BASELINE-ORIENTED MODELING: UNA APROXIMACIÓN MDA BASADA EN LÍNEAS DE PRODUCTOS SOFTWARE PARA EL DESARROLLO DE APLICACIONES

Tesis presentada por:

María Eugenia Cabello Espinosa

al **Departamento de Sistemas Informáticos y Computación**
como requisito parcial para la obtención del título de
Doctor en Programación Declarativa e Ingeniería de la Programación

Director de tesis: **Dr. Isidro Ramos Salavert**

Valencia-España, Noviembre de 2008

RESUMEN

Esta tesis presenta la aproximación denominada *Baseline-Oriented Modeling* (BOM). BOM es un *framework* que genera aplicaciones, basado en Líneas de Productos Software (LPS). Para ilustrar BOM, se ha elegido al dominio de los Sistemas Expertos que realizan tareas de diagnóstico (SED), como el tipo de aplicación desarrollada en BOM.

En BOM se generan aplicaciones en un dominio específico, basadas en la estructura y comportamiento de los SED, como modelos arquitectónicos PRISMA, utilizando técnicas de LPS. Este proceso implica, por un lado construir una *Baseline* (como repositorio que contiene todos los *assets* necesarios para obtener un producto de la LPS), y por otro lado realizar el plan de producción de la LPS.

BOM gestiona la especificación de la variabilidad y la funcionalidad en modelos separados, representados a través de modelos conceptuales. La variabilidad se maneja en dos pasos: en el primer paso la variabilidad del dominio se refleja en las arquitecturas base de la LPS, que comparten una arquitectura genérica, y en el segundo paso la variabilidad del dominio de aplicación específico se instancia para obtener la arquitectura específica PRISMA como el producto final de la LPS.

BOM sigue la iniciativa *Model-Driven Architecture* (MDA) del *Object Management Group* (OMG) para construir modelos de dominio (como Modelos Independientes de Computación), que son transformados en modelos arquitectónicos PRISMA (como Modelos Independientes de Plataforma) y compilados a una aplicación ejecutable en .NET (como Modelos Específicos de Plataforma). Lenguajes Específicos de Dominio han sido utilizados para que la interfaz del usuario sea amigable, clara y sencilla.

PALABRAS CLAVE

reutilización del software, líneas de producto software, variabilidad, modelos conceptuales, arquitecturas software, vistas software, transformación de modelos, sistemas expertos, diagnóstico.

RESUM

Aquesta tesi presenta l'aproximació anomenada *Baseline-Oriented Modeling* (BOM). BOM és un *framework* que genera aplicacions, basat en Línies de Producte Software (LPS). Per tal d' il.lustrar BOM, s'ha escollit el domini dels Sistemes Experts que realitzen tasques de diagnòstic (SED), com el tipus d'aplicacions a desenvolupar en BOM.

En BOM, es generen aplicacions en un domini específic, basades en l'estructura i comportament dels SED, com a models arquitectònics PRISMA, utilitzant tècniques de LPS. Aquest procés implica, d'una banda construir una *Baseline* (com a magatzem que conté tots els *assets* necesaris per a construir un producte de la LPS), i per l'altra realitzar el pla de producció de la LPS.

BOM gestiona l'especificació de la variabilitat i la funcionalitat en models separats, representats mitjanaçant models conceptuals. La variabilitat es maneja en dos pasos: en el primer pas, la variabilitat del domini és reflectada en les arquitectures base de la LPS, que comparteixen una arquitectura genèrica, i en el segón pas, la variabilitat del domini d'aplicació específic, és instanciada per a obtindre l'arquitectura específica PRISMA com el producte final de la LPS.

BOM segueix l'iniciativa *Model-Driven Architecture* (MDA) del *Object Management Group* (OMG) per a construir models de domini (com a Models Independents de Computació), que són transformats automàticament en models arquitectònics PRISMA (com a Models Independents de Plataforma) i compilats en una aplicació executable en .NET (com a Models Específics de Plataforma). Un Llenguatge Específic de Domini ha sigut utilitzat per a que l'interfice d'usuari siga amistosa, clara i senzilla.

PARAULES CLAU

reutilització del software, línies de producte software, variabilitat, models conceptuals, arquitectures software, vistes software, transformació de models,. sistemes experts, diagnòstic.

ABSTRACT

This thesis presents the approach called Baseline-Oriented Modeling (BOM). BOM is a framework that generates applications based on Software Product Lines (SPL). In order to illustrate BOM, it´s been chosen the Expert Systems domain which realize diagnostic tasks (ESD), as the type of application developed in BOM.

In this context, in BOM, the applications are generated in an specific domain, based in the structure and behaviour of the ESD, as PRISMA architectural models using SPL techniques. This process involves in the first place, to build a Baseline (as a repository that contains all the necessary assets to obtain an SPL product), and in second place to realize the Production Plan of the SPL.

BOM manages the variability and the functionality specification in separate models; represented through conceptual models. The variability is managed in two steps: in the first step the domain variability is reflected in the base architectures of the SPL that share a generic architecture, and in the second step the specific application domain variability is instanced in the base architectures in order to obtain the specific PRISMA architecture as the final product of the SPL.

BOM follows the Model-Driven Architecture (MDA), initiative of the Object Management Group (OMG), to build domain models (as Computational Independent Models) that are transformed into PRISMA architectural models (as Platform Independent Models) and then compiled to an executable application in .NET (as Platform Specific Models). Domain Specific Languages have been used in order to offer friendly and simple GUI to the BOM end-user .

KEY WORDS

software reuse, software product lines, variability, conceptual models, software architectures, software views, model transformations, expert systems, diagnosis.

ÍNDICE GENERAL

PARTE I. INTRODUCCIÓN 21

Capítulo 1. Introducción 23

1.1 Planteamiento del problema y justificación del trabajo 23
1.2 Hipótesis y objetivos de la tesis 27
1.2.1 Hipótesis 27
1.2.2 Objetivos 27
1.2.2.1 Objetivo general 27
1.2.2.2 Objetivos específicos 27
1.3 Estructura de la tesis 29

PARTE II. ESTADO DEL ARTE 31

Capítulo 2. Estado del arte 33

2.1 Introducción 33
2.2 Trabajos relacionados 33
2.3 Ingeniería de software dirigida por modelos 37
2.3.1 Arquitecturas software dirigida por modelos 39
2.3.1.1 MDA y las vistas de arquitecturas software 41
2.4 Líneas de productos software 42
2.4.1 Repositorio de la línea de productos software 46
2.4.2 Arquitectura de la línea de productos software 46
2.4.3 Aproximación para los procesos de la ingeniería de la línea de productos software 48
2.4.4 Estándares de la OMG utilizados en BOM como técnicas de modelado 50
2.4.4.1 Metamodelo de ingeniería de procesos de software 51
2.4.4.2 Especificación de activos reutilizables 52
2.4.4.3 Notación para el modelado de procesos de negocio 52
2.4.4.4 Lenguaje de modelado unificado 53
2.4.4.5 Facilidad de Meta Objetos 54
2.4.5 La variabilidad en las Líneas de Productos Software 55
2.4.5.1 Representación de la variabilidad 57
2.4.5.1.1 Primera clasificación para representar la variabilidad: lenguajes de modelado 58
2.4.5.1.2 Segunda clasificación para representar la variabilidad: lenguajes textual y gráfico 59
2.4.5.2 Gestión de la variabilidad 63
2.4.6 Herramientas utilizadas en la línea de productos software 63
2.5 El modelo PRISMA 68
2.6 Los Sistemas Expertos 71
2.7 Conclusiones 75

PARTE III. PRELIMINARES 77

Capítulo 3. Las arquitecturas software de la Línea de Productos definida en BOM 79

3.1 Introdución 79
3.2 La arquitectura de la Línea de Productos Software 79
3.3 La arquitectura de los Sistemas Expertos 81

3.4 La arquitectura genérica y las arquitecturas base de los Sistemas Expertos de Diagnóstico en BOM 82
3.5 Conclusiones 89

Capítulo 4. El Diagnóstico: Variabilidad del dominio 91
4.1 Introducción 91
4.2 Estudio de campo 91
4.2.1 Casos de estudio 95
4.2.1.1 El diagnóstico médico 95
4.2.1.2 El diagnóstico de emergencias 96
4.2.1.3 El diagnóstico educativo (caso: diagnóstico de candidatos a becas) 98
4.2.1.4 El diagnóstico de programas educativos (caso: diagnóstico de programas educativos) 99
4.2.1.5 El diagnóstico televisivo 100
4.3 Análisis de las características asociadas al diagnóstico 103
4.3.1 Análisis de las características involucradas en el proceso del diagnóstico 103
4.3.2 Análisis de las características involucradas en los requisitos del usuario 104
4.3.3 Análisis de las características del dominio 104
4.3.4 Análisis de las características del dominio de aplicación 108
4.4 Conclusiones 109

Capítulo 5. Los Sistemas Expertos de Diagnóstico: Funcionalidad del dominio 111
5.1 Introducción 111
5.2 Variabilidad en la estructura de los sistemas expertos de diagnóstico 111
5.3 Variabilidad en el comportamiento de los sistemas expertos de diagnóstico 115
5.3.1 Procesos de inferencia 115
5.3.1.1 Proceso de inferencia estático 116
5.3.1.2 Proceso de inferencia dinámico 118
5.4 Conclusiones 121

PARTE IV. APROXIMACIÓN: BASELINE-ORIENTED MODELING 123

Capítulo 6. Gestión de la variabilidad en BOM 125
6.1 Introducción 125
6.2 La gestión de la variabilidad en BOM 125
6.3 La primera variabilidad en BOM 127
6.4 La segunda variabilidad en BOM 129
6.5 Conclusiones 136

Capítulo 7. Aproximación BOM-EAGER: Desarrollo de la LPS mediante técnicas "ad-hoc" utilizadas en BOM para el tratamiento de la variabilidad 139
7.1 Introducción 139
7.2 Tratamiento de la primera variabilidad: de la arquitectura genérica a las arquitecturas base. 139
7.3 Tratamiento de la segunda variabilidad: de la arquitectura base a la arquitectura PRISMA. 142
7.4 Desarrollo de Sistemas Expertos de Diagnóstico en BOM 143
7.5 Conclusiones 146

Capítulo 8. Aproximación BOM-LAZY: Desarrollo de la LPS mediante técnicas de transformación de modelos para el tratamiento de la variabilidad 149
8.1 Introducción 149
8.2 Vistas de los sistemas expertos de diagnóstico en BOM 149
8.3 Metamodelos de las vistas software 150

8.4 Relaciones entre metamodelos y transformaciones entre modelos 154
8.5 Relaciones entre vistas software.. 156
8.6 Transformaciones entre modelos... 159
8.7 El proceso BOM de transformación de modelos.. 162
8.8 Conclusiones 164

Capítulo 9. Modelado del proceso de desarrollo de la LPS en BOM........................... 165
9.1 Introducción 165
9.2 Modelado del proceso del desarrollo de software para la creación de la LPS... 165
9.3 Ingeniería del dominio: creación de la Baseline.. 170
9.3.1 Análisis del dominio.. 171
9.3.2 Desarrollo de los componentes básicos reutilizables.................................. 171
9.4 Ingeniería de la aplicación: ejecución del Plan de Producción........................... 187
9.4.1 Plan de producción de la LPS en la aproximación BOM-EAGER............. 188
9.4.1.1 Caracterización del producto 190
9.4.1.2 Síntesis del producto.. 193
9.4.1.3 Construcción del producto... 196
9.4.2 Plan de producción de la LPS en la aproximación BOM-LAZY 197
9.4.3 Ejecución del sistema (producto final) 199
9.5 Conclusiones 199

PARTE V. IMPLEMENTACIÓN.. 201

Capítulo 10. Implementación de la aproximación BOM-EAGER: ProtoBOM............ 203
10.1 Introducción 203
10.2 Características de ProtoBOM... 203
10.3 ProtoBOM en la ingeniería del dominio. 206
10.4 ProtoBOM en la ingeniería de la aplicación.. 214
10.5 Conclusiones 223

Capítulo 11. Implementación de la aproximación BOM-LAZY................................... 225
11.1 Introducción 225
11.2 Características de la implementación BOM-LAZY en QVT 225
11.3 Desarrollo de la herramienta 227
11.4 Conclusiones 243

PARTE VI. CONCLUSIONES 245

Capítulo 12. Conclusiones y trabajo futuro... 247
12.1 Conclusiones generales 247
12.2 Contribuciones 250
12.3 Trabajo futuro.. 253

BIBLIOGRAFÍA 255

APÉNDICES 273

Apéndice A. Terminología del diagnóstico 275

Apéndice B. Conceptos utilizados en el modelado de los procesos de la ingeniería en la Línea de Productos Software 281

Apéndice C. Caso de estudio: Diagnóstico de Programas Educativos......................... 287

Apéndice D. Caso de estudio: Diagnóstico Médico... 315

Apéndice E. Implementación de ProtoBOM *355*
Apéndice F. Implementación en QVT-Relations *377*

