mediante una interacción del usuario con la interfaz o bien a través de una actualización de la *edad* como consecuencia de la ejecución de otra regla, el valor de *esAdulto* deberá ser refrescado en la presentación del usuario para que se visualice el nuevo valor derivado por el motor de inferencia al disparar la regla que infiere su valor.

Aunque este tipo de interacción puede ser adecuada en distintos tipos de interfaces de usuario en distintas plataformas, es especialmente adecuada para aplicaciones web ricas que emplean AJAX para la actualización de los contenidos de la vista de usuario sin necesidad de comunicación completa entre el cliente y el servidor. Por tanto, para nuestra aproximación este tipo de interacción es una aportación fundamental.

- ***isNotifiedBy****: Attribute [0..*]*: la relación inversa a *notifiesTo*. En el ejemplo anterior, *esAdulto isNotifiedBy edad*.

- ***valueFrom****: Attribute [0..1]*: permite establecer el atributo que contiene la lista de valores que pueden ser elegidos como valor para el atributo actual.

  Por ejemplo, supongamos que la clase *Persona* tiene los atributos *amigos: Persona[*]* y *mejorAmigo: Persona*. Para indicar que el

valor de *mejorAmigo* debe ser un elemento de la lista *amigos*, en *mejorAmigo* se especifica *valueFrom* = *Persona.amigos*.

En el ejemplo de la familia, la propiedad *isDerived* es *true* para todos los atributos de *Person* cuyos valores se infieren mediante reglas, como son *siblings*, *brothers*, *sisters*, *children*, *sons*, *daughters*, *uncles*, *aunts*, etc. La propiedad *notifiesTo* del atributo *father* contiene a todos los atributos que se ven afectados con un cambio en el valor de *father*, como son *siblings*, *brothers*, *sisters*, *uncles* y *aunts*. La propiedad *isNotifiedBy* del atributo *uncles* y *aunts* contiene los atributos que implican una actualización de sus valores, que son *father* y *mother*, ya que un cambio de padre o madre implica que los tíos y tías deben actualizarse. Por último, si se añade a *Person* un nuevo atributo *favoriteUncle* que permita seleccionar uno de los tíos como favorito, *favoriteUncle* toma su valor de *uncles*, y esto se especifica en la propiedad *valueFrom*.

La Fig. 4.1 muestra un ejemplo de como se representa la interacción del atributo *father* con los atributos *parents* y *siblings* mediante las propiedades *isDerived* y *notifiesTo*, y cómo esto debe afectar a la interfaz web resultante, en concreto a la vista de usuario que permite editar las instancias de la clase *Person*. Los atributos *parents* y *siblings* son atributos derivados así que sus valores no podrán ser editados por el usuario

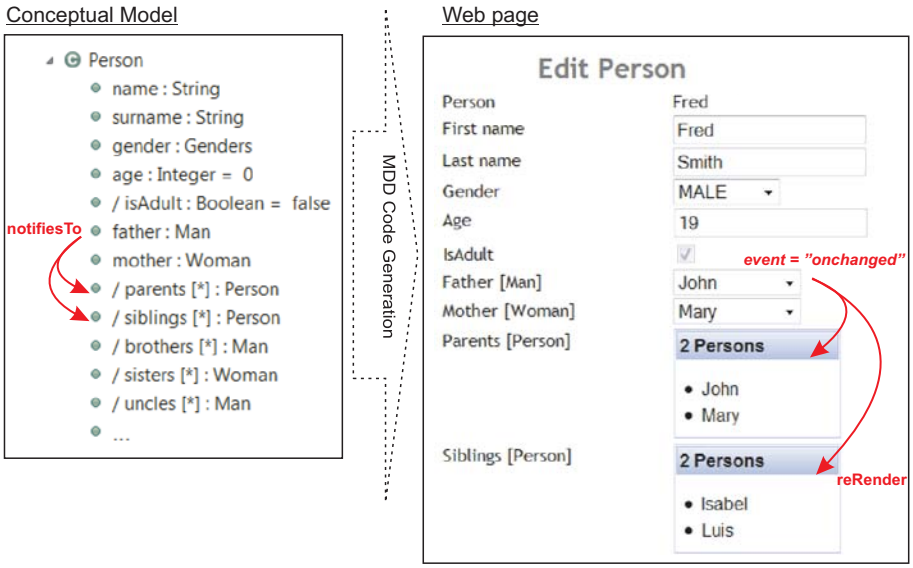


Figura 4.1: Modelado de la interacción con la propiedad *NotifiesTo*

sino que serán deducidos por el motor de inferencia. En el editor de modelos CML, los atributos derivados se identifican por estar precedidos por el carácter "/". Además, *father* notifica a *parents* y *siblings*, así que cuando el valor de *father* cambie por una acción del usuario el motor de inferencia debe ejecutarse y tras ello la lista de valores contenidos en *parents* y *siblings* debe ser refrescada, actualizando así sus valores con los nuevos valores deducidos por el motor de inferencia. Desde el punto de vista de la implementación en el framework usado, esto se consigue mediante el evento *onChanged* que detecta un cambio en el valor de un campo de la interfaz de usuario, en este caso *father*, y la acción *reRender* de AJAX asociada a los campos que deben actualizarse.

4.2.2 Modelado de la presentación

Las características del modelado de presentación permiten especificar propiedades de visualización de los elementos del modelo conceptual, permitiendo la personalización de interfaz de usuario. Estas propiedades están asociadas a varias primitivas del metamodelo de CML, fundamentalmente a las metaclases *Concepto* (ver sección A.2.1) y *Atrituto* (ver sección A.2.1).

Por ejemplo, hace posible seleccionar que conceptos se muestran en el componente menú de la aplicación, y que atributos se incluyen como columnas en las tablas que permiten visualizar el listado de todas las instancias de un concepto o clase.

Las siguientes propiedades se añaden a la metaclase *Concepto* (ver tabla A.5, son atributos cuya descripción comienza por la etiqueta *[UI]*), para enriquecer el modelo con características de presentación.

- ***showInMenu**: Boolean*: permite indicar si el nombre de concepto (o clase) va a aparecer en el menú de la aplicación, lo que permitirá acceder a sus instancias y realizar las operaciones predefinidas para las instancias de ese concepto o clase. Para ocultar el concepto y que no aparezca en el menú, esta propiedad se establece a falso.

Las siguientes propiedades se añaden a la metaclase *Atributo* (ver tabla A.2), para enriquecer el modelo con características de presentación para los atributos.

- ***key**: Boolean*: permite indicar el o los atributos que van a ser el valor clave para cada instancia de la clase, y que se van a utilizar como identificador en la interfaz de usuario. El valor de esos

atributos marcados como clave se almacenarán concatenados en el atributo *id_NombreDelConcepto* que se crea de forma predeterminada para cada concepto y que sirve como clave primaria.

Por ejemplo, para una *persona* podemos considerar clave su *nombre*. Al especificar la propiedad *key* de ese atributo como verdadero, los componentes de la interfaz de usuario que muestren las instancias de tipo *persona*, como son menús desplegables, listados de elementos relacionados, etc, visualizaran el valor del atributo *nombre* para identificar las instancias de *persona* existentes o relacionadas en la aplicación (ver Fig. 4.2).

- ***label**: String*: Etiqueta asociada al atributo que se mostrará en la interfaz de usuario. Por ejemplo, en la Fig. 4.2 se puede apreciar que los atributos *name* y *surname* se muestran en la vista web con

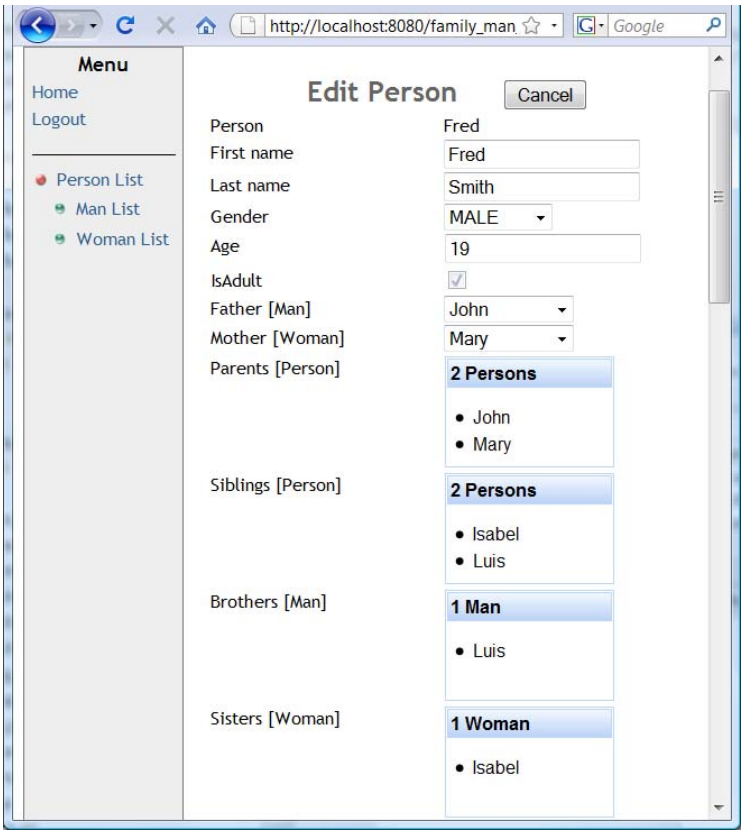


Figura 4.2: Aplicación web de Familia: edición de instancia de Persona

las etiquetas *First name* y *Last name*, respectivamente, ya que esos son los valores especificados en la propiedad *label* para cada uno de ellos.

- ***showRelationType**: Boolean*: permite indicar si en la vista se muestra el tipo del atributo junto a la etiqueta del nombre del atributo o no se muestra. Esta característica se tiene en cuenta solamente a los atributos que representan relaciones entre conceptos. Por ejemplo, en la Fig. 4.2 los atributos *father*, *mother*, *parents*, *siblings*, *brothers* y *sisters* tiene esta propiedad activada (*true*) por ello junto al nombre del atributo se muestra el tipo entre corchetes.

- ***showColumn**: Boolean*: permite indicar si el atributo se va a visualizar o no como una columna de la tabla que muestra el listado de todas las instancias de la clase a la que pertenece el atributo.

En el ejemplo de la familia, hemos especificado la propiedad *showInMenu* a *true* para los tres conceptos del modelo, *Person*, *Man* y *Woman*, por lo que los tres aparecen en el menú de la aplicación (ver Fig. 4.3). Así mismo, la Fig. 4.3 muestra el resultado de marcar la propiedad *showColumn* a *true* para los atributos *idPerson*, *gender*, *age*, *isAdult*, *father*, *mother* y *siblings* del concepto *Person*, visualizandose dichos atributos en el listado de personas. Otros atributos como *brothers* y *sisters* tienen esa propiedad desactivada por lo que no se muestran en el listado.

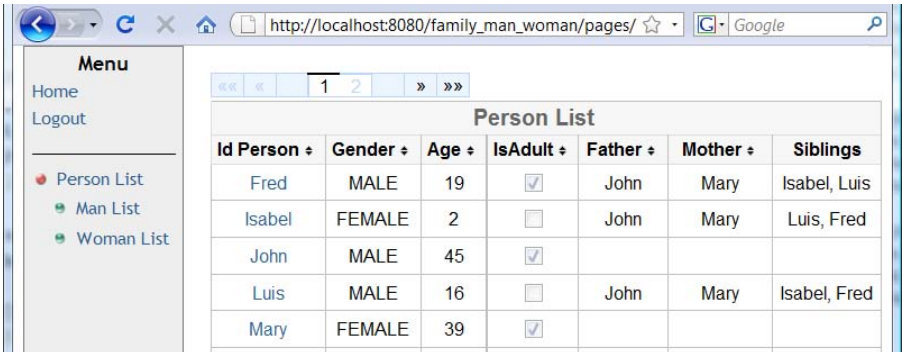


Figura 4.3: Aplicación web de Familia: listado de Personas

4.3 Modelado de la arquitectura web para Java JSF

Esta sección describe el metamodelo especifico de plataforma para representar una arquitectura web basada en Java JSF diseñado en esta tesis para aplicar la propuesta MDD. Se ha denominado *JSFWebmm* .

La aplicación web generada aplicando el proceso dirigido por modelos se divide en dos capas: la capa de negocio y la interfaz de usuario. Se ha optado por obviar la tercera capa habitual en una arquitectura web de 3 capas, la capa de persistencia, por ser un problema ya resuelto en la bibliografía (Ceri y cols., 2002a,b).

La capa de negocio está compuesta por:

- el paquete de clases Java, los JavaBeans, obtenido a partir de los conceptos del modelo de entrada. Estos JavaBeans constituyen el modelo en la arquitectura MVC.
- un conjunto de clases Java de soporte necesarios para implementar diversas funcionalidades. Estas clases
- y una clase denominada *JessEngineBean* que integra las reglas Jess en la aplicación web usando la librería Jess API para llamar a el motor de inferencia de Jess. Esta clase también se enmcarca en el controlador de la arquitectura MVC.

Por su parte, la capa de interfaz de usuario, correspondiente a la vista en la arquitectura MVC, está compuesta por una lista de páginas web basadas en JSF que permiten al usuario interactuar con el sistema y realizar las funcionalidades predeterminadas.

Para conseguir la generación de la aplicación web que implemente esta integración con Jess, se ha diseñado el metamodelo de arquitectura web JSF que se muestra en la figura 4.4. Este metamodelo representa los componentes de la aplicación web JSF, compuesta de un conjunto de páginas web JSP (el formato de base para JSF), un archivo de configuración (*faces-config.xml*), un archivo de mensajes que guarda los valores de texto que se se visualizarán en las páginas web y un conjunto de clases Java tipo JavaBean agrupadas en paquetes. La descripción detallada del metamodelo se encuentra en el Apéndice C.

Este metamodelo es la base para el diseño de las transformaciones que permiten obtener los artefactos de la arquitectura web (código) a partir de los modelos origen.

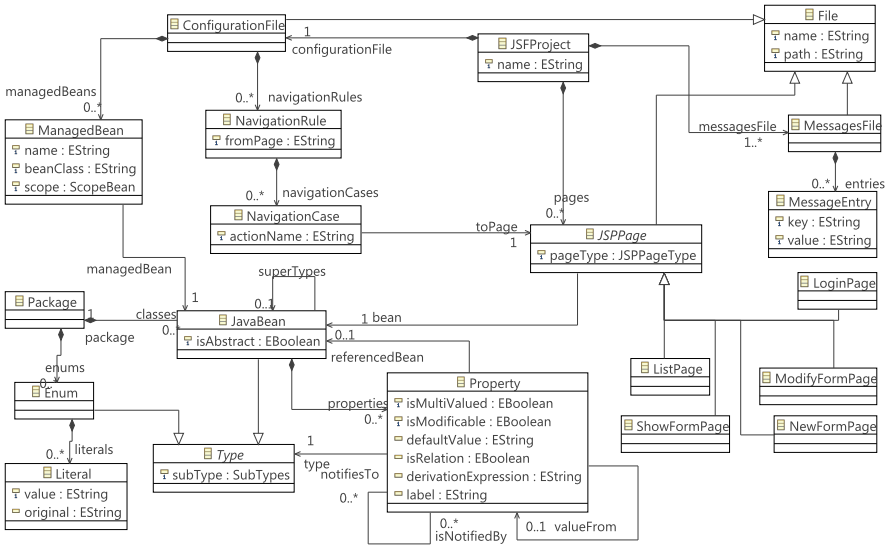


Figura 4.4: *JSFWebmm*: Metamodelo de la arquitectura Web JSF

4.4 Transformaciones de modelos desde CML a la arquitectura web

En esta sección se describe cómo se han diseñado las transformaciones de modelos que permiten obtener un modelo de arquitectura web JSF a partir de los modelos origen CML enriquecidos.

4.4.1 Transformación Modelo a Modelo

El modelo de arquitectura web basado en JSFWebmm es el destino la transformación M2M que parte de CML enriquecido. En esta transformación de modelos conceptuales origen CML a modelo JSFWebmm, cada concepto CML se mapea en varios elementos: una clase JavaBean, una página JSF para la creación y edición de instancias de ese concepto, una página JSF para el listado de todas las instancias, y un elemento bean gestionado (managed bean) para ser incluido en el archivo de configuración.

Así mismo, las características de interacción y presentación que enriquecen a los modelos originales CML son tenidas en cuenta en este nivel de transformaciones dentro del proceso dirigido por modelos, para conseguir así que el resultado final contemple las características añadidas

para modelar la interfaz de usuario.

La transformación ha sido implementada en ATL. En total está compuesta por un conjunto de 9 reglas de transformación que fundamentalmente convierten los elementos del esquema de dominio de CML (conceptos, atributos, valores enumerados y relaciones) en uno o varios elementos del metamodelo JSFWebmm, y 6 *helpers* con funciones necesarias en la transformación. La figura 4.5 muestra el listado de componentes de la transformación *CML2JavaBeans* tal y como se visualizan en la vista resumen de Eclipse.

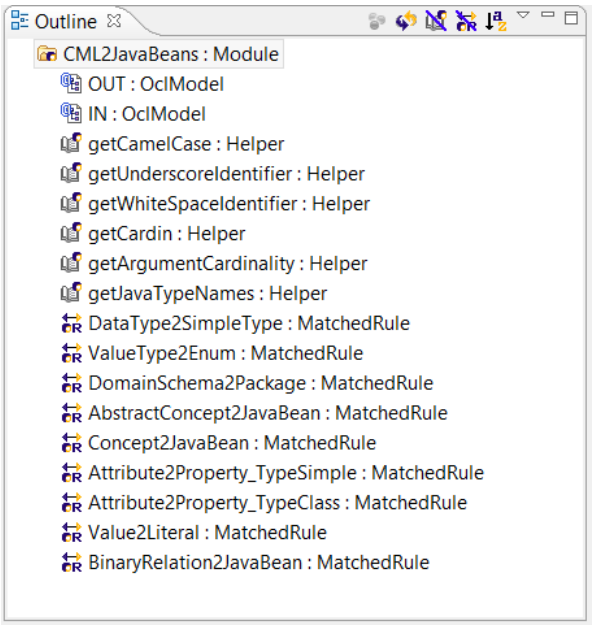


Figura 4.5: Resumen de la transformación ATL *CML2JavaBeans*

Un fragmento del modelo conforme con el metamodelo JSFWebmm resultante de la aplicación de la transformación al ejemplo de la *Familia* presentado en la figura 3.6 se muestra en la figura 4.6.

4.4.2 Transformación Modelo a Texto

Como último paso del proceso MDD, una transformación modelo a texto procesa cada elemento del modelo de arquitectura web generando el código correspondiente para cada uno de los componentes de la arquitectura, y obteniendo así como resultado final a aplicación web

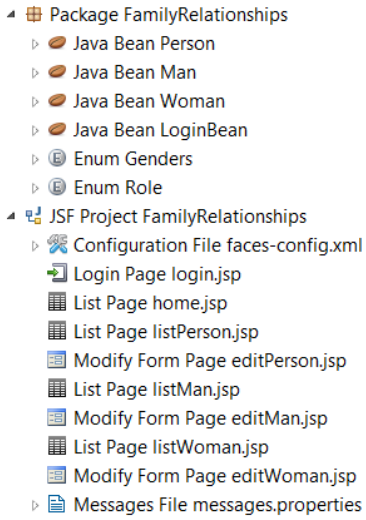


Figura 4.6: Fragmento del modelo basado en *JSFWebmm* del ejemplo Familia

deseada. Las figuras 4.2 y 4.3 son ejemplos de la aplicación resultante. La transformación implementada con JET genera el código fuente de interfaz de usuario utilizando los componentes del framework para RIAs predeterminado, en nuestro caso RichFaces. En importante destacar que varios elementos como definiciones de estilos CSS, fuentes de texto, iconos, etc., son incorporados al resultado final de forma predeterminada, ya que dichos elementos no dependen del modelo de presentación ni de la ontología.

4.5 Modelado de la interfaz de usuario para OWL

A continuacion se describe el caso de utilizar ontologías OWL como modelos de dominio origen en el proceso de desarrollo dirigido por modelos que genera la arquitectura web de la aplicación basada en reglas. Como se comentó en la introducción de este capítulo, el modelado de las características de la interfaz de usuario en este caso implica la necesidad de utilizar un modelo independiente que complemente a OWL, siguiendo así el enfoque de separación de aspectos aplicado en otras disciplinas como la Ingeniería Web (Rossi y cols., 2008). El objetivo es obtener, a partir de la ontología del dominio definida en OWL, los elementos de la

arquitectura web y la interfaz de usuario en un framework de componentes reutilizables para RIAs.

Este enfoque se basa en la presunción de que la interfaz de usuario se puede obtener a partir de la ontología de clases, propiedades y aserciones en OWL. Sin embargo, puesto que la semántica de OWL es más rica (y distinta) a la semántica de los componentes de una interfaz de usuario, solamente una parte de la ontología OWL puede ser trasladada a elementos de la interfaz de usuario. Para mejorar la generación de interfaz de usuario se añade un *modelo de presentación*, que está fuertemente acoplado a la ontología, y que proporciona el mecanismo para especificar características propias del diseño de la interfaz de usuario web.

Una primera versión del modelo de presentación, basada en el metamodelo *OWL2Web* diseñado en esta tesis y que se describe en la siguiente sección, es generada a partir de la ontología, lo que permite partir de un modelo inicial sobre el que modificar y/o añadir las características deseables para la interfaz. El proceso de generación de una versión inicial del modelo de presentación a partir de la ontología se describe más adelante en la sección 4.5.1. Este modelo de presentación inicial se crea con características predeterminadas, sirviendo como punto de partida y puede ser modificado por el diseñador, de manera que se especifiquen o se modifiquen manualmente las características deseadas para la presentación e interacción en la interfaz de usuario.

4.5.1 Metamodelo de presentación web para OWL

En esa sección se describe el metamodelo de presentación propuesto para complementar OWL que permite la especificación de las características propias del diseño de la interfaz de usuario en aplicaciones web ricas basadas en reglas. La especificación detallada de dicho metamodelo, *OWL2Web*, se encuentra en el Apéndice D.

Para conseguir la generación de una interfaz de usuario potente, el modelo de presentación para OWL captura las características de los componentes de la interfaz de usuario usado en aplicaciones web para especificar, por ejemplo en que orden deben aparecer, su aspecto visual y disposición dentro de la vista, etc. Básicamente, se proponen tres tipos principales de elementos de presentación en la interfaz de usuario de aplicaciones web basadas en reglas:

- Un *menu*, incluyendo una jerarquía o árbol con la taxonomía de

clases.

- Una página *list* (lista) para cada clase de la ontología, que permita listar todas las instancias de la clase.

- Una página *form* (formulario) para cada clase de la ontología, que permita visualizar y editar las instancias individuales.

La Figura 4.7 muestra el metamodelo *OWL2Web*. En él se definen las primitivas que pueden ser usadas en los modelos de presentación. Las metaclases están diseñadas para permitir la extensión de los modelos de presentación mediante la incorporación de nuevas características o modificando las existentes, permitiendo así la evolución del proceso propuesto. Destacan los siguientes elementos:

- Una *Class* está relacionada con un *MenuItem* que almacena la(s) característica(s) para mostrar la clase en el menú de la aplicación, y con una *TableList* que almacena la(s) característica(s) para mostrar la clase en la página de listado. La tabla 4.1 incluye la descripción detallada de la clase *Class* y sus clases relacionadas *MenuItem* y *TableList*.

- De forma similar, una *Property* está relacionada con un *FormField* el cual almacena las características para mostrar la propiedad como una campo de la página de formulario, y con *ListColumn* que almacena las características para mostrar la propiedad como una columna dentro de la página de listado. La tabla 4.2 incluye la descripción detallada de la clase *Property* y sus clases relacionadas *FormField* y *ListColumn*.

El uso este modelo de presentación para la generación de los elementos de la interfaz de usuario será descrito más adelante en la sección 4.6.

4.6 Transformaciones de modelos desde OWL a la arquitectura web

La Figura 4.8 muestra el esquema dirigido por modelos propuesto para obtener la interfaz de usuario web a partir de una ontología OWL. El proceso parte de la ontología OWL del dominio del problema. En un primer paso, a partir de la ontología OWL se deriva un modelo de

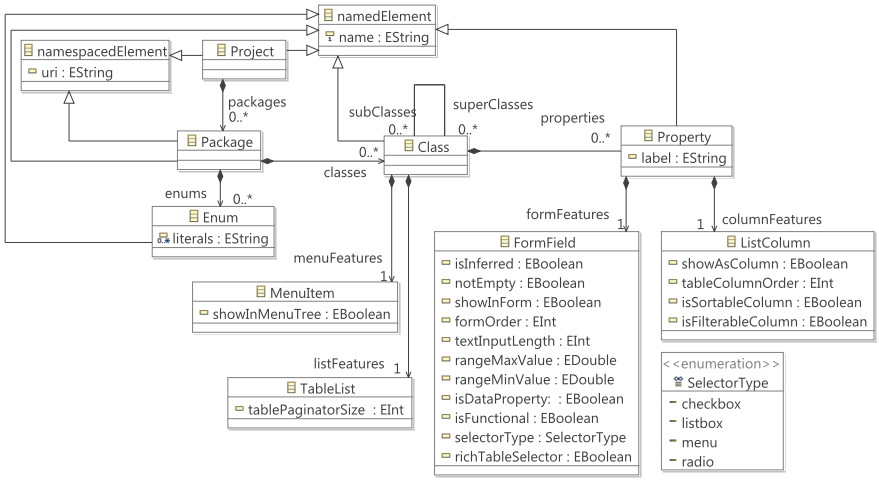


Figura 4.7: Metamodelo de Presentación *OWL2Web*

presentación conforme al metamodelo *OWL2Web* con valores predeterminados, ejecutando una transformación modelo a modelo (M2M) desde OWL al modelo de presentación propuesto en la sección 4.5.1. Ciertas propiedades del modelo de presentación no se pueden obtener de la ontología OWL, ya que la semántica de OWL no está diseñada para representar esas propiedades, por lo que se inicializan con valores predeterminados tal y como se ha muestra en las tablas 4.1 y 4.2 (ver columna *valor predeterminado*). A continuación, el desarrollador puede modificar y personalizar el modelo de presentación para mejorar la generación de la interfaz de usuario. Tras ello se ejecuta una segunda transformación M2M que usa dos modelos de entrada: la ontologia OWL y el modelo de presentación conforme a *OWL2Web* obtenido en el paso anterior. El resultado de esta segunda transformación M2M es un modelo basado en el metamodelo Java JSF presentado en la sección 4.3. Por último, la transformación M2T descrita anteriormente en la sección 4.4.2 genera el código fuente de interfaz de usuario utilizando los componentes del framework para RIAs predeterminado.

Es importante aclarar que el modelo de arquitectura web basado en JSFWebmm es el destino común para las dos transformaciones que parten bien de CML (ver sección 4.4) o bien de OWL y *OWL2Web*. Por tanto, se trata de dos transformaciones distintas, una para cada uno de los lenguajes de modelado origen. Por su parte, la transformación de modelo JSFWebmm a código para obtener los artefactos finales, es

Tabla 4.1: Metamodelo de Presentación *OWL2Web*: Class

Metaclase	*Característica*	*Tipo*	*Usado en Página*	*Descripción*	*Valor predet.*
Class	*name*	string	all	nombre de la clase	OWL2
	subClasses	Class[0..*]	menu	colección de subclases	OWL2
	superClasses	Class[0..*]	menu	colección de superclases	OWL2
	properties	Property[0..*]	form, list	colección de propiedades que tienen la calse como dominio	OWL2
	menuFeatures	MenuItem	menu	enlace a las caracterísitcas de presentacion para la calse en la página menú	
	listFeatures	TableList	list	enlace a las características de presentación para la case la página de listado	
MenuItem	*showInMenuTree*	boolean	menu	true si la clase es mostrada en el menú (árbol)	true
TableList	*tablePaginatorSize*	integer	list	número de filas a visualizar a la vez en la página de listado	10

única.

Las tecnologías de soporte empleadas para implementar la cadena de transformaciones has sido las mismas que se han utilizado hasta ahora en esta tesis: Eclipse Modeling Project como base, la herramienta TwoUse Toolkit para la creación de ontologías OWL; Ecore para la definición del metamodelo y el editor de modelos *OWL2Web*, ATL para las transformaciones M2M y JET para las transformaciones M2T.

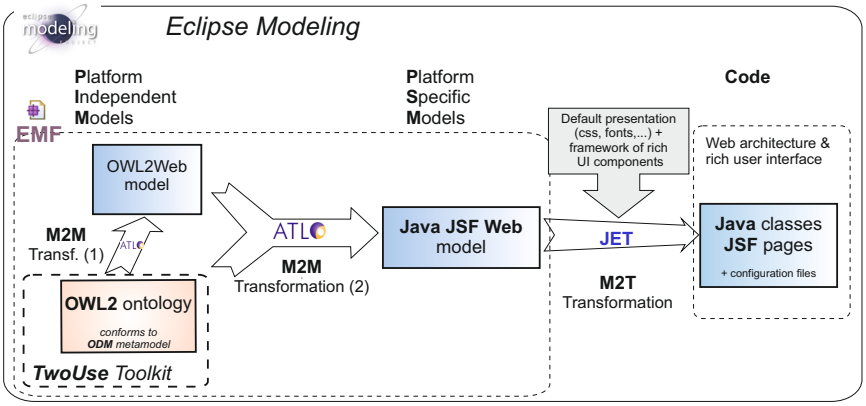


Figura 4.8: MDD para generar la arquitetura web desde OWL

4.6.1 Clases y propiedades OWL

Para la descripción de la transformación a continuación se indica cómo se obtienen los principales elementos de la interfaz de usuario generada a partir de OWL y el modelo del modelo de presentación intermedio:

Menú tipo árbol. Los conceptos del dominio se representan en OWL como clases con nombre (named classes), pueden tener subclases y superclases, formando así una jerarquía de clases que define la taxonomía del dominio. Para mostrar una jerarquía así en la interfaz de usuario se utiliza un menú en forma de árbol. Para hacer eso, RichFaces proporciona el componente *rich:tree* que muestra un control de tipo árbol en la vista de usuario. Para que la clase se muestre en el menú, la propiedad *showInMenuTree* del objeto *MenuItem* asociado a cada clase debe ser *true*. De forma predeterminada, todas las clases se mostrarán en el menú de la aplicación generada.

Página de formulario. Para visualizar y editar las instancias de una clase concreta con todas sus propiedades se utiliza una página de formulario rico que contiene campos para todas las propiedades de la clase. Tal y como se introdujo en la sección 3.5, en OWL se distingue entre propiedades de tipo de dato y propiedades objeto. Así, dependiendo del rango de cada propiedad, se utilizará el tipo de campo de formulario que sea más adecuado para ese tipo:

- un campo de entrada de texto para tipos de datos primitivos como números o cadenas de caracteres,
- un campo de selección entre varias opciones (verdadero o falso) cuando se trate de un tipo de dato lógico o booleano,
- un campo que permita seleccionar uno o varios valores de una lista cuando se trate de una propiedad de objeto, etc.

El comportamiento predeterminado en OWL determina que una propiedad puede relacionar una instancia del dominio con múltiples instancias del rango. Sin embargo, una propiedad definida como *functional* relaciona una instancia del dominio solamente con un valor del rango. De igual manera, una restricción de cardinalidad permite establecer el rango de una propiedad a valor 1.

Además, una propiedad puede no aparecer en la página de formulario de la clase cambiando a *false* la característica *showInForm* asociada a la propiedad de la clase en el modelo de presentación.

ID	First Name	Last Name	Phone
140	Joshua	Brown	555-4579
150	Christopher	Brown	555-4580
160	Matthew	Brown	555-4581
170	Ryan	Jones	555-4582
180	Jason	Jones	555-4583
200	Kevin	Jones	555-4584
220	Daniel	Jones	555-4585
210	Matthew	Jones	555-4586

1 2 3 4

Figura 4.9: Tabla con paginación (3 filas seleccionadas, fondo gris)

De esta forma, dependiendo de si una propiedad es funcional o no (o de cardinalidad 1), la generación de los campos del formulario correspondiente a cada propiedad implica que: para propiedades no funcionales se utilizan widgets de tipo selectores múltiples, como *selectManyCheckbox*, *selectManyListbox* y *selectManyMenu*, mientras que para propiedades funcionales, se pueden usar widgets de selección de ún único valor: *selectOneListbox*, *selectOneMenu* y *selectOneRadio*.

Cuando una propiedad de objeto tiene a true la característica *richTableSelector*, la lista de valores de la selección se muestra en una tabla en lugar de un componente tipo selector. Richfaces proporciona el componente *rich:extendedDataTable* que permite realizar una selección de uno o varios elementos. La Figura 4.9 muestra un ejemplo de tabla de selección múltiple.

Para obtener la lista de valores a seleccionar, se muestran todos los individuos de un tipo rango. Cuando una propiedad de objeto tiene la misma clase como tipo en su dominio y en su rango, entonces se deben considerar las propiedades de axioma reflexiva o irreflexiva. En una propiedad de objeto irreflexiva, el propio individuo (dominio) no está incluido in la lista de valores a ser seleccionados (rango), mientras que en una reflexiva, todos los individuos son seleccionables.

OWL proporciona varios constructores de extensión de clase para definir clases anónimas sin nombre. la expresión “oneOf” permite la definición de una clase enumerada a través de la lista de individuos de la clase. Cuando el rango de una propiedad es un enumerado definido a través de una clase anónoima “oneOf” , los literales de la enumeración obtenidos de los individuos asociados a la enumeración son usados como la lista de valores permitidos que pueden ser seleccionados. Un ejemplo es la propiedad funcional *hasSex* de la clase *Person* que tiene asociada

la clase enumerado con los valores *{female, male}*. En este caso, en la interfaz de usuario se puede usar uno de los tres selectores *selectOne*.

Página de listado. Por último, para listar todas las instancias o individuos de cada clase con nombre se genera una página con un componente de tipo tabla. Las columnas de la tabla son aquellas propiedades de la clase que tengan valor *true* en la característica *showAsColumn* del modelo de presentación, y la posición de la columna en la tabla se establece por el valor de la propiedad *tableColumnOrder*. Los framework de componentes ricos RichFaces y ICEfaces proporcionan un componente *datatable* con funcionalidad avanzada, incluyendo una opción *paginator* muy útil para tablas con un elevado número de filas, que permite limitar el número de filas de la tabla a visualizar y añadir un contador de páginas para avanzar en las filas mostradas, en lugar de mostrar una sola tabla con todas las filas (ver figura 4.9); una característica *sortColumn* que permite al usuario final ordenar los datos de la tabla por los valores de la columna, y una característica *filterValue* que permite filtrar los datos de la tabla. Las propiedades correspondientes en el modelo de presentación nos permite personalizar todos estos elementos.

4.7 Conclusiones del capítulo

En este capítulo se ha presentado el proceso dirigido por modelos propuesto para la generación de una arquitectura de aplicación web basada en la tecnología Java JSF, que integra el motor de inferencia Jess, a partir de modelos conceptuales basados en conocimiento. Primero se ha descrito el proceso a partir de modelos CML y después a partir ontologías ligeras OWL. Como aportaciones principales, se han diseñado los metamodelos y las transformaciones de modelos para la puesta en práctica del proceso. Para el caso de CML como modelos origen, se ha enriquecido el metamodelo de CML para modelar las características de la interfaz de usuario no presentes en los modelos conceptuales y permitir así la generación de una interfaz de usuario más potente y rica. Además, se ha diseñado el metamodelo de la arquitectura web basado en Java JSF necesario como modelo intermedio en la cadena de transformaciones MDD. Igualmente se han diseñado e implementado las transformaciones de modelos aplicadas en el proceso: la transformación M2M desde CML enriquecido al modelo de arquitectura web Java JSF, y la transformación M2T desde el modelo de arquitectura web hasta el código.

Para el caso de OWL y SWRL como modelo origen del proceso, las aportaciones principales han sido el diseño de un nuevo metamodelo de presentación web, *OWL2Web*, que complementa a OWL para el modelado de las características de la interfaz de usuario, así como la implementación de dos nuevas transformaciones M2M, la primera desde OWL hasta el metamodelo *OWL2Web* para la creación de una versión predeterminada del modelo de presentación web a partir de la ontología, y una segunda, desde OWL y el modelo de presentación web hasta el modelo de arquitectura web Java JSF, sobre el que aplicando la misma transformación M2T para el caso anterior, permite disponer de un proceso dirigido por modelos partiendo de OWL y SWRL.

De las dos aproximaciones propuestas, la primera enriqueciendo el metamodelo de CML para modelado de la interfaz de usuario, y la segunda creando un metamodelo específico que complementa a OWL, la principal ventaja de la primera respecto a la segunda es que un mismo modelo permite tener representado todo lo necesario para la aplicación de MDD y obtener la aplicación final. Pero tiene el inconveniente de que no están separados los aspectos en distintos modelos, algo que es una ventaja de la segunda aproximación, que facilita el mantenimiento y la reutilización. Estos enfoques que han sido aplicados al diseño de interfaces de usuario web ricas, pueden ser trasladados a otras plataformas tecnológicas.

Tabla 4.2: Metamodelo de Presentación *OWL2Web*: Property

Metaclase	*Característica*	*Tipo*	*Usado en Página*	*Descripción*	*Valor predet.*
Property	*name*	string	form, list	name of the property	OWL2
	label	boolean	form, list	text to show in the property field (form) or the table column (list)	prop. name
	fromFeatures	FormField	form	link to presentation features for a property in a form field	
	columnFeatures	ListColumn	list	link to presentation features for a property as a column of the table list	
FormField	*isInferred*	boolean	form	true if the value is not editable	false
	notEmpty	boolean	form	true if the value can not be empty	true
	showInForm	boolean	form	true if the form includes a field for the property	true
	formOrder	integer	form	field order in the form	null
	textInputLegth	integer	form	field size for the property value	30
	rangeMaxValue	real	form	maximum value allowed for number type properties	null
	rangeMinValue	real	form	minimum value allowed for number type properties	null
	isDataProperty	boolean	form	true if the property type is primitive, false otherwise (object property)	OWL2
	isFunctional	boolean	form	true if the property cardinality is as much one, false otherwise	OWL2
	selectorType	SelectorType	form	for object properties, kind of selector showing related instances	menu
	richTableSelector	boolean	form	for object properties, use a rich datatable for selecting related instances	false
ListColumn	*showAsColumn*	boolean	list	true if the property is showed as a column in the list	true
	tableColumnOrder	int	list	column order of the property in the table	null
	isSortableColumn	boolean	list	true if the table can be sorted by column values	true
	isFilterableColumn	boolean	list	true if the table can be filtered by column values	false

Capítulo 5

Caso de estudio y evaluación

En este capitulo se muestra la aplicación de proceso propuesto a un caso de modelado del conocimiento de mayor tamaño donde se trabaja con múltiples bases de conocimiento y la aplicación web resultante es de mayor complejidad. Además de mostrar el proceso completo de la generación de este sistema basado en conocimiento para el control de plagas y enfermedades en el cultivo de la uva de mesa, en la segunda sección de este capítulo se muestra la evaluación empírica que se ha realizado del ejemplo simple de las familias.

5.1 Un caso de estudio en el dominio agrícola

El proceso de desarrollo propuesto ha sido aplicado en SAVIA (Sistema de Vigilancia Intensiva en Agricultura), un sistema de información web basado en reglas para la gestión de información y el control de plagas en cultivos agrícolas.

La productividad de la industria agrícola actual se ha multiplicado por la incorporación de nuevas innovaciones tecnológicas sobre los sistemas tradicionales. Este es el caso del sureste español, donde la agricultura en uno de los sectores económicos más importantes especialmente en la provincia de Almería. Denominada “el Jardín de Europa” por la elevada producción intensiva en invernaderos, casi la totalidad de la producción hortícola de Almería se dedica a la exportación al resto de Europa. Sin embargo, en los últimos años, el uso inadecuado de pestici-

das ha ocasionado una imagen negativa de los productos procedentes de Almería tras la detección, en algunas partidas de productos almerienses, de residuos de pesticidas por encima de los límites legales permitidos. Esto llevó a que algunos países europeos cerraran sus fronteras durante un tiempo a productos hortícolas almerienses.

La salud vegetal y, más concretamente, el control de plagas es por tanto un factor de vital importancia en el negocio agrícola de hoy día debido al impacto que tiene sobre la economía del sector, la venta de los productos en el mercado y sus efectos tanto sobre los consumidores como sobre el entorno medioambiental. La utilización inadecuada de pesticidas puede producir perdidas significativas tanto económicas como ecológicas. Un método adecuado para reducir el impacto de plagas y enfermedades es la Producción Integrada (PI) (Norris y cols., 2003). Consiste en un sistema de producción agrícola que trata de mantener el impacto de las plagas por debajo de un umbral de acción, maximizando el uso de los recursos de producción y mecanismos naturales para asegurar la sostenibilidad de la producción agrícola a largo plazo. En PI los tratamientos químicos y biológicos son seleccionados cuidadosamente teniendo en cuenta las demandas del mercado, los aspectos económicos y la protección del medio ambiente.

Uno de las grandes contribuciones de las organizaciones científicas y las autoridades en la promoción de la PI ha sido la definición y desarrollo de marcas y programas de calidad en las políticas agrarias para asegurar la calidad y salubridad de las producciones agrícolas que aplican IP. Un ejemplo es la directiva europea 2009/128/EC que requiere planes de acción nacionales para la reducción de pesticidas y la implementación de la PI para 2014 en los países miembros de la Unión Europea (EC, 2009). Es por ello que el gobierno español y las comunidades autónomas han desarrollado una regulación de calidad relacionada con la PI.

Las tecnologías de la información y específicamente los sistemas basados en conocimiento, pueden ser aplicados al control de plagas en PI para mejorar la gestión y efectividad del proceso productivo. En general, l a utilización de sistemas software de ayuda a la toma de decisiones en agricultura no es nueva. Las primeras referencias de sistemas expertos agrícolas (Beck y cols., 1989; Plant, 1989) llevaron en años posteriores a sistemas mayores y más complejos (Chauhan y cols., 2011; Gonzalez-Andujar y cols., 2006; Lopez-Morales y cols., 2008; Mansingh y cols., 2007). Más tarde, esos sistemas de ayuda a la decisión comenzaron a incorporar técnicas de inteligencia artificial y enfoques de ingeniería del

conocimiento para proporcionar ayuda inteligente a la toma de decisiones en algunas de las tareas más importantes de la producción agrícola, como el diagnóstico de plagas y enfermedades (Lopez-Morales y cols., 2008; Mansingh y cols., 2007), el análisis de las condiciones del agua (Cardona y cols., 2011), la aplicación de fertilizantes (Busato y cols., 2013; Gonzalez-Andujar y cols., 2006), y el control de riego (Chauhan y cols., 2011), entre otras. La mayoría de esos sistemas de ayuda a la decisión basados en conocimiento han sido desarrollados para beneficiarse de las posibilidades que ofrece Internet (Grove, 2000; Yao y Yao, 2003). Por ejemplo, el uso de sistemas de ayuda a la decisión en Internet en agricultura incluye un sistema de toma de decisiones sobre riego que emplea una arquitectura orientada a servicios (Xu y cols., 2011), un prototipo de sistema de ayuda a la decisión web capaz de asistir a los agricultores mediante tecnología móvil (Antonopoulou y cols., 2010), la estimación de producción de biomasas y costes de trasporte con respeto a los requisitos de entrada, procesos internos y salida (Busato y Berruto, 2014), y el sistema de información web SAIFA que permite la monitorización de la producción integrada del cultivo de olivo en España (Orellana y cols., 2011).

El sistema de ayuda a la decisión que nos ocupa, SAVIA, aplica razonamiento basado en reglas para ayudar a agricultores y técnicos agrícolas en el cumplimiento de la regulación de calidad IP respecto la control de plagas, informando si es necesario o no aplicar un tratamiento para el control de la plaga a partir de los valores observados en el muestreo del cultivo. En la Figura 5.1 se muestra de forma jerárquica el proceso de razonamiento general seguido por los expertos. Este esquema es considerado como un marco general que ha de ser instanciado según cada plaga que aparezca en el cultivo.

5.1.1 Modelo conceptual de SAVIA

El problema de control de plagas se formula como una tarea de administración de terapia que consiste en una composición de tres tareas intensivas en conocimiento: monitorización, diagnóstico y valoración (Águila y cols., 2015). Un cultivo se considera como un sistema complejo compuesto de terreno, plantas, plagas y fauna auxiliar que puede reducir el daño producido por las plagas. Este sistema se ve afectado por variables externas como son las condiciones climáticas, humedad, precio del producto, etc. Para mantener la estabilidad del sistema se aplican medidas de control que representen el menor riesgo posible para el cultivo,

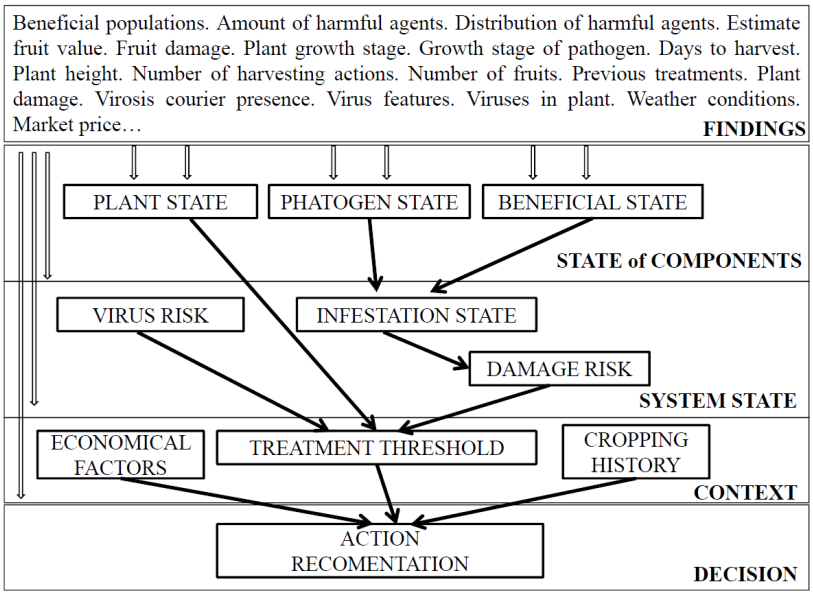


Figura 5.1: Jerarquía de niveles en el proceso seguido por los expertos

la fauna auxiliar y el medio ambiente.

El modelo de conocimiento de SAVIA se ha realizado utilizando el editor de modelos CML desarrollado como parte de esta tesis. Consiste en un esquema de dominio (*Domain Schema*) y varias bases de conocimiento (*Knowledge Base*). En el esquema de dominio se representa el dominio de aplicación en cuanto a los conceptos y sus propiedades, los tipos de reglas, los valores enumerados y la relaciones entre conceptos. La figura 5.2 muestra los conceptos del dominio, como son *Visita*, *Parcela*, *Muestreo*, *Agente nocivo*, *Plaga* y *Enfermedad* como subclases de *Agente nocivo*, *Estado Fenológico*, etc. Desde el punto de vista del razonamiento basado en reglas, *Muestreo* es el concepto más importante porque las propiedades del mismo, como son *fecha*, *posiblesFenologías*, *fenología*, *agentesPosibles*, etc., son la base para ofrecer un criterio de intervención sobre el cultivo. Los detalles del concepto *Muestreo* se muestran en la figura 5.3. Así mismo, la figura 5.4 muestra los detalles de la propiedad *criterioIntervencionTrips* de *Muestreo*, una propiedad derivada cuyo valor es obtenido por el motor de inferencia como consecuencia de la ejecución de las reglas.

La figura 5.5 muestra los detalles de tres tipos de reglas que constituyen las “plantillas” para definir instancias de reglas concretas en la

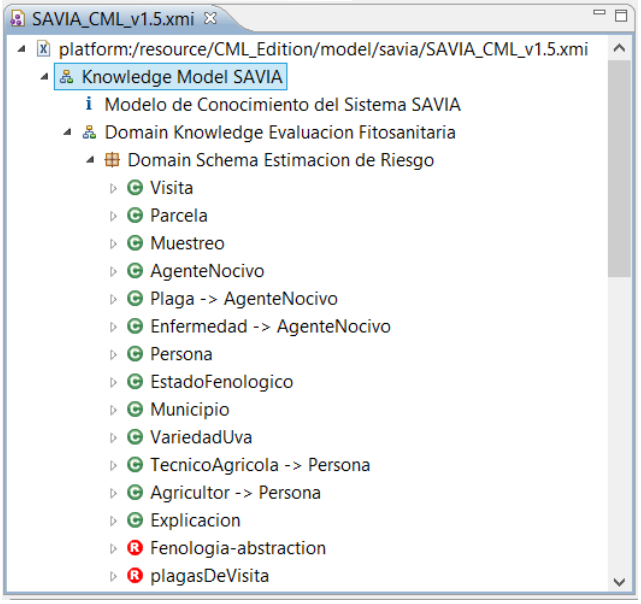


Figura 5.2: Esquema de dominio de *SAVIA* (listado general)

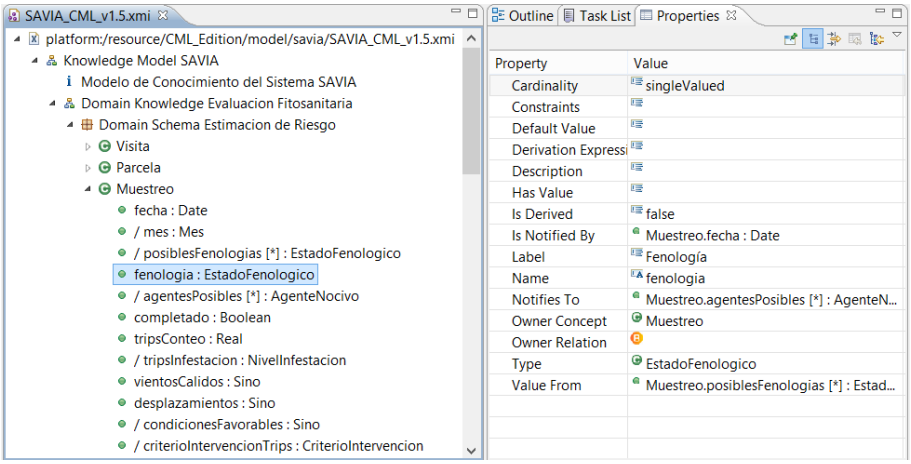


Figura 5.3: Propiedades de *Muestreo* y detalles de la propiedad *fenologia*

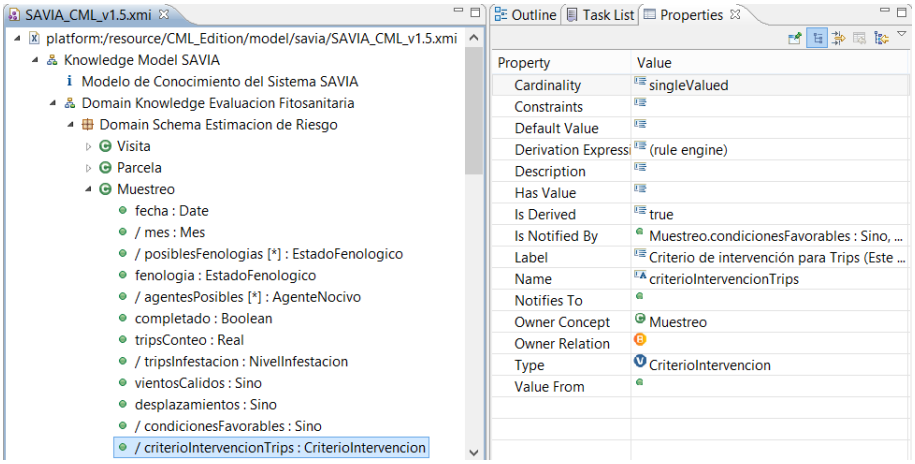


Figura 5.4: Detalles de la propiedad derivada *criterioIntervencionTrips* de *Muestreo*

base de conocimiento, como se muestra más adelante.

Los valores enumerados se muestran en la figura 5.6. Como ejemplo, *Especie Vegetal* define los dos valores que se han incluido en el dominio, por ser los cultivos más representativos de la zona de estudio: *Parral* en Murcia y *Tomate* en Almería. El modelo SAVIA presentado aquí se centra en el cultivo del *Parral* en Murcia. Un modelo similar se ha definido para el cultivo del *Tomate*.

Para finalizar con el esquema de dominio, la figura 5.7 muestra las relaciones binarias entre los conceptos.

5.1.2 Reglas de decisión para control fitosanitario

En esta sección se describe el modelado de las reglas de decisión que ofrece SAVIA para el cultivo del parral. Estas reglas están diseñadas para evaluar la necesidad de tratamiento de los agentes nocivos (plagas y enfermedades) que afectan al cultivo. Tal como se ha comentado previamente, cada plaga o enfermedad tiene su propia instanciación para el cultivo en función de su biología. La figura 5.8 muestra la instancia del esquema genérico (figura 5.1) para el caso de la *araña roja* en el cultivo del parral, que como se puede apreciar, no necesita hacer un análisis del riesgo de virus porque la biología de esta plaga no la convierte en un vector de virosis.

En el modelo de conocimiento de SAVIA se han definido varias bases

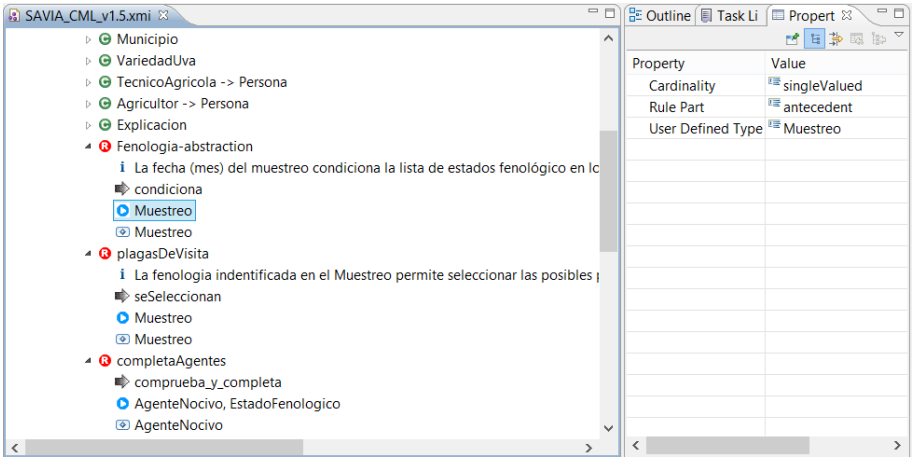


Figura 5.5: Tipos de reglas

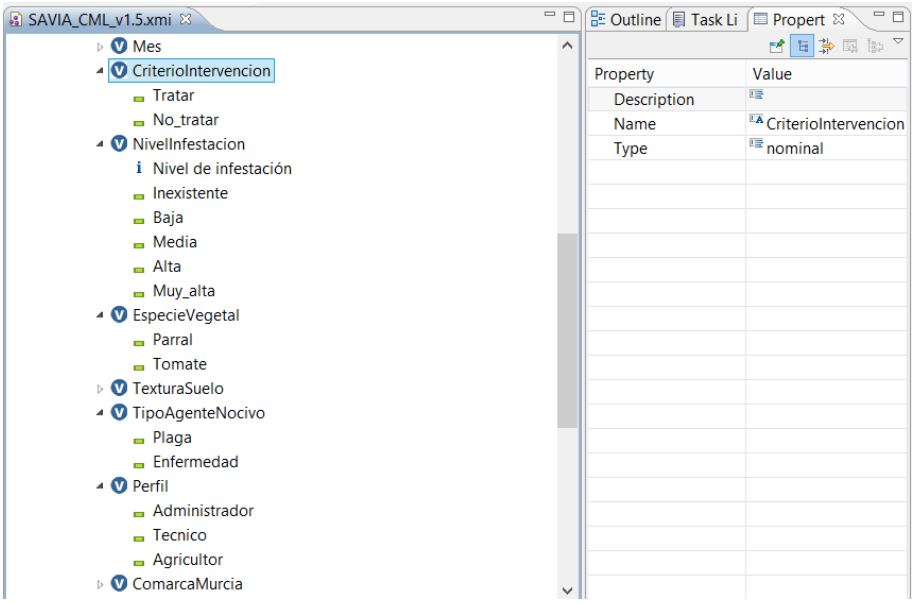


Figura 5.6: Tipos valor (enumerados)

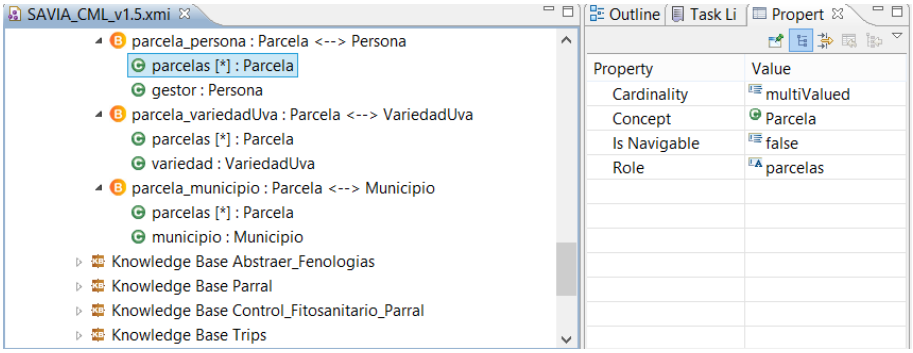


Figura 5.7: Relaciones binarias entre conceptos

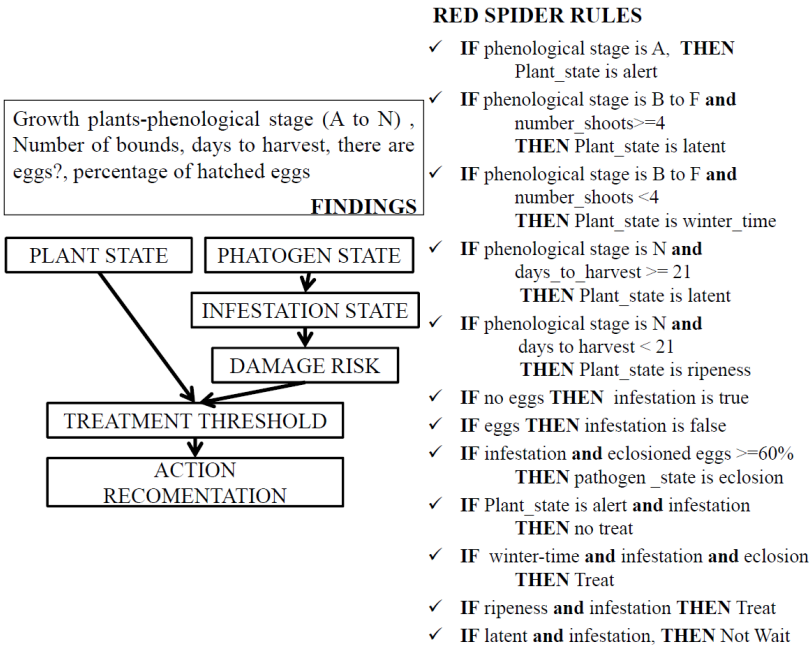


Figura 5.8: Caso de la araña roja

de conocimiento en las que agrupar instancias de los elementos definidos en el esquema de dominio:

- *Parral*
- *Control Fitosanitario Parral*
- *Abstraer Fenologías*
- *Trips*
- *Pulgón*
- *Araña Roja*
- ... y así una base de conocimiento por cada agente nocivo

La base de conocimiento denominada *Parral* almacena instancias de conceptos genéricas para el cultivo del parral, como son las variedades del cultivo y los estados fenológicos (ver figura 5.9)

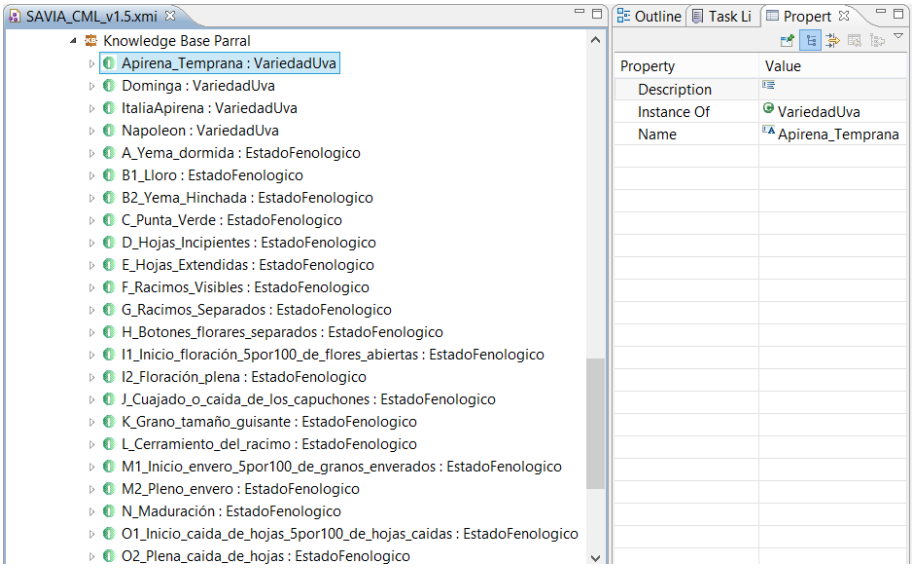


Figura 5.9: Instancias de los conceptos *VariedadUva* y *EstadoFenológico* para *parral*

La base de conocimiento *Control_Fitosanitario_Parral* (ver figura 5.10) almacena instancias de conceptos y de reglas relacionadas con el control de la salud vegetal de este cultivo, como son instancias de agentes

nocivos de subtipo *Plaga* y *Enfermedad*, e instancias de *Muestreo*: instancias de *Agente nocivo* son Trips *(Frankliniella occidentalis)*, Pulgón *(Aphis gossypii)*, Araña Roja *(Tetranychus urticae)*), Hilandero *(Lobesia Botrana)*, Botrytis *(Botrytis cinerea)*, etc.