LISTADO DE FIGURAS

Figura 1 Nociones básicas en MDE (figura adaptada de [Bézivin, 2005]) 38
Figura 2 Las tres actividades esenciales para la LPS [Clements et al., 2001] 49
Figura 3 Desarrollo de la LPS [Clements et al., 2001] 50
Figura 4 Metamodelo de variabilidad [Bachmann et al., 2003] 61
Figura 5 Notación gráfica utilizada en el modelo de características clásico 62
Figura 6 Vistas interna y externa de un elemento arquitectónico PRISMA [Pérez, 2006] 68
Figura 7 Tareas de los sistemas expertos [JIPDEC, 1989] 74
Figura 8 Taxonomía de los sistemas software 81
Figura 9 Arquitectura de un Sistema Experto 82
Figura 10 Metáfora visual de la relación que exite entre los representantes canónicos de la LPS 83
Figura 11 Módulos esenciales que integran un Sistema Experto de Diagnóstico 84
Figura 12 Arquitectura genérica en el dominio de los Sistemas Expertos de Diagnóstico 84
Figura 13 Correspondencia entre (a) la arquitectura genérica de un SED y (b) una arquitectura base 85
Figura 14 Correspondencia entre (a) la arquitectura genérica de un SED y (b) una arquitectura base 86
Figura 15 Metáfora visual de (a) una única arquitectura genérica, y (b) dos arquitecturas base 86
Figura 16 Mapping entre la taxonomía de los SED y las arquitecturas de la LPS 88
Figura 17 Grafo del diagnóstico médico 96
Figura 18 Grafo del diagnóstico en desastres 97
Figura 19 Grafo del diagnóstico de candidatos a beca 98
Figura 20 Grafo del diagnóstico de programas educativos 99
Figura 21 Grafo del diagnóstico televisivo 100
Figura 22 Modelo de características del dominio del diagnóstico 106
Figura 23 Árbol de decisión de las características de la LPS 107
Figura 24 Modelo Conceptual del dominio del diagnóstico 108
Figura 25 Modelo conceptual del dominio de aplicación 109
Figura 26 Diagrama UML-casos de uso del diagnóstico médico 112
Figura 27 Modelos arquitectónicos correspondientes a cada caso de uso 113
Figura 28 Modelo arquitectónico de un sistema de diagnóstico médico 114
Figura 29 Grafo del proceso de inferencia estático 116
Figura 30 Modelado del proceso de inferencia estático en BPMN 117
Figura 31 Diagrama de Transición de estados del proceso de inferencia estático 117
Figura 32 Grafo del proceso de inferencia dinámico 119
Figura 33 Modelado del proceso de inferencia dinámico en BPMN 119
Figura 34 Diagrama de Transición de estados del proceso de inferencia dinámico 120
Figura 35 Clasificación de las características de la variabilidad gestionada por BOM . 127
Figura 36 Proceso de inserción de las características en un esqueleto para formar su tipo PRISMA 132
Figura 37 Proceso desde la inserción de características en los esqueletos, hasta la configuración de la arquitectura 132
Figura 38 Metáfora visual de una arquitectura base y dos arquitecturas PRISMA 134
Figura 39 Tratamiento de la variabilidad V1 a través de BOM 141
Figura 40 Tratamiento de la variabilidad V2 a través de BOM 143
Figura 41 Desarrollo de los SED en BOM 144

Figura 42 Información de entrada y salida de un SED desde el punto de vista del usuario final *146*
Figura 43 Vista del metamodelo modular (notación MOF) *152*
Figura 44 Vista del metamodelo C-C (notación MOF) *153*
Figura 45 Ejemplo de: (a) Relaciones entre dos vistas en el nivel de metamodelo, (b) Transformación de un modelo en otro modelo en el nivel de modelo [Limón et al., 2007] *154*
Figura 46 Relaciones entre metamodelos y transformaciones entre modelos, en el escenario de LPS [Limón et al., 2007] *156*
Figura 47 Diagramas y códigos de las relaciones: (a) moduleToComponent, (b) functionToService, (c) rUseModToConnector, (d) rCompositionModToComp [Limón et al., 2007] *158*
Figura 48 Las transformaciones T1 y T2 de modelos *160*
Figura 49 Las transformaciones T1 y T2 de modelos (expresados en niveles de MOF) *161*
Figura 50 Tareas y elementos involucrados en el proceso de transformación de los modelos (figura adaptada de [Limón et al., 2007]) *163*
Figura 51 Creación del Modelo de Características (notación SPEM) *172*
Figura 52 Modelo de Características *172*
Figura 53 Creación del Árbol de Decisión (notación SPEM) *172*
Figura 54 Árbol de Decisión *173*
Figura 55 Creación del Proceso de selección de assets (notación SPEM) *174*
Figura 56 Proceso de selección de assets *174*
Figura 57 Creación del Modelo Conceptual de Dominio (notación SPEM) *175*
Figura 58 Modelo Conceptual del Dominio del Diagnóstico *176*
Figura 59 Creación del Modelo Conceptual del Dominio de Aplicación (notación SPEM) *176*
Figura 60 Modelo Conceptual del Dominio de Aplicación *177*
Figura 61 Creación de los (aspectos) esqueletos (notación SPEM) *178*
Figura 62 Metáfora visual de un esqueleto (aspecto) *178*
Figura 63 Metáfora visual de una interfaz tipo PRISMA *179*
Figura 64 Metáfora visual de un componente (a) y un conector (b) tipos PRISMA *179*
Figura 65 Creación de los artefactos tipo PRISMA (notación SPEM) *180*
Figura 66 Creación de las configuraciones de los modelos arquitectónicos (notación SPEM) *180*
Figura 67 Creación del Proceso principal de inserción de características (notación SPEM) *181*
Figura 68 Proceso principal de inserción de características *181*
Figura 69 Creación de los Procesos de inserción de características (notación SPEM) .. *182*
Figura 70 Creación de los esqueletos aspecto-proceso (notación SPEM) *182*
Figura 71 Creación de los Híbridos empaquetados (notación SPEM) *183*
Figura 72 Creación de los modelos RAS (notación SPEM) *183*
Figura 73 Creación de las cajas (notación SPEM) *184*
Figura 74 Ejemplo de una caja (kit-box) *184*
Figura 75 Creación del Plan de Producción (notación SPEM) *185*
Figura 76 El plan de producción de la LPS en BOM *186*
Figura 77 Creación de la Baseline (notación SPEM) *186*
Figura 78 La Baseline *187*
Figura 79 Plan de Producción de la aproximación BOM-EAGER (notación SPEM) *189*
Figura 80 Proceso en SPEM para obtener las características del dominio *190*
Figura 81 Proceso en SPEM para seleccionar asset *191*
Figura 82 Proceso en SPEM para desempaquetar una caja *192*

Figura 83 Proceso en SPEM para obtener las características del dominio de aplicación 192
Figura 84 Proceso en SPEM para obtener los aspectos tipo PRISMA 193
Figura 85 Proceso en SPEM para construir la especificación PRISMA 194
Figura 86 Metáfora visual del sistema del diagnóstico de programas educativos (realizada con la herramienta PRISMA-CASE).. 195
Figura 87 Proceso en SPEM para compilar el modelo arquitectónico PRISMA............. 196
Figura 88 Proceso en SPEM para crear el sistema ejecutable 197
Figura 89 Plan de Producción de la aproximación BOM-LAZY (notación SPEM)........ 198
Figura 90 Ejecución de un sistema de la LPS (con notación en SPEM) 199
Figura 91 Casos de uso en la ingeniería del dominio 205
Figura 92 Casos de uso en la ingeniería de la aplicación 205
Figura 93 IGU inicial de ProtoBOM.. 206
Figura 94 IGU para crear la Baseline 207
Figura 95 IGU para empaquetar una caja a la Baseline 208
Figura 96 IGU para empaquetar artefactos software en una caja................................... 209
Figura 97 IGU para empaquetar un aspecto-proceso esqueleto...................................... 210
Figura 98 IGU para empaquetar un híbrido 211
Figura 99 IGU para empaquetar el Plan de Producción en la Baseline......................... 212
Figura 100 IGU para crear el Modelo de Características.. 213
Figura 101 IGU para crear el Árbol de Decisión 214
Figura 102 IGU para la ejecución del Plan de Producción.. 215
Figura 103 IGU para obtener las características del dominio 216
Figura 104 IGU para obtener las características del dominio de aplicación.................. 217
Figura 105 IGU para obtener las propiedades del dominio de aplicación..................... 217
Figura 106 IGU para obtener las hipótesis del dominio de aplicación 218
Figura 107 IGU para obtener las reglasde derivación del dominio de aplicación.......... 218
Figura 108 IGU para construir la especificación PRISMA 219
Figura 109 IGU para ingresar la especificación PRISMA a las herramientas PRISMA y crear el ejecutable del sistema final 220
Figura 110 IGUs inicial y final del sistema ejecutable (caso diagnóstico de programas educativos).. 221
Figura 111 IGUs inicial y final del sistema ejecutable (caso diagnóstico médico) 222
Figura 112 Arquitectura de la implementación BOM-LAZY... 226
Figura 113 IGU de las invocaciones para ejecutar las transformaciones....................... 227
Figura 114 IGU del meta modelo Modular... 229
Figura 115 IGU del metamodelo Componente-Conector.. 230
Figura 116 IGU del modelo modular de los SED 231
Figura 117 IGU del árbol del modelo modular de los SE.. 232
Figura 118 IGU de las QVT-Relations Caso:Diagnóstico de programas educativos...... 233
***Figura 119 IGU del Modelo Conceptual del Dominio del Diagnóstico**........................ 234*
Figura 120 IGU del modelo C-C del SE del diagnóstico de programas educativos 235
Figura 121 IGU del árbol del modelo C-C del SED del diagnóstico de programas educativos 236
Figura 122 IGU de las trazas entre los modelos modular y C-C del SED del diagnóstico de programas educativos.. 237
Figura 123 IGU de la instancia del DCM del diagnóstico de programas educativos...... 238
Figura 124 IGU del árbol del modelo C-C del SED del diagnóstico médico.................. 239
Figura 125 IGU del árbol del modelo C-C del SED del diagnóstico médico.................. 240
Figura 126 IGU de las trazas entre los modelos modular y C-C del SED del diagnóstico médico... 241

Figura 127 IGU de la instancia del DCM del diagnóstico médico 242
Figura 128 Grafo del razonamiento deductivo.. 278
Figura 129 Grafo del razonamiento inductivo 279
Figura 130 Grafo del razonamiento diferencial... 280
Figura 131 Características del diagnóstico de programas educativos indicadas en el Modelo de Características.. 289
Figura 132 Características del diagnóstico de programas educativos indicadas en el Árbol de Decisión 290
Figura 133   Grafo que muestra un diagnóstico de programas educativos...................... 290
Figura 134   Diagrama de casos de uso correspondiente al diagnóstico de programas educativos 291
Figura 135   Trayectoria del árbol de decisión computada por el proceso de selección de assets correspondiente al diagnóstico de programas educativos 292
Figura 136   Modelo arquitectónico del caso de estudio: diagnóstico de programas educativos 294
Figura 137   Proceso de inferencia estático del caso de estudio: diagnóstico de programas educativos 296
Figura 138 Entrada y salida del sistema de diagnóstico de programas educativos......... 297
Figura 139 IGUs del sistema ejecutable   (caso de estudio: diagnóstico de programas educativos)... 298
Figura 140 Características del diagnóstico médico indicadas en el Modelo de Características.. 318
Figura 141 Características del diagnóstico médico indicadas en el Árbol de Decisión .. 318
Figura 142   Grafo que muestra un diagnóstico médico 319
Figura 143   Diagrama de casos de uso correspondiente al diagnóstico médico 320
Figura 144 Trayectoria del árbol de decisión computada por el proceso de selección de assets correspondiente al diagnóstico médico.. 321
Figura 145 Modelo arquitectónico del caso de estudio: diagnóstico médico 323
Figura 146 Proceso de inferencia dinámico del caso de estudio: diagnóstico médico 324
Figura 147 Entrada y salida del sistema de diagnóstico médico 325
Figura 148 IGUs del sistema ejecutable   (caso de estudio: diagnóstico médico) 326
Figura 149   Arquitectura de ProtoBOM 355
Figura 150 Tablas de la base de datos 356
Figura 151 Metáfora visual del resultado del proceso de inserción de características ... 365

LISTADO DE TABLAS

Tabla 1 Procesos de la ingeniería en la LPS [Clements et al., 2001] *49*
Tabla 2 Comparación (a nivel funcional) de las herramientas para el desarrollo de LPS (Tabla adaptada del AMPLE D.3.1 project, [AMPLE,2007]) *67*
Tabla 3 Características de los casos de estudio *102*
Tabla 4 Resumen de la variabilidad del dominio en los casos de estudio *105*
Tabla 5 Ejemplo de un aspecto esqueleto y su correspondiente tipo PRISMA *133*
Tabla 6 Ejemplo de un aspecto esqueleto y dos aspectos tipo PRISMA (casos: Diagnóstico de programas educativos y Diagnóstico televisivo) *135*
Tabla 7 Metáfora visual de la variabilidad reflejada en las arquitecturas de los SED de los casos de estudio *145*
Tabla 8 Algunas relaciones identificadas en el metamodelo modular y el metamodelo C-C [Limón et al., 2007] *155*
Tabla 9 Iconos estándar de SPEM utilizados en el modelado de la LPS *166*
Tabla 10 Iconos de SPEM creados exprofeso para el modelado de la LPS *168*
Tabla 11 Puntos de variabilidad y sus variantes *173*
Tabla 12 Trayectorias del árbol de decisión *175*
Tabla 13 Iconos estándar de SPEM utilizados en el modelado de la LPS *284*
Tabla 14 Iconos de SPEM creados exprofeso para el modelado de la LPS *285*

ACRÓNIMOS

ALP Arquitectura de la Línea de Productos

BOM Baseline-Oriented Modeling

CIM Computational Independent Model (Modelo Independiente de Computación)

DSL Domian Specific Languaje **(**Lenguaje Específico de Dominio)

DSBC Desarrollo de Software Basado en Componentes

DSOA Desarrollo de Software Orientado a Aspectos

FOM Feature Oriented Modeling (Modelado Orientado a Caracterísiticas)

FOP Feature Oriented Programming (Programación Orientada a Caracterísiticas)

IGU Interfaz Gráfica del Usuario

LDA Lenguaje de Descripción de Arquitecturas

LPS Línea de Productos Software

MDA Model Driven Architecture (Arquitectura Dirigida por Modelos)

MDE Model Driven Engineering (Ingeniería Dirigida por Modelos)

MOF Meta Object Facility

OMG Object Management Group

PIM Platform Independent Model (Modelo Independiente de Plataforma)

PSM Platform Specific Model (Modelo Específico de Plataforma)

RAS Reusable Asset Specification (Especificación de Activos Reutilizables)

SE Sistema Experto

SED Sistema Experto de Diagnóstico

SPEM Software Process Engineering Metamodel (Metamodelo para la Ingeniería de Procesos Software)

UML Unified Modeling Language (Lenguaje Unificado de Modelado)

VFS Vista Funcional del Sistema

VVS Vista de la Variabilidad del Sistema

PARTE I. INTRODUCCIÓN

GESTACIÓN

CAPÍTULO 1

INTRODUCCIÓN

Seis honrados servidores me enseñaron cuanto sé; sus nombres son cómo, cuándo, dónde, qué, quién y por qué.
Rudyard Kipling (1865-1936). Novelista británico

Esta tesis presenta la aproximación denominada Baseline-Oriented Modeling (BOM).

BOM es un *framework* que genera aplicaciones en un dominio específico. BOM captura en modelos conceptuales la información del dominio, utilizando técnicas del Model Driven Architecture (MDA) [MDA, 2003] y de Líneas de Productos Software (LPS) [LPS] para generar sistemas como arquitecturas software PRISMA [Pérez, 2006].

Para ilustar la aproximación BOM en esta tesis, se ha elegido el dominio de los Sistemas Expertos (SE) [Turban et al., 2001] que realizan diagnóstico.

La estructura de este capítulo es la siguiente: En la sección 1 se presenta el planteamiento del problema y la justificación de este trabajo de investigación. La sección 2 contiene la hipótesis y los objetivos de esta tesis. En la sección 3, se comenta brevemente el contenido de los capítulos restantes.

1.1 Planteamiento del problema y justificación del trabajo

Una clase de sistemas que está cobrando interés en los últimos tiempos son los SE. Pero el desarrollo de este tipo de sistemas es complejo, dado que los elementos básicos que conforman su arquitectura varían tanto en su comportamiento como en su estructura. Por consiguiente, existe la necesidad de soportarlos adecuadamente. Las metodologías y aplicaciones desarrolladas de los SE forman una amplia categoría de productos de investigación, ofreciendo ideas y soluciones a dichos sistemas en dominios específicos. Cabe señalar que es notable la importancia que

han cobrado en los últimos años los sistemas expertos orientados a diagnóstico o sistemas de diagnóstico simplemente.

Por otro lado, en nuestros días existen una gran cantidad y variedad de trabajos relacionados con el desarrollo de sistemas software complejos: modelos arquitectónicos, componentes, aspectos y lenguajes de definición de arquitecturas. Todo ello persigue un consenso, aún no logrado, en los conceptos básicos de sistemas software complejos, su definición precisa y la metodología a aplicar para su desarrollo.

Para dar cuenta de los sistemas con requisitos software complejos surge el modelo PRISMA, presentando ventajas en la construcción de modelos arquitectónicos complejos, y cubriendo el hueco existente en el modelado de sistemas software altamente reconfigurables y reutilizables dentro de un marco de calidad controlada. El modelo arquitectónico PRISMA integra dos aproximaciones: el Desarrollo de Software Basado en Componentes-DSBC [Szyperski, 1998] y el Desarrollo de Software Orientado a Aspectos-DSOA [AOSD, 2001]. Esta integración se consigue definiendo los elementos arquitectónicos mediante aspectos. De esta forma, el modelo PRISMA, además de definir los elementos arquitectónicos básicos y especificar su sintaxis y semántica, también especifica los aspectos que detallan las propiedades necesarias de cada uno de ellos.

Además, el desarrollo de sistemas complejos se está complicando cada vez más debido a una serie de factores, como la aparición de nuevas tecnologías (Internet e Intranet), la interconexión de varios sistemas y plataformas, la necesidad de integrar viejos sistemas aun válidos (Legacy Systems), la adaptación personalizada del software a cada tipo de usuario, las necesidades específicas de un sistema y las diferentes plataformas de implementación. Esta situación lleva a necesitar, en cortos periodos, múltiples versiones de la misma o parecida aplicación. Por ello, la Ingeniería del Software debe proporcionar herramientas y métodos que permitan desarrollar una familia de productos con distintas capacidades y adaptables a situaciones variables, y no sólo un único producto.

Ante esta situación, surge el concepto de LPS [Clements et al, 2001], con la finalidad de controlar y minimizar los altos costos del desarrollo de software. Esta situación provocó la creación de un diseño que puede ser compartido por todos los miembros de una familia de programas (la arquitectura básica). De esta manera, un diseño hecho explícitamente para un producto, beneficia al software común y puede ser usado en diferentes productos, reduciendo los gastos generales y el tiempo para construir nuevos productos. Asimismo, se destaca la importancia de los planes de producción, preferiblemente automatizables, al ofrecer tiempos y costes de ejecución predecibles.

Así mismo, se considera que existen varias razones para que los SE que realizan tareas de diagnóstico estén más inclinados al paradigma LPS que al software tradicional:

- La primera razón es la heterogeneidad de las aplicaciones del dominio, dado que en cada caso de estudio, las características del diagnóstico difieren de un caso al otro. La variabilidad de las LPS permite abordar esa heterogeneidad.
- La segunda razón se debe a que los sistemas de diagnóstico en dominios específicos han sido tratados de forma particular para realizar el diagnóstico de un caso de estudio, sin embargo las aplicaciones del diagnóstico involucran una amplia gama de áreas de aplicación.
- Una tercera razón es la rapidez con la que cambia la tecnología. La LPS facilita el desarrollo de los productos en distintas plataformas y su uso en distintas tecnologías.
- La cuarta razón se basa en el propio proceso de inferencia del diagnóstico, el cual cambia según el dominio de aplicación, por lo que varias estrategias de razonamiento están implicadas en los procesos del diagnóstico. La LPS permite considerar estas estrategias de razonamiento como una más de las "características" que diferenciarán a la LPS.
- Finalmente, la quinta razón (expresada por [Atkinson et al., 2000]), se debe a que el desarrollo de software basado en componentes y la ingeniería de la línea de productos comparten los mismos fines desde diferentes puntos de vista, y por lo tanto, se pueden obtener beneficios en su integración.

Uno de los elementos clave para una LPS es la representación y gestión de la variabilidad. En este contexto se ha usado la iniciativa de la MDA en la construcción de modelos de dominio y su posible transformación en modelos arquitectónicos. La novedad que propone MDA es la posibilidad de automatizar la transformación que especifica cómo se convierten los modelos en una aplicación ejecutable.

Pero "nada sale de la nada", por lo que la generación automática de sistemas software es posible si existe algún *framework* que lo respalde. Ante esto, The Object Management Group (OMG) [OMG] dentro de la tendencia Model Driven Engineering (MDE) [Kent, 2002] (que promueve el uso de los modelos, sus relaciones y sus transformaciones, como artefactos de primera clase a partir de los cuales generar código) propone The Model-Driven Architecture (MDA) [MDA, 2003] (que defiende el uso de estándares y potencia la independencia de plataforma en los procesos de desarrollo de software dirigidos por modelos, como una nueva forma de generar aplicaciones).

Para tratar la variabilidad han sido realizadas y discutidas diferentes aproximaciones en los últimos años. Sin embargo, actualmente no existe una forma estándar para representarla. Muchas de las aproximaciones propuestas agregan

anotaciones de variabilidad directamente en los modelos que captan la funcionalidad del software (como el Lenguaje de Modelado Unificado - del inglés Unified Modeling Language-UML [UML, 2005], los Lenguajes de Descripción de Arquitectruas - del inglés Architectural Description Languages-ADLs, etc.), mientras que otras aproximaciones más poderosas separan la especificación de la variabilidad en modelos específicos.

Tomando en cuenta lo anteriormente expuesto, y ante la necesidad de:

- crear sistemas en diferentes dominios,
- minimizar los costes de producción reutilizando "paquetes de software",
- generar código automáticamente con la finalidad de incrementar la productividad y la calidad, y disminuir el tiempo al mercado,
- construir un sistema de una forma sencilla utilizando las ontologías de dominio que faciliten la comunicación hombre-máquina, y
- desarrollar sistemas independiente de plataforma y que sean abordados desde la perspectiva del problema y no de la solución, permitiendo generalidad a la aproximación desarrollada y aplicabilidad a diversos dominios,

esta tesis plantea cómo generar sistemas en un dominio específico, basados en líneas de producto, a través del *framework* Baseline-Oriented Modeling (BOM). Sin embargo, no se automatiza todo, es necesaria la intervención de un usuario que interaccione con el sistema, aportando información sobre la variabilidad del dominio a un alto nivel de abstracción. El usuario de BOM, en el rol del ingeniero de aplicación, para desarrollar un sistema concreto aportará al sistema la información mínima necesaria del dominio particular.

El trabajo de esta tesis integra diferentes espacios tecnológicos con el fin de disminuir la complejidad del problema en el desarrollo del software de los SE:

a) Los Sistemas Expertos, para capturar el conocimiento de los expertos y tratar de imitar sus procesos de razonamiento cuando resuelven un problema en un cierto dominio (por ejemplo el diagnóstico);
b) El Desarrollo Dirigido por Modelos (de la OMG), para el modelado con un alto nivel de abstracción (CIM y PIM);
c) El *Framework* PRISMA, como herramienta y entorno destino (PSM);
d) Las Líneas de Productos Software, como una técnica para sistematizar el reuso en productos software.

1.2 Hipótesis y objetivos de la tesis

1.2.1 Hipótesis

Con base en lo anteriormente descrito, la hipótesis planteada en esta tesis expresa que:
"Es posible facilitar la generación de aplicaciones, utilizando técnicas de MDA (niveles de abstracción de modelado y transformación de modelos) y de líneas de producto software."

1.2.2 Objetivos

1.2.2.1 Objetivo general

El objetivo general de esta tesis, derivada de la hipótesis, es:
"Desarrollar un *framework* que permita la generación de aplicaciones, basado en líneas de producto."

1.2.2.2 Objetivos específicos

Para la realización del objetivo general se plantearon los siguientes objetivos específicos:

- Definir el *framework* BOM
- Desarrollar un prototipo de BOM
- Aplicar BOM en distintos casos de estudio

que a continuación se comentan.

1. Definir el framework BOM

Con la finalidad de ilustrar en esta tesis la aproximación BOM, se ha elegido el dominio del diagnóstico. De esta manera, primeramente se debe de realizar un estudio de campo sobre los sistemas expertos que realizan tareas de diagnóstico, y asimismo un estudio de campo del diagnóstico, con el fin de conocer la variabilidad representada por las características particulares y generales de dicho dominio. Como resultado del análisis de campo, y basándose en técnicas MDA y de LPS, el modelo PRISMA, y la estructura y funcionamiento de los SE, proceder a realizar el proceso para desarrollar la LPS para sistemas de diagnóstico. Dicho proceso implica principalmente modelar la Baseline (como repositorio de todos los *assets* necesarios para construir un producto de la LPS) y definir las transformaciones de modelos como ejecución del plan de producción de la LPS.

1.a) Realizar un estudio de campo

Para conocer el dominio del diagnóstico, se ha realizado un estudio de campo considerando cinco casos de estudio que involucran al diagnóstico médico, de programas educativos, de asignación de becas, televisivo y de emergencias. Asimismo se ha realizado un estudio de campo en los SE que realizan tareas de diagnóstico.

1.b) Desarrollar BOM

- 1.b.i) Gestionar la variabilidad en BOM.