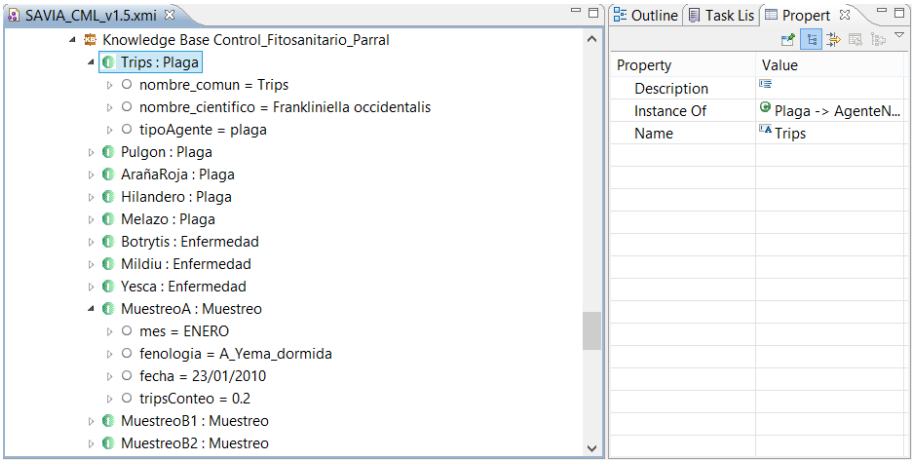


Figura 5.10: Instancias de conceptos de *Plaga*, *Enfermedad* y *Muestreo*

Las reglas de las bases de conocimiento son las que permiten derivar la decisión de necesidad de tratamiento en función de los datos de muestreo introducidos. La base de conocimiento *Abstraer Fenologías* contiene las instancias de reglas específicas para derivar los posibles estados fenológicos del cultivo del parral en función del mes del año en el que se realice el muestreo (ver figura 5.11). Estas reglas han sido definidas a partir de la información facilitada por los expertos en este cultivo, obteniendo doce reglas de este tipo, una para cada mes. Un ejemplo para la regla del mes de enero se muestra en la figura 5.12: *Si la fecha de realización del muestreo en el mes de enero (antecedente), entonces las posibles fenologías son* A-Yema Dormida*,* B1-Lloro *y* B2-Yema Hinchada *(consecuente)*.

Seguidamente se define una base de conocimiento por agente nocivo, en la que se establecen las reglas de criterio de intervención sobre el agente a partir de los datos de muestreo específicos para ese agente nocivo. Igualmente, las reglas utilizan datos genéricos de muestreo como los relacionados con las condiciones climáticas. El listado de reglas definido en la base de conocimiento para la plaga *Trips* se muestra en la figura 5.13.

La decisión a alcanzar por el sistema, que consiste en derivar si el

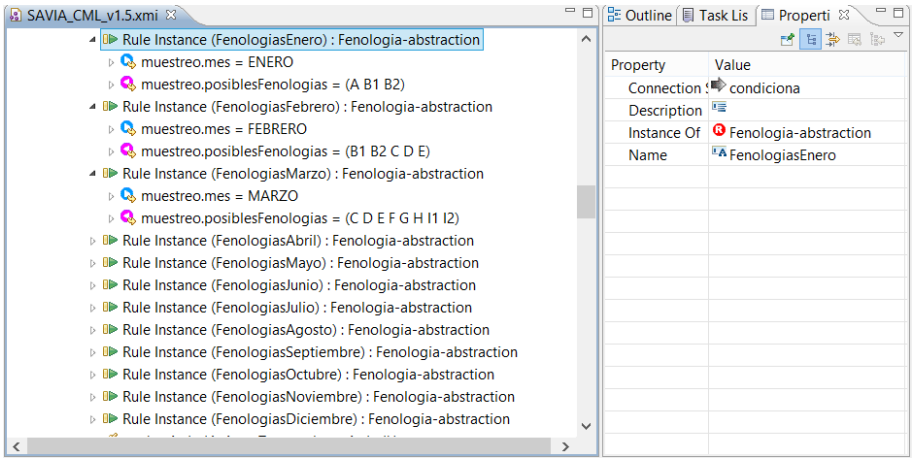


Figura 5.11: Doce instancias de tipo de regla *Fenologia-abstracción*

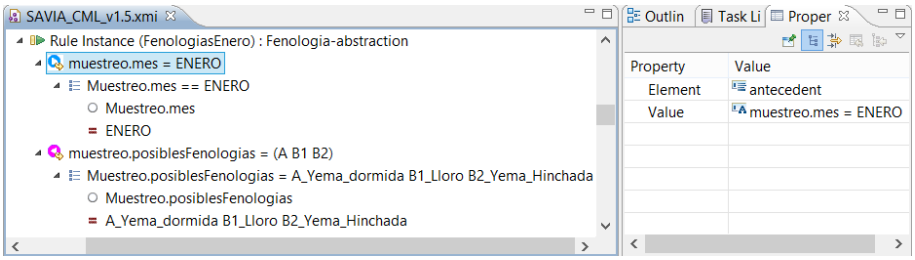


Figura 5.12: Detalles de una instancia de regla *Fenologia-abstracción*

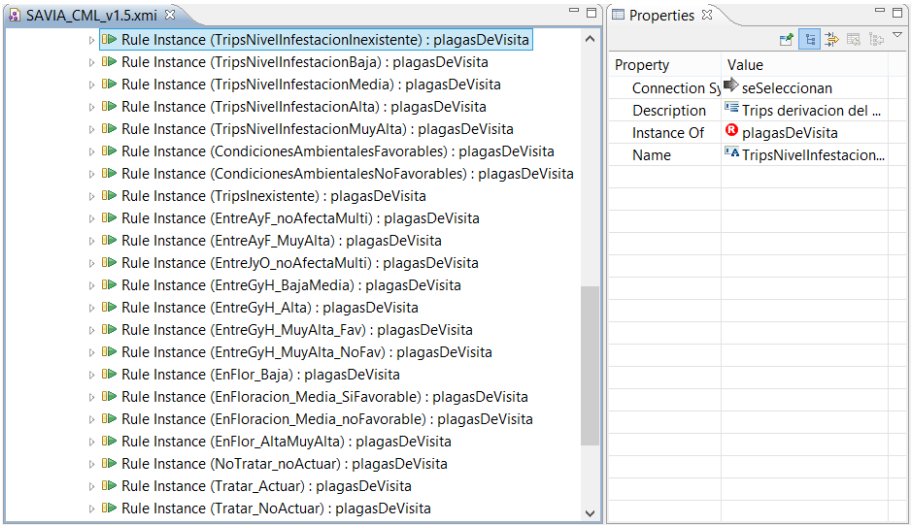


Figura 5.13: Lista de reglas de la base de conocimiento del *trips*

criterio de intervención es actuar sobre el agente nocivo o no actuar, se obtiene a partir de ese conjunto de reglas que permiten obtener indicadores intermedios en función de los cuales se derivará la decisión final. Así, se definen reglas para evaluar el nivel de infestación del cultivo debido a cada agente nocivo, que en el caso de los *trips* puede tener valor *inexistente*, *baja*, *media*, *alta* y *muy alta* dependiendo del conteo de *trips* realizado en el muestreo del cultivo. Un ejemplo de regla que deriva el nivel de infestación *alta* se muestra en la figura 5.14.

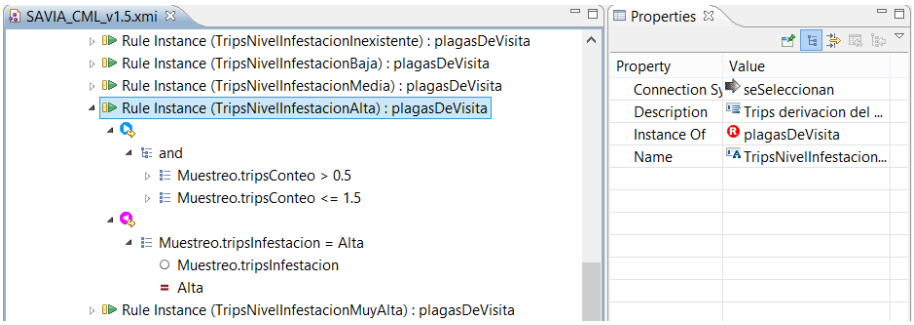


Figura 5.14: Detalles de regla que evalúa la infestación por *trips*

A partir de los datos del muestreo y de las condiciones climáticas se derivan otros resultados intermedios como las *condiciones ambientales*

favorables, que junto al *estado fenológico* permite decidir el *criterio de intervención* (*tratar* o *no tratar*). La figura 5.15 muestra el detalle de la regla que deriva el criterio de intervención en caso de nivel de infestación media, fenología en floración y condiciones ambientales favorables. El consecuente de esta regla, además de establecer criterio de intervención a *tratar*, añade un texto de explicación para que el usuario reciba información sobre la decisión ofrecida por el sistema.

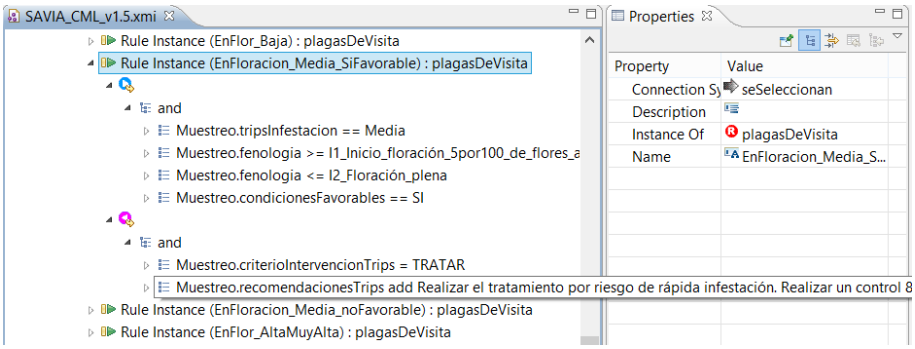


Figura 5.15: Detalles de una regla que evalúa el criterio de intervención para los *trips*

5.1.3 Transformación a modelo especifico de base de reglas y código Jess

La aplicación de las transformaciones descritas en el capítulo 3 se han aplicado al modelo conceptual de SAVIA para obtener el modelo específico de plataforma para reglas primero, y a partir de este generar el código Jess para cada uno de los elementos del modelo específico de reglas.

A continuación se muestran varios ejemplos del resultado de la aplicación de las transformaciones de modelos sobre los elementos definidos en el modelo CML de SAVia.

La figura 5.16 muestra el listado de plantillas de hechos, *fact templates*, obtenidos a partir de las clases del modelo SAVIA (ver figura 5.2). En detalle se muestran los slots del fact template *Muestreo*, cuyo modelo se presentó en la figura 5.3. La aplicación de la transformación modelo a texto genera el código Jess mostrado en la figura 5.17.

Las reglas definidas a nivel conceptual en CML como elementos de las bases de conocimiento se transforman en reglas del modelo especifico

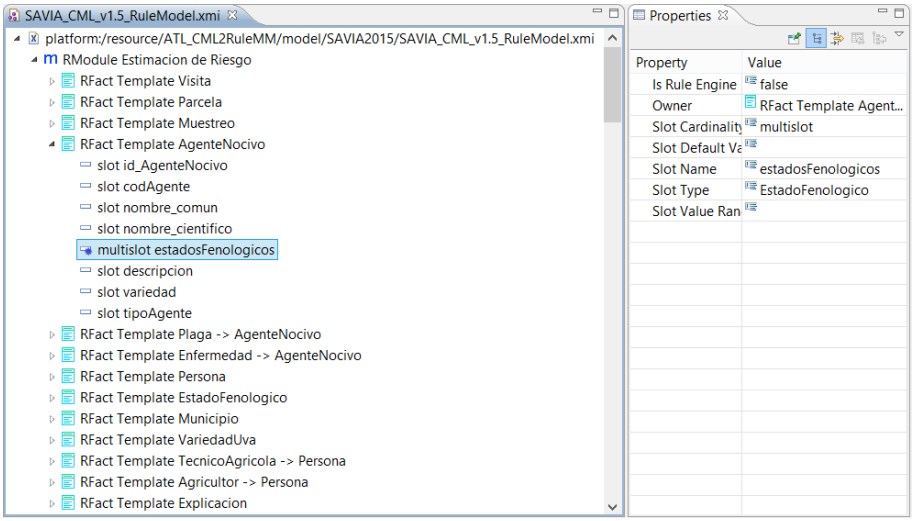
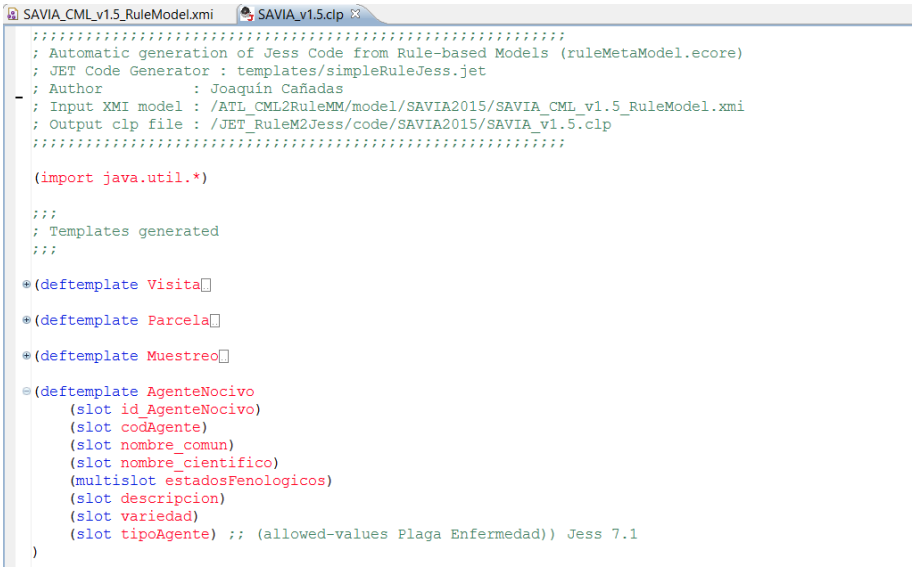


Figura 5.16: Listado de Fact Templates


SAVIA_CML_v1.5_RuleModel.xmi
SAVIA_v1.5.clp
;;;

de reglas y finalmente reglas Jess. La figura 5.18 muestra la regla que se obtiene de aplicar la transformación de modelos para la regla descrita en la figura 5.11 que establece las posibles fenologías para el cultivo del parral en el mes de enero. El código Jess resultante de aplicar la transformación de modelo a texto es el que aparece en la figura 5.19.

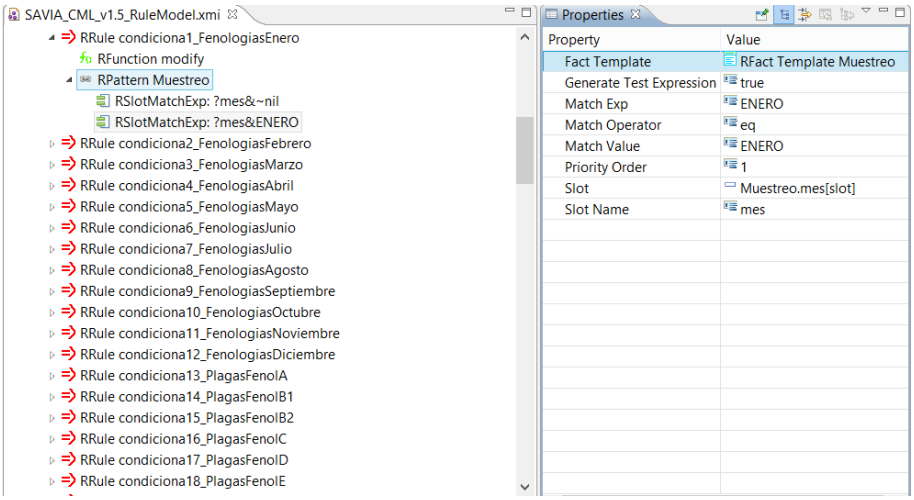


Figura 5.18: Regla que establece las posibles fenologias en el mes de enero

Otro ejemplo de regla se presenta en la figura 5.20: muestra la regla que se obtiene de aplicar la transformación M2M para la regla que deriva el criterio de intervención para los *trips* presentada en la figura 5.15. El codigo Jess resultante de aplicar la transformación M2T es el que aparece en la figura 5.21.

Las instancias modeladas en las bases de conocimiento CML detallan individuos de las clases a nivel conceptual. Dichas instancias se convierten en hechos, *facts*, en Jess. La figura 5.22 presenta un subconjunto de los hechos obtenidos para los *trips*, en concreto las instancias de recomendaciones definidas a nivel conceptual, y que son usados en las reglas para ofrecer información adicional al usuario del sistema SAVIA relacionada con el criterio de intervención derivado por la aplicación. Igualmente, los hechos se convierten de código Jess tal y como muestra la figura 5.23.

Por último, en el modelo específico de regla se incluyen determinados elementos que no están definidos a nivel conceptual pero que son necesarios para el correcto funcionamiento de una aplicación basada en

```
SAVIA_CML_v1.5_RuleModel.xmi   SAVIA_v1.5.clp
;;;
; Rules generated
;;;

(defrule condiciona1_FenologiasEnero
    "Regla antecedente SIMPLE, (RuleTypeInstance 1) (equations Antecedent: 1) (Concepts en Antecedent: 1)
    (declare (salience 10))
    ?Muestreo <- (Muestreo (mes  ?mes))
    (test (neq  ?mes nil))
    (test (eq  ?mes ENERO))
 =>
    (modify ?Muestreo (posiblesFenologias A_Yema_dormida B1_Lloro B2_Yema_Hinchada))
)

(defrule condiciona2_FenologiasFebrero

(defrule condiciona3_FenologiasMarzo

(defrule condiciona4_FenologiasAbril

(defrule condiciona5_FenologiasMayo

(defrule condiciona6_FenologiasJunio
```

Figura 5.19: Código Jess de la regla que establece las posibles fenologias en el mes de enero

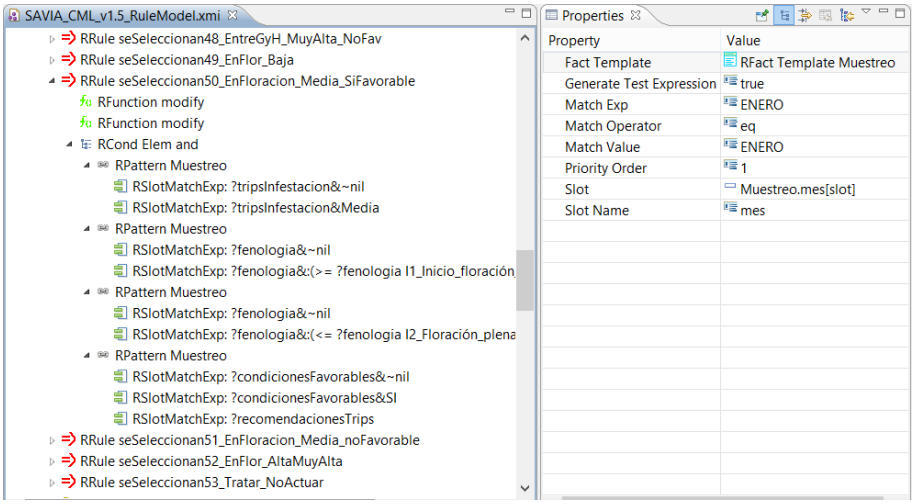


Figura 5.20: Detalles de una regla que evalúa el criterio de intervención para los *trips*

```
SAVIA_CML_v1.5_RuleModel.xmi    SAVIA_v1.5.clp
(defrule seSeleccionan50_EnFloracion_Media_SiFavorable
    "Regla antecedente SIMPLE, (RuleTypeInstance 50) (equations Antecedent: 4) (Concepts en Antecedent:
    (declare (salience 10))
    ?Muestreo <- (Muestreo (id_Muestreo  ?id_Muestreo) (fecha  ?fecha) (mes  ?mes) (posiblesFenologias
            (test (neq  ?tripsInfestacion nil))
            (test (eq  ?tripsInfestacion Media))
            (test (neq  ?fenologia nil))
            (test (>=  ?fenologia I1_Inicio_floración_5por100_de_flores_abiertas))
            (test (neq  ?fenologia nil))
            (test (<=  ?fenologia I2_Floración_plena))
            (test (neq  ?condicionesFavorables nil))
            (test (eq  ?condicionesFavorables SI))
            (test (not (member$ "Realizar el tratamiento por riesgo de rápida infestación. Realizar un
 =>
    (modify ?Muestreo (criterioIntervencionTrips TRATAR))
    (modify ?Muestreo (recomendacionesTrips  (create$ ?recomendacionesTrips "Realizar el tratamiento po
)

(defrule seSeleccionan51_EnFloracion_Media_noFavorable

(defrule seSeleccionan52_EnFlor_AltaMuyAlta

(defrule seSeleccionan53_Tratar_NoActuar
```

Figura 5.21: Código Jess de una regla que evalúa el criterio de intervención para los *trips*

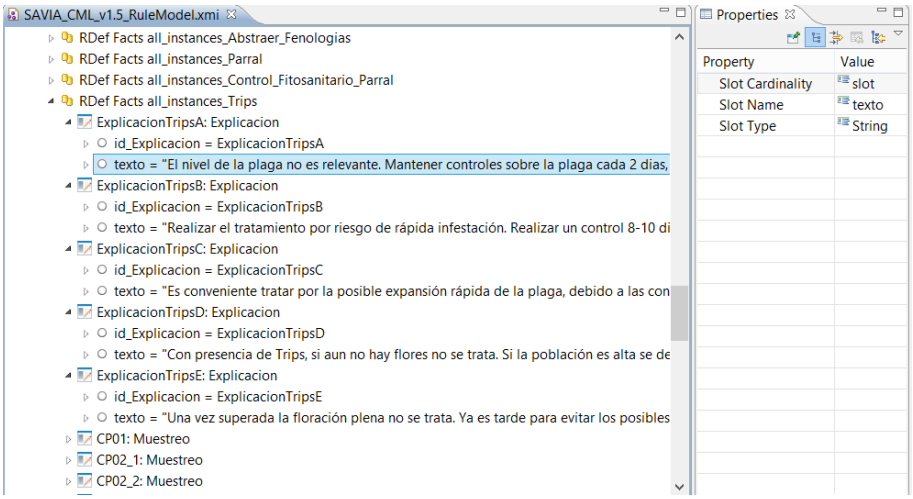


Figura 5.22: Lista de bases de hechos y detalles de hechos con recomendaciones para los *trips*

```
SAVIA_CML_v1.5_RuleModel.xmi   SAVIA_v1.5.clp
;;;
; DefFacts generated
;;;

(deffacts all_instances_Abstraer_Fenologias
(deffacts all_instances_Parral
(deffacts all_instances_Control_Fitosanitario_Parral
(deffacts all_instances_Trips
    "All facts definitions from instances of kb Trips"

    (Explicacion (id_Explicacion ExplicacionTripsA) (texto "El nivel de la plaga no es relevante. Mantener controles
    (Explicacion (id_Explicacion ExplicacionTripsB) (texto "Realizar el tratamiento por riesgo de rápida infestación
    (Explicacion (id_Explicacion ExplicacionTripsC) (texto "Es conveniente tratar por la posible expansión rápida de
    (Explicacion (id_Explicacion ExplicacionTripsD) (texto "Con presencia de Trips, si aun no hay flores no se trata
    (Explicacion (id_Explicacion ExplicacionTripsE) (texto "Una vez superada la floración plena no se trata. Ya es t
    (Muestreo (id_Muestreo CP01) (mes MARZO) (fenologia E_Hojas_Extendidas) (fecha 26/03/2010) (tripsConteo 0.0))
    (Muestreo (id_Muestreo CP02_1) (mes MARZO) (fenologia E_Hojas_Extendidas) (fecha 26/03/2010) (tripsConteo 0.2))
    (Muestreo (id_Muestreo CP02_2) (mes MARZO) (fenologia E_Hojas_Extendidas) (fecha 26/03/2010) (tripsConteo 0.4))
    (Muestreo (id_Muestreo CP02_3) (mes MARZO) (fenologia E_Hojas_Extendidas) (fecha 26/03/2010) (tripsConteo 1.1))
```

Figura 5.23: Código Jess de detalles de hechos con recomendaciones para los *trips*

reglas Jess. Es el caso de las *queries*, que se generan a partir de cada una de las clases de modelo conceptual, y que permiten consultar los hechos de la base de hechos de Jess en tiempo de ejecución. Se definen dos tipos de *queries*, una para consultar todos los hechos de un determinado tipo, *RQuery all_nombreClase* y otra para obtener un hecho concreto de un tipo a partir de su clava, *RQuery a_nombreClase*. La figura 5.24 muestra el listado de *queries* para SAVIA, y el codigo Jess generado a partir de ellas se muestra en la figura 5.25.