- 1.b.ii) Desarrollar la LPS en niveles de abstracción de modelado, a través de la aproximación BOM-EAGER (mediante técnicas *ad-hoc* para el tratamiento de la variabilidad) y de la aproximación BOM-LAZY (mediante técnicas de transformación de modelos para el tratamiento de la variabilidad)

- 1.b.iii) Modelar en el estándar denominado Metamodelo de Ingeniería de Procesos de Software (del inglés Software Process Engineering Metamodel –SPEM) [SPEM, 2006], el proceso del desarrollo de la LPS en las fases de la ingeniería del dominio y de la ingeniería de la aplicación (ejecución del Plan de Producción):

- Modelar la Baseline.- Crear una Baseline, usando el estándar denominado Especificación de Recursos Reutilizables (del inglés Reusable Asset Specification-RAS) [RAS, 2005], para empaquetar e incorporar en ella todos los *assets* de su contenido. Explícitamente se modelan todos los *assets* contenidos en la Baseline, incluyendo entre ellos el plan de producción de la LPS:
    - modelando los requisitos y la funcionalidad de los SE, así como de la variabilidad de un dominio específico, mediante Modelos Independientes de Computación-CIM;
    - modelando la funcionalidad de los SE y la variabilidad de un dominio específico y de sus dominios de aplicación en modelos conceptuales independientes abstrayéndose de los detalles tecnológicos de la plataforma de implementación, mediante Modelos Independientes de Plataforma-PIM, y definiendo transformaciones entre modelos.

- Modelar la ejecución del plan de producción de LPS.- Combinar las especificaciones contenidas en el PIM con los detalles de la plataforma de implementación, originando Modelos Específicos de Plataforma-PSM. De esta manera BOM en interacción con el usuario: selecciona los assets que conforman la base de los productos de la LPS, y crea las arquitecturas PRISMA correspondientes.

2.- Desarrollar un prototipo de BOM: ProtoBOM
Implementar una herramienta que permita mostrar el uso y las ventajas de BOM. Utilizar dicha herramienta para generar aplicaciones ejecutables.

3.- Aplicar BOM en distintos casos de estudio
Aplicar BOM en un dominio específico (los Sistemas Expertos de diagnóstico), y en dos dominios de aplicación significativos: (el diagnóstico médico y el diagnóstico educativo) que ayuden a mostrar cómo BOM gestiona la variabilidad, y cómo se generan dichos sistemas (hasta su ejecución).

1.3 Estructura de la tesis

Los capítulos restantes de esta tesis están organizados de la siguiente manera:

Capítulo 2.- Estado del arte
Este capítulo presenta un panorama de las tendencias actuales más importantes en investigación en el tema de la tesis y los trabajos relacionados con los espacios tecnológicos usados: las técnicas de MDA y de LPS, el modelo arquitectónico PRISMA y los SE.

Capítulo 3.- Las arquitecturas software de la línea de productos definida en BOM
Este capítulo presenta un panorama general de la línea de productos generada por BOM: los sistemas expertos. Así mismo se presenta la arquitectura genérica de dichos sistemas y las arquitecturas base de la LPS.

Capítulo 4.- El diagnóstico: la variabilidad del dominio
En este capítulo se realiza el análisis de un dominio específico: el diagnóstico, mediante un estudio de campo realizado con cinco casos, con el fin de detectar las características más representativas de dicho dominio y sus campos de aplicación y descubrir los puntos la variabilidad.

Capítulo 5.- Los Sistemas Expertos de diagnóstico : funcionalidad del dominio
En este capítulo se realiza un análisis detallado de la variabilidad de la estructura y del comportamiento en el caso específico de los sistemas expertos que realizan tareas de diagnóstico.

Capítulo 6.- La gestión de la variabilidad en BOM
Este capítulo presenta cómo es gestionada la variabilidad del dominio a través de BOM, para desarrollar sistemas expertos.

Capítulo 7.- El desarrollo de la LPS mediante técnicas "ad-hoc" a la aproximación BOM-EAGER para el tratamiento de la variabilidad.
Este capítulo presenta el desarrollo de los sistemas expertos, mediante técnicas de árbol de decisión y de FOM (Feature Oriented Modeling) específicas de la aproximación BOM-EAGER para el tratamiento de la variabilidad.

Capítulo 8.- El desarrollo de la LPS mediante técnicas "ad-hoc" a la aproximación BOM-LAZY para el tratamiento de la variabilidad
En este capítulo se presenta el desarrollo de los sistemas expertos a través del *framework* BOM, usando técnicas de transformación de modelos para el tratamiento de la variabilidad.

Capítulo 9.- El modelado del desarrollo de la LPS en BOM
En este capítulo se modela el desarrollo de la LPS en BOM, a través de dos procesos: la ingeniería del dominio (creación de la Baseline) y la ingeniería de la aplicación (uso de la Baseline en la ejecución del plan de producción de la LPS). Se hace uso de los estándares RAS (para conformar la Baseline) y SPEM (para modelar dichos procesos).

Capítulo 10.- La implementación de la aproximación BOM-EAGER: ProtoBOM
En este capítulo se muestra la implementación de BOM-EAGER, a través de un prototipo realizado exprofeso en esta tesis.

Capítulo 11.- La implementación de la aproximación BOM-LAZY.
En este capítulo se muestra la implementación de las transformaciones de los modelos modular a componente-conector utilizando Query/View/Transformations-Relations-QVT-relations [QVT,2005], realizadas exprofeso en esta tesis.

Capítulo 12.- Conclusiones
Las conclusiones derivadas de la tesis, incluyendo las principales aportaciones de este trabajo de investigación, son presentadas en este capítulo. Así mismo son propuestas algunas ideas para trabajos de investigación en el futuro.

Apéndices
Apéndice A.- Terminología del diagnóstico
Apéndice B.- Conceptos utilizados en el modelado de los procesos de la ingeniería en la LPS
Apéndice C.- Caso de estudio: el diagnóstico de programas educativos
Apéndice D.- Caso de estudio: el diagnóstico médico
Apéndice E.- Implementación de ProtoBOM
Apéndice F.- Implementación en QVT-Relations

PARTE II. ESTADO DEL ARTE

OLLIN TONATIUH (PIEDRA DEL SOL)
(1479)

CAPÍTULO 2

ESTADO DEL ARTE

No sabe más el que más cosas sabe, sino el que sabe las que más importan.
Bernardino Rebolledo (1597-1676). Militar, poeta y diplomático español

2.1 Introducción

En este capítulo se presentan las tendencias actuales más importantes en investigación sobre los espacios tecnológicos usados en la tesis. Enmarcados en esta presentación, se incluyen algunos trabajos relacionados con la investigación realizada para el desarrollo de BOM.

La estructura de este capítulo es la siguiente: En la sección 2 se comentan los trabajos relacionados con la aproximación presentada en esta tesis. De las secciones 3 a la 7 se presentan los diferentes espacios tecnológicos contemplados en BOM: la ingeniería de software dirigida por modelos, la arquitectura software, las líneas de producto software, el modelo PRISMA y los sistemas expertos. La sección 8 contiene las conclusiones de este capítulo.

2.2 Trabajos relacionados

Existen un gran número de trabajos relacionados con la aproximación presentada en esta tesis. Las metodologías y aplicaciones en esta temática han producido una amplia variedad de productos de investigación, ofreciendo sugerencias y soluciones en dominios específicos. En concreto, el trabajo de investigación de esta tesis tiene como puntos de partida los trabajos que se exponen a continuación:

- La definición de modelos a un alto nivel de abstracción, de acuerdo con MDA, independientes de la tecnología (PIM) cuyas instancias soportarán la información del dominio, tanto de su funcionalidad como de su variabilidad. En

esta tesis, se ha considerado esta línea con un enfoque hacia los SE basados en líneas de producto.

- Las investigaciones realizadas por [Liao, 2005]:
    - al examinar las metodologías de los SE y clasificarlos en once categorías. Dos de esas categorías corresponden a los sistemas basados en reglas y los sistemas basados en el conocimiento. Estas dos categorías han sido tomadas en cuenta en esta tesis al utilizar el conocimiento representado en forma de reglas (cláusulas de Horn) y hechos (variables observables, i.e. propiedades de las entidades a diagnosticar).
    - al expresar que las aplicaciones de los SE se desarrollan como sistemas orientados a un problema en un dominio específico. Esta tesis está enfocada al dominio del diagnóstico.
    - al señalar que el desarrollo de los SE se caracteriza por separar el conocimiento de los procesos, como unidades independientes. En el modelo arquitectónico presentado en esta tesis, los elementos arquitectónicos son definidos tomado en cuenta este concepto, específicamente cuando se considera un componente que contiene el conocimiento del dominio de aplicación y otro componente distinto que ejecuta los procesos de inferencia para llevar a cabo el diagnóstico.
- Las investigaciones realizadas por [Giarratano et al., 2004] y otros autores, consideran que las arquitecturas de los SE están basadas en componentes. En esta tesis, la arquitectura genérica de la LPS (i.e. la arquitectura genérica de los SE) está conformada por módulos que se corresponden con los componentes mencionados por Giarratano.
- La implementación de los SE que ha sido realizada en diferentes paradigmas de programación, tales como la estructurada, la lógica y la orientada a objetos. Estos paradigmas están orientados a lenguajes de cuarta generación y métodos de programación visual para dar una comunicación amigable con el usuario. En esta tesis, se usa el Lenguaje de Descripción de Arquitecturas (LDA) de PRISMA para definir los modelos arquitectónicos.
- La detección de componentes basada en la descomposición funcional del problema, es decir del sistema, compatible con la metodología Architecture Based Design Method [Bachman et. al, 2000], propuesta por The Software Engineering Institute of The Carnegie Mellon University para diseñar arquitecturas software de un dominio de aplicación. Esta metodología ha sido aplicada para la construcción del modelo arquitectónico de los SE.
- El trabajo realizado por [Garlan, 2001], como punto de referencia para establecer los elementos de un sistema software complejo, en el cual se introducen conceptos como componente, conector, sistema, puertos de entrada y de salida. Dichos conceptos han sido contemplados en el modelo propuesto.

- El concepto de contrato de [Andrade et al., 1999], ya que se han definido los conectores de los modelos arquitectónicos de la LPS como una extensión de dicho concepto; incorporando la coreografía en los conectores, especificada como el protocolo del aspecto de coordinación de dichos elementos arquitectónicos.
- La integración de las aproximaciones que combinan DSBC y DSOA, introducida por [Constantinides et al., 2000]. Esta integración es contemplada en el modelo PRISMA [Pérez, 2006]. Los aspectos y requisitos descritos en PRISMA son considerados en la arquitectura de la LPS presentada en esta tesis. Asimismo, [Giarratano et al., 2004] y otros autores en el campo de los SE, consideran que las arquitecturas de esos sistemas están basadas solamente en componentes. En esta tesis se integran las aproximaciones que combinan componentes (DSBC) con aspectos (DSOA), obteniendo las ventajas de cada una de ellas e incrementando de esta forma, la reusabilidad y el mantenimiento de los SE.
- Las líneas de producto software, que incorporan técnicas y metodologías relacionadas con esta tesis. Entre ellos se encuentran los trabajos de:
    - [Batory et al., 2006] donde se expresan las características del dominio en un Modelo de Características. En esta tesis se han representado las características del diagnóstico en un modelo de características y han sido plasmadas en un árbol de decisión y capturadas en un modelo conceptual.
    - [Trujillo, 2007] quien ha utilizado FOP como técnica para insertar características en documentos XML a través de XSLT. En esta tesis se ha utilizado esta técnica para insertar las características del dominio de aplicación por medio de plantillas XSLT en los aspectos (esqueletos) PRISMA de los componentes del sistema experto, representados como documentos XML. Además en el trabajo de investigación de esta tesis, se ha incorporado FOP pero a nivel de modelos, de forma que ha sido definido y usado como técnica el Feature-Oriented Modeling (FOM).
    - [González-Baixauli et al., 2005] aplican la propuesta de MDA y la Ingeniería de Requisitos para Líneas de Productos. En la tesis se ha utilizado técnicas de MDA para producir aplicaciones, así como contemplar los requisitos del usuario final para construir los elementos arquitectónicos de la LPS. i.e. los activos reutilizables.
    - [Clements et al., 2001] que usan la aproximación de desarrollo de LPS, estableciendo una división entre la ingeniería del dominio y la ingeniería de la aplicación, para el reuso y la automatización de los procesos software. Así mismo, el método Kobra divide la ingeniería del dominio (para desarrollar la familia de productos, incluyendo identificación de componentes comunes y variables) de la ingeniería de la aplicación (para la derivación de un producto específico dentro de la familia). Esta aproximación es utilizada en BOM para desarrollar su LPS en las

fases de ingeniería del dominio e ingeniería de la aplicación respectivamnete.

- [Ávila-García et al., 2006] que desarrollaron un editor de modelos RAS (como un plugin de Eclipse) para el empaquetamiento de los activos reutlizables y su manipulación; así como el uso del estándar de OMG, SPEM para procesos de modelado, ofreciendo facilidades de creación y manipulación de LPS. En esta tesis se han integrado estos dos estándares de la OMG para la manipulación y creación de la Baseline, incluyendo en ella el plan de producción de la LPS. Se ha utilizado el estándar SPEM para modelar todo el proceso de desarrollo de la LPS, y el estándar RAS para empaquetar todos los activos hasta conformar la Baseline.
- [Santos, 2007] que propone el desarrollo de una técnica basada en MDA para gestionar la variabilidad en las LPS, tomando en cuenta el dominio específico. En esta tesis se utiliza dicha técnica de MDA, al utilizar dos modelos conceptuales (el modelo conceptual de dominio: MCD y el modelo conceptual de la aplicación: MCDA), que se usan para generar el software necesario que captura las características del dominio y del dominio de aplicación como instancias de dichos modelos, respectivamente.
- [Bachmann et al., 2003] que proponen la separación de la descripción de la variabilidad y de la funcionalidad de los artefactos afectados. En BOM la especificación de la variabilidad y de la funcionalidad se captan en modelos separados. Además a través de instancias de los dos modelos conceptuales, MCD y MCDA, se permite captar la variabilidad en dos etapas, la primera al definir la información de las características del dominio usadas para seleccionar los *assets* adecuados de la Baseline, y la segunda al definir las características del dominio de la aplicación para configurar la aplicación final respectivamente.
- [Gomma et al., 2007] utilizan un modelo de variabilidad semi-ortogonal donde la variabilidad es representada en una vista separada, pero los puntos de variabilidad y las variantes aparecen estereotipadas en diagramas UML. En BOM la vistas de la variabilidad y de la funcionalidad son tratadas separadamente, pero a diferencia de Gomma, las variantes están incorporadas en los modelos de variabilidad específicos.
- [Bosch, 2000] que describe tres dimensiones en las que se pueden descomponer los conceptos involucrados en las LPS. De esas tres, la que se corresponde con esta tesis es su primera dimensión, que considera un sistema como un conjunto de "*assets*" formado por los sistemas construidos sobre la base de la arquitectura y los componentes de la línea de productos. Esta actividad requiere la

adaptación de la arquitectura de la línea de productos para ajustarse a la arquitectura de sistema, que puede requerir eliminar o añadir componentes o relaciones entre ellos, desarrollar extensiones a los componentes existentes, configurar los componentes y desarrollar elementos software específicos para el sistema. En BOM se parte de una arquitectura genérica que comparten las arquitecturas base, y que seran enriquecidas con las características del dominio de aplicación para obtener las arquitecturas PRISMA como producto final.

- [Clements et al. 2002b], que propusieron la distinción de tres modelos de creación de LPS: el proactivo, el extractivo y el reactivo. Esta tesis se inclinó por el modelo reactivo, en el cual no se establece desde un principio el dominio en extenso de la línea de producto, sino que se irán incluyendo nuevos productos a medida que aparezca la necesidad de producirlos. La creación de activos es por tanto más incremental y progresiva que con los otros modelos de adopción.

2.3 Ingeniería de software dirigida por modelos

La Ingeniería de Software Dirigida por Modelos (del inglés Model Driven Engineering-MDE) [Kent, 2002] constituye una aproximación para el desarrollo de sistemas software basada en la separación de la especificación de la funcionalidad del sistema de su implementación en plataformas específicas. MDE persigue elevar el nivel de abstracción dándole una mayor importancia al modelado conceptual y al papel de los modelos en el desarrollo de software.
En la literatura se pueden encontrar los términos Model-Driven Software Development y Model-Driven Engineering como sinónimos. En esta tesis se usará el término Model-Driven Engineering (MDE).

En el paradigma MDE, los modelos constituyen los elementos centrales en el desarrollo de software, al contrario que en el enfoque tradicional que está más centrado en la implementación. Los modelos se expresan usando conceptos más abstractos que los utilizados en los lenguajes de programación, y la descripción del sistema es independiente de la tecnología utilizada. Con ello los modelos son fácilmente especificados y su mantenimiento es más sencillo, sirviendo de guía para el desarrollo del sistema mediante la tecnología aplicada. Uno de los objetivos de MDE es automatizar todo este proceso de desarrollo. Los modelos permiten capturar el conocimiento y la lógica del negocio de forma independiente de la tecnología utilizada para implementar los sistemas, con el fin de proteger la inversión frente a cambios y evoluciones en la tecnología.

En MDE los modelos, como entidades de primera clase, son usados en todo el ciclo de vida de la ingeniería, el cual es visto como una cadena de transformaciones de modelos. El objetivo principal de MDE es usar modelos para incrementar el nivel de abstracción, en el cual los desarrolladores de aplicaciones crean software para simplificar y formalizar las actividades y tareas que comprende el ciclo de vida del software.

En MDE, un sistema software es creado a través de la definición de diferentes modelos en varios niveles de abstracción, donde cada modelo es una versión más refinada de los modelos definidos en el nivel de abstracción superior. Los refinamientos continúan hasta que es obtenida la implementación ejecutable del sistema.

La derivación de los modelos de un cierto nivel de abstracción de modelos de un nivel superior se realiza por medio de transformaciones. Una transformación de modelos especifica como un modelo de salida se construye basado en los elementos de uno (o varios) modelo(s) de entrada. Con los lenguajes de transformación de modelos se automatiza el proceso para derivar un modelo de otro modelo.

Por ello, en MDE podemos identificar como conceptos básicos: modelos, transformación de modelos, metamodelos y meta-meta modelos de acuerdo con Meta Object Facility-MOF [MOF, 2006] de la OMG [OMG]. Las relaciones entre modelos, metamodelos y meta-metamodelos se presentan en la **figura 1**. Una vista de un sistema puede ser representada por un modelo. Cada modelo debe conformar a su metamodelo, el cual debe ser escrito en el lenguaje de ese metamodelo. De la misma manera, el metamodelo debe conformar a su meta-metamodelo que se describe por el mismo.

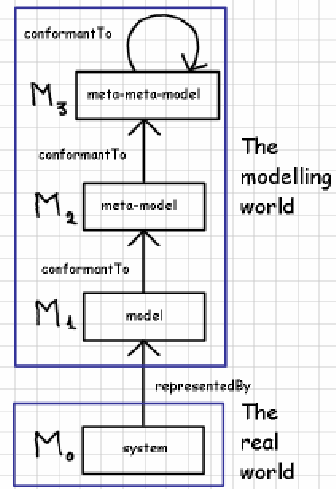


Figura 1 Nociones básicas en MDE (figura adaptada de [Bézivin, 2005])

En la actualidad existen dos iniciativas principales y diferentes de la MDE. Una de ellas es el enfoque de la OMG, a través de la estrategia Model-Driven Architecture-MDA, y la otra iniciativa es la de Microsoft, a través de Software Factories-SF y de Domain Specific Languages-DSL [Greenfield et al., 2004].

2.3.1 Arquitecturas software dirigida por modelos

MDA ha definido tres niveles conceptuales de modelado, en diferentes niveles de abstracción. Estos pueden ser modelos formales, y por lo tanto, entendibles por el ordenador. Cabe señalar que el *framework* BOM implementado en esta tesis se corresponde directamente con estas ideas.

En el primer nivel de abstracción, se modelan los requisitos del sistema mediante Modelos Independientes de Computación (Computer Independent Model-CIM) que sirven de puente entre los expertos del dominio y los desarrolladores que afrontan la realización del sistema. Los CIM no representan detalles del sistema dependientes del cálculo, del ordenador, sino del dominio de aplicación del mismo.

En el segundo nivel de abstracción, se encuentran los Modelos Independientes de Plataforma (Platform Independent Model-PIM) que sirven para la representación de la funcionalidad y la estructura del sistema, aislándose de los detalles tecnológicos de la plataforma de implementación. Con los PIM se desarrollan los sistemas con una tecnología neutral, que proporciona servicios que se implementarán posteriormente de forma diferente en cada plataforma concreta. Los modelos del nivel PIM pueden ser refinados tantas veces como se quiera hasta obtener una descripción del sistema con el nivel de claridad y abstracción deseado. Un PIM es una especificación de un sistema en términos de conceptos del dominio y con independencia de plataformas tecnológicas.

En el tercer nivel de abstracción, se encuentran los Modelos Específicos de Plataforma (Platform Specific Model-PSM), que combinan las especificaciones contenidas en un PIM, con los detalles de la plataforma elegida. Un PIM puede transformarse en un PSM, y también a partir de distintos PSM se pueden generar distintas implementaciones del mismo sistema.

De acuerdo con MDA, una transformación de modelos "es el proceso de convertir un modelo en otro modelo del mismo sistema". En general, una transformación de modelos es el proceso de generación automática, partiendo de un modelo origen, de un modelo destino, de acuerdo a una definición de transformación, la cual es expresada en un lenguaje de transformación de modelos [Kurtev, 2005].

MDA propone algunas restricciones en las transformaciones entre estos niveles de abstracción:

- definir reglas de transformación entre modelos PIM y PSM, y PIM-PIM y
- mantener una relación de trazabilidad entre los requisitos del sistema (expresados en el modelo CIM) y los artefactos representados en los modelos PIM y PSM que permiten llevar a cabo tales requisitos.

Por ello, las transformaciones contempladas en MDA pueden ser agrupadas en dos tipos [France et al., 2001]: verticales y horizontales.

1 Una transformación vertical es aquella en la que los niveles de abstracción del modelo origen y el modelo destino son diferentes. MDA contempla varias transformaciones verticales; por ejemplo:
    - cuando un PIM está suficientemente refinado se transforma en un modelo dependiente de la infraestructura final de ejecución, de modo que un PIM se transforma en uno o varios PSM,
    - cuando partiendo de una implementación en una plataforma concreta, se desea abstraerse de los detalles concretos de dicha plataforma, se está aplicando una transformación vertical para pasar del PSM al PIM.

    Ejemplos de estas transformaciones son el refinar un modelo o implementarlo en un lenguaje de programación concreto.

2 Una transformación horizontal es aquella en la que el modelo origen y el modelo destino corresponden al mismo nivel de abstracción. La principal aplicación de este tipo de transformaciones es la evolución de modelos; dicha evolución puede ser:
    - perfectiva, para mejorar un diseño, cambiar el metamodelo,...
    - correctiva, para corregir errores en el diseño, o
    - adaptativa, para introducir en un diseño nuevos requisitos o restricciones.

    Una aplicación de estas transformaciones permite obtener modelos de diferentes vistas del sistema (por ejemplo, de una vista modular a una vista componente-conector) pasando de un PIM a otro, o bien obtener también modelos específicos para distintas plataformas pasando de un PSM a otro.

La OMG ha promovido la especificación de transformaciones entre modelos, a través del estándar denominado Query View Transformations-QVT [QVT, 2005]. Cabe señalar que en la actualidad no existe una implementación de referencia que abarque la propuesta completa de QVT, aunque si han surgido propuestas que lo soportan en parte, tales como la herramienta MOMENT [Queralt et al., 2006].

Además de las transformaciones modelo a modelo, existen las transformaciones modelo a código o texto, que son principalmente utilizadas para transformar modelos diseñados en los niveles mas bajos a código de un lenguaje de programación específico, y también para generar la implementación de artefactos.

2.3.1.1 MDA y las vistas de arquitecturas software

Las arquitecturas software pueden ser representadas por un conjunto de vistas de las diferentes estructuras de un sistema software [Shaw et al., 2006], denominadas vistas de arquitecturas software o del inglés Software Architecture Views (SAV).

En la literatura existen varias aproximaciones que describen una SAV, entre ellas están: SEI [Shaw et al., 2006], IEEE [IEEE-1471], y Kruchten [Kruchten , 1995]. Aunque hay algunas diferencias en cómo se interpreta una vista, también hay coincidencias en el tipo de elementos y sus relaciones. En todas esas aproximaciones, los componentes y conectores son considerados como bloques de construcción básicos para la vista componente-conector, y el módulo para la vista modular.

Las SAVs pueden ser usadas para separar asuntos("concerns"), representadas por varios modelos. Las SAVs pueden formarse como modelos abstractos, i.e. PIMs, aunque representen sistemas específicos. Las SAVs tienen características que son aplicables a un conjunto similar de sistemas (familias de sistemas software) de modo que la misma SAV puede ser adaptada a sistemas en diferentes plataformas, i.e. a diferentes modelos específicos de plataforma, i.e. PSMs. Además las SAVs están formadas con elementos en un alto nivel de abstracción, descritos por lenguajes de descripción de arquitecturas (LDAs) o notaciones que son traducidas a nivel código. Por lo tanto, MDA puede ser usado no sólo para refinar una arquitectura software, sino también para realizar transformaciones entre vistas arquitectónicas que están en el mismo nivel de abstracción [Kleppe et al., 2003].