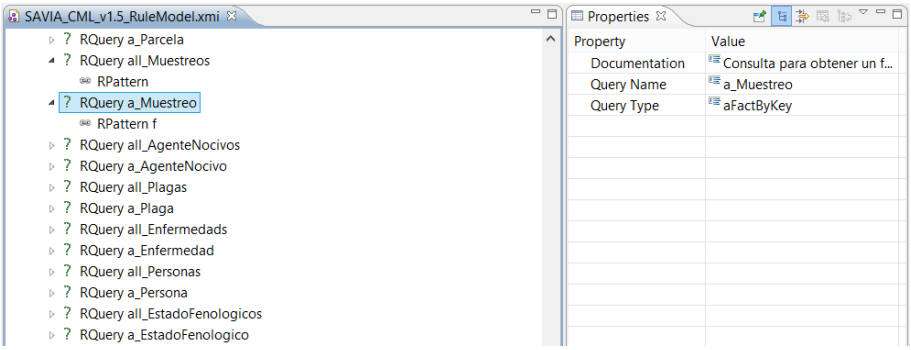


Figura 5.24: Lista de *Queries* necesarias para consultar la base de hechos de Jess

5.1.4 Modelado de la aplicación Web

Las características de modelado específicas de la aplicación web están incrustadas en el modelo CML como propiedades adicionales en cada uno

```
SAVIA_CML_v1.5_RuleModel.xmi    SAVIA_v1.5.clp
;;;
;; defQueries an-instance
;;;

(defquery a_Visita

(defquery a_Parcela

(defquery a_Muestreo
    "Consulta para obtener un fact del tipo : Muestreo"
    (declare (variables ?id_Muestreo))
    ?f <- (Muestreo (id_Muestreo ?id_Muestreo) (fecha ?fecha) (mes ?mes) (posiblesFenologias $?posiblesFenologias
)

(defquery a_AgenteNocivo

(defquery a_Plaga

(defquery a_Enfermedad
```

Figura 5.25: Código Jess de una *Query* necesaria para consultar la base de hechos de Jess

de los elementos del modelo conceptual. En la sección 4.2 se ha descrito el modelado de las características de interacción y presentación sobre CML. En el caso de estudio que nos ocupa, SAVIA, estas características se ven reflejadas en los siguientes ejemplos.

En la clase *Muestreo* (ver figuras 5.4 y 5.3), la característica *isDerived* del modelado de interacción se establece a valor *true* en varios atributos derivados, como son *mes, posiblesFenologias, tripsInfestación, condicionesFaborales, criterioIntervenciónTrips, criterioActuaciónTrips y recomendacionesTrips*. De todos ellos, *mes* tiene definido en su expresión de derivación el valor *fecha.MONTH()*, mientras que el resto tienen como expresión de derivación *(rule engine)*, lo que representa que serán las reglas de inferencia las que deriven su valor.

La actualización de los valores de los atributos cuando hay cambios en los valores de otros atributos es representada mediante las características *notifiesTo* y *isNotifiedBy*, que combinado con la definición de atributo derivado permite al motor de inferencia actualizar los valores derivados mediante las reglas. Así por ejemplo, *criterioIntervenciónTrips* es notificado por los atributos que aparecen en condiciones de reglas que tendrán como resultado obtener el valor de *criterioIntervenciónTrips*, en este caso *condicionesFavorables, fenología, tripsConteo, vientosCalidos y desplazamientos* (ver figura 5.26). De forma inversa, *condicionesFavorables, fenología, tripsConteo, vientosCalidos y desplazamientos* notifican a *criterioIntervenciónTrips*.

En la figura 5.3 se puede ver como la característica *valueFrom* del atributo *fenología* tiene el valor *Muestreo.posiblesFenologías*. Esto representa que en la aplicación web el valor de *fenología* se seleccionará de

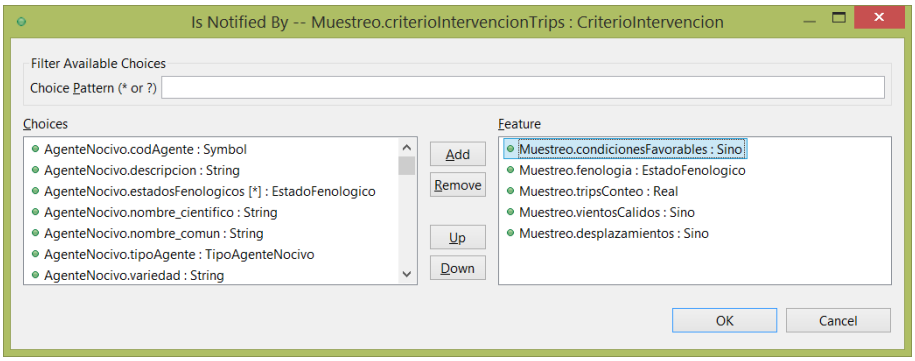


Figura 5.26: Atributos por los que es notificado el atributo *criterioIntervecióTrips* de *Muestreo*

la lista de valores que contenga *posiblesFenologías*. Así mismo, *fenología* es notificado por *fecha*, por lo que un cambio en la fecha desencadenará una ejecución de las reglas que derivan las posibles fenologías a partir del mes de la fecha, y por tanto la interfaz web debe actualizar la lista de valores sobre los que elegir el valor de *fenología* (ver figura 5.33).

5.1.5 Transformación a modelo de la arquitectura web

La aplicación de las transformaciones descritas en el capítulo 4 se han aplicado al modelo conceptual de SAVIA enriquecido con las características de diseño web para obtener el modelo específico de plataforma para de arquitectura web Java JSF, y a partir de este generar el código de los componentes de la arquitectura web.

A continuación se muestran varios ejemplos del resultado de la aplicación de las transformaciones sobre los elementos definidos en el modelo CML.

La figura 5.27 muestra los principales componentes del modelo de arquitectura web Java JSF: el *paquete* que contienen las clases Java que implementan la lógica de negocio de la aplicación Web, y el projecto JSF que contiene las páginas JSF de la interfaz de usuario generada y resto de componentes de la arquitectura.

La figura 5.28 muestra el contenido del paquete *EstimacionDeRiesgo* con las clases Java Bean asociadas a cada clase del modelo. Igualmente para las enumeraciones.

El detalle del Java Bean *Muestreo* se puede ver en la figura 5.29 con propiedades para cada atributo de la clase. Los detalles de la propiedad

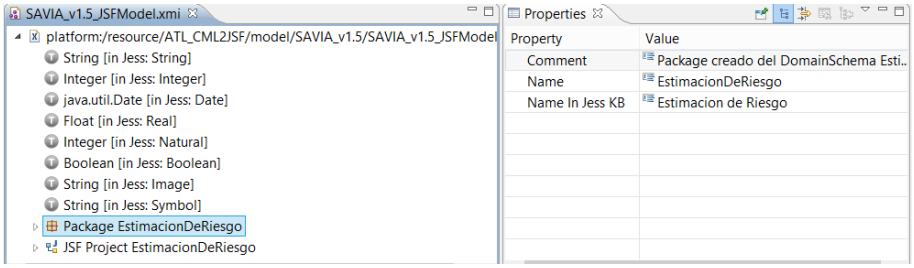


Figura 5.27: Elementos del modelo de aplicación web basado en JSF: Tipos, Paquete y JSF Project

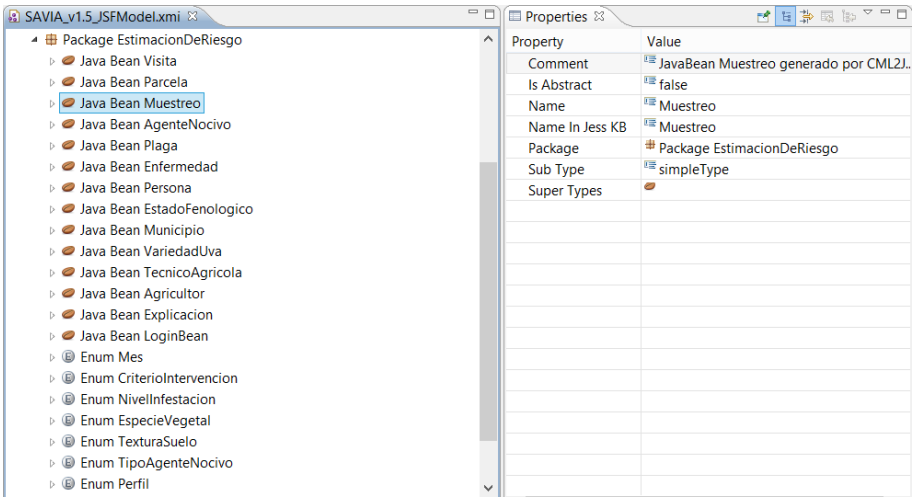


Figura 5.28: JavaBeans y Enumeraciones del paquete Java

agentesPosibles contienen toda la información necesaria para la descripción de esta propiedad. Como puede verse, se trata de una propiedad derivada cuyo valor es obtenido por el motor de inferencia, y multivaluada.

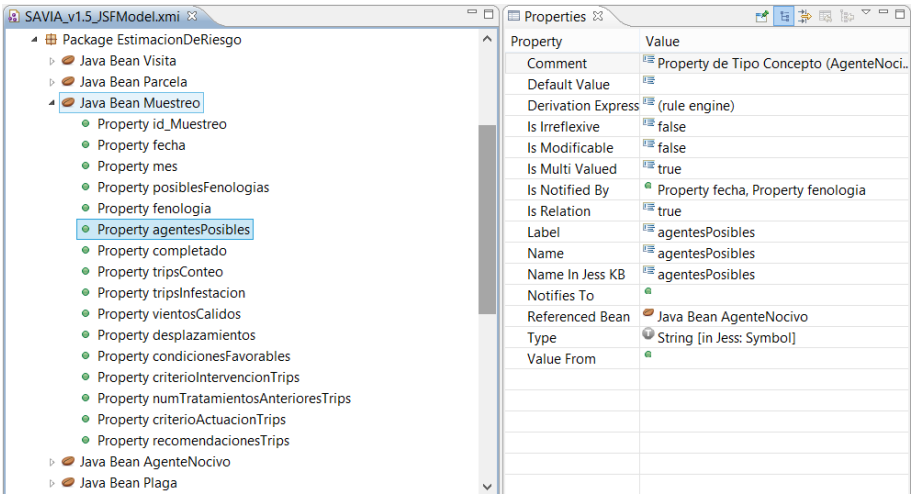


Figura 5.29: Detalles del atributo multivaluado *agentesPosibles* del JavaBean *Muestreo*

Dentro del componente projecto JSF se incluye el archivo de configuración de la aplicación Web (faces-config.xml) y las páginas de edición y listado para cada una de las clases del modelo (ver figura 5.30

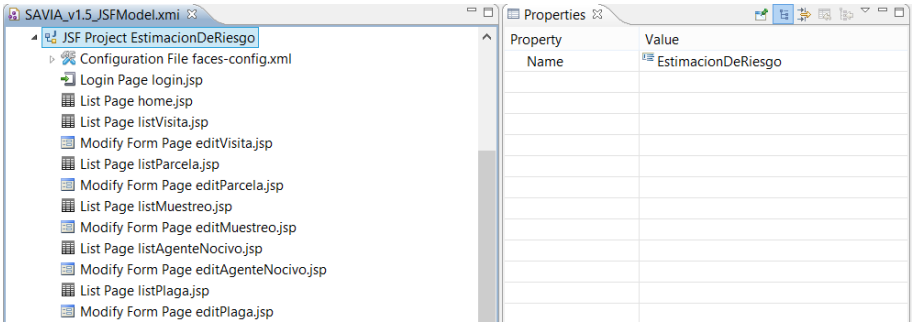


Figura 5.30: Componentes del proyecto JSF: archivo de configuracion, páginas de edición y de listado

El archivo de configuración debe contener un *bean gestionado* por cada una de las clases del modelo así como una serie de reglas de nave-

gación que definen la navegación entre las páginas de la aplicación web. La figura 5.31 muestra los componentes del archivo de configuración.

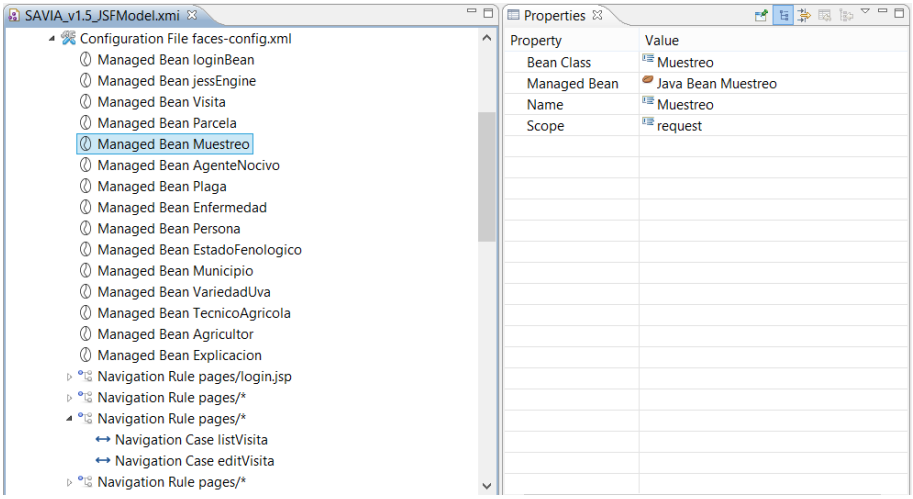


Figura 5.31: Detalles del archivo de configuración *faces-config.xml*: beans gestionados y reglas de navegación

5.1.6 Uso de SAVIA

En una parcela que pretenda aplicar la Producción Integrada, el estado del cultivo debe ser monitorizado periódicamente con visitas a la parcela por parte del técnico agrícola cada cierto tiempo. La información acerca del estado de crecimiento del cultivo, la presencia de plagas y sus efectos y el resto de información útil para conocer la salud del cultivo, es recogida mediante los datos del muestreo. El sistema gestiona toda esa información. Una lista de muestreos de cultivos se muestra en la figura 5.32. La lógica basada en reglas implementada en SAVIA proporciona ayuda a la decisión respecto a si es necesario o no realizar una acción sobre el cultivo. Sin embargo, el criterio de intervención ofrecido es orientativo, ya que es el técnico agrícola responsable de la explotación quien decide en última instancia qué acción tomar, realizando ajustes o modificaciones sobre el criterio ofrecido por el sistema.

La figura 5.33 muestra la página de edición de un muestreo, donde se introducen los datos generales del muestreo. En primer lugar se introduce la *Fecha*, y a partir ese valor el motor de inferencia deriva el listado de *Estados Fenológicos* posibles para esa fecha. En la figura, para la fecha

SAVIA tecnico1 Fecha Actual 12-abr-2014

Menu | Inicio | Parcelas | Muestreos | Agentes nocivos | Estados fenológicos

« « 1 2 » »»

Id Muestreo	Fecha	Mes	PosiblesFenologias	Fenologia	AgentesPosibles
CP03	26/mar/2010	Marzo	C_Punta_Verde D_Hojas_Incipientes E_Hojas_Extendidas F_Racimos_Visibles G_Racimos_Separados H_Botones_florares_separados I1_Inicio_floración_5por100_de_flores_abiertas I2_Floración_plena	E_Hojas_Extendidas	ArañaRoja
CP04	26/jun/2010	Junio	J_Cuajado_o_caida_de_los_capuchones K_Grano_tamaño_guisante L_Cerramiento_del_racimo M1_Inicio_envero_5por100_de_granos_enverados	M1_Inicio_envero_5por100_de_granos_enverados	Hilandero ArañaAmarilla Melazo MosquitoVerde Oidio Yesca PodredumbreAcida
CP05_1	20/abr/2010	Abril	F_Racimos_Visibles G_Racimos_Separados H_Botones_florares_separados I1_Inicio_floración_5por100_de_flores_abiertas I2_Floración_plena J_Cuajado_o_caida_de_los_capuchones K_Grano_tamaño_guisante	I1_Inicio_floración_5por100_de_flores_abiertas	Hilandero Trips Pulgon ArañaAmarilla Oidio Mildiu Botrytis
CP05_2	20/abr/2010	Abril	F_Racimos_Visibles G_Racimos_Separados H_Botones_florares_separados I1_Inicio_floración_5por100_de_flores_abiertas I2_Floración_plena J_Cuajado_o_caida_de_los_capuchones K_Grano_tamaño_guisante	G_Racimos_Separados	Hilandero Trips Pulgon Oidio Mildiu
CP06_1	20/abr/2010	Abril	F_Racimos_Visibles G_Racimos_Separados H_Botones_florares_separados I1_Inicio_floración_5por100_de_flores_abiertas I2_Floración_plena J_Cuajado_o_caida_de_los_capuchones K_Grano_tamaño_guisante	G_Racimos_Separados	Hilandero Trips Pulgon Oidio Mildiu

Figura 5.32: Lista de muestreos de un cultivo

seleccionada en el mes de abril existen 7 estados fenológicos en los que se puede encontrar el cultivo muestreado. El usuario debe seleccionar el estado fenológico del cultivo de la lista de estados posibles, usando el menú desplegable *Fenologia*. Una vez seleccionada la fenología, el sistema deriva automáticamente la lista de agentes nocivos que pueden afectar al cultivo en esa fenología, que en este caso también son 7.

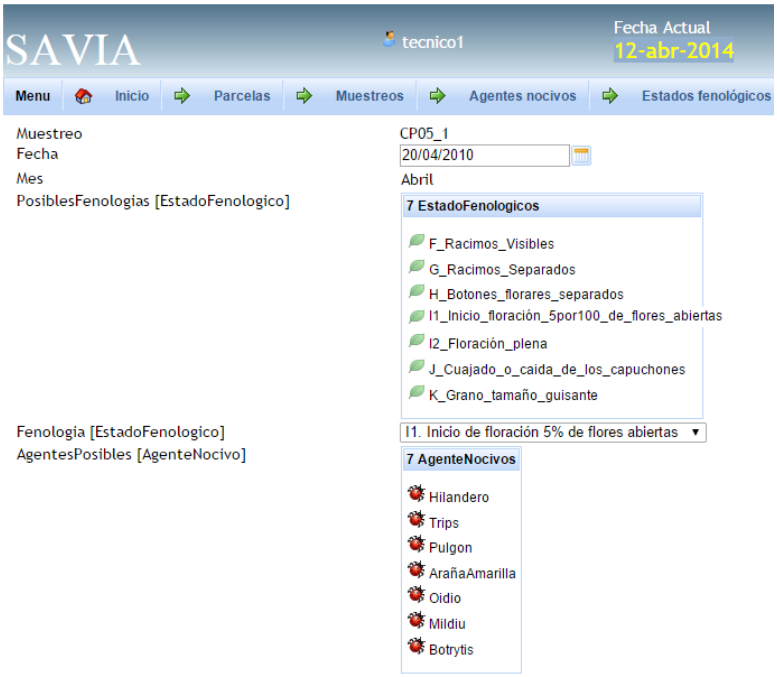


Figura 5.33: Vista de la edición de un muestreo. Datos generales

A continuación el sistema pide al usuario datos específicos del muestreo de la plaga en estudio. La figura 5.34 muestra los datos introducidos para un ejemplo de muestro de la plaga de los trips. El sistema ejecuta el mecanismo de inferencia y ofrece de forma interactiva un criterio de intervención en cuanto los datos son introducidos. En el caso de la figura, el criterio ofrecido es *Tratar*.

La figura 5.35 muestra como cambiando un dato del muestreo, en este caso el *número de tratamientos*, el sistema cambia el criterio de intervención revisado para la plaga, ya que el número de tratamientos ha pasado de 0 a 2, y según las reglas no se pueden sobrepasar 2 tratamientos para los trips.

Completado	☐
Formas móviles por racimo	1,90
Nivel de infestación por Trips	Muy alta
¿Hay vientos cálidos?	No
¿Hay niveles altos de población en malas hierbas o cultivos cercanos?	No
Condiciones ambientales favorables	No
Criterio de intervención para Trips (Este criterio de intervención es orientativo. Los ajustes o modificaciones del mismo deben ser realizados por el Técnico responsable de la explotación)	Tratar
Número de tratamientos	0
Criterio de intervención para Trips revisado teniendo en cuenta el historial de tratamientos	Tratar

Recomendaciones para los Trips

1 RecomendacionesTrips

- "Es conveniente tratar por el alto nivel de plaga y estado en floración"

Save Muestreo

Figura 5.34: Datos de muestreo para Trips: resultado *Tratar*

Completado	☐
Formas móviles por racimo	1,90
Nivel de infestación por Trips	Muy alta
¿Hay vientos cálidos?	No
¿Hay niveles altos de población en malas hierbas o cultivos cercanos?	No
Condiciones ambientales favorables	No
Criterio de intervención para Trips (Este criterio de intervención es orientativo. Los ajustes o modificaciones del mismo deben ser realizados por el Técnico responsable de la explotación)	Tratar
Número de tratamientos	2
Criterio de intervención para Trips revisado teniendo en cuenta el historial de tratamientos	No tratar

Recomendaciones para los Trips

2 RecomendacionesTrips

- "Es conveniente tratar por el alto nivel de plaga y estado en floración"
- "Los valores de conteo y condiciones de riesgo indican que es aconsejable el tratamiento. Sin embargo, ya ha habido 2 tratamientos para los trips, por tanto se ha cubierto el número máximo y no se debe realizar ningún tratamiento más."

Save Muestreo

Figura 5.35: Datos de muestreo para Trips: resultado *No Tratar*

5.2 Evaluación de la aplicación web del ejemplo *Familia*

Para evaluar la aplicación web resultante de aplicar el proceso dirigido por modelos propuesto en esta tesis se ha medido el rendimiento de la aplicación web generada para el caso de estudio *Familia*, así como aspectos de usabilidad de la misma.

La aplicación web basada en reglas del caso *Familia* ha sido probada utilizando bases de conocimiento de diferentes tamaños (10, 20, 50, 100, 200, 500 y 1000 individuos o instancias) y dos conjuntos de reglas diferentes (5 y 20 reglas). Los resultados de medir los tiempos de respuesta se muestran en la figura 5.36. Se muestra el tiempo que tarda en cargar la base de conocimiento, es decir, el tiempo que emplea el motor de inferencia en ejecutar las reglas sobre todos los individuos de la base de conocimiento cuando estos son cargados por primera vez. Puesto que antes de realizar esta carga inicial los individuos solo tienen unos pocos e indispensables valores en sus propiedades, como son *age*, *father* y *mother*, el proceso inicial de carga debe derivar el resto de relaciones familiares para cada uno de los individuos (*siblings*, *brothers*, *sisters*, *uncles*, etc.). Esta tarea consume más y más tiempo cuando el número de individuos aumenta, tal y como muestra la figura 5.36. El número de reglas también provoca un efecto extremo sobre el tiempo de carga inicial. La escala de tiempo para el caso de 5 reglas (escala de la barra vertical izquierda en la figura), del orden de decenas de segundos, es mucho mas pequeña que la escala de tiempo para el caso de 20 reglas (escala de la derecha en la figura), del orden de cientos de segundos hasta casi 2000 segundos. Sin embargo, ambos conjuntos de reglas se comportan de forma similar cuando el número de individuos asciende ya que ambas curvas muestran la misma tendencia de crecimiento.

Es importante destacar que, una vez finalizada esta carga inicial, la ejecución de la aplicación web se mantiene interactiva sin tiempos de espera considerables, ya que la creación, modificación o eliminación de instancias solamente afecta a unos pocos individuos, veasé, la persona editada y sus familiares relacionados, si los tiene. Así que, cuando el usuario final interacciona con la aplicación, el motor de inferencia dispara las reglas apropiadas en orden de milisegundos, proporcionando una respuesta inmediata al usuario. Este rendimiento ha sido probado en el caso de estudio evaluado. Sin embargo, un motor de inferencia como Jess podría ofrecer una respuesta diferente en otros casos, ya que cuando un

hecho es emparejado en muchas reglas con muchas combinaciones, puede llevar un tiempo razonablemente grande modificarlo (Friedman-Hill, 2003).

La aplicación web basada en reglas resultante está dirigida a ser utilizada por usuarios finales. Proporciona una usabilidad mejorada gracias al uso de tecnología AJAX y a la aplicación de buenas prácticas de diseño. De hecho, tanto el motor de reglas embebido como la tecnología AJAX mejoran la creación y edición de instancias en la aplicación web: los formularios web están implementados con componentes RichFaces que incorporan tecnología AJAX la cual permite que cada valor de formulario pueda ser validado y enviado individualmente al servidor una vez que este es introducido por el usuario en el navegador web. Esta característica permite que el motor de inferencia ejecute las reglas adecuadas que afectan al valor recién introducido, derivando nueva información a partir de ese valor recién introducido, lo que dirige la creación o edición de la instancia. Un ejemplo de esto lo percibimos cuando en la propiedad *Padre* de una persona se introduce un nuevo valor, la aplicación automáticamente actualiza los valores de ciertos campos del propio formulario.

Las figuras 5.37 y 5.38 muestran para el ejemplo de Familia un ejemplo de página de listado y página del formulario de edición para la clase Persona, respectivamente.

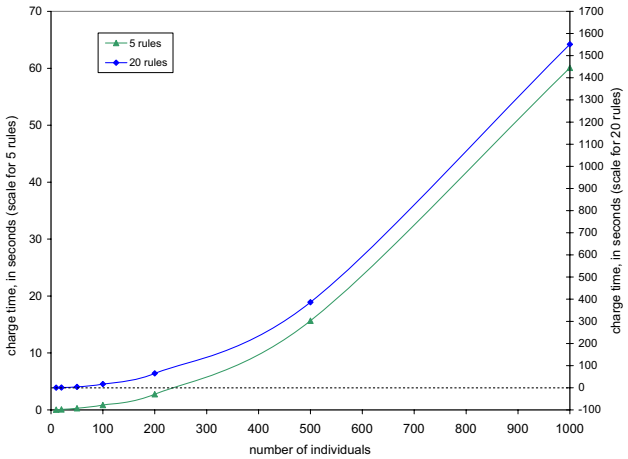


Figura 5.36: Tiempos de inicializacion para la aplicación web *Familia*

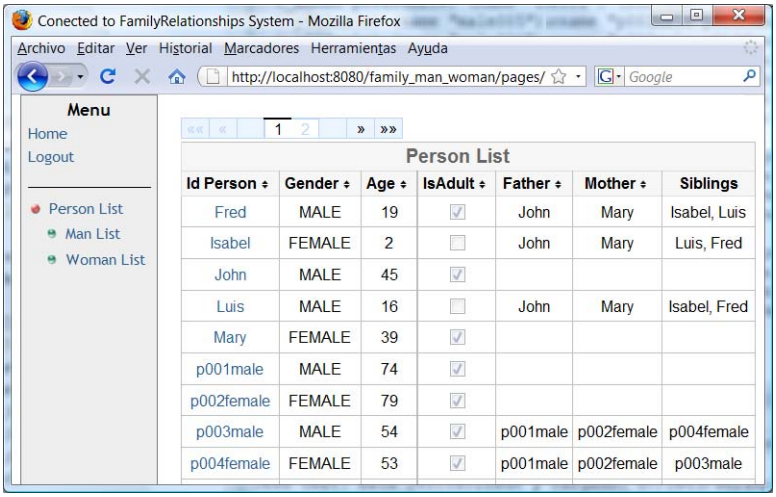


Figura 5.37: Aplicación web de Familia: listado de personas

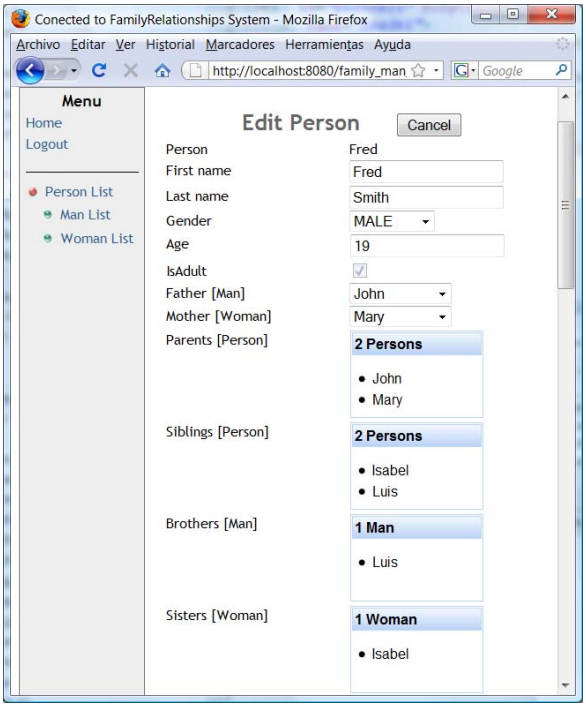


Figura 5.38: Aplicación web de Familia: edición de persona

5.3 Discusión

El proceso de desarrollo dirigido por modelos propuesto tiene tanto fortalezas como debilidades. La principal ventaja es la drástica reducción en el tiempo y esfuerzo de desarrollo de una nueva aplicación web basada en reglas, ya que la herramienta de soporte que implementa el proceso realiza el trabajo duro. El desarrollo incremental, propio de las metodologías de desarrollo de software actuales, también se ve favorecido con este enfoque porque los cambios en el modelo son trasladados directamente a la implementación del sistema. Ademas, como característica propia de los enfoques MDA/MDD, los modelos son ahora los principales artefactos en el desarrollo de software, no una mera documentación que puede quedar obsoleta. También, la calidad del código final se incrementa, ya que las buenas prácticas de diseño y codificación están implementadas en las transformaciones, produciendo código de alta calidad. La creciente aplicación de MDE a nivel industrial en los últimos años así lo corrobora (Cabot y cols., 2014).