En BOM se han seleccionado las vistas de arquitectura software descritas en [Limón, 2006] como las aproximaciones más apropiadas con la propuesta de esta tesis, i.e. la vista modular y la vista componente-conector. Dichas vistas son tratadas en BOM como la Vista Funcional del Sistema, representando la funcionalidad de los sistemas expertos. Sin embargo en BOM el tratamiento de la variabilidad del dominio y la variabilidad de la funcionalidad de los sistemas es manejado separadamente (en modelos independientes). Por ello en BOM es también considerada la Vista de la Variabilidad de Dominio.

2.4 Líneas de productos software

Una de las aproximaciones más importantes en la ingeniería del software es la Línea de Productos Software, paradigma enriquecido en los últimos años por la Ingeniería Dirigida por Modelos [Schmidt, 2006]

Una LPS puede ser vista como un conjunto de sistemas intensivos de software compartiendo un conjunto común de características manejables, y que son desarrolladas usando un conjunto de activos básicos comunes (artefactos que pueden ser software, herramientas, documentos, modelos, etc.).

Los productos de una LPS son diferenciados por sus características, donde una característica es un incremento en la funcionalidad del producto. Es evidente que las características pueden ser usadas en distintos miembros de una LPS.

Es importante considerar técnicas de análisis de requisitos funcionales y no funcionales de la nueva línea de producto, así como técnicas de clasificación y estudio de las relaciones entre los requisitos del sistema, que a la vez sugieren una utilización de las bibliotecas de reutilización como gestores de este tipo de componentes software reutilizables o "*assets*". Existen algunas propuestas muy útiles, que pueden ser consideradas como puntos de partida y que deben complementarse con las propuestas de las nuevas técnicas. En particular, Feature-Oriented Domain Analysis-FODA [Kang et al., 1990] que sistematiza las etapas iniciales del desarrollo de las LPS. Sin embargo, se necesitan herramientas más potentes para obtener modelos de requisitos más precisos que los propuestos en estos métodos, así como cambiar su enfoque del espacio de la solución al espacio del problema que se pretende solucionar.

Diversas definiciones de las LPS han sido dadas por varios investigadores del área. En esta tesis se ha recurrido a la bibliografía especializada, donde existen diversas definiciones, pero tal vez la más conocida es la dada por [Clements et al., 2001] que define a la LPS como "un conjunto de sistemas software que comparten un conjunto común y gestionado de características, que satisface las necesidades específicas de un segmento de mercado o misión y que son desarrolladas a partir de un conjunto central de "*assets*" de una manera preestablecida" [Clements et al., 2001]. Esta definición es detallada en los siguientes cinco conceptos:

Productos: "Un conjunto de sistemas software…". Las LPS cambian el enfoque del desarrollo de software al desarrollo de las LPS. Los procesos de desarrollo no intentan construir una aplicación, sino un número de ellas. [Clements et al., 2001].

Características: "… que comparten un conjunto común y gestionado de características…". Las características son "unidades por medio de las cuales

diferentes productos pueden ser distinguidos y definidos en una LPS" [Batory et al., 2004].

Dominio: "…que satisface las necesidades específicas de un segmento de mercado o misión…". Una LPS es creada en el ámbito de un dominio. Un dominio es un área de aplicación de productos software Un dominio es definido como "un bloque de conocimiento especializado, un área de experticia o un conjunto de funcionalidades relacionadas" [Northrop, 2002].

Activos reutilizables de software: "…y que son desarrolladas a partir de un conjunto central de *assets*…". Un activo de software reutilizable es "un artefacto o recurso que es usado en la producción de una línea de productos software mas que en un producto" [Clements et al., 2001].

Plan de producción: "…de una manera preestablecida". Se establece cómo es producido cada producto y como se componen todos los activos de software reutilizables para permitir la producción del producto final. Una línea de productos está formada por un conjunto de aplicaciones muy parecidas que pertenecen a un dominio determinado, como por ejemplo, el dominio del diagnóstico. El plan de producción es "una descripción de cómo son usados los activos de software reutilizables para desarrollar un producto de una línea de productos y especificar como usar el plan de producción para construir el producto final" [Chastek et al., 2002].

Las LPS tienen su origen en la década de los ochenta, en las escuelas de negocio, con un claro objetivo económico mediante el desarrollo sinérgico de productos [Knauber et al., 2001]. La reducción de costes, la disminución del tiempo de mercado, y la mejora en la calidad, son beneficios que se derivan de una estrategia basada en las líneas de productos. Sin embargo, para cada posible beneficio derivado de la reutilización existe un coste asociado. En [Clements et al., 1998] se presentan los elementos que afectan tanto a la línea de productos como a los nuevos productos, junto a los beneficios y los costes asociados.

El objetivo principal de una LPS es reducir el tiempo, el esfuerzo, el coste y la complejidad de crear y mantener los productos de la LPS mediante:

- la detección de los aspectos comunes de la línea de productos, a través de la consolidación y reutilización de los activos de entrada a la línea de productos, y
- el manejo de los aspectos variables de los productos de la línea, a través de los puntos de variación de los activos y los modelos de decisión [Krueger, 2006].

La reutilización del software es uno de los objetivos fundamentales dentro de la ingeniería del software. Las LPS potencian la reutilización estratégica. Los "*assets*" de una LPS van más allá de sólo la reutilización de código. Cada producto de la línea

de productos toma la ventaja de las etapas de análisis, diseño, implementación, planificación y prueba, realizados en cada uno de los productos desarrollados previamente en la LPS.

De los componentes de bajo nivel inicialmente reutilizados (básicamente código fuente), se ha ido ampliando el espectro a niveles de abstracción cada vez más altos. [Krueger, 1992] clasifica en los niveles más altos a los "*assets*", como generadores de aplicaciones o arquitecturas software. [Mili et al., 1995] consideran sistemas transformacionales reutilizables y lenguajes de muy alto nivel. Una tendencia general es crear elementos reutilizables de grano más grueso, como por ejemplo bibliotecas de clases y "frameworks" [Wirfs-Brock et al., 1990].

Los *frameworks* capturan las decisiones comunes del diseño de un tipo de aplicaciones, estableciendo un modelo común a todas ellas (en BOM Arquitectura Genérica), asignando responsabilidades y estableciendo colaboraciones entre las clases que forman dicho modelo. Este modelo común contiene puntos de variabilidad, conocidos como puntos calientes [Pree, 1995], tomando en cuenta los distintos comportamientos de las aplicaciones representadas por el *framework*. Los *frameworks* constituyen un hito importante en la reutilización de software, ya que permiten la reutilización de código y también la reutilización del diseño de sistemas o subsistemas. La utilización de *frameworks*, como base de arquitecturas de referencia adaptables, junto con la utilización de técnicas de descripción (e implementación) de los puntos de variación de una familia de productos ha permitido el éxito del concepto de LPS [Bosch, 2000], [Clements et al., 2001].

Clements y Krueger [Clements et al., 2002b] propusieron la distinción de tres modelos de creación de LPS: el proactivo, el extractivo y el reactivo:

- El modelo proactivo es el más clásico; propone un gran esfuerzo e inversión iniciales en una ingeniería de dominio exhaustiva, donde se establece el dominio exacto de los productos de la línea, para luego crear los activos. Esta aproximación puede requerir tiempos y costes de adopción prohibitivos para una empresa.
- El modelo extractivo plantea la utilización, durante la ingeniería de dominio, de ingeniería inversa de los productos ya existentes en el dominio para obtener conocimiento más preciso sobre éste más rápidamente. Con el desarrollo de activos en esta aproximación se obtiene una reducción considerable de coste y tiempo.
- En el modelo reactivo, no se establece desde un principio el dominio de la línea de producto, sino que se irán incluyendo nuevos productos a medida que aparezca la necesidad de producirlos. La creación de activos es por tanto más incremental y progresiva que con los modelos de adopción anteriores.

El concepto de LPS [Clements et al, 2001] aparece formalizando técnicas de fábrica de software. Asimismo, se anota la importancia de los planes de producción, preferiblemente automáticos, al ofrecer tiempos y costes de ejecución predecibles.

Las LPS permiten la reutilización sistemática de las familias de productos, es decir productos similares diferenciados por algunas características, promoviendo de este modo la industrialización del desarrollo software.

Una familia de productos de software es un conjunto de productos de software asociados a un dominio determinado [García et al., 2002] definen una familia de productos como: "el conjunto de productos diferentes que pueden ser producidos desde un diseño común, *assets* compartidos y mediante un proceso de ingeniería de aplicaciones. La pertenencia a este conjunto depende de la abstracción que unifica los *assets* en un sistema que funciona: una arquitectura, las reglas físicas o de negocio, o la plataforma hardware" o como "el conjunto de productos que comparten una plataforma común, pero tiene características y funcionalidad requeridas por el cliente".

Algunas veces es confundido el término de líneas de producto con los términos familia de productos y dominio, o bien se realiza una correspondencia biunívoca entre línea de productos y unidad de negocio. A continuación son aclarados estos términos:

- Una LPS no necesita construirse como una familia de productos, aunque es la forma de obtener los mayores beneficios. Una familia de productos no necesita constituir una línea de productos si los productos resultantes tienen poco en común en términos de sus características.

- Una LPS no es un grupo de productos producidos por una única unidad de negocio. Puede esperarse que haya una gran correlación entre las líneas de productos y las unidades de negocio, pero conceptualmente son cosas distintas. Una unidad de negocio se forma por razones financieras o de organización, y puede ser responsable de una o más líneas de productos. Mientras que en una línea de productos se comparten y gestionan un conjunto común de características que satisfacen las necesidades específicas de un mercado seleccionado. Las líneas de productos que provienen de diferentes unidades de negocio pueden mezclarse para formar una "super línea de productos".

- Una LPS no es sinónimo de dominio. Un dominio es un cuerpo especializado de conocimiento, un área de experiencia o una colección de funcionalidad relacionada: La línea de productos hace referencia a los sistemas software que comparten las características particulares de un sector de mercado.

2.4.1 Repositorio de la línea de productos software

La LPS requiere almacenar sus activos de software en repositorios o "Baselines". La Baseline es una base de datos especializada que almacena activos de software y facilita su recuperación y mantenimiento. Su objetivo es asegurar la disponibilidad de activos para apoyar el desarrollo de productos de la LPS.

De todos los *assets* contenidos en la Baseline, la arquitectura software juega el papel más central. El diseño de una arquitectura software deberá dar cobertura a todos los productos de la LPS e incluir a todas las características que se comparten entre los productos. Los componentes que han sido identificados en la arquitectura deben reflejar la funcionalidad requerida, pero además deben soportar la variabilidad identificada en la arquitectura de la línea de productos.

2.4.2 Arquitectura de la línea de productos software

Con respecto a la arquitectura, existe un amplio rango de definiciones realizadas por varios autores, de modo que aunque no existe una sola definición que capte perfectamente el significado de arquitectura, se pueden obtener los elementos comunes de esas definiciones: estructura, composición, organización, comportamiento, decisiones, ámbito, estilos y cualidades. Muchas otras definiciones incluyen referencias explícitas a componentes (la unidad computacional principal), conectores (las interacciones entre diferentes componentes), e interfaces (propiedades de los componentes), los cuales son abstracciones básicas de los lenguajes de descripción de arquitecturas (ADLs) [Medvidovic et al., 2000]. [Eeles, 2006] agrega a ésto el propósito, las necesidades de los "*stakeholders*", el ámbito y la estructura.

El Software Engineering Institute (SEI) encontró 77 definiciones que la comunidad ha dado a la arquitectura, de las cuales las más representativas son.

- [IEEE-1471, 2000]: "La arquitectura es la parte fundamental de un sistema incorporada en sus componentes, sus relaciones con otros, y el entorno, y los principios que guían su diseño y evolución".

- [SEI, 2007]: "Una arquitectura es la descripción de las estructuras de un sistema, puede haber varias (descomposición modular, de proceso, de despliegue, de capas, etc.) La arquitectura es el primer artefacto que puede ser analizado para determinar como son logrados los atributos de calidad, y también sirve como el plan del proyecto. Una arquitectura sirve como el vehículo de comunicación, es la manifestación de las decisiones de diseño tempranas, y es una abstracción reusable que puede ser transferida a nuevos sistemas "

El contar con arquitecturas estandarizadas permite definir los contextos en los que los elementos reutilizables pueden ser desarrollados. Así pues, la ingeniería de dominio surge como una evolución de la reutilización sistemática del software basada en modelos donde las arquitecturas software juegan un papel importante.

En una LPS, la arquitectura de la LPS es la expresión de los aspectos no variantes, mientras que con una arquitectura convencional con cualquier instancia se obtiene el comportamiento de un solo sistema. En la LPS, identificar las variaciones permitidas e incorporar mecanismos para conseguirlas, es parte de la responsabilidad al construir una arquitectura. Los productos en una LPS existen simultáneamente y puede variar en términos de su comportamiento, calidad de atributos, plataforma, configuración física, middelware, ente otros.

La esencia de construir exitosamente una LPS consiste en discriminar entre lo que permanece constante y lo que varía, en todos los miembros de la familia. La arquitectura software maneja esta dualidad, dado que todas las arquitecturas son abstracciones que admiten una pluralidad de instancias, lo que permite concentrarse en diseños de una serie de implementaciones diferentes.