Por supuesto, también existen muchas limitaciones ligadas fundamentalmente al enfoque MDA/MDD. Principalmente, el tiempo y esfuerzo necesario para el desarrollo de la herramienta de soporte al proceso MDD es realmente significativo, por lo que deben ser evaluados los beneficios que se van a obtener al disponer de la herramienta MDD respecto al importante esfuerzo necesario en su diseño y construcción. De esta forma, podemos decir que cuando el objetivo es el desarrollo de un único sistema, en nuestro caso se trataría de un sistema basado en reglas en un dominio especifico, la construcción del software desde cero va a ser la mejor opción. Sin embargo, cuando un tipo de solución software puede ser aplicada a muchos sistemas similares en muchos dominios de aplicación, la construcción de una herramienta de soporte MDD como la presentada es en esta tesis es una solución más que apropiada.

Otra importante limitación está relacionada con la evolución del proceso de desarrollo y las herramientas de soporte. Los modelos están limitados a la expresividad de los metamodelos, y la generación de código también está limitada a lo que puede ser expresado en los modelos. Por eso cuando es necesario un cambio en la expresividad de los modelos o en la generación de código, entonces hay que realizar cambios en los metamodelos, los modelos y las transformaciones, haciendo que la herramienta MDD sea difícil de evolucionar.

Respecto a las ventajas e inconvenientes ligados a la propuesta concreta presentada en esta tesis doctoral, los lenguajes de modelado de

conocimiento utilizados ofrecen cada uno sus ventajas: CML a pesar de ser un lenguaje relativamente anticuado y no estandarizado, nos permite total flexibilidad a la hora de modificar sus propiedades y añadir nueva expresividad al metamodelo no contemplada en su definición original. Por otro lado, el uso de OWL y SWRL nos ha permitido evaluar la propuesta empleando lenguajes estandarizados ampliamente usados por la comunidad.

En cuanto a los inconvenientes, destacar que los lenguajes de modelado utilizados no están pensados para el diseño de interfaces de usuario, por lo que en ambos casos ha sido necesario la ampliación de los metamodelos con características específicas para el diseño de interfaces de usuario como el caso de CML, o bien la creación de modelos paralelos que permitan especificar estas características, como el caso de OWL. También es reseñable indicar que debido a la extensa semántica de OWL, mayor que la de los motores de inferencia como Jess, solamente parte del modelo OWL es transformado a Jess. La utilización de un razonador específico para OWL como Pellet (Sirin y cols., 2007) resolvería esta limitación, aunque también eliminaría la necesidad de la transformación entre distintos lenguajes ya que Pellet interpreta directamente ontologías OWL, por lo que estaríamos hablando de un problema distinto que queda fuera del ámbito de esta tesis.

5.4 Conclusiones del capítulo

En este capítulo se ha presentado SAVIA, un sistema de información web basado en reglas para el control de plagas en cultivos agrícolas, conforme a la normativa Producción Integrada. La principal aportación ha sido la puesta en práctica del proceso MDD propuesto en esta tesis en el desarrollo de SAVIA. Se ha descrito el modelo de conocimiento de SAVIA, el resultado de aplicación de las transformaciones de modelos y fragmentos del código resultante. Igualmente se ha descrito el uso del sistema SAVIA relacionándolo con las características especificadas en el modelo. Tras ello se han presentado los resultados de una evaluación empírica de la aplicación web del ejemplo de la Familia, en relación al tiempo de respuesta y usabilidad de la aplicación. Por último se ha realizado una discusión sobre el método aplicado en este trabajo.

Capítulo 6
Conclusiones

6.1 Conclusiones

La principal conclusión de esta tesis doctoral ha sido que la hipótesis de partida ha sido demostrada: el desarrollo de software dirigido por modelos ha permitido formalizar y automatizar el desarrollo de aplicaciones web basadas en reglas a partir de los modelos conceptuales definidos en lenguajes propios de la Ingeniería del Conocimiento.

Las conclusiones y principales aportaciones en relación a los objetivos marcados inicialmente han sido las siguientes:

1. Se ha definido un propuesta genérica de aplicación del desarrollo dirigido por modelos para sistematizar el proceso de construcción de aplicaciones web basadas en reglas a partir de los modelos conceptuales de la IC, que está dividida en dos subprocesos independientes: por un lado, el que permite el desarrollo de la base de reglas, y por otro, el que permite el desarrollo de la aplicación web funcional que integra el motor de inferencia y la base de reglas generada por el primer proceso.

2. Se ha formalizado el proceso dirigido por modelos para la generación de la base de reglas en Jess a partir de modelos conceptuales, primero a partir de modelos CML y después a partir ontologías ligeras OWL y SWRL:

    - Para el caso de usar CML como modelos origen, las principales aportaciones han sido la creación del metamodelo del

conocimiento del dominio de CML, el cual se ha enriquecido con operadores sobre colecciones para proporcionar una mayor expresividad al lenguaje. Igualmente se ha diseñado el metamodelo de reglas Jess. Para completar el proceso se han diseñado e implementado las transformaciones de modelos aplicadas en el proceso: la transformación M2M desde CML al modelo de reglas Jess, y la transformación M2T desde el modelo de reglas Jess hasta código Jess.

- Para el caso de OWL y SWRL como modelo origen del proceso MDD, se ha utilizado el metamodelo de OWL y SWRL estándar ODM ya disponible, y la aportación principal ha sido una nueva transformación M2M desde OWL y SWRL al metamodelo de reglas Jess comentado en el caso anterior, lo que, reutilizando la misma transformación M2T para el caso anterior, permite aplicar un proceso dirigido por modelos partiendo de OWL y SWRL.

3. Se ha definido el proceso dirigido por modelos propuesto para la generación de una arquitectura de aplicación web basada en la tecnología Java JSF que integra el motor de inferencia Jess a partir de modelos conceptuales basados en conocimiento enriquecidos mediante la incorporación del modelado de características propias de aplicaciones web:

   - Para el caso de CML como modelos origen, como principales aportaciones se ha enriquecido el metamodelo de CML para modelar las características de la interfaz de usuario no presentes en los modelos conceptuales y permitir así la generación de una interfaz de usuario más potente y rica. Además, se ha diseñado el metamodelo de la arquitectura web basado en Java JSF necesario como modelo intermedio en la cadena de transformaciones MDD. Igualmente se han diseñado e implementado las transformaciones de modelos aplicadas en el proceso: la transformación M2M desde CML enriquecido al modelo de arquitectura web Java JSF, y la transformación M2T desde el modelo de arquitectura web hasta el código.
   - Para el caso de OWL y SWRL como modelo origen del proceso, las aportaciones principales han sido el diseño de un nuevo metamodelo de presentación web, *OWL2Web*, que complementa a OWL para el modelado de las características de la

interfaz de usuario, así como la implementación de dos nuevas transformaciones M2M: la primera desde OWL hasta el metamodelo *OWL2Web* para la creación de una versión predeterminada del modelo de presentación web a partir de la ontología; y la segunda, desde OWL y el modelo de presentación web hasta el modelo de arquitectura web Java JSF, sobre el que aplicando la misma transformación M2T para el caso anterior de CML, permite disponer de un proceso dirigido por modelos partiendo de OWL y SWRL.

Esta arquitectura tiene la ventaja de que, debido a la naturaleza declarativa de las reglas, la lógica de decisión a nivel de negocio se externaliza del código fuente del núcleo de la aplicación, lo que permite obtener aplicaciones web más fáciles de mantener y evolucionar.

4. El proceso propuesto se ha evaluado en el desarrollo de SAVIA, un sistema de información web basado en reglas para el control de plagas en cultivos agrícolas que ofrece recomendaciones a los agricultores y técnicos agrícolas sobre la necesidad de tratamiento de determinadas plagas y enfermedades en los cultivos. Se ha descrito el modelo de conocimiento de SAVIA, el resultado de aplicación de las transformaciones de modelos y fragmentos del código resultante. Igualmente se ha descrito el uso del sistema SAVIA relacionándolo con las características especificadas en el modelo. Además, se han presentado los resultados de una evaluación empírica de la aplicación web del ejemplo sencilo de las relaciones familiares, en relación al tiempo de respuesta y usabilidad de la aplicación.

También es adecuado remarcar que el tiempo y esfuerzo necesarios en el desarrollo del proceso MDD propuesto han sido considerablemente elevados, tanto por la complejidad de la propuesta como por el elevado número de tecnologías y lenguajes que han de utilizarse para definir un proceso como el presentado en esta tesis. Por ello, se puede concluir que un enfoque dirigido por modelos es una solución apropiada solamente cuando esta vaya a ser usada en la construcción de un numero considerable de sistemas similares en diversos dominios de aplicación. En esos casos, la drástica reducción en el coste de desarrollo de los sistemas, una vez que la herramienta de soporte MDD ha sido desarrollada, justifican el elevado esfuerzo inicial. Igualmente, el desarrollo iterativo e incremen-

tal se ve favorecido con este enfoque ya que los cambios en el modelo pueden ser directamente reflejados en la implementación del sistema.

6.2 Trabajos futuros

Como trabajo futuro se plantean varias lineas principales, que se describen a continuación:

- La utilización de otros lenguajes de modelado de reglas como modelos origen para el proceso MDD, como por ejemplo SBVR (Semantics of Business Vocabulary and Business Rules) (Object Management Group, 2013) que es el formalismo adoptado por la OMG como estándar en reglas de negocio. Así mismo se plantea utilizar otras tecnologías de implementación de sistemas basados en reglas, como por ejemplo Drools (JBoss, 2009).

- La arquitectura de la aplicación web generada se puede ser más compleja, incorporando una capa de persistencia mediante un gestor de bases de datos relacional.

- Se plantea extender la aplicación web basada en reglas generada con funcionalidades de la Web Semántica, como búsquedas semánticas basadas en la jerarquía de clases de la ontología así como búsquedas a nivel de instancias. Para salvar la diferencia de semántica entre OWL y el motor de inferencia, una solución podría ser utilizar un razonador como Pellet que interpreta directamente OWL y SWRL, e incorporarlo en la aplicación web como mecanismo de razonamiento.

- Igualmente se plantea aplicar un enfoque dirigido por modelos similar para una arquitectura software diferente sobre la que integrar un motor de inferencia. De esta forma, el proceso de desarrollo dirigido por modelos de la base de reglas se podría reutilizar y diseñar un nuevo proceso MDD orientado a la nueva arquitectura de aplicación, por ejemplo, una aplicación móvil, donde ya existen trabajos publicados que plantean los retos de integrar un motor de inferencia en una aplicación móvil (Slazynski y cols., 2014).

- Otra linea de trabajo es la de la automatización de pruebas para aplicaciones basadas en reglas. Actualmente existen numerosas tecnologías de automatización de pruebas del software (Beizer, 2002),

contexto en el que se ha aplicado con éxito MDA (Baker y cols., 2007). El reto se plantea en las pruebas automatizadas de SBC y la aplicación de un enfoque MDA para la generación de los casos de prueba partiendo de los modelos de reglas, tanto a nivel de pruebas unitarias (Nalepa y cols., 2014), como a nivel de pruebas de aceptación.

- La aplicación de la integración continua (Duvall y cols., 2007) y el despliegue continuo (Humble y Farley, 2010), así como tecnologías propias de Computación en la Nube (McGrath y McGrath, 2012) nos permitiría diseñar un entorno automatizado de integración y despliegue continuo que complemente el proceso MDE propuesto. La idea consiste en que la aplicación web generada se construya y despliegue automáticamente tras la generación de nuevo código obtenido por el proceso MDD tras un cambio realizado en los modelos. Después de la ejecución de las transformaciones de modelos, el código actualizado se guarda en un repositorio de código fuente (*Subversion* o *Git*). La actualización del repositorio dispara el proceso de construcción automatizada (*build*) en un sistema de integración continua como *Jenkins* (Smart, 2011), que recoge las actualizaciones en el código y ejecuta el proceso de construcción de la nueva versión de la aplicación. Igualmente, si no hay problemas en la construcción, *Jenkins* es capaz de desplegar la aplicación en el servidor de despliegue en la Nube. La nueva versión actualizada de la aplicación estaría disponible para el usuario final con los cambios incorporados en el modelo. El proceso automatizaría todos los pasos necesarios desde el modelado hasta el despliegue de la aplicación web en la Nube.

- Los editores de modelos creados en esta tesis, tanto de CML como de los modelos específicos de plataforma (Jess, JSF) pueden ser llevados a la Nube haciendo uso de EMF-REST (Ed-Douibi y cols., 2014). EMF-REST es una herramienta actualmente en desarrollo que toma los modelos EMF como entrada y genera una API Web siguiendo los principios de servicios web RESTful, soportado por librerías y estándares conocidos. Trata de proporcionar mayor agilidad al proceso de desarrollo web proporcionando APIs REST listas para ejecutar y probar a partir de modelos. Para los profesionales de MDE, este enfoque promete proporcionar las bases para el desarrollo de soluciones basadas en modelos que se desplieguen

en la Nube, así como un soporte a la colaboración mejorado para aplicaciones web de modelado.

- La comunidad aplica MDE en otros escenarios además de la automatización en el desarrollo de software. MDE puede ser aplicado a la validación y verificación de SBC, mediante la transformación de los modelos conceptuales de reglas utilizados en esta tesis, tanto CML como OWL y SWRL, en formatos interpretables por analizadores formales de bases de reglas, que permiten detectar problemas clásicos de inconsistencia, incompletitud, redundancias y ciclos, etc. (Vermesan y Coenen, 2013).

- Como consecuencia de la realización de esta memoria de tesis, se ha diseñado una transformación de EMF a LaTeXcon el objetivo de generar automáticamente la documentación de los metamodelos, que ha servido para obtener el contenido de los apéndices que describen detalladamente los metamodelos propuestos en esta tesis (apéndices A, B, C, y D). Se plantea publicar esta transformación con licencia de código abierto y ponerla a disposición de la comunidad, como un generador de documentación LaTeXa partir de la descripción detallada de metamodelos EMF. Es posible además enriquecer la documentación generada mediante la incorporación de nuevas métricas de los modelos además de las que ya se muestran en la sección *resumen*.

6.3 Contribuciones

Para facilitar la valoración de los resultados en el contexto de la investigación sobre el trabajo presentado, en esta sección se enumeran las publicaciones relacionadas con este trabajo de tesis.

6.3.1 Revistas Internacionales

- Joaquín Cañadas, José Palma, Samuel Túnez. **Defining the semantics of rule-based Web applications through model-driven development**. International Journal of Applied Mathematics and Computer Science 21(1):41–55, March, 2011.
  JCR Impact Factor 2011: 0,487 (Q3)

- Isabel María del Águila, Joaquín Cañadas, Samuel Túnez. **Decision making models embedded into a web-based tool for**

assessing pest infestation risk. Biosystems Engineering 133: 102–115, May, 2015.
JCR Impact Factor 2014: 1.619 (Q1)

6.3.2 Congresos Internacionales

- Joaquín Cañadas, José Palma, Samuel Túnez. **From Knowledge Level to Symbolic Level. A Metamodeling Approach**. In Proceedings of Knowledge Engineering and Software Engineering Workshop (KESE2005) at the 28th German Conference on Artificial Intelligence (KI-2005). Fachberichte Informatik 8–2005, University Koblenz-Landau. Joachim Baumeister, Dietmar Seipel (Eds.). ISSN 1860-4471. Koblenz, Germany. pp 13-23. 2005

- Isabel María del Águila, Joaquín Cañadas, José Palma, Samuel Túnez. **Towards a Methodology for Hybrid Systems Software Development**. Proceedings of the Eighteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2006). pp. 188-193, San Francisco, USA, July, 2006.

- Joaquín Cañadas, José Palma, Samuel Túnez. **InSCo-Gen: A MDD Tool for Web Rule-Based Applications**. Web Engineering, 9th International Conference, ICWE 2009. Lecture Notes in Computer Science 5648:523-526, San Sebastián, Spain, June, 2009.

- Joaquín Cañadas, José Palma, Samuel Túnez. **A Model-Driven Method for automatic generation of Rule-Based Web applications**. Proceedings of the 5th Workshop on Knowledge Engineering and Software Engineering (KESE-2009). CEUR Workshop Proceedings Vol-486:13-24, Paderborn, Germany, September, 2009

- Joaquín Cañadas, F. Javier Orellana, Raquel Salmerón, Alfonso Bosch. **RIAs for Collaborative Knowledge Modeling. An application in Agriculture**. Proceedings of The 13th International Business Information Management Association Conference (IBIMA'09) pp. 1448-1452, Marrakech, Morocco, November, 2009.

- Joaquín Cañadas, F. Javier Orellana, Raquel Salmerón, Samuel Túnez. **RuleModeler: a rich internet application for knowledge modeling**. I-SEMANTICS '10: Proceedings of the 6th In-

ternational Conference on Semantic Systems, Graz, Austria, September, 2010

- Joaquín Cañadas, José Palma, Samuel Túnez. **A MDD approach for generating Rule-based Web Applications from OWL and SWRL**. Proceedings of the 3rd Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE 2010). CEUR Workshop Proceedings Vol-604, Málaga, Spain, June, 2010

- Joaquín Cañadas, José Palma, Samuel Túnez. **A Tool for MDD of Rule-based Web Applications based on OWL and SWRL**. Proceedings of the 6th Workshop on Knowledge Engineering and Software Engineering (KESE-2010). CEUR Workshop Proceedings Vol-636, Karlsruhe, Germany, September, 2010

- Joaquín Cañadas, José Palma, Samuel Túnez. **Model-Driven Rich User Interface Generation from Ontologies for Data-Intensive Web Applications**. Proceedings of the 7th Workshop on Knowledge Engineering and Software Engineering (KESE7). CEUR Workshop Proceedings Vol-805, La Laguna (Tenerife), Spain, November, 2011

6.3.3 Congresos Nacionales

- Joaquín Cañadas, Samuel Túnez, José Palma. **Aplicación de MDA a la Ingeniería del Conocimiento**. En Actas del I Taller sobre Desarrollo Dirigido por Modelos, MDA y Aplicaciones (DSDM'04). pp. 33-42. ISBN:84-688-8870-2. Málaga (Spain), Noviembre, 2004.

- Joaquín Cañadas, Isabel María del Águila, Alfonso Bosch, Samuel Túnez. **Un sistema de ayuda a la toma de decisiones para control fitosanitario en agricultura**. En Actas de la XI Conferencia de la Asociación Española para la Inteligencia Artificial CAEPIA 2005. pp 391-400. Santiago de Compostela, España. Noviembre 2005.

- Joaquín Cañadas, Isabel Mª Segovia, Francisco Rodríguez, Samuel Túnez. **Aplicación de los sistemas expertos en la supervisión y control de cultivos bajo invernadero**. Actas del V Con-

greso Nacional y II Congreso Ibérico AGROINGENIERÍA 2009. pp. 413-414, Lugo, España, Septiembre, 2009

- Joaquín Cañadas, F. Javier Orellana, Isabel María del Águila, José Palma, Samuel Túnez. **A Tool Suite for Hybrid Intelligence Information Systems**. Actas de la XIII Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA-TTIA 2009) pp. 147-156, Sevilla, España, November, 2009

Bibliografía

Abdullah, Mohd Syazwan; Benest, Ian D.; Paige, Richard F. y Kimble, Chris: «Using Unified Modeling Language for Conceptual Modelling of Knowledge-Based Systems». En: Christine Parent; Klaus-Dieter Schewe; Veda C. Storey y Bernhard Thalheim (Eds.), *26th International Conference on Conceptual Modeling, ER 2007, Auckland, New Zealand*, volumen 4801 de *Lecture Notes in Computer Science*, pp. 438–453. Springer. ISBN 978-3-540-75562-3, 2007.

Águila, Isabel María del: *InSCo. Un modelo de proceso para la integración entre la ingeniería del software y la ingeniería del conocimiento.* Tesis doctoral, Universidad de Almería, 2010.

Águila, Isabel María del; Cañadas, Joaquín; Bosch, Alfonso; Túnez, Samuel y Marín, Roque: «Knowledge Model of a Therapy Administration Task - Applied to an Agricultural Domain». En: Vasile Palade; Robert J. Howlett y Lakhmi C. Jain (Eds.), *Knowledge-Based Intelligent Information and Engineering Systems, 7th International Conference, KES 2003, Oxford, UK*, volumen 2774 de *Lecture Notes in Computer Science*, pp. 1277–1283. Springer. ISBN 3-540-40804-5, 2003.

Águila, Isabel María del; Cañadas, Joaquín; Palma, José y Túnez, Samuel: «Towards a Methodology for Hybrid Systems Software Development». En: Kang Zhang; George Spanoudakis y Giuseppe Visaggio (Eds.), *Proceedings of the Eighteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2006)*, pp. 188–193. ISBN 1-891706-18-7, 2006.

Águila, Isabel María del; Cañadas, Joaquín y Túnez, Samuel: «Decision making models embedded into a web-based tool for asses-

sing pest infestation risk». *Biosystems Engineering*, 2015, **133(0)**, pp. 102 – 115. ISSN 1537-5110.

Águila, Isabel María del; Palma, José y Túnez, Samuel: «Milestones in software engineering and knowledge engineering history: A comparative review». *The Scientific World Journal*, 2014, **2014**.

Águila, Isabel María del; Túnez, Samuel; Cañadas, Joaquín; Bosch, Alfonso y Marín, Roque: «A Proposal for Project Management Using CommonKADS». En: Roberto Moreno-Díaz; Bruno Buchberger y José Luis Freire (Eds.), *Computer Aided Systems Theory - EUROCAST 2001*, volumen 2178 de *Lecture Notes in Computer Science*, pp. 160–171. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42959-3.

Albert, Manoli; Cabot, Jordi; Gómez, Cristina y Pelechano, Vicente: «Automatic generation of basic behavior schemas from UML class diagrams». *Software & Systems Modeling*, 2010, **9(1)**, pp. 47–67. ISSN 1619-1366.

Alonso, F.; Juristo, N. y Pazos, J.: «Trends in life-cycle models for SE and KE: Proposal for a spiral-conical life-cycle approach». *International Journal of Software Engineering and Knowledge Engineering*, 1995, **5(3)**, pp. 445–465. ISSN 0218-1940.

Alonso, Fernando; Juristo, N; Maté, José Luis y Pazos, Juan: «Software engineering and knowledge engineering: Towards a common life cycle». *Journal of Systems and Software*, 1996, **33(1)**, pp. 65–79.

Alonso Betanzos, Amparo; Guijarro Berdiñas, Bertha; Lozano Tello, Adolfo y Palma Méndez, José Tomas: *Ingeniería del conocimiento: aspectos metodológicos*. Pearson Educación, 2004. ISBN 9788420541921.

Angele, J.; Fensel, D.; Landes, D. y Studer, R.: «Developing Knowledge-Based Systems with MIKE». *Journal of Automated Software Engineering*, 1998, **5(4)**, pp. 389–418.

Antonopoulou, E.; Karetsos, S.T.; Maliappis, M. y Sideridis, A.B.: «Web and mobile technologies in a prototype DSS for major field crops». *Computers and Electronics in Agriculture*, 2010, **70(2)**, pp. 292 – 301. ISSN 0168-1699.

Bailey, James; Maier, David; Schewe, Klaus-Dieter; Thalheim, Bernhard y Wang, Xiaoyang Sean (Eds.): *Web Information Systems Engineering - WISE 2008, 9th International Conference, Auckland, New Zealand, September 1-3, 2008. Proceedings*, volumen 5175 de *Lecture Notes in Computer Science*. Springer, 2008. ISBN 978-3-540-85480-7.

Baker, Paul; Dai, Zhen Ru; Grabowski, Jens; Schieferdecker, Ina y Williams, Clay: *Model-driven testing: Using the UML testing profile*. Springer Science & Business Media, 2007.

Baresi, Luciano; Garzotto, Franca y Paolini, Paolo: «From Web Sites to Web Applications: New Issues for Conceptual Modeling». En: Stephen W. Liddle; Heinrich C. Mayr y Bernhard Thalheim (Eds.), *Conceptual Modeling for E-Business and the Web, ER 2000 Workshops on Conceptual Modeling Approaches for E-Business and The World Wide Web and Conceptual Modeling, Salt Lake City, Utah, USA, October 9-12, 2000, Proceedings*, volumen 1921 de *Lecture Notes in Computer Science*, pp. 89–100. Springer. ISBN 3-540-41073-2, 2000.

Beck, Howard W.; Jones, Pierce y Jones, J.W.: «SOYBUG: An expert system for soybean insect pest management». *Agricultural Systems*, 1989, **30(3)**, pp. 269 – 286. ISSN 0308-521X.

Beizer, Boris: *Software testing techniques*. Dreamtech Press, 2002.

Berners-Lee, Tim; Hendler, James y Lassila, Ora: «The Semantic Web». *Scientific American*, 2001, **284(5)**, pp. 34–43.

Berners-Lee, Tim J: «The world-wide web». *Computer Networks and ISDN Systems*, 1992, **25(4)**, pp. 454–459.

Boley, Harold; Tabet, Said y Wagner, Gerd: «Design Rationale for RuleML: A Markup Language for Semantic Web Rules». En: Isabel F. Cruz; Stefan Decker; Jérôme Euzenat y Deborah L. McGuinness (Eds.), *Proceedings of SWWS'01, The first Semantic Web Working Symposium, Stanford University, California, USA*, pp. 381–401, 2001.

Brachman, Ronald J. y Levesque, Hector J.: *Knowledge representation and reasoning*. Morgan Kaufmann, San Francisco, CA, USA, 2004.

Brambilla, Marco; Cabot, Jordi y Wimmer, Manuel: *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2012.

Bressan, Stéphane; Goh, Cheng Hian; Levina, Natalia; Madnick, Stuart E.; Shah, Ahmed y Siegel, Michael: «Context Knowledge Representation and Reasoning in the Context Interchange System». *Applied Intelligence*, 2000, **13(2)**, pp. 165–180.

Buchanan, Bruce G; Barstow, David; Bechtal, Robert; Bennett, James; Clancey, William; Kulikowski, Casimir; Mitchell, Tom y Waterman, Donald A: «Constructing an expert system». *Building expert systems*, 1983, **50**, pp. 127–167.

Buchanan, Bruce G. y Shortliffe, Edward H.: *Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. ISBN 0201101726.

Busato, Patrizia y Berruto, Remigio: «A web-based tool for biomass production systems». *Biosystems Engineering*, 2014, **120**, pp. 102–116.

Busato, Patrizia; Sørensen, Claus G.; Pavlou, Dimitris; Bochtis, Dionysis D.; Berruto, Remigio y Orfanou, Anna: «DSS tool for the implementation and operation of an umbilical system applying organic fertiliser». *Biosystems Engineering*, 2013, **114(1)**, pp. 9–20.

Cabot, Jordi; García Molina, Jesús y Rossi, Gustavo: «Presentación. Una introducción a MDE y su creciente adopción industrial». *Novatica*, 2014, **(228)**, pp. 5 – 9.

Cañadas, Joaquín; Águila, Isabel María del; Bosch, Alfonso y Túnez, Samuel: «An Intelligent System for Therapy Control in a Distributed Organization». En: Hassan Shafazand y A. Min Tjoa (Eds.), *EurAsia-ICT 2002: Information and Communication Technology, First EurAsian Conference, Shiraz, Iran*, volumen 2510 de *Lecture Notes in Computer Science*, pp. 19–26. Springer. ISBN 3-540-00028-3, 2002.

Cañadas, Joaquín; Orellana, Francisco Javier; Salmerón, Raquel y Túnez, Samuel: «RuleModeler: a rich internet application

for knowledge modeling». En: Adrian Paschke; Nicola Henze y Tassilo Pellegrini (Eds.), *Proceedings the 6th International Conference on Semantic Systems, I-SEMANTICS 2010, Graz, Austria, September 1-3, 2010*, ACM International Conference Proceeding Series. ACM. ISBN 978-1-4503-0014-8, 2010.

Cañadas, Joaquín; Palma, José y Túnez, Samuel: «InSCo-Gen: A MDD Tool for Web Rule-Based Applications». En: Martin Gaedke; Michael Grossniklaus y Oscar Díaz (Eds.), *Web Engineering, 9th International Conference, ICWE 2009, San Sebastián, Spain*, volumen 5648 de *Lecture Notes in Computer Science*, pp. 523–526. Springer. ISBN 978-3-642-02817-5, 2009.

Cardona, C.M.; Martin, C.; Salterain, A.; Castro, A.; Martín, D. San y Ayesa, E.: «CALHIDRA 3.0 – New software application for river water quality prediction based on RWQM1». *Environmental Modelling & Software*, 2011, **26(7)**, pp. 973 – 979. ISSN 1364-8152.

Carroll, Jeremy J; Dickinson, Ian; Dollin, Chris; Reynolds, Dave; Seaborne, Andy y Wilkinson, Kevin: «Jena: implementing the semantic web recommendations». En: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pp. 74–83, 2004.

Ceri, S.; Fraternali, P. y Matera, M.: «Conceptual modeling of data-intensive Web applications». *Internet Computing, IEEE*, 2002a, **6(4)**, pp. 20–30. ISSN 1089-7801.

Ceri, Stefano; Fraternali, Piero; Bongio, Aldo; Brambilla, Marco; Comai, Sara y Matera, Maristella: *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2002b. ISBN 9781558608436.

Chandrasekharan, B: «Generic tasks in knowledge-based reasoning: High-level building blocks for expert systems design». *IEEE Expert, Fall*, 1986, pp. 125–164.

Chauhan, Yashvir; Wright, Graeme; Holzworth, Dean; Rachaputi, Rao y Payero, José: «AQUAMAN: a web-based decision support system for irrigation scheduling in peanuts». *Irrigation Science*, 2011, pp. 1–13. ISSN 0342-7188.

Chaur G. Wu: «Modeling Rule-Based Systems with EMF». Eclipse Corner Articles http://www.eclipse.org/articles/, 2004.

Corcho, Oscar; Fernández-López, Mariano y Gómez-Pérez, Asunción: «Methodologies, tools and languages for building ontologies. Where is their meeting point?» *Data & Knowledge Engineering*, 2003, **46(1)**, pp. 41 – 64. ISSN 0169-023X.

Dean, Mike; Schreiber, Guus; Bechhofer, Sean; van Harmelen, Frank; Hendler, Jim; Horrocks, Ian; McGuinness, Deborah L.; Patel-Schneider, Peter F. y Stein, Lynn Andrea: *OWL Web Ontology Language Reference.* W3C Recommendation, 2004. Available at http://www.w3.org/TR/owl-ref.

Dean, Mike; Schreiber, Guus; Bechhofer, Sean; van Harmelen, Frank; Hendler, Jim; Horrocks, Ian; McGuinness, Deborah L.; Patel-Schneider, Peter F. y Stein, Lynn Andrea: *OWL 2 Web Ontology Language Document Overview (Second Edition).* W3C Recommendation, 2012. Available at http://www.w3.org/TR/owl2-overview/.

Diouf, Mouhamed; Maabout, Sofian y Musumbu, Kaninda: «Merging Model Driven Architecture and Semantic Web for Business Rules Generation». En: Massimo Marchiori; Jeff Z. Pan y Christian de Sainte Marie (Eds.), *Web Reasoning and Rule Systems, First International Conference, RR 2007, Innsbruck , Austria*, volumen 4524 de *Lecture Notes in Computer Science*, pp. 118–132. Springer. ISBN 978-3-540-72981-5, 2007.

Driver, M.; Valdes, R. y Phifer, G.: «Rich Internet Applications are the next evolution of the Web». *Informe técnico*, Gartner Research Note. G, 2005.

Duan, Y.; Edwards, J.S. y Xu, M.X.: «Web-based expert systems: benefits and challenges». *Information & Management*, 2005, **42(6)**, pp. 799 – 811. ISSN 0378-7206.

Durkin, John: *Expert Systems: Catalog of Applications.* Akron (Ohio), Intelligent Computer Systems Inc., 1993.

Duvall, Paul M; Matyas, Steve y Glover, Andrew: *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007.

EC: «Directive 2009/128/EC of the European Parliament and of the Council of 21 October 2009 establishing a framework for Community action to achieve the sustainable use of pesticides». *Official Journal European Union*, 2009, **L309**, pp. 71–86.

Eclipse Modeling Project: «JET - Java Emitter Templates». http://www.eclipse.org/modeling/m2t/?project=jet, 2007.

Ed-Douibi, Hamza; Álvarez, Camilo; Cánovas, Javier y Cabot, Jordi: «EMF-REST». http://emf-rest.com, 2014.

Efftinge, Sven y Völter, Markus: «oAW xText: A framework for textual DSLs». En: *Workshop on Modeling Symposium at Eclipse Summit*, volumen 32, p. 118, 2006.

Eiter, Thomas; Ianni, Giovambattista; Krennwallner, Thomas y Polleres, Axel: «Rules and Ontologies for the Semantic Web». En: *Reasoning Web*, pp. 1–53, 2008.

Epsilon: «Epsilon». http://www.eclipse.org/epsilon/, 2007.

Eriksson, H.: «Using JessTab to integrate Protege and Jess». *Intelligent Systems, IEEE*, 2003, **18(2)**, pp. 43–50. ISSN 1541-1672.

Eriksson, Henrik; Shahar, Yuval; Tu, Samson W.; Puerta, Angel R. y Musen, Mark A.: «Task modeling with reusable problem-solving methods». *Artificial Intelligence*, 1995, **79(2)**, pp. 293–326. ISSN 0004-3702.

Ernst, Erik: «Separation of concerns». En: *Proceedings of the AOSD 2003 Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), Boston, MA, USA*, , 2003.

Fensel, Dieter: *Knowledge Acquisition and Representation Language Karl*. Kluwer Academic Publishers, 1995. ISBN 0792396014.

Fernández-López, Mariano y Gómez-Pérez, Asunción: «Overview and analysis of methodologies for building ontologies». *The Knowledge Engineering Review*, 2003, **17(02)**, pp. 129–156.

Forgy, Charles L: «Rete: A fast algorithm for the many pattern/-many object pattern match problem». *Artificial intelligence*, 1982, **19(1)**, pp. 17–37.

Frankel, D.; Hayes, P.; Kendall, E. y McGuinness, D.: «The Model Driven Semantic Web». En: *1st International Workshop on the Model-Driven Semantic Web (MDSW2004), Monterey, California, USA*, , 2004.

Friedman-Hill, Ernest: *Jess in Action: Java Rule-Based Systems.* Manning Publications, Greenwich, CT, USA, 2003. ISBN 1930110898.

García, J.; García, F.O.; Pelechano, V.; Vallecillo, A.; Vara, J.M. y Vicente-Chicote, C.: *DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS: CONCEPTOS, MÉTODOS Y HERRAMIENTAS.* RA-MA, 2013.

Gašević, Dragan; Djurić, Dragan y Devedžić, Vladan: *Model Driven Architecture and Ontology Development.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540321802.

Geary, David y Horstmann, Cay S.: *Core JavaServer Faces.* Prentice Hall, 2ª edición, 2007. ISBN 0131738860.

Giarratano, Joseph C. y Riley, Gary D.: *Expert Systems: Principles and Programming, Fourth Edition.* Course Technology, Boston, 4 editionª edición, 2004. ISBN 9780534384470.

Gómez, A.; Juristo, N.; Montes, C. y Pazos, J.: *Ingeniería del Conocimiento.* Editorial Centro de estudios Ramón Areces, 1997.

Gómez-Pérez, Asunción; Fernández-López, Mariano y Corcho, Oscar: *Ontological Engineering.* Springer, 2004. ISBN 1852335513, 9781852335519.

Gonzalez-Andujar, J. L.; Fernandez-Quintanilla, C.; Izquierdo, J. y Urbano, J. M.: «SIMCE: An expert system for seedling weed identification in cereals». *Computers and electronics in agriculture*, 2006, **54(2)**, pp. 115–123. ISSN 0168-1699.

Groenewegen, Danny M.; Hemel, Zef; Kats, Lennart C. L. y Visser, Eelco: «WebDSL: a domain-specific language for dynamic web applications». En: Gail E. Harris (Ed.), *23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, Nashville, TN, USA*, pp. 779–780. ACM. ISBN 978-1-60558-220-7, 2008.

GROVE, RALPH: «Internet-based expert systems». *Expert Systems*, 2000, **17(3)**, pp. 129–135. ISSN 1468-0394.

GRUBER, THOMAS R: «Toward principles for the design of ontologies used for knowledge sharing?» *International journal of human-computer studies*, 1995, **43(5)**, pp. 907–928.

GRUNINGER, M.; FOX, M.S. y cols.: «Methodology for the Design and Evaluation of Ontologies». En: *Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing, IJCAI-95*, volumen 95, pp. 6.1–6.10, 1995.

GUIDA, GIOVANNI y TASSO, CARLO: *Design and development of knowledge-based systems: from life cycle to methodology.* John Wiley & Sons, Inc., Chichester, UK, 1994. ISBN 0471928089.

HAYES-ROTH, FREDERICK; WATERMAN, DONALD y LENAT, DOUGLAS: «Building expert systems», 1984.

HERNÁNDEZ, J LUIS MATE y SIERRA, JUAN PAZOS: *Ingeniería del conocimiento: Diseño y construcción de sistemas expertos*, 1988.

HORROCKS, I.; PATEL-SCHNEIDER, P.F.; BOLEY, H.; TABET, S.; GROSOF, B. y DEAN, M.: «SWRL: A Semantic Web Rule Language combining OWL and RuleML». http://www.w3.org/Submission/SWRL/, 2004.

HUMBLE, JEZ y FARLEY, DAVID: *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education, 2010.

HUNTINGTON, DUSTIN: «Web-based Expert Systems Are on the Way: Java-based Web Delivery». *PC AI*, 2000, **14(6)**, pp. 34–36. ISSN 0894-0711.

HUSSMANN, H.; MEIXNER, G. y ZUEHLKE, D.: *Model-Driven Development of Advanced User Interfaces.* Springer-Verlag New York Inc, 2011.

IEEE: *IEEE Standards Software Engineering. Customer and Terminology Standards.* volumen 1, 1999.

IIBA: *Guide to the Business Analysis Body of Knowledge.* volumen 3, 2015.

JBoss: «Drools documentation». http://www.drools.org/learn/documentation.html, 2009.

JBoss: «RichFaces». http://www.jboss.org/jbossrichfaces/, 2009.

Jouault, Frédéric; Allilaire, Freddy; Bézivin, Jean y Kurtev, Ivan: «ATL: A model transformation tool». *Science of Computer Programming*, 2008, **72(1)**, pp. 31–39. ISSN 0167-6423. Special Issue on Second issue of experimental software and toolkits (EST).

Kifer, Michael: «Rule Interchange Format: The Framework». En: Diego Calvanese y Georg Lausen (Eds.), *Web Reasoning and Rule Systems, Second International Conference, Karlsruhe, Germany*, volumen 5341 de *Lecture Notes in Computer Science*, pp. 1–11. Springer. ISBN 978-3-540-88736-2, 2008.

Koch, Nora; Knapp, Alexander; Zhang, Gefei y Baumeister, Hubert: «UML-Based Web Engineering. An Approach Based on Standards». En: Gustavo Rossi; Oscar Pastor; Daniel Schwabe y Luis Olsina (Eds.), *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pp. 157–191. Springer London, 2008. ISBN 978-1-84628-923-1.

Kogut, Paul; Cranefield, Stephen; Hart, Lewis; Dutra, Mark; Baclawski, Kenneth; Kokar, Mieczyslaw y Smith, Jeffrey: «UML for Ontology Development». *The Knowledge Engineering Review*, 2002, **17(01)**, pp. 61–64.

Kolovos, Dimitrios S.: «Exeed: EXtended Emf EDitor - User Manual», 2007.

Lenat, Douglas B: «CYC: a large-scale investment in knowledge infrastructure». *Commun. ACM*, 1995, **38**, p. 3338. ISSN 0001-0782.

Linaje, Marino; Preciado, Juan Carlos y Sanchez-Figueroa, Fernando: «Engineering Rich Internet Application User Interfaces over Legacy Web Models». *Internet Computing, IEEE*, 2007, **11(6)**, pp. 53–59. ISSN 1089-7801.

Lopez-Morales, Virgilio; López-Ortega, Omar; Ramos-Fernández, J. y Muñoz, L. B.: «JAPIEST: An integral intelligent system for the diagnosis and control of tomatoes diseases and pests in hydroponic greenhouses». *Expert Syst. Appl.*, 2008, **35(4)**, pp. 1506–1512.

Lukichev, Sergey y Wagner, Gerd: «Visual Rules Modeling». En: Irina Virbitskaite y Andrei Voronkov (Eds.), *Perspectives of Systems Informatics*, volumen 4378 de *Lecture Notes in Computer Science*, pp. 467–473. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-70880-3.

Mansingh, Gunjan; Reichgelt, Han y Bryson, Kweku-Muata Osei: «CPEST: An expert system for the management of pests and diseases in the Jamaican coffee industry». *Expert Systems with Applications*, 2007, **32(1)**, pp. 184 – 192. ISSN 0957-4174.

McGrath, Michael y McGrath, Michael P: *Understanding PaaS*. .º'Reilly Media, Inc.", 2012.

Mellor, Stephen J.; Clark, Anthony N. y Futagami, Takao: «Guest Editors' Introduction: Model-Driven Development». *IEEE Software*, 2003, **20(5)**, pp. 14–18.

Moreno, Nathalie; Meliá, Santiago; Koch, Nora y Vallecillo, Antonio: «Addressing New Concerns in Model-Driven Web Engineering Approaches». En: (Bailey y cols., 2008), pp. 426–442, 2008a.

Moreno, Nathalie; Romero, Jose Raul y Vallecillo, Antonio: «An Overview of Model-Driven Web Engineering and the MDA». En: Gustavo Rossi; Oscar Pastor; Daniel Schwabe y Luis Olsina (Eds.), *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pp. 353–382. Springer London, 2008b.

Motik, Boris; Sattler, Ulrike y Studer, Rudi: «Query Answering for OWL-DL with rules». *Journal of Web Semantics*, 2005, **3(1)**, pp. 41–60.

Motta, Enrico: «An overview of the OCML modelling language». En: *8th Workshop on Knowledge Engineering Methods and Languages (KEML '98)*, , 1998.

Musset, Jonathan; Juliot, Étienne; Lacrampe, Stéphane; Piers, William; Brun, Cédric; Goubet, Laurent; Lussaud, Yvan y Allilaire, Freddy: «Acceleo user guide», 2006.

Nalepa, Grzegorz J.; Kutt, Krzysztof y Kaczor, Krzysztof: «Can the Generation of Test Cases for Unit Testing be Automated

with Rules?» En: Leszek Rutkowski; Marcin Korytkowski; Rafal Scherer; Ryszard Tadeusiewicz; Lotfi A. Zadeh y Jacek M. Zurada (Eds.), *Artificial Intelligence and Soft Computing - 13th International Conference, ICAISC 2014, Zakopane, Poland, June 1-5, 2014, Proceedings, Part II*, volumen 8468 de *Lecture Notes in Computer Science*, pp. 536–547. Springer. ISBN 978-3-319-07175-6, 2014.

Nalepa, Grzegorz J. y Ligeza, Antoni: «The HeKatE methodology. Hybrid engineering of intelligent systems». *Applied Mathematics and Computer Science*, 2010, **20(1)**, pp. 35–53.

Neches, R.; Fikes, R.; Finin, T.; Gruber, T.; Patil, R.; Senator, T. y Swartout, W.: «Enabling Technology for Knowledge Sharing». *AI Magazine*, 1991, pp. 36–56.

Newell, A. y Simon, H.: *Production systems: Models of Control Structure.* Academic Press, 1973.

Newell, Allen: «The knowledge level». *Artificial Intelligence*, 1982, **18(1)**, pp. 87–127. ISSN 0004-3702.

Nicola, Antonio De; Missikoff, Michele y Navigli, Roberto: «A software engineering approach to ontology building». *Information Systems*, 2009, **34(2)**, pp. 258 – 275. ISSN 0306-4379.

Norris, Robert F.; Caswell-Chen, Edward P. y Kogan, Marcos: *Concepts in integrated pest management.* Prentice Hall PTR, 2003. ISBN 9780130870162.

Object Management Group: «"Model Driven Architecture - A Technical Perspective». OMG document: ormsc/01-07-01, 2001.

Object Management Group: «MDA Guide Version 1.0.1». OMG document: omg/2003-06-01, 2003a.

Object Management Group: «Meta Object Facility (MOF) 2.0 Core Specification». OMG document: ptc/03-10-04, 2003b.

Object Management Group: «UML 2.0 Infrastructure Specification». OMG document: ptc/03-09-15, 2003c.

Object Management Group: «MOF Model to Text Transformation Language, v1.0». http://www.omg.org/spec/MOFM2T/1.0/, 2008a.

Object Management Group: «Query/View/Transformation, v1.1». http://www.omg.org/spec/QVT/1.1/, 2008b.

Object Management Group: «Semantics of Business Vocabulary and Business Rules (SBVR)». http://www.omg.org/spec/SBVR/1.0, 2008c.

Object Management Group: «Ontology Definition Metamodel. Version 1.0». http://www.omg.org/spec/ODM/1.0/, 2009a.

Object Management Group: «Production Rule Representation (PRR)». http://www.omg.org/spec/PRR/1.0/, 2009b.

Object Management Group: «Object Constraint Language (OCL)». http://www.omg.org/spec/OCL/, 2012.

Object Management Group: «Semantics of Business Vocabulary and Business Rules (SBVR) V1.2». http://www.omg.org/spec/SBVR/1.2, 2013.

Oracle: «Java Platform, Enterprise Edition (Java EE)». http://www.oracle.com/technetwork/java/javaee, 2009.

Orellana, Francisco Javier; Cañadas, Joaquín; Águila, Isabel María del y Túnez, Samuel: «INSCO Requisite - A Web-Based RM-Tool to support Hybrid Software Development». En: José Cordeiro y Joaquim Filipe (Eds.), *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume ISAS-1, Barcelona, Spain, June 12-16, 2008*, pp. 326–329. ISBN 978-989-8111-38-8, 2008.

Orellana, Francisco Javier; Sagrado, José del y Águila, Isabel María del and: «SAIFA: A web-based system for Integrated Production of olive cultivation». *Computers and Electronics in Agriculture*, 2011, **78(2)**, pp. 231–237.

Palma, J.T.; Paniagua, E.; Martin, F. y Marin, R.: «Ingeniería del Conocimiento. De la Extracción al Modelado de Conocimimiento». *Revista Iberoamericana de Inteligencia Artificial*, 2000, **4(11)**, pp. 46–72.

Paptaxiarhis, Vassilis; Tsetsos, Vassileios; Karali, Isambo y Stamotopoulos, Panagiotis: «Developing Rule-Based Web Applications: Methodologies and Tools». En: *Handbook of Research on*

Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, pp. 371–392. IGI Global, 2009. ISBN 9781605664026.

Parreiras, Fernando Silva y Staab, Steffen: «Using ontologies with UML class-based modeling: The TwoUse approach». *Data & Knowledge Engineering*, 2010, **69(11)**, pp. 1194–1207.

Parreiras, Fernando Silva; Staab, Steffen y Winter, Andreas: «TwoUse: Integrating UML models and OWL ontologies». *Informe técnico*, Fachbereich Informatik 16/2007, Universität Koblenz-Landau, 2007.

Plant, R.E.: «An integrated expert decision support system for agricultural management». *Agricultural Systems*, 1989, **29(1)**, pp. 49–66. ISSN 0308-521X.

Ribarić, Marko; Gašević, Dragan; Milanović, Milan; Giurca, Adrian; Lukichev, Sergey y Wagner, Gerd: «Model-Driven Engineering of Rules for Web Services». En: Ralf Lämmel; Joost Visser y João Saraiva (Eds.), *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal*, volumen 5235 de *Lecture Notes in Computer Science*, pp. 377–395. Springer. ISBN 978-3-540-88642-6, 2007.

Ross, Ronald G: *Principles of the business rule approach.* Addison-Wesley Professional, 2003.

Rossi, Gustavo; Pastor, Oscar; Schwabe, Daniel y Olsina, Luis (Eds.): *Web Engineering: Modelling and Implementing Web Applications.* Human-Computer Interaction Series. Springer London, London, 2008. ISBN 978-1-84628-922-4.

Russell, Stuart J. y Norvig, Peter: *Artificial intelligence: a modern approach.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995. ISBN 0-13-103805-2.

Schmidt, D.C.: «Guest Editor's Introduction: Model-Driven Engineering». *Computer*, 2006, **39(2)**, pp. 25–31.

Schreiber, Guus; Akkermans, Hans; Anjewierden, Anjo; de Hoog, Robert; Shadbolt, Nigel; de Velde, Walter Van y Wielinga, Bob: *Knowledge Engineering and Management: The CommonKADS Methodology.* The MIT Press, Cambridge, 2000. ISBN 0262193000.

Simpson, J; Kingston, J y Molony, N: «Internet-based decision support for evidence-based medicine». *Knowledge-Based Systems*, 1999, **12(5-6)**, pp. 247–255. ISSN 0950-7051.

Sirin, Evren; Parsia, Bijan; Grau, Bernardo Cuenca; Kalyanpur, Aditya y Katz, Yarden: «Pellet: A practical OWL-DL reasoner». *Web Semantics: Science, Services and Agents on the World Wide Web*, 2007, **5(2)**, pp. 51–53. ISSN 1570-8268. Software Engineering and the Semantic Web.

Slazynski, Mateusz; Bobek, Szymon y Nalepa, Grzegorz J.: «Migration of Rule Inference Engine to Mobile Platform. Challenges and Case Study». En: Grzegorz J. Nalepa y Joachim Baumeister (Eds.), *Proceedings of 10th Workshop on Knowledge Engineering and Software Engineering (KESE10) co-located with 21st European Conference on Artificial Intelligence (ECAI 2014), Prague, Czech Republic, August 19 2014.*, volumen 1289 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.

Smart, John: *Jenkins: the definitive guide*. .º'Reilly Media, Inc.", 2011.

Steinberg, Dave; Budinsky, Frank; Paternostro, Marcelo y Merks, Ed: *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition[a] edición, 2008. ISBN 978-0-321-33188-5.

Studer, Rudi; Benjamins, V Richard y Fensel, Dieter: «Knowledge engineering: principles and methods». *Data & knowledge engineering*, 1998, **25(1)**, pp. 161–197.

Sure, York; Staab, Steffen y Studer, Rudi: «On-To-Knowledge Methodology (OTKM)». En: Steffen Staab y Rudi Studer (Eds.), *Handbook on Ontologies*, International Handbooks on Information Systems, pp. 117–132. Springer, 2004. ISBN 3-540-40834-7.

Vermesan, Anca y Coenen, Frans: *Validation and Verification of Knowledge Based Systems: Theory, Tools and Practice*. Springer Science & Business Media, 2013.

Volter, Marcus: «MD* Best Practices». *Journal of Object Technology*, 2009, **8(6)**, pp. 79–102. ISSN 1660-1769.

Wagner, Gerd; Giurca, Adrian y Lukichev, Sergey: «A Usable Interchange Format for Rich Syntax Rules Integrating OCL, RuleML

and SWRL». En: *Proc. of WSh. Reasoning on the Web (RoW2006)*, , 2006.

Wright, Jevon M. y Dietrich, Jens: «Requirements for Rich Internet Application Design Methodologies». En: (Bailey y cols., 2008), pp. 106–119, 2008.

Xu, L.; Chen, L.; Chen, T. y Gao, Y.: «SOA-based precision irrigation decision support system». *Mathematical and Computer Modelling*, 2011, **54(3-4)**, pp. 944–949.

Yao, J.T. y Yao, Y.Y.: «Web-based support systems». En: *Proceedings of the Workshop on Applications, Products and Services of Web-based Support Systems (WSS-03)*, pp. 1–5, 2003.

Apéndice A
Metamodelo CML

A.1 Introducción

Las figuras A.1 , A.2 , A.3 , A.4 representan el paquete *simpleCML*, que contiene:

- Clases: 35
- Enumeraciones: 7

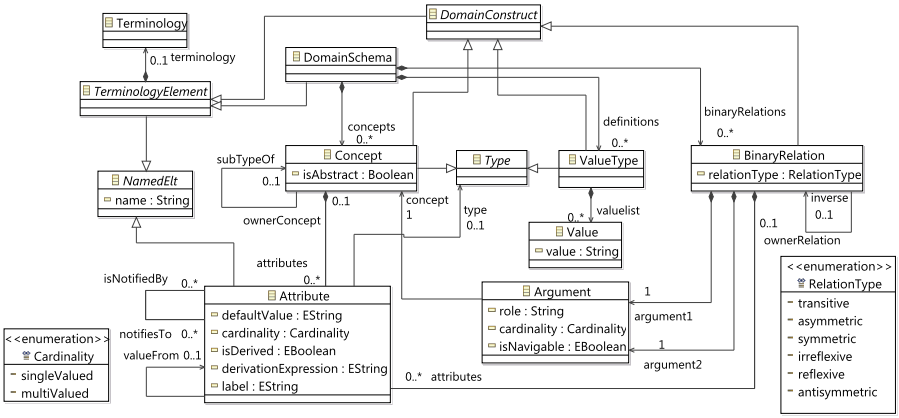


Figura A.1: Metamodelo de CML: Conocimiento del Dominio, Conceptos y Relaciones

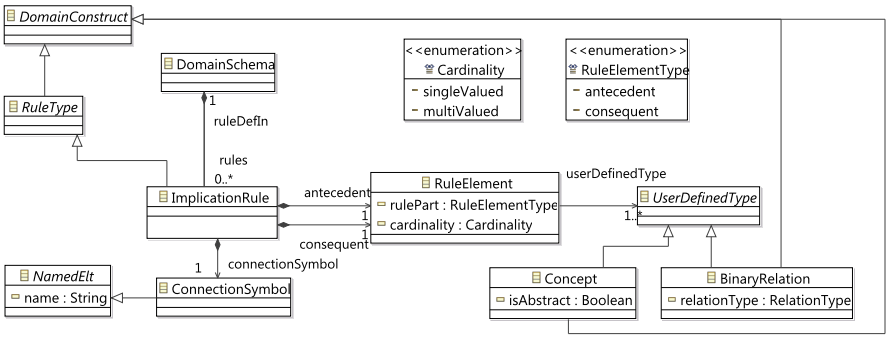


Figura A.2: Metamodelo de CML: Conocimiento del Dominio, Tipos de regla

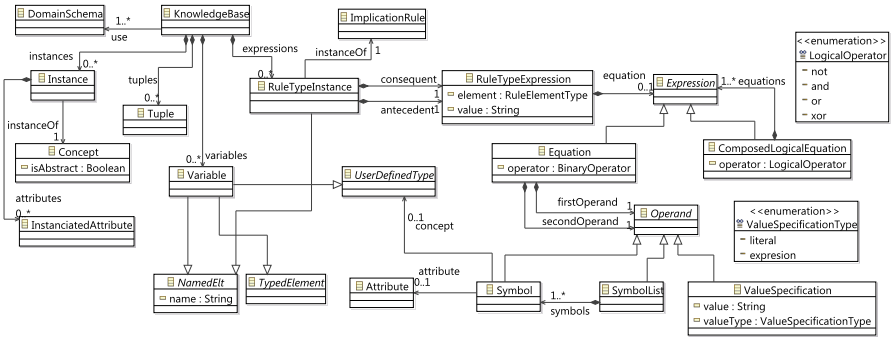


Figura A.3: Metamodelo de CML: Base de conocimiento e Instancias de Tipo de Regla

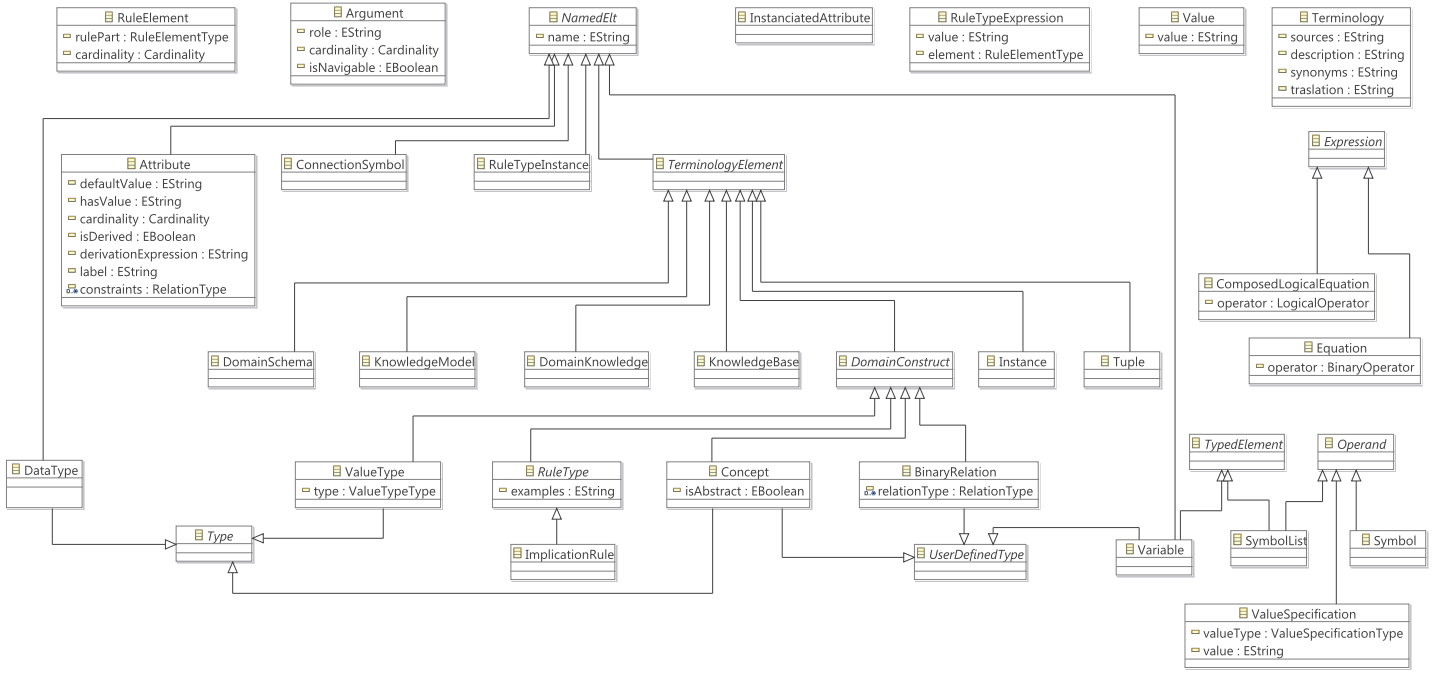


Figura A.4: Metamodelo de CML: Relaciones de herencia (metamodelo completo)

A.2 Paquete *simpleCML*

A.2.1 Metaclases

Clase **Argument**

Esta clase representa un Argumento en una relación Binaria. Está relacionado con un Concepto, al que se le asocia un rol y una cardinalidad en la relación. Los atributos y asociaciones de la clase `Argument` se muestran en la tabla A.1.