Por ello, la arquitectura de línea de productos (ALP) - también denominada arquitectura de dominio - es la clave para la reutilización sistemática, ya que describe la estructura de los productos del dominio, mostrando sus componentes y las relaciones entre los mismos. Definir una adecuada ALP requiere armonizar las cualidades que se persiguen para los productos de la LPS con la definición de elementos opcionales, alternativos o variables. La ALP debe ser particularizada cada vez que se desarrolla un producto de la línea.

La ALP es una arquitectura software genérica que:

- describe la estructura de toda la familia de productos y no solamente la de un producto particular, y
- captura los aspectos comunes y variables de un familia de productos de software:
    - los aspectos comunes de la arquitectura son capturados por los componentes de software que son comunes a toda la familia.
    - los aspectos variables de la arquitectura son capturados por los componentes de software que varían entre los miembros de la familia.

2.4.3 Aproximación para los procesos de la ingeniería de la línea de productos software

El proceso de desarrollo de la LPS no intenta construir una aplicación, sino una familia de ellas. Este enfoque produce un cambio de un desarrollo orientado a un único producto software a un desarrollo de varios productos que contienen unas características comunes, formando una familia de productos. Esto implica una reestructuración en el desarrollo del software, surgiendo dos procesos de ingeniería distintos: la ingeniería del dominio y la ingeniería de aplicación.

La ingeniería del dominio se centra en el desarrollo de elementos reutilizables que formarán la familia de productos, mientras que la ingeniería de aplicación se orienta hacia la construcción o desarrollo de productos individuales, pertenecientes a la familia de productos, y que satisfacen un conjunto de requisitos y restricciones expresados por un usuario específico, reutilizando, adaptando e integrando los elementos reutilizables existentes y producidos en la ingeniería de dominio [García et al., 2002].

Por ello, la división entre la ingeniería del dominio y la ingeniería de la aplicación es fundamental en la aproximación de las LPS [Clements et al., 2001]. Esta partición es la base de cualquier intento de reutilización y automatización en procesos de software [Czarnecki et al, 2000].

Frecuentemente, los productos y los activos principales son construidos en sinergia con el otro. La **figura 2** muestra esta triada de actividades, donde se observa que cada círculo representa una de las actividades esenciales. Las tres están ligadas en un movimiento conjunto, mostrando que las tres son esenciales y que están unidas, y que el desarrollo puede ocurrir en cualquier orden de forma altamente iterativa. Las flechas rotativas indican no solamente que los activos principales son usados para desarrollar productos, sino también que las revisiones de activos principales existentes o incluso nuevos activos podrían, evolucionar en el desarrollo del producto.

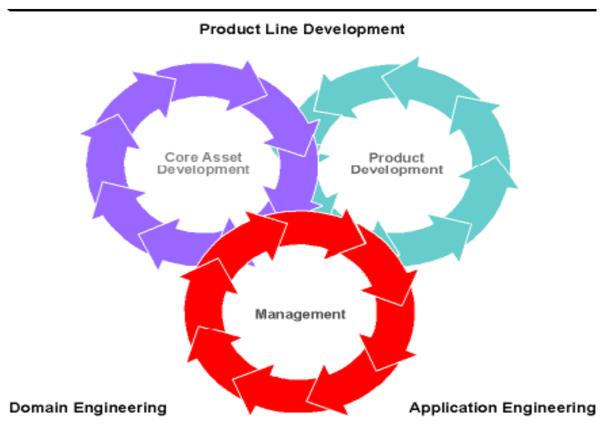


Figura 2 Las tres actividades esenciales para la LPS [Clements et al., 2001]

BOM se fundamenta en la aproximación para desarrollar LPS mencionada por [Clements et al., 2001], que es mostrada en la **figura 3**. La **tabla 1** bosqueja el proceso total:

Ingeniería del dominio	Análsis del dominio	Desarrollo de componentes básicos reutilizables (*assets*)	Planeación de la producción
Ingeniería de la aplicación	Caracterización del producto	Síntesis del producto	Construcción del producto

Tabla 1 Procesos de la ingeniería en la LPS [Clements et al., 2001]

El análisis del dominio estudia la variabilidad del dominio. Frecuentemente este estudio se realiza en términos de características del dominio y se representa usando un modelo de características.

En el desarrollo de los componentes básicos reutilizables se concibe, diseña e implementan los componentes básicos reutilizables. Esto no sólo involucra el desarrollo de la funcionalidad del dominio, sino también define cómo los componentes básicos reutilizables deben ser extendidos.

En el planeamiento de la producción se define cómo los productos individuales son creados. En general implica la capacidad de producción de la LPS.

En la caracterización del producto se eligen las características que diferencian un producto seleccionado. Este proceso se inicia con la selección de las características.

La síntesis del producto reúne los componentes básicos reutilizables para obtener la "materia prima" (un producto está compuesto de). Las técnicas de variabilidad son usadas en este proceso.

La construcción del producto procesa la "materia prima" siguiendo el plan de construcción (e.g. compilar, generar código, ejecutar, etc.), para obtener un producto final.

Cada una de las etapas del proceso de desarrollo de una LPS es realizada por un usuario que desempeña un rol particular. En la etapa de la ingeniería del dominio, el usuario cumple con el rol del ingeniero de la LPS o ingeniero del dominio. En la etapa de la ingeniería de la aplicación, el usuario cumple con el rol del ingeniero de la aplicación software o ingeniero de la aplicación. Esta situación se muestra en la **figura 3**, donde un tercer usuario (el cliente) aparece, el cual hace uso de la aplicación resultante.

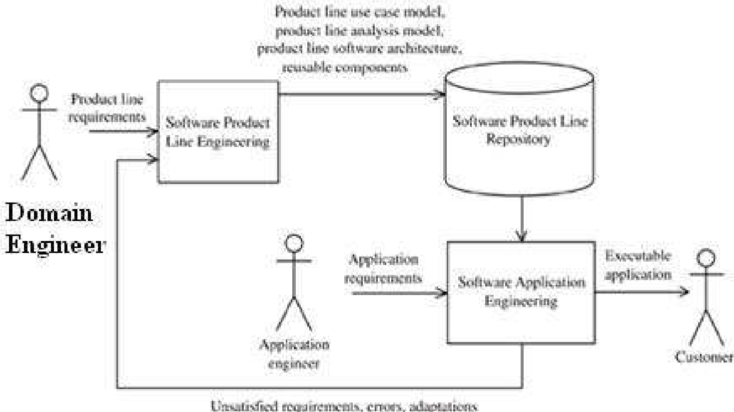


Figura 3 Desarrollo de la LPS [Clements et al., 2001].

2.4.4 Estándares de la OMG utilizados en BOM como técnicas de modelado

Las técnicas de modelado en BOM utilizan los estándares de la OMG. En particular, se usan estándares tradicionales como:

- el Lenguaje de Modelado Unificado [UML, 2005] (del inglés Unified Modeling Language-UML) que permite modelar, construir y documentar los elementos que forman un sistema software, y
- el Meta Object Facility [MOF, 2006] propuesta de la OMG para el caso específico de MDA, como un lenguaje de metamodelado,

así como nuevos estándares que en los últimos años ha propuesto la OMG **para la MDA** y que complementan los anteriores:
- el Metamodelo de Ingeniería de Procesos de Software (del inglés Software Process Engineering Metamodel –SPEM) [SPEM, 2006] que define un lenguaje para modelar procesos de desarrollo de software,
- la Especificación de Recursos Reutilizables (del inglés Reusable Asset Specification-RAS) [RAS, 2005] que define de manera estándar toda la información asociada a un activo o recurso, a través de la cual es posible manipularlo y reutilizarlo, y
- la Notación para el Modelado de Procesos de Negocio (del inglés Business Process Model Notation-BPMN) [BPMN, 2006] que define las técnicas de modelado de procesos de negocio.

2.4.4.1 Metamodelo de ingeniería de procesos de software

Los procesos de desarrollo de software se han vuelto tan complejos que es necesario el uso de lenguajes formales para modelarlos. Para ello, la OMG en el año 2001 ha creado el estándar Software Process Engineering Metamodel-SPEM, que define un lenguaje estándar de modelado que permite describir tales procesos.

Esta línea de investigación tuvo su origen en un estudio que exploraba las sinergias entre las Líneas de Productos Software, la Ingeniería Dirigida por Modelos y la Ingeniería de Procesos Software [Ávila-García et al, 2006]. SPEM es un metamodelo y un UML-Profile dedicado al modelado de procesos software. SPEM se utiliza para modelar una familia de procesos software relacionadas, usando la notación UML. El metamodelo de SPEM es muy extenso, permitiendo el modelado de muchos aspectos y problemas del proceso de desarrollo. SPEM permite modelar estos procesos de una manera estándar.

En SPEM, la idea de proceso de desarrollo de software está centrada en la colaboración entre entidades abstractas llamadas roles, que realizan operaciones llamadas actividades sobre entidades de trabajo concretas llamadas productos de trabajo. El metamodelo SPEM permite relacionar cualquier elemento (como actividades, tareas, artefactos, etc.) con elementos guía que dan soporte a su manipulación. Estas guías pueden ser desde tutoriales y ejemplos hasta plantillas y activos reutilizables.

El proceso de configuración de activos para generar una aplicación concreta puede ser modelado a través de SPEM. El ingeniero de la aplicación debe de realizar la tarea de configuración. Este será el ejecutor de este proceso de desarrollo, que previamente habrá sido diseñado por un ingeniero de dominio, encargado además de crear los activos reutilizables así como las guías de configuración adecuadas para generar cada uno de los productos de la línea y de almacenarlos en la Baseline.

En la sección 9.2 de esta tesis se utilizó el estándar SPEM, en todas las tareas llevadas a cabo para el modelado de los *assets*, incluyendo al Plan de Producción de la LPS.

2.4.4.2 Especificación de activos reutilizables

La Especificación de Recursos Reutilizables (Reusable Asset Specification-RAS) define la manera estándar de empaquetar activos reutilizables, permitiendo identificar, describir y empaquetar un activo, cumpliendo los requisitos para una reutilización efectiva.

Ávila-García [Ávila-García et al, 2006] define: Un activo es una colección cohesiva de artefactos que resuelven un problema específico o conjunto de problemas, así como metainformación para facilitar la reutilización. Un artefacto es un producto de trabajo del ciclo de desarrollo de software, como por ejemplo especificaciones de requerimientos, modelos, ficheros de código fuente, documentos XML, etc. En este contexto un artefacto es un fichero, La metainformación vendrá definida por un fichero XML que contendrá el modelo RAS del activo (como se muestra en los trabajos de [Ávila-García et al, 2006]). El modelo hace referencia a los artefactos contenidos en el activo, así como las actividades que definen procesos o guías de uso para el mismo. Estas actividades permitirán, entre otras cosas, configurar la variabilidad de los artefactos contenidos en el activo para obtener del mismo soluciones concretas.

En el capítulo 9 de esta tesis, se muestra como se utilizó la técnica del estándar RAS al empaquetar y desempaquetar los artefactos software del repositorio de assets (la Baseline).

2.4.4.3 Notación para el modelado de procesos de negocio

Como consecuencia del creciente interés en las técnicas de modelado de procesos de negocio, en junio de 2005 la Iniciativa para la Gestión de los Procesos de Negocio (BPMI) [BPMI, 2005] entró a formar parte de la OMG, publicando en

febrero de 2006 la versión 1.0 final de la Notación para el Modelado de Procesos de Negocio (BPMN).

Antes de la absorción de BPMI por la OMG, los diagramas de actividad se usaban para el modelado de procesos de negocio, pero en la actualidad, todo parece indicar que la propia OMG ha elegido a BPMN para el modelado de procesos de negocio.

Según los propios autores: "BPMN se centra en los procesos de negocio y los diagramas de actividad de UML se centran en el diseño de software y por tanto no son competidoras, sino diferentes puntos de vista sobre un sistema" [BPMN, 2006].

En lo relativo a los elementos gráficos de las dos notaciones, algunos trabajos como [White, 2004] afirman que BPMN es más rico gráficamente, con menos símbolos base y más variaciones de éstos lo que facilita la comunicación de la complejidad de los procesos de negocio entre los distintos usuarios involucrados. Según White, los diagramas de actividad son una notación más técnica mientras que BPMN está dirigido a los usuarios de negocio, lo que puede interpretarse como que la notación BPMN es más comprensible para la gran mayoría de los usuarios.

En la sección 5.3.1 de esta tesis se muestra el uso de este estándar al modelar los procesos de inferencia de los sistemas expertos.

2.4.4.4 Lenguaje de modelado unificado

El Lenguaje de Modelado Unificado (del inglés Unified Modeling Language-UML) es un lenguaje que permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos. UML está respaldado por la OMG.

Para comprender lo qué es UML, a continuación se analizan cada una de las palabras que lo componen:

- *Lenguaje*: UML cuenta con una sintaxis bien definida y una semántica intuitiva. Por lo tanto, al modelar un concepto en UML, existen reglas sobre cómo deben agruparse los elementos del lenguaje y el significado de esta agrupación.
- *Modelado*: mediante la sintaxis de UML se modelan distintos aspectos del mundo real, que permiten una mejor interpretación y entendimiento de éste; i.e. UML es visual.
- *Unificado*: UML unifica varias técnicas de modelado en una única.