Tabla A.1: Propiedades de la clase: `Argument`

Features	*Type*		*Description*
`role`	`EString`	[0..1]	Nombre del rol que recibe el concepto en la relación.
`cardinality`	`Cardinality`	[0..1]	Cardinalidad del Concepto en la relación. Puede ser valor único o múltiple.
`isNavigable`	`EBoolean`	[0..1]	Indica si la relacion es navegable en la dirección del argumento.
`concept`	`Concept`	[1..1]	Concepto asociado al Argumento.

Clase **Attribute**

Esta clase representa un atributo de un Concepto o Relación Binaria. La clase `Attribute` hereda de *`NamedElt`*. Los atributos y asociaciones de la clase `Attribute` se muestran en la tabla A.2.

Clase **BinaryRelation**

Esta clase representa una relación Binaria. Tiene dos Argumentos que representan el Concepto origen y el Concepto destino de la asociación. La clase `BinaryRelation` hereda de *`DomainConstruct`*, *`UserDefinedType`*. Los atributos y asociaciones de la clase `BinaryRelation` se muestran en la tabla A.3.

Clase **ComposedLogicalEquation**

Esta clase premite respresentar una ecuación lógica compuesta, que está formada por varias ecuaciones (simples o a su vez compuestas) unidas por un operador lógico . La clase `ComposedLogicalEquation` hereda de *`Expression`*. Los atributos y asociaciones de la clase `ComposedLogicalEquati` se muestran en la tabla A.4.

Tabla A.2: Propiedades de la clase: `Attribute`

Features	*Type*		*Description*
defaultValue	EString	[0..1]	Valor predeterminado para el atributo.
hasValue	EString	[0..1]	Valor del atributo.
cardinality	Cardinality	[0..1]	cardinalidad del atributo: puede ser valor único o múltiple (lista de valores).
isDerived	EBoolean	[0..1]	Indica si el valor del atributo es derivado a partir de otros valores o reglas.
derivationExpression	EString	[0..1]	Expresión de derivación que describe cómo se obtiene el valor del atributo derivado.
constraints	RelationType	[0..*]	Restriciones asociadas al tipo de relación (para atributos de relaciones).
key	EBoolean	[0..1]	[UI] Permite indicar si el atributo van a ser clave de las instancias del concepto..
label	EString	[0..1]	[UI] Etiqueta asociada al atributo (se mostrará en la interfaz de usuario).
showRelationType	EBoolean	[0..1]	[UI] Indica si se muestra o no el tipo del atributo junto a la etiqueta del nombre del atributo en la interfaz de usuario.
showColumn	EBoolean	[0..1]	[UI] Permite indicar si el atributo se va a visualizar o no como una columna de la tabla que presenta el listado de todas las instancias de la clase a la que pertenece el atributo.
ownerConcept	Concept	[0..1]	Concepto al que pertenece el atributo.
type	Type	[0..1]	Tipo del Atributo.
ownerRelation	BinaryRelation	[0..1]	Relación a la que pertenece el Atributo.
valueFrom	Attribute	[0..1]	[UI] Indica el atributo del que se obtiene el valor.
notifiesTo	Attribute	[0..*]	[UI] Indica que otros atributos deben actualizar su valor cuando el valor del atributo cambia. Esa lista de atributos son los atributos a notificar, que se almacentan en esta propiedad.
isNotifiedBy	Attribute	[0..*]	[UI] Indica que otros atributos notifican al atributo, para que el valor del atributo se actualice al cambiar los valores de los atributos que le notifican.

Clase **`Concept`**

Esta clase representa un concepto del dominio. La clase `Concept` hereda de *`DomainConstruct`*, *`Type`*, *`UserDefinedType`*. Los atributos y asociaciones de la clase `Concept` se muestran en la tabla A.5.

Tabla A.3: Propiedades de la clase: `BinaryRelation`

Features	*Type*	*Description*
relationType	RelationType [0..*]	Tipo de relación binaria. Los valores que puede tomar se definen en el enumerado RelationType.
inverse	BinaryRelation[0..1]	relación inversa.
argument1	Argument [1..1]	Origen de la relación.
argument2	Argument [1..1]	Destino de la relación.
attributes	Attribute [0..*]	Atributos de la relación binaria.

Tabla A.4: Propiedades de la clase: `ComposedLogicalEquation`

Features	*Type*	*Description*
operator	LogicalOperator[0..1]	Operador lógico.
equations	Expression [1..*]	Ecuaciones simples o compuestas que componen la ecuación compuesta.

Tabla A.5: Propiedades de la clase: `Concept`

Features	*Type*	*Description*
isAbstract	EBoolean [0..1]	Indica si el concepto es abstracto (no tiene instancias).
showInMenu	EBoolean [0..1]	[UI] Permite indicar si el nombre de concepto (o clase) va a aparecer en el menú de la aplicación.
icon	EString [0..1]	[UI] Nombre del icono (imagen) que se mostrará junto a las instancias de la clase en los listados.
attributes	Attribute[0..*]	Lista de atributos del Concepto.
subTypeOf	Concept [0..1]	relación de herencia simple. Indica que el concepto es subtipo de otro concepto del dominio.

Clase **ConnectionSymbol**

Palabra o texto que premite expresar la semántica del tipo de regla, relacionando el antecedente con el consecuente en lenguaje natural. La clase `ConnectionSymbol` hereda de *`NamedElt`*. La clase `ConnectionSymbol` no tiene atributos ni asociaciones.

Clase **DataType**

Esta clase representa un tipo de dato del modelo de conocimiento. Un tipo de dato será un tipo cercano a un tipo primitivo.. La clase `DataType` hereda de *`NamedElt`*, *`Type`*. La clase `DataType` no tiene atributos ni asociaciones.

Clase abstracta ***DomainConstruct***

Esta clase abstracta representa un constructor de dominio. Se definen cuatro subclases de constructor de dominio: Concepto, relación binaria, Tipo valor (enumerado), y Tipo de regla. La clase abstracta *DomainConstruct* hereda de *TerminologyElement*. La clase abstracta *DomainConstruct* no tiene atributos ni asociaciones.

Clase **DomainKnowledge**

Esta clase representa un conocimiento del dominio, que forma parte de un modelo de conocimiento. La clase DomainKnowledge hereda de *TerminologyElement*. Los atributos y asociaciones de la clase DomainKnowledge se muestran en la tabla A.6.

Tabla A.6: Propiedades de la clase: DomainKnowledge

Features	*Type*	*Description*
domainSchemas	DomainSchema [0..*]	Esquema(s) de dominio que conforman el conocimiento del dominio.
knowledgeBases	KnowledgeBase[0..*]	Bases de conocimiento asociadas al dominio.

Clase **DomainSchema**

Esta clase representa un esquema de dominio, que se enmarca dentro del conocimiento del dominio. La clase DomainSchema hereda de *TerminologyElement*. Los atributos y asociaciones de la clase DomainSchema se muestran en la tabla A.7.

Tabla A.7: Propiedades de la clase: DomainSchema

Features	*Type*	*Description*
concepts	Concept [0..*]	Conceptos que forman parte del esquema de dominio.
rules	ImplicationRule[0..*]	Tipos de reglas de implicación definidos en el esquema de dominio.
definitions	ValueType [0..*]	Definiciones de Tipos Valor que forman parte del esquema de dominio.
binaryRelations	BinaryRelation [0..*]	Relaciones binarias que forman parte del esquema de dominio.

Clase **Equation**

Esta clase representa una ecuacion o expresión simple, formada por un primer operando, un segundo operando y un operador. La clase `Equation` hereda de *`Expression`*. Los atributos y asociaciones de la clase `Equation` se muestran en la tabla A.8.

Tabla A.8: Propiedades de la clase: `Equation`

Features	*Type*		*Description*
`operator`	`BinaryOperator`	[0..1]	Operador binario.
`firstOperand`	`Operand`	[1..1]	primer operando de la expresión.
`secondOperand`	`Operand`	[1..1]	segundo operando de la expresión.
`implicationRule`	`ImplicationRule`	[1..1]	Tipo de regla asociada a la ecuación.

Clase abstracta ***Expression***

Esta clase abstracta representa una Expresión lógica que forma constituye el antecedente o consecuente de una instancia de regla. Tiene dos subtipos: Ecuación (simple) y Ecuación lógica compuesta. El diseño corresponde al patrón que permite que una ecuación pueda ser una ecuación simple o una composición de ecuaciones simples y compuestas. La clase abstracta *`Expression`* no tiene atributos ni asociaciones.

Clase **ImplicationRule**

Esta clase permite modelar las reglas de implicación (reglas de producción) con un antecedente y un consecuente. La clase `ImplicationRule` hereda de *`RuleType`*. Los atributos y asociaciones de la clase `ImplicationRule` se muestran en la tabla A.9.

Tabla A.9: Propiedades de la clase: `ImplicationRule`

Features	*Type*		*Description*
`connectionSymbol`	`ConnectionSymbol`	[1..1]	Símbolo de conexión del la regla, expresar la semántica que conecta el antecedente con el consecuente.
`antecedent`	`RuleElement`	[1..1]	antecedente de la regla.
`consequent`	`RuleElement`	[1..1]	consecuente de la regla.
`ruleDefIn`	`DomainSchema`	[1..1]	Tipos de reglas que forman parte del esquema de dominio.

Clase **Instance**

Esta clase representa una instancia de un concepto, es decir, lo que se entiende por un objeto oun individuo de una clase. La clase `Instance` hereda de *`TerminologyElement`*. Los atributos y asociaciones de la clase `Instance` se muestran en la tabla A.10.

Tabla A.10: Propiedades de la clase: `Instance`

Features	*Type*	*Description*
`instanceOf`	`Concept` [1..1]	Concepto del que es instancia.
`attributes`	`InstanciatedAttribute` [0..*]	lista de attributos instanciados (con valores concretos).

Clase **InstanciatedAttribute**

Esta clase representa un atributo instanciado, es decir un valor concreto para un atributo. Los atributos y asociaciones de la clase `InstanciatedAttri` se muestran en la tabla A.11.

Tabla A.11: Propiedades de la clase: `InstanciatedAttribute`

Features	*Type*	*Description*
`feature`	`Attribute` [1..1]	atributo al que corresponde el valor instanciado.
`value`	`ValueSpecification` [1..1]	Valor del atributo instanciado.

Clase **KnowledgeBase**

Esta clase respresenta una base de conocimiento que contiene fundamentalmente las instancias de los elementos definidos en el dominio, como son las instancias de los conceptos (individuos), instancias de las relaciones (tuplas) y las instancias de las reglas. La clase `KnowledgeBase` hereda de *`TerminologyElement`*. Los atributos y asociaciones de la clase `KnowledgeBase` se muestran en la tabla A.12.

Clase **KnowledgeModel**

Esta clase representa un modelo de conocimiento. La clase `KnowledgeModel` hereda de *`TerminologyElement`*. Los atributos y asociaciones de la clase `KnowledgeModel` se muestran en la tabla A.13.

Tabla A.12: Propiedades de la clase: `KnowledgeBase`

Features	*Type*		*Description*
KBDefIn	DomainKnowledge	[1..1]	Conocimiento del dominio al que pertenece la base de conocimiento.
use	DomainSchema	[1..*]	Esquemas de dominio usados en la base de conocimiento.
instances	Instance	[0..*]	Instancias de conceptos (individuos).
expressions	RuleTypeInstance	[0..*]	Instancias de reglas.
tuples	Tuple	[0..*]	Tuplas: instancias de relaciones binarias.
variables	Variable	[0..*]	variables.

Tabla A.13: Propiedades de la clase: `KnowledgeModel`

Features	*Type*		*Description*
domainKnowledge	DomainKnowledge	[1..1]	Conocimiento del dominio que forma parte del modelo.
DataType	DataType	[0..*]	tipos de datos definidos en el modelo de conocimiento.

Clase abstracta ***NamedElt***

Esta clase abstracta representa un elemento con nombre. Los atributos y asociaciones de la clase abstracta *`NamedElt`* se muestran en la tabla A.14.

Tabla A.14: Propiedades de la clase: *`NamedElt`*

Features	*Type*	*Description*
name	EString[0..1]	Nombre del elemento.
description	EString[0..1]	Descripción del elemento.

Clase abstracta ***Operand***

Esta clase permite representar un operando de una expresión arirmético-lógica. El operando será un valor único, o bien una lista de valores, o bien un símbolo de la base de conocimiento como una variable o instancia. . La clase abstracta *`Operand`* no tiene atributos ni asociaciones.

Clase **`RuleElement`**

Esta clase permite modelar un elemento de la regla, que básicamente puede ser el antecedente y el consecuente. Los atributos y asociaciones de la clase `RuleElement` se muestran en la tabla A.15.

Tabla A.15: Propiedades de la clase: `RuleElement`

Features	*Type*	*Description*
`rulePart`	`RuleElementType`[0..1]	Parte de la regla a la que pertenece el elemento: antecedente o consecuente.
`cardinality`	`Cardinality` [0..1]	Cardinalidad en la que se presenta el Tipo asociado al elemento de la regla mediante la asociación userDefinedType.
`userDefinedType`	`UserDefinedType`[1..*]	Tipos (conceptos, relaciones binarias, ...) a los que va a estar asociado el elemento de la regla.

Clase abstracta ***RuleType***

Clase abstracta para representar los tipos de reglas. Como subtipo tiene la clase que representa las reglas de implicación (reglas de producción). La clase abstracta *`RuleType`* hereda de *`DomainConstruct`*. Los atributos y asociaciones de la clase abstracta *`RuleType`* se muestran en la tabla A.16.

Tabla A.16: Propiedades de la clase: *`RuleType`*

Features	*Type*	*Description*
`examples`	`EString`[0..1]	ejemplos de reglas en lenguaje natural.

Clase **`RuleTypeExpression`**

Esta clase representa una expresión de una instancia de regla, que puede corresponder al antecedente o bien al consecuente. El atributo element indica si la expresión es del antecedente o del consecuente. . Los atributos y asociaciones de la clase `RuleTypeExpression` se muestran en la tabla A.17.

Tabla A.17: Propiedades de la clase: `RuleTypeExpression`

Features	*Type*	*Description*
`value`	`EString` [0..1]	Valor de la expresión, en caso de ser una expresión simple (sin ecuación).
`element`	`RuleElementType`[0..1]	tipo de expresión: antecedente o consecuente.
`equation`	`Expression` [0..1]	Ecuación que forma parte del elemento de la regla.

Clase **RuleTypeInstance**

Esta clase representa una instancia de un tipo de regla, es decir, una regla concreta que se forma especificando condiciones y acciones en antecedente y consecuente respectivamente, afectando a los conceptos definidos en el tipo de regla. La clase `RuleTypeInstance` hereda de *`NamedElt`*. Los atributos y asociaciones de la clase `RuleTypeInstance` se muestran en la tabla A.18.

Tabla A.18: Propiedades de la clase: `RuleTypeInstance`

Features	*Type*	*Description*
instanceOf	ImplicationRule [1..1]	Regla de la que es instancia.
connectionSymbol	ConnectionSymbol [1..1]	Símbolo de conexión del la regla, expresar la semántica que conecta el antecedente con el consecuente.
antecedent	RuleTypeExpression[1..1]	Expresión del antecedente.
consequent	RuleTypeExpression[1..1]	Expresión del consecuente.

Clase **Symbol**

Esta clase representa un símbolo que es el constructor que permite expresar un atributo de un concepto como operando en una expresión. Por ejemplo, permite indicar expresiones del tipo "Persona.edad + 1", donde Persona.edad se representa como un símbolo.. La clase `Symbol` hereda de *`Operand`*. Los atributos y asociaciones de la clase `Symbol` se muestran en la tabla A.19.

Tabla A.19: Propiedades de la clase: `Symbol`

Features	*Type*	*Description*
concept	UserDefinedType[0..1]	Tipo definido por el usuario, que puede ser un concepto o relacion binaria, sobre la que se expresa el símbolo.
attribute	Attribute [0..1]	atributo del símbolo.

Clase **SymbolList**

Esta clase representa una lista de símbolos. La clase `SymbolList` hereda de *`Operand`*, *`TypedElement`*. Los atributos y asociaciones de la clase `SymbolList` se muestran en la tabla A.20.

Tabla A.20: Propiedades de la clase: `SymbolList`

Features	*Type*	*Description*
symbols	Symbol[1..*]	símbolos de la lista.

Clase **Terminology**

Esta clase representa un conjunto de propiedades que documentan un elemento del modelo de conocimiento, como su descripción, sinónimos, traduccion, etc.. Los atributos y asociaciones de la clase `Terminology` se muestran en la tabla A.21.

Tabla A.21: Propiedades de la clase: `Terminology`

Features	*Type*	*Description*
sources	EString[0..1]	Fuente u origen de dónde se ha obtenido el elemento.
description	EString[0..1]	Descripción del elemento.
synonyms	EString[0..1]	Términos usados en el dominio con idéntico significado.
traslation	EString[0..1]	Traducción a distintos idiomas.

Clase abstracta ***TerminologyElement***

Esta clase abstracta representa un elemento con varias propiedades descriptivas, definidas en la clase Terminology. La clase abstracta *`TerminologyElement`* hereda de *`NamedElt`*. Los atributos y asociaciones de la clase abstracta *`TerminologyElement`* se muestran en la tabla A.22.

Tabla A.22: Propiedades de la clase: *`TerminologyElement`*

Features	*Type*	*Description*
terminology	Terminology[0..1]	Terminología asociada al elemento.

Clase **Tuple**

Esta clase representa una tupla, es decir una instancia de una relación binaria, y consiste en una asociación entre dos intancias de los conceptos relacionados por la relacion binaria. La clase `Tuple` hereda de *`TerminologyElement`*. Los atributos y asociaciones de la clase `Tuple` se muestran en la tabla A.23.

Tabla A.23: Propiedades de la clase: `Tuple`

Features	*Type*		*Description*
instanceOf	BinaryRelation	[1..1]	relacion binaria que instancia la tupla.
argument1	Instance	[1..1]	primera instancia de la tupla.
argument2	Instance	[1..1]	segunda instancia de la tupla.
attributes	InstanciatedAttribute	[0..*]	lista de atributos instanciados para una tupla (relación con atributos).

Clase abstracta *Type*

Esta clase abstracta representa un Tipo en el modelo de dominio. Se definen como tipos los Conceptos, los Tipos Valor (enumerados) y los Tipos de Datos.. La clase abstracta *Type* no tiene atributos ni asociaciones.

Clase abstracta *TypedElement*

Esta clase abstracta representa un elemento con un Tipo asociado. Tiene como subclases Variable y SimpolList. Los atributos y asociaciones de la clase abstracta *TypedElement* se muestran en la tabla A.24.

Tabla A.24: Propiedades de la clase: *TypedElement*

Features	*Type*	*Description*
type	Type[0..1]	Tipo asociado al elemento.

Clase abstracta *UserDefinedType*

Esta clase abstracta representa un tipo definido por el usuario. Puede ser un Concepto, una relación Binaria, o una Variable. La clase abstracta *UserDefinedType* no tiene atributos ni asociaciones.

Clase **Value**

Esta clase representa un valor literal de un Tipo Valor (Enumeración). Los atributos y asociaciones de la clase `Value` se muestran en la tabla A.25.

Tabla A.25: Propiedades de la clase: `Value`

Features	*Type*	*Description*
`value`	`EString`[0..1]	Valor literal de un Tipo Valor.

Clase **`ValueSpecification`**

Esta clase representa una especificación de un valor concreto. La clase `ValueSpecification` hereda de *`Operand`*. Los atributos y asociaciones de la clase `ValueSpecification` se muestran en la tabla A.26.

Tabla A.26: Propiedades de la clase: `ValueSpecification`

Features	*Type*	*Description*
`valueType`	`ValueSpecificationType`[0..1]	Tipo del valor especificado.
`value`	`EString` [0..1]	valor concreto.
`type`	`Type` [0..1]	tipo del valor especificado.

Clase **`ValueType`**

Esta clase representa un Tipo Valor, que básicamente constituye un tipo enumerado al que se le asocian un conjunto de literales.. La clase `ValueType` hereda de *`DomainConstruct`*, *`Type`*. Los atributos y asociaciones de la clase `ValueType` se muestran en la tabla A.27.

Tabla A.27: Propiedades de la clase: `ValueType`

Features	*Type*	*Description*
`type`	`ValueTypeType`[0..1]	Tipo del Tipo Valor. Toma valore del enumerado ValueTypeType.
`valuelist`	`Value` [0..*]	Lista de valores literales del Tipo Valor (Enumeración).

Clase **`Variable`**

Esta clase permite definir variables de un tipo básico. La clase `Variable` hereda de *`NamedElt`*, *`TypedElement`*, *`UserDefinedType`*. La clase `Variable` no tiene atributos ni asociaciones.

A.2.2 Enumeraciones

Enumeración `BinaryOperator`

Enumeración que contiene los operadores para expresiones binarias: aritméticos, lógicos, para listas.. Los literales de la enumeración *BinaryOperator* se muestran en la tabla A.28.

Tabla A.28: Enum literals: *BinaryOperator*

Literal	*Description*
`lt`	menor.
`gt`	mayor.
`leq`	menor o igual.
`geq`	mayor o igual.
`eq`	igual.
`neq`	distinto.
`plus`	suma aritmética.
`minus`	diferencia aritmética.
`prod`	producto aritmético.
`div`	división aritmética.
`asig`	asignación.
`add`	añadir elemento a la lista.
`remove`	eliminar un elemento de la lista.
`includes`	verdadero si la lista contiene al elemento; falso en otro caso.
`excludes`	verdadero si la lista no contiene al elemento; falso en otro caso.
`includesAll`	devuelve verdadero si una lista contiene todos los elementos de otra lista que se pasa como parámetro.
`excludesAll`	devuelve verdadero si una lista no contiene a todos los elementos de otra lista que se pasa como parámetro.
`union`	union de listas.

Enumeración `Cardinality`

Enumeración con la cardinalidad que pueden tomar los elementos en el modelo. Esta puede ser sencilla (valor único) o múltiple (lista de valores). Los literales de la enumeración *Cardinality* se muestran en la tabla A.29.

Tabla A.29: Enum literals: *Cardinality*

Literal	*Description*
`singleValued`	Cardinalidad simple o sencilla (valor único) .
`multiValued`	Cardinalidad múltiple (lista de valores).

Enumeración `LogicalOperator`

negación lógica. Los literales de la enumeración *LogicalOperator* se muestran en la tabla A.30.

Tabla A.30: Enum literals: *LogicalOperator*

Literal	*Description*
`not`	negación lógica.
`and`	conjución lógica.
`or`	disyunción lógica.
`xor`	disyunción exclusiva.

Enumeración `RelationType`

Enumerado con los tipos de relaciones binarias. Los literales de la enumeración *RelationType* se muestran en la tabla A.31.

Tabla A.31: Enum literals: *RelationType*

Literal	*Description*
`transitive`	relación transitiva.
`asymmetric`	relación asimétrica.
`symmetric`	relación simétrica.
`irreflexive`	relación irreflexiva.
`reflexive`	relación reflexiva.
`antisymmetric`	relación antisimétrica.

Enumeración `RuleElementType`

Enumeración que permite definir el Tipo de elemento de regla, que puede ser antecedente o consecuente. Los literales de la enumeración *RuleElementType* se muestran en la tabla A.32.

Tabla A.32: Enum literals: *RuleElementType*

Literal	*Description*
`antecedent`	antecedente de la regla (condiciones).
`consequent`	consecuente de la regla (acciones).

Enumeración `ValueSpecificationType`

Enumeración que contiene los tipos de especificacion de valor: valor literal o valor definido por una expresión. Los literales de la enumeración

ValueSpecificationType se muestran en la tabla A.33.

Tabla A.33: Enum literals: *ValueSpecificationType*

Literal	*Description*
literal	valor literal.
expresion	expresión.

Enumeración `ValueTypeType`

Enumeración que contiene los tipos de Tipo Valor: nominal (literales cualesquiera), ordinal (literales que representan un orden). Los literales de la enumeración *ValueTypeType* se muestran en la tabla A.34.

Tabla A.34: Enum literals: *ValueTypeType*

Literal	*Description*
nominal	Literales cualesquiera.
ordinal	Literales que representan un orden.

A.3 Resumen

A modo de resumen, a continuación se presentan una serie de métricas sobre el metamodelo descrito. El paquete está formado por:

- Clases: 35 (9 abstractas)
  - Número total de atributos: 34
  - Número total de asociaciones: 56
  - Número total de relaciones de herencia: 31
  - La media de atributos por clase es: 0.97
  - La media de asociaciones por clase es: 1.6
  - El número medio de hijos por clase es: 0.88
- Enumeraciones: 7
  - Número total de literales: 36
  - La media de literales por enumeración es: 5.14

Apéndice B

Metamodelo para Reglas JESS

B.1 Introducción

La figura B.1 representa el paquete *ruleMetaModel*, que contiene:

- Clases: 14
- Enumeraciones: 7

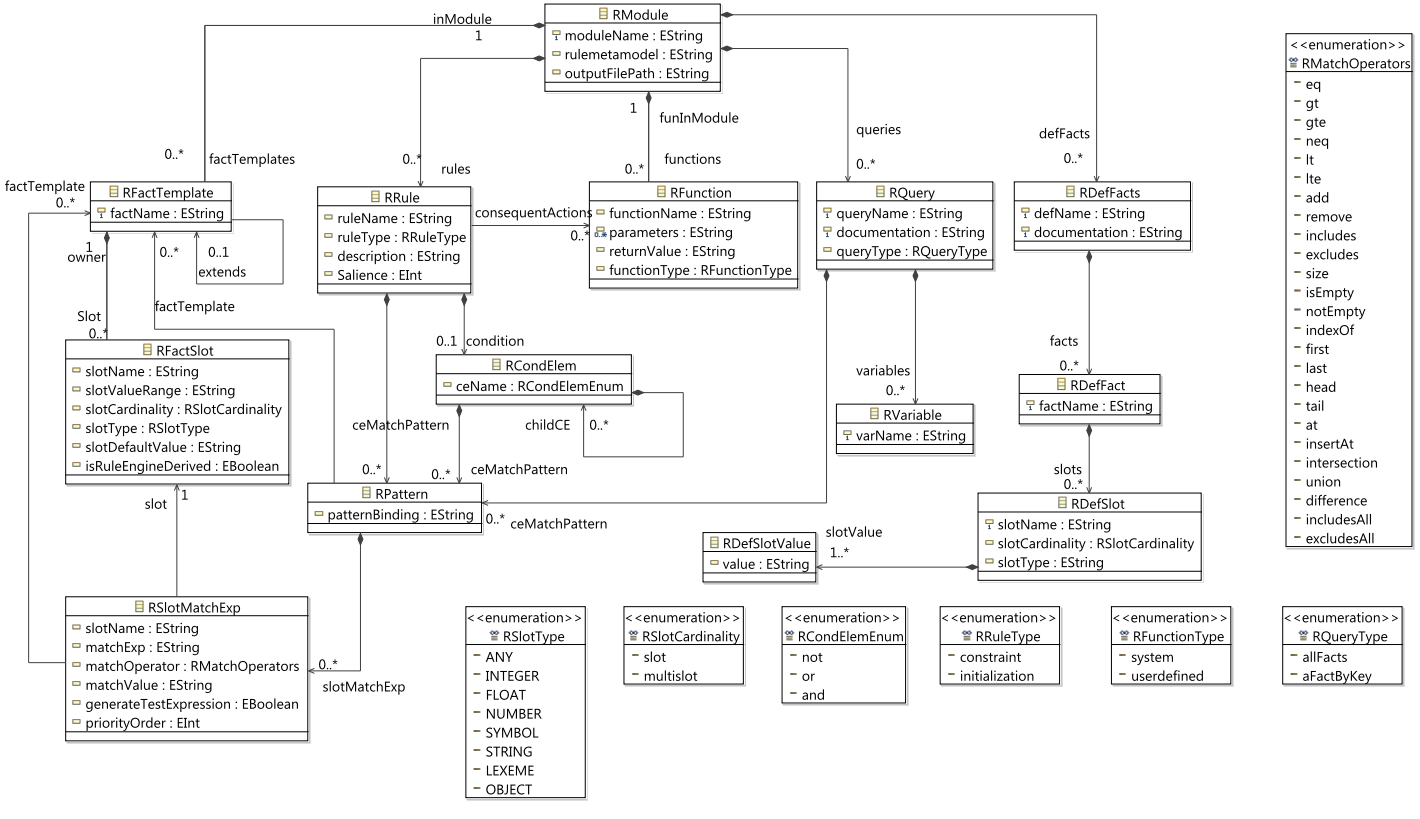


Figura B.1: Jess Rules Metamodel

B.2 Paquete *ruleMetaModel*

B.2.1 Metaclases

Clase **RCondElem**

Esta clase representa un bloque de condiciones de la regla, unidas por un operador lógico and, or, not. Los atributos y asociaciones de la clase `RCondElem` se muestran en la tabla B.1.

Tabla B.1: Propiedades de la clase: `RCondElem`

Features	*Type*	*Description*
ceName	RCondElemEnum[0..1]	Operador lógico que afecta a todo el bloque de condiciones: and, or, not.
ceMatchPattern	RPattern [0..*]	Expresiones de coincidencia de patrones (pattern matching) que forman parte de la condición de la regla.
childCE	RCondElem [0..*]	Asociación a otro bloque de condiciones, lo que permite componer condiciones en la regla..

Clase **RDefFact**

Esta clase representa una definicion de un hecho (individual), al que se darán valores para los slots que lo componen. Los atributos y asociaciones de la clase `RDefFact` se muestran en la tabla B.2.

Tabla B.2: Propiedades de la clase: `RDefFact`

Features	*Type*	*Description*
factName	EString [1..1]	Nombre del hecho.
slots	RDefSlot[0..*]	Definición de los valores de los slots que componen el hecho.

Clase **RDefFacts**

Esta clase representa una definicion de hechos (individuals) con la que es inicializada la base de conocimiento. Los atributos y asociaciones de la clase `RDefFacts` se muestran en la tabla B.3.

Clase **RDefSlot**

Esta clase define un slot de un hecho, al que se le asignará un valor. Los atributos y asociaciones de la clase `RDefSlot` se muestran en la

Tabla B.3: Propiedades de la clase: `RDefFacts`

Features	*Type*	*Description*
defName	EString [1..1]	Nombre del bloque de definición de hechos (individuals).
documentation	EString [1..1]	Documentación descriptiva.
facts	RDefFact[0..*]	Lista de hechos definidos.

tabla B.4.

Tabla B.4: Propiedades de la clase: `RDefSlot`

Features	*Type*	*Description*
slotName	EString [1..1]	Nombre del slot.
slotCardinality	RSlotCardinality[0..1]	Cardinalidad del slot: simple o múltiple.
slotType	EString [0..1]	Tipo del slot, que debe coincidir con el definido en la plantilla.
slotValue	RDefSlotValue [1..*]	Definición del valor para el slot.

Clase **RDefSlotValue**

Esta clase permite representar la definición del valor para un slot de un hecho. Los atributos y asociaciones de la clase `RDefSlotValue` se muestran en la tabla B.5.

Tabla B.5: Propiedades de la clase: `RDefSlotValue`

Features	*Type*	*Description*
value	EString[0..1]	Valor del slot.

Clase **RFactSlot**

Esta clase representa un slot de una plantilla de hechos. Básicamente se corresponde con el concepto de atributo de una clase. Los atributos y asociaciones de la clase `RFactSlot` se muestran en la tabla B.6.

Clase **RFactTemplate**

Esta clase representa una plantilla de hechos en la base de reglas. Representa un concepto o clase del dominio. Los atributos y asociaciones de la clase `RFactTemplate` se muestran en la tabla B.7.

Tabla B.6: Propiedades de la clase: `RFactSlot`

Features	*Type*		*Description*
slotName	EString	[0..1]	Nombre del slot.
slotValueRange	EString	[0..1]	Rango de valores que puede tomar el slot.
slotCardinality	RSlotCardinality	[0..1]	Cardinalidad del slot: simple o múltiple.
slotType	RSlotType	[0..1]	Tipo del slot. Puede indicarse o no, en tal caso será uno de los tipos primitivos ANY, INTEGER, FLOAT, NUMBER, SYMBOL, STRING, LEXEME, and OBJECT. .
slotDefaultValue	EString	[0..1]	Valor predeterminado para el slot.
isRuleEngineDerived	EBoolean	[0..1]	Indica si el valor que toma el slot es derivado por el motor de inferencia..
owner	RFactTemplate	[1..1]	Plantilla a la que pertenece el slot.

Tabla B.7: Propiedades de la clase: `RFactTemplate`

Features	*Type*		*Description*
factName	EString	[1..1]	Nombre de la plantilla de hechos.
Slot	RFactSlot	[0..*]	Slots de la plantilla de hechos..
inModule	RModule	[1..1]	módulo al que pertenece la plantilla de hechos..
extends	RFactTemplate	[0..1]	Herencia simple entre plantillas de hechos. Representa la plantilla padre..

Clase **RFunction**

Esta clase representa una función en el módulo de la base de reglas. Los atributos y asociaciones de la clase `RFunction` se muestran en la tabla B.8.

Tabla B.8: Propiedades de la clase: `RFunction`

Features	*Type*		*Description*
functionName	EString	[0..1]	Nombre de la función.
parameters	EString	[0..*]	Parámetros de la función.
returnValue	EString	[0..1]	Tipo del valor que devuelve la función.
functionType	RFunctionType	[0..1]	Tipo de función: system, userdefined.
funInModule	RModule	[1..1]	módulo al que pertenece la función.

Clase **RModule**

Esta clase representa un módulo de la base de reglas. Un módulo permite agrupar los elementos que conforman la base de reglas, como

reglas, consultas, funciones, y plantillas de hechos, en bloques independientes que pueden ser cargados y ejecutados independientemente. Los atributos y asociaciones de la clase `RModule` se muestran en la tabla B.9.

Tabla B.9: Propiedades de la clase: `RModule`

Features	*Type*		*Description*
moduleName	EString	[1..1]	Nombre del módulo.
rulemetamodel	EString	[0..1]	Nombre del metamodelo.
outputFilePath	EString	[0..1]	Ruta donde se guardará el archivo con la base de conocimiento generada.
factTemplates	RFactTemplate	[0..*]	Plantillas de hechos del módulo.
functions	RFunction	[0..*]	Funciones del módulo.
rules	RRule	[0..*]	Reglas del módulo.
defFacts	RDefFacts	[0..*]	Definiciones de hechos del módulo.
queries	RQuery	[0..*]	Consultas del módulo.

Clase **RPattern**

Esta clase representa una expresión de coincidencia de patrones (pattern matching). Los atributos y asociaciones de la clase `RPattern` se muestran en la tabla B.10.

Tabla B.10: Propiedades de la clase: `RPattern`

Features	*Type*		*Description*
patternBinding	EString	[0..1]	Expresión de unión de patrones en Jess.
factTemplate	RFactTemplate	[0..*]	Plantillas de hechos sobre las que se define la expresión de coincidencia de patrones.
slotMatchExp	RSlotMatchExp	[0..*]	Slot o slots concretos a los que afecta la expresión de coincidencia de patrones.

Clase **RQuery**

Esta clase representa una consulta en la base de reglas. Una consulta permite acceder a hechos existentes en la memoria de trabajo. Los atributos y asociaciones de la clase `RQuery` se muestran en la tabla B.11.

Clase **RRule**

Esta clase representa una regla en la base de reglas. Los atributos y asociaciones de la clase `RRule` se muestran en la tabla B.12.

Tabla B.11: Propiedades de la clase: `RQuery`

Features	*Type*	*Description*
queryName	EString [1..1]	Nombre de la consulta.
documentation	EString [1..1]	Documentación descriptiva.
queryType	RQueryType[0..1]	Tipo de consulta: allFacts, aFactByKey.
variables	RVariable [0..*]	Variables que forman parte de la consulta.
ceMatchPattern	RPattern [0..*]	Lista de patrones de coincidencia que se emplean en la consulta, y permiten filtrar los hechos de la base de hechos de la memoria de trabajo que devuelve la consulta.

Tabla B.12: Propiedades de la clase: `RRule`

Features	*Type*	*Description*
ruleName	EString [0..1]	Nombre de la regla.
ruleType	RRuleType[0..1]	Tipo de regla: constraint, initialization. .
description	EString [0..1]	Texto descriptivo de la regla.
Salience	EInt [0..1]	Prioridad de la regla.
consequentActions	RFunction[0..*]	Acciones del consecuente de la regla.
condition	RCondElem[0..1]	Condiciones del antecedente de la regla. Se utiliza para expresar varias condiciones que van unidas por un operador lógico (and, or).
ceMatchPattern	RPattern [0..*]	Expresiones de coincidencia de patrones (pattern matching) en la condición de la regla. Expresa una o varias condiciones de la regla que no van unidas por operador lógico (and, or).

Clase **RSlotMatchExp**

Esta clase permite expresar una expresión de coincidencia de patrones individual, que afecta a un slot concreto sobre el que se evalua una expresión. Los atributos y asociaciones de la clase `RSlotMatchExp` se muestran en la tabla B.13.

Clase **RVariable**

Esta clase permite representar una variable que se usa en una consulta. Los atributos y asociaciones de la clase `RVariable` se muestran en la tabla B.14.

B.2.2 Enumeraciones

Enumeración **RCondElemEnum**

Esta enumeración contiene los literales que representan los distintos tipos de bloques de condiciones de una regla. Los literales de la enumeración *RCondElemEnum* se muestran en la tabla B.15.

Tabla B.13: Propiedades de la clase: `RSlotMatchExp`

Features	*Type*		*Description*
slotName	EString	[0..1]	Nombre del slot sobre el que se evalua la expresión.
matchExp	EString	[0..1]	Expresión de coincidencia.
matchOperator	RMatchOperators	[0..1]	Operador de la expresión. Puede ser un operador sobre un valor simple o sobre una lista de valores.
matchValue	EString	[0..1]	Valor de coincidencia de la expresión.
generateTestExpression	EBoolean	[0..1]	Generar expresión de prueba.
priorityOrder	EInt	[0..1]	Orden de prioridad de la expresión.
factTemplate	RFactTemplate	[0..*]	Plantillas de hechos sobre las que se define la expresión de coincidencia de patrones.
slot	RFactSlot	[1..1]	Slot sobre el que se define la expresión de coincidencia de patrones.

Tabla B.14: Propiedades de la clase: `RVariable`

Features	*Type*	*Description*
varName	EString[1..1]	Nombre de la variable.

Tabla B.15: Enum literals: *RCondElemEnum*

Literal	*Description*
not	negación lógica.
or	disyunción lógica.
and	conjunción logica.

Enumeración `RFunctionType`

Esta enumeración contiene los literales que representan los distintos tipos de funciones. Los literales de la enumeración *RFunctionType* se muestran en la tabla B.16.

Tabla B.16: Enum literals: *RFunctionType*

Literal	*Description*
system	función predefinida en el sistema.
userdefined	funcion definida por el usuario.

Enumeración `RMatchOperators`

Los literales de la enumeración *RMatchOperators* se muestran en la tabla B.17.

Tabla B.17: Enum literals: *RMatchOperators*

Literal	*Description*
`eq`	igual.
`gt`	mayor.
`gte`	mayor o igual.
`neq`	distinto.
`lt`	menor.
`lte`	menor o igual.
`add`	añadir elemento a la lista.
`remove`	eliminar un elemento de la lista.
`includes`	verdadero si la lista contiene al elemento; falso en otro caso.
`excludes`	verdadero si la lista no contiene al elemento; falso en otro caso.
`size`	numero de elementos de la lista.
`isEmpty`	devuelve verdadero si la lista está vacía.
`notEmpty`	devuelve verdadero si la lista no está vacía.
`indexOf`	posición de un elemento dentro de la lista.
`first`	primero de la lista.
`last`	último de la lista.
`head`	cabeza de la lista.
`tail`	cola de la lista.
`at`	elemento de la lista en una posición que se pasa como parámetro.
`insertAt`	añadir un elemento a la lista en la posición que se pasa como parámetro.
`intersection`	intersección de listas .
`union`	union de listas.
`difference`	diferencia de listas.
`includesAll`	devuelve verdadero si una lista contiene todos los elementos de otra lista que se pasa como parámetro.
`excludesAll`	devuelve verdadero si una lista no contiene a todos los elementos de otra lista que se pasa como parámetro.

Enumeración `RQueryType`

Esta enumeración contiene los literales que representan los distintos tipos de consultas. Los literales de la enumeración *RQueryType* se muestran en la tabla B.18.

Tabla B.18: Enum literals: *RQueryType*

Literal	*Description*
`allFacts`	consulta que devuelve todos los hechos (facts) de un determinado tipo.
`aFactByKey`	consulta que devuelve un único hecho (facts) a partir de su clave.

Enumeración `RRuleType`

Esta enumeración contiene los literales que representan los distintos tipos de reglas. Los literales de la enumeración *RRuleType* se muestran en la tabla B.19.

Tabla B.19: Enum literals: *RRuleType*

Literal	*Description*
`constraint`	regla tipo restricción.
`initialization`	regla de tipo inicialización.

Enumeración `RSlotCardinality`

Esta enumeración contiene los literales que representan los distintos tipos de slots. Los literales de la enumeración *RSlotCardinality* se muestran en la tabla B.20.

Tabla B.20: Enum literals: *RSlotCardinality*

Literal	*Description*
`slot`	slot que almacena un valor único.
`multislot`	almacena una lista de valores.

Enumeración `RSlotType`

Esta enumeración contiene los literales que representan los distintos tipos de datos que puede tomar un slot. Los literales de la enumeración *RSlotType* se muestran en la tabla B.21.

Tabla B.21: Enum literals: *RSlotType*

Literal	*Description*
`ANY`	Cualquiera.
`INTEGER`	Entero.
`FLOAT`	Real.
`NUMBER`	Número.
`SYMBOL`	Símbolo (token que puede ser el tomar un identificador en Jess).
`STRING`	Cadena de caracteres.
`LEXEME`	Lexema (symbol o string).
`OBJECT`	Objeto (Java).

B.3 Resumen

A modo de resumen, a continuación se presentan una serie de métricas sobre el metamodelo descrito. El paquete está formado por:

- Clases: 14 (0 abstractas)
  - Número total de atributos: 37
  - Número total de asociaciones: 24
  - Número total de relaciones de herencia: 0
  - La media de atributos por clase es: 2.64
  - La media de asociaciones por clase es: 1.71
  - El número medio de hijos por clase es: 0
- Enumeraciones: 7
  - Número total de literales: 44
  - La media de literales por enumeración es: 6.28

Apéndice C

Metamodelo Arquitectura Web JSF

C.1 Introducción

El paquete *JavaBeans* contiene:

- Clases: 22
- Enumeraciones: 3

C.2 Paquete *JavaBeans*

C.2.1 Metaclases

Clase `ConfigurationFile`

La clase `ConfigurationFile` hereda de `File`. Los atributos y asociaciones de la clase `ConfigurationFile` se muestran en la tabla C.1.

Tabla C.1: Propiedades de la clase: `ConfigurationFile`

Features	*Type*	*Description*
managedBeans	ManagedBean [0..*]	
navigationRules	NavigationRule[0..*]	

Clase **Enum**

La clase `Enum` hereda de *`JavaBeanEntity`*, *`Type`*. Los atributos y asociaciones de la clase `Enum` se muestran en la tabla C.2.

Tabla C.2: Propiedades de la clase: `Enum`

Features	*Type*	*Description*
literals	Literal[0..*]	

Clase **File**

Los atributos y asociaciones de la clase `File` se muestran en la tabla C.3.

Tabla C.3: Propiedades de la clase: `File`

Features	*Type*	*Description*
name	EString[1..1]	
path	EString[1..1]	

Clase **JSFProject**

Los atributos y asociaciones de la clase `JSFProject` se muestran en la tabla C.4.

Tabla C.4: Propiedades de la clase: `JSFProject`

Features	*Type*	*Description*
name	EString [1..1]	
configurationFile	ConfigurationFile[1..1]	
pages	JSPPage [0..*]	
messagesFile	MessagesFile [1..*]	

Clase abstracta ***JSPPage***

La clase abstracta *`JSPPage`* hereda de `File`. Los atributos y asociaciones de la clase abstracta *`JSPPage`* se muestran en la tabla C.5.

Tabla C.5: Propiedades de la clase: *JSPPage*

Features	Type	Description
pageType	JSPPageType[1..1]	
bean	JavaBean [1..1]	

Clase **JavaBean**

La clase JavaBean hereda de *JavaBeanEntity*, *Type*. Los atributos y asociaciones de la clase JavaBean se muestran en la tabla C.6.

Tabla C.6: Propiedades de la clase: JavaBean

Features	Type	Description
isAbstract	EBoolean[1..1]	
showInMenu	EBoolean[0..1]	
icon	EString [0..1]	
package	Package [1..1]	
properties	Property[0..*]	
superTypes	JavaBean[0..1]	

Clase abstracta ***JavaBeanEntity***

Los atributos y asociaciones de la clase abstracta *JavaBeanEntity* se muestran en la tabla C.7.

Tabla C.7: Propiedades de la clase: *JavaBeanEntity*

Features	Type	Description
name	EString[1..1]	
comment	EString[0..1]	
nameInJessKB	EString[0..1]	

Clase **ListPage**

La clase ListPage hereda de *JSPPage*. La clase ListPage no tiene atributos ni asociaciones.

Clase **Literal**

Los atributos y asociaciones de la clase Literal se muestran en la tabla C.8.

Tabla C.8: Propiedades de la clase: `Literal`

Features	Type	Description
value	EString[1..1]	
original	EString[0..1]	

Clase **LoginPage**

La clase `LoginPage` hereda de *`JSPPage`*. La clase `LoginPage` no tiene atributos ni asociaciones.

Clase **ManagedBean**

Los atributos y asociaciones de la clase `ManagedBean` se muestran en la tabla C.9.

Tabla C.9: Propiedades de la clase: `ManagedBean`

Features	Type	Description
name	EString [1..1]	
beanClass	EString [1..1]	
scope	ScopeBean[1..1]	
managedBean	JavaBean [1..1]	

Clase **MessageEntry**

Los atributos y asociaciones de la clase `MessageEntry` se muestran en la tabla C.10.

Tabla C.10: Propiedades de la clase: `MessageEntry`

Features	Type	Description
key	EString[1..1]	
value	EString[1..1]	

Clase **MessagesFile**

La clase `MessagesFile` hereda de `File`. Los atributos y asociaciones de la clase `MessagesFile` se muestran en la tabla C.11.

Tabla C.11: Propiedades de la clase: `MessagesFile`

Features	*Type*	*Description*
entries	MessageEntry[0..*]	

Clase **ModifyFormPage**

La clase `ModifyFormPage` hereda de *`JSPPage`*. La clase `ModifyFormPage` no tiene atributos ni asociaciones.

Clase **NavigationCase**

Los atributos y asociaciones de la clase `NavigationCase` se muestran en la tabla C.12.

Tabla C.12: Propiedades de la clase: `NavigationCase`

Features	*Type*	*Description*
actionName	EString[1..1]	
toPage	JSPPage[1..1]	

Clase **NavigationRule**

Los atributos y asociaciones de la clase `NavigationRule` se muestran en la tabla C.13.

Tabla C.13: Propiedades de la clase: `NavigationRule`

Features	*Type*	*Description*
fromPage	EString [1..1]	
navigationCases	NavigationCase[0..*]	

Clase **NewFormPage**

La clase `NewFormPage` hereda de *`JSPPage`*. La clase `NewFormPage` no tiene atributos ni asociaciones.

Clase **Package**

La clase `Package` hereda de *`JavaBeanEntity`*. Los atributos y asociaciones de la clase `Package` se muestran en la tabla C.14.

Tabla C.14: Propiedades de la clase: `Package`

Features	*Type*	*Description*
classes	JavaBean[0..*]	
enums	Enum [0..*]	

Clase **`Property`**

La clase `Property` hereda de *`JavaBeanEntity`*. Los atributos y asociaciones de la clase `Property` se muestran en la tabla C.15.

Tabla C.15: Propiedades de la clase: `Property`

Features	*Type*	*Description*
isMultiValued	EBoolean[1..1]	
isModificable	EBoolean[1..1]	
defaultValue	EString [0..1]	
isRelation	EBoolean[0..1]	
derivationExpression	EString [0..1]	
label	EString [0..1]	
isIrreflexive	EBoolean[0..1]	
type	Type [1..1]	
referencedBean	JavaBean[0..1]	
valueFrom	Property[0..1]	
notifiesTo	Property[0..*]	
isNotifiedBy	Property[0..*]	

Clase **`ShowFormPage`**

La clase `ShowFormPage` hereda de *`JSPPage`*. La clase `ShowFormPage` no tiene atributos ni asociaciones.

Clase **`SimpleType`**

La clase `SimpleType` hereda de *`JavaBeanEntity`*, *`Type`*. La clase `SimpleType` no tiene atributos ni asociaciones.

Clase abstracta ***`Type`***

Los atributos y asociaciones de la clase abstracta *`Type`* se muestran en la tabla C.16.

Tabla C.16: Propiedades de la clase: `Type`

Features	Type	Description
subType	SubTypes[1..1]	

C.2.2 Enumeraciones

Enumeración `JSPPageType`

Los literales de la enumeración *JSPPageType* se muestran en la tabla C.17.

Tabla C.17: Enum literals: *JSPPageType*

Literal	Description
listInstances	
showForm	
newForm	
modifyForm	
loginPage	
home	

Enumeración `ScopeBean`

Los literales de la enumeración *ScopeBean* se muestran en la tabla C.18.

Tabla C.18: Enum literals: *ScopeBean*

Literal	Description
request	
session	
application	
none	

Enumeración `SubTypes`

Los literales de la enumeración *SubTypes* se muestran en la tabla C.19.

Tabla C.19: Enum literals: *SubTypes*

Literal	*Description*
simpleType	
enum	
javaBean	

C.3 Resumen

A modo de resumen, a continuación se presentan una serie de métricas sobre el metamodelo descrito. El paquete está formado por:

- Clases: 22 (3 abstractas)
  - Número total de atributos: 27
  - Número total de asociaciones: 21
  - Número total de relaciones de herencia: 16
  - La media de atributos por clase es: 1.22
  - La media de asociaciones por clase es: 0.95
  - El número medio de hijos por clase es: 0.72
- Enumeraciones: 3
  - Número total de literales: 13
  - La media de literales por enumeración es: 4.33

Apéndice D

Metamodelo OWL2Web

D.1 Introducción

La figura D.1 representa el paquete *owl2web*, que contiene:

- Clases: 11
- Enumeraciones: 1

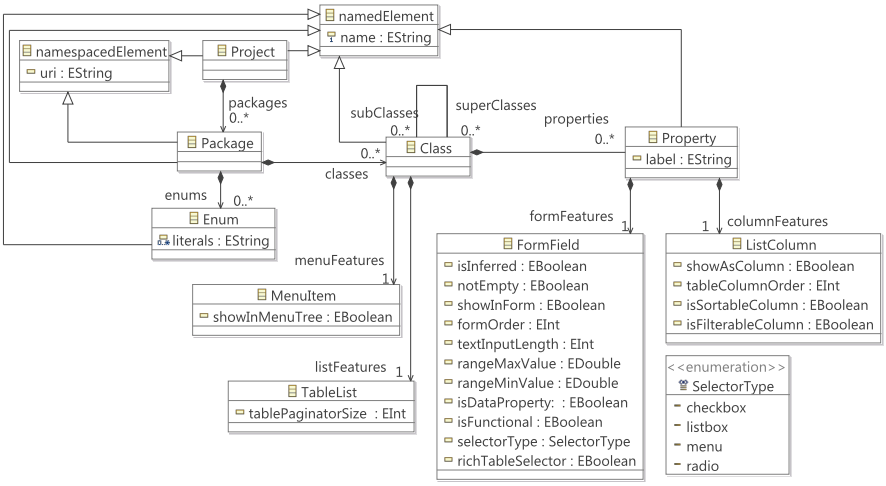


Figura D.1: Metamodelo de Presentación Web para complementar OWL

D.2 Paquete *owl2web*

D.2.1 Metaclases

Clase **Class**

La clase `Class` hereda de `namedElement`. Los atributos y asociaciones de la clase `Class` se muestran en la tabla D.1.

Tabla D.1: Propiedades de la clase: `Class`

Features	*Type*	*Description*
properties	Property [0..*]	
superClasses	Class [0..*]	
subClasses	Class [0..*]	
menuFeatures	MenuItem [1..1]	
listFeatures	TableList[1..1]	

Clase **Enum**

La clase `Enum` hereda de `namedElement`. Los atributos y asociaciones de la clase `Enum` se muestran en la tabla D.2.

Tabla D.2: Propiedades de la clase: `Enum`

Features	*Type*	*Description*
literals	EString[0..*]	

Clase **FormField**

Los atributos y asociaciones de la clase `FormField` se muestran en la tabla D.3.

Clase **ListColumn**

Los atributos y asociaciones de la clase `ListColumn` se muestran en la tabla D.4.

Clase **MenuItem**

Los atributos y asociaciones de la clase `MenuItem` se muestran en la tabla D.5.

Tabla D.3: Propiedades de la clase: `FormField`

Features	*Type*	*Description*
isInferred	EBoolean [0..1]	
notEmpty	EBoolean [0..1]	
showInForm	EBoolean [0..1]	
formOrder	EInt [0..1]	
textInputLength	EInt [0..1]	
rangeMaxValue	EDouble [0..1]	
rangeMinValue	EDouble [0..1]	
isDataProperty:	EBoolean [0..1]	
isFunctional	EBoolean [0..1]	
selectorType	SelectorType [0..1]	
richTableSelector	EBoolean [0..1]	

Tabla D.4: Propiedades de la clase: `ListColumn`

Features	*Type*	*Description*
showAsColumn	EBoolean[0..1]	
tableColumnOrder	EInt [0..1]	
isSortableColumn	EBoolean[0..1]	
isFilterableColumn	EBoolean[0..1]	

Tabla D.5: Propiedades de la clase: `MenuItem`

Features	*Type*	*Description*
showInMenuTree	EBoolean[0..1]	

Clase **Package**

La clase `Package` hereda de `namespacedElement`, `namedElement`. Los atributos y asociaciones de la clase `Package` se muestran en la tabla D.6.

Tabla D.6: Propiedades de la clase: `Package`

Features	*Type*	*Description*
classes	Class[0..*]	
enums	Enum [0..*]	

Clase **Project**

La clase `Project` hereda de `namespacedElement`, `namedElement`. Los atributos y asociaciones de la clase `Project` se muestran en la tabla D.7.

Tabla D.7: Propiedades de la clase: `Project`

Features	*Type*	*Description*
packages	Package[0..*]	

Clase **Property**

La clase `Property` hereda de `namedElement`. Los atributos y asociaciones de la clase `Property` se muestran en la tabla D.8.

Tabla D.8: Propiedades de la clase: `Property`

Features	*Type*	*Description*
label	EString [0..1]	
notifiesTo	Property [0..*]	
isNotifiedBy	Property [0..*]	
domainClass	Class [1..1]	
formFeatures	FormField [1..1]	
columnFeatures	ListColumn[1..1]	

Clase **TableList**

Los atributos y asociaciones de la clase `TableList` se muestran en la tabla D.9.

Tabla D.9: Propiedades de la clase: `TableList`

Features	*Type*	*Description*
tablePaginatorSize	EInt[0..1]	

Clase **namedElement**

Los atributos y asociaciones de la clase `namedElement` se muestran en la tabla D.10.

Tabla D.10: Propiedades de la clase: `namedElement`

Features	*Type*	*Description*
name	EString[1..1]	

Clase **namespacedElement**

Los atributos y asociaciones de la clase `namespacedElement` se muestran en la tabla D.11.

Tabla D.11: Propiedades de la clase: `namespacedElement`

Features	*Type*	*Description*
uri	EString[0..1]	

D.2.2 Enumeraciones

Enumeración **SelectorType**

Los literales de la enumeración *SelectorType* se muestran en la tabla D.12.

Tabla D.12: Enum literals: *SelectorType*

Literal	*Description*
checkbox	
listbox	
menu	
radio	

D.3 Resumen

A modo de resumen, a continuación se presentan una serie de métricas sobre el metamodelo descrito. El paquete está formado por:

- Clases: 11 (0 abstractas)
  - Número total de atributos: 21
  - Número total de asociaciones: 13
  - Número total de relaciones de herencia: 7
  - La media de atributos por clase es: 1.90
  - La media de asociaciones por clase es: 1.18
  - El número medio de hijos por clase es: 0.63
- Enumeraciones: 1

- Número total de literales: 4
- La media de literales por enumeración es: 4