Por lo anterior se puede concluir que UML es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para construir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos y funciones del sistema, y aspectos concretos

como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

Es importante resaltar que UML es un lenguaje para especificar (no para describir) métodos o procesos. Se utiliza para definir un sistema de software y para detallar los artefactos software en el sistema, en otras palabras, es el lenguaje en el que está descrito el modelo.

Al desarrollar el nuevo estándar 2.0 de UML, la OMG se planteó, entre otros, dos objetivos principales, debido a la influencia de éstos en la nueva versión:

- Hacer el lenguaje de modelado más extensible.
- Permitir la validación y ejecución de modelos.

Un modelo representa a un sistema software desde una perspectiva específica. UML 2.0 cuenta con 13 tipos de diagramas, los cuales muestran diferentes aspectos de las entidades representadas. Cada uno de estos modelos, permite fijarse en un aspecto distinto del sistema:

Los modelos de UML 2.0 utilizados en esta tesis son los siguientes:

- Diagrama de Clases.
- Diagrama de Casos de Uso.
- Diagrama de Estados.

2.4.4.5 Facilidad de Meta Objetos

Para el caso específico de MDA, la OMG propone la Facilidad de Meta Objetos (del inglés Meta Object Facility-MOF) para expresar los metamodelos, i.e. MOF es un lenguaje de metamodelado.

MOF usa las Query/Views/Transformations (QVT) [QVT, 2005] para establecer las relaciones entre los metamodelos. Específicamente, el lenguaje de las QVT-Relations es usado para describir las relaciones.

El metamodelado es un mecanismo que permite construir formalmente lenguajes de modelado, como lo es UML. La Arquitectura 4 capas de Modelado es la propuesta de OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.
Los niveles definidos en esta arquitectura se denominan comúnmente: M3, M2, M1, M0:

- El nivel M3 (el meta-metamodelo) es el nivel más abstracto, donde se encuentra MOF, que permite definir metamodelos concretos, por ejemplo el metamodelo de UML.

- El nivel M2 (el metamodelo), sus elementos son lenguajes de modelado, por ejemplo el de UML. Los conceptos a este nivel pueden ser clase, atributo, asociación, agregación.
- El nivel M1 (el modelo del sistema), sus elementos son modelos de datos, por ejemplo entidades como "persona", atributos como "nombre", y relaciones entre las entidades.
- El nivel M0 (instancias) modela al sistema real. Sus elementos son datos, por ejemplo "Isidro".

La definición de lenguajes para transformación de modelos se encuentra en la capa M3 de la arquitectura de modelado 4 capas, ya que una instancia específica de transformación se ubica en la capa M2 para poder relacionar instancias genéricas de metamodelos concretos (que se ubican en M2), entre cuyas instancias se produce la transformación. Es decir, los modelos que concretamente están involucrados en la transformación (capa M1) son parámetros para el lenguaje de transformación. El metamodelo para transformaciones y su instanciación no pueden convivir en la misma capa, ya que representan distintos niveles de abstracción.

En la capa M3 se ubica MOF, que provee un lenguaje para expresar los metamodelos. MOF es usado para especificarse a sí mismo (en el nivel M3) y para especificar los metamodelos de las vistas y sus interacciones (en nivel M2).

MOF representa un meta-metamodelo cerrado sobre el que se instancian metamodelos (instancias de MOF). En consecuencia, el metamodelo para transformaciones se ubica en la capa M2, junto con el resto de los metamodelos (por ejemplo el de UML).

En esta tesis se utilizó la versión MOF 2.0

2.4.5 La variabilidad en las Líneas de Productos Software

La variabilidad es particularmente relevante en el campo de las líneas de productos software. La variabilidad entre productos de una línea de productos puede ser expresada en términos de características [Kang et al., 1998], [Bosch, 2000a]. Una característica es una unidad lógica de comportamiento que es especificada por un conjunto de requisitos funcionales o de calidad [Bosch, 2000a].

Uno de los elementos clave para una LPS es la representación y gestión de la variabilidad. La variabilidad ha sido descrita como: "…la habilidad de un sistema software o artefacto para ser cambiado, personalizado o configurado para usarse en un contexto particular" [Van Grurp et al., 2002]

Actualmente, la variación de las características puede ser implícitamente definida en MDA a través del uso de perfiles de dominios particulares; por lo tanto, la variabilidad debe ser modelada explícitamente, con el fin de obtener productos óptimos.

En este contexto, se encuentra la iniciativa de MDA de construcción de modelos de dominio (expresados como grafos de características) y su posible transformación en modelos arquitectónicos.

La novedad que propone MDA es la posibilidad de automatizar (al menos parcialmente) la transformación que especifica cómo se convierten las instancias del modelo de características en una aplicación ejecutable. La transformación sería equivalente a la compilación de un modelo descrito por un lenguaje específico de dominio. El requisito previo antes de evaluar esta posibilidad es disponer de un metamodelo que represente uniformemente los conceptos de cada una de las técnicas utilizadas.

La ingeniería de la LPS pretende soportar un grupo de productos. Esos productos pueden atender diferentes clientes individuales o diferentes segmentos de mercado. La variabilidad es un concepto clave en cualquiera de esas aproximaciones. En lugar de entender cada uno de los sistemas individuales, la ingeniería de LPS mira a la línea de producto como un todo y la variación entre los sistemas individuales. Esta variabilidad debe ser definida, representada, explotada e implementada, a través de la ingeniería de LPS.

Cuando manejamos la variabilidad en una línea de productos, necesitamos distinguir tres tipos principales:

- *Comunes*.- una característica (funcional o no funcional) puede ser común a todos los productos en la línea de producto. Esta es implementada como parte de la plataforma.
- *Variables*.- una característica puede ser común a algunos productos, pero no a todos. Entonces debe ser explícitamente modelada como una posible variabilidad y debe ser implementada en una forma que permita tenerla en solamente los productos seleccionados.
- *Específicos del producto*.- una característica puede ser parte de solamente un producto. Tales especialidades son escasamente requeridas por el mercado, pero si por los intereses de los clientes individuales.

Mientras las características comunes y variables se manejan esencialmente en la ingeniería del dominio, las características específicas del producto son manejadas exclusivamente en la ingeniería de la aplicación.

2.4.5.1 Representación de la variabilidad

Para representar la variabilidad se han realizado y discutido diferentes aproximaciones en los últimos años. Las aproximaciones más modernas soportan la caracterización de la variabilidad por medio de características que atraviesan diferentes vistas. Así se describe si la información de la variabilidad está totalmente integrada en otros modelos o no. Mientras unos modelan la variabilidad integrada en la notación del modelo, otras aproximaciones modelan la variabilidad con un modelo de variabilidad específico y distinto al modelo del sistema principal, lo cual es mucho más fácil de aplicar en ambientes complejos y a mayor escala [Bachmann et al., 2003]. La notación que distingue entre el modelo de variabilidad y el modelo de un sistema básico es llamada modelado de variabilidad ortogonal [Pohl et al., 2005].

La variabilidad puede describirse mediante una notación gráfica que conforma un diagrama de variabilidad. La notación gráfica usada en [Pohl et al., 2005] y en [Bachmann et al., 2003] consiste en los siguientes elementos:

- *Punto de variabilidad.*-que describe donde existen diferencias en los productos finales. El descubrimiento de los puntos de variabilidad se realiza durante el proceso del diseño de la arquitectura como opciones para implementar las variaciones identificadas durante los procesos de requisitos o variaciones normales durante el diseño.
- *Variante.*- las diferentes posibilidades que existen para instanciar un punto de variación son llamadas variantes. Un punto de variabilidad está definido a través de sus variantes. La identificación de la variabilidad es una actividad continua, porque un producto puede variar de muchas maneras, al identificar las variantes en cualquier tiempo durante el proceso de desarrollo. Algunas variantes son identificadas durante la elicitación de los requisitos de la línea de productos, otras, durante el diseño de la arquitectura, y otras durante la implementación.
- *Dependencias de variabilidad.*- son usadas para cualificar las diferentes elecciones (variantes) que son posibles en un punto de variación. La notación incluye una cardinalidad que determina cuantas variantes pueden ser seleccionadas simultáneamente. Las dependencias de variabilidad pueden ser tres: alternativa <<*alternative*>>, opcional <<*optional*>> y obligada <<*mandatory*>>.
- *Dependencias de restricciones.*- que describen las dependencias entre ciertas variantes seleccionadas. Existen dos formas de restricciones:
    - Requiere <<*requires*>>.- la selección de una variante puede requerir la selección de otra variantes (quizás en un punto de variabilidad diferente)
    - Exluye <<*excludes*>>.- la selección de una variante puede prohibir la selección de otra variante (quizás en un punto de variabilidad diferente)

Un modelo único de variabilidad y funcionalidad por sí mismo representa con dificultad todo el significado de la variabilidad en la ingeniería de la LPS, ya que dicho modelo sería muy confuso, al incluir todas las restricciones y plasmar la funcionalidad de los sistemas. Esto ocasiona que no exista una forma estándar para representar la variabilidad.

Por ello, algunas de las aproximaciones propuestas agregan notaciones de variabilidad directamente a los modelos, mientras otras aproximaciones separan la especificación de la variabilidad en un modelo separado, i.e. en dos vistas independientes: la vista de la variabilidad y la vista de la funcionalidad.

En la literatura [AMPLE, 2007] se encuentran dos clasificaciones para representar la variabilidad. La primera, clasifica la especificación de la variabilidad según los diagramas que la soportan: sobre diagramas UML, sobre lenguajes ADL o usando DSLs. La segunda clasificación, distingue dos formas de plasmarla: textualmente o por un diagrama de modelos.

2.4.5.1.1 Primera clasificación para representar la variabilidad: lenguajes de modelado

En una primera clasificación, en la literatura se encuentran tres formas de representar la variabilidad:

i) **En el lenguaje de modelado unificado** (del inglés Unified Modeling Language-UML), a través de un UML-Profile se modela la variabilidad en modelos de clases o diagramas de secuencia añadiendo extensiones estereotipadas. De esta forma se expresa de manera implícita la variabilidad, en un solo modelo que mezcla la funcionalidad y la variabilidad. En este contexto se encuentran la propuesta de Clauss [Clauss, 2001a] [Clauss, 2001b] que presenta un UML Profile para modelar la variabilidad. Asimismo Gomma and Shin [Gomma et al., 2007] utilizan un modelo de variabilidad semi-ortogonal donde la variabilidad es representada en una vista separada, pero donde los puntos de variabilidad y las variantes aparecen estereotipadas en diagramas UML.

ii) **En los lenguajes de descripción de arquitecturas** (del inglés Architecture Description Language-ADL) han sido propuestas notaciones de modelado para soportar desarrollo basado en arquitecturas. Al igual que en UML, existe un modelo donde se mezcla la funcionalidad y la variabilidad. En el proyecto AMPLE D2.1 [AMPLE, 2007] se describe un ADL basado en XML altamente extensible, para modelar arquitecturas de líneas de producto: el xADL 2.0.

iii) El uso de **los lenguajes específicos de dominio** (del inglés Domain Specific Language-DSL) permite expresar la variabilidad en el lenguaje del dominio del problema. Los DSLs, en contraste con los lenguajes de propósito general ADL y UML, permiten describir sistemas software y arquitecturas usando términos cercanos al dominio de las aplicaciones que son entendidos de mejor manera por los expertos del dominio, sin que ellos requieran un amplio conocimiento de conceptos arquitectónicos. En el desarrollo de LPS, múltiples aplicaciones pueden ser generadas para el mismo dominio específico; por lo tanto, puede decirse que la LPS estimula la creación y uso de DSLs. Pero por otro lado, crear, diseñar, implementar y mantener un DSL requiere un cierto costo, ya que el tiempo usado en la creación, diseño, implementación y mantenimiento del DSL, es mayor que en un lenguaje de propósito general, dado que un nuevo DSL es creado por cada nueva LPS. Esta forma de representar la variabilidad implica un modelado explícito para definir la variabilidad, es decir, se manejan dos modelos distintos, uno para la funcionalidad y otro para la variabilidad.

De estas tres formas de representar la variabilidad, en esta tesis se ha elegido la tercera, i.e. BOM es una aproximación donde la especificación de la variabilidad y la funcionalidad se manejan en modelos separados, a través de modelos conceptuales de variabilidad ortogonal, que conforman a sus respectivos metamodelos.

2.4.5.1.2 Segunda clasificación para representar la variabilidad: lenguajes textual y gráfico

La variabilidad puede ser representada textualmente o bien por un diagrama de modelo. Cada aproximación tiene sus ventajas y desventajas. La representación textual permite muchos detalles a expensas de alguna ambigüedad debido a la interpretación del lenguaje natural. La aproximación de diagramas no siempre puede proveer las abstracciones efectivas para transmitir información más detallada. A continuación se comentan ambas aproximaciones.

i) Aproximación textual

El lenguaje natural.- La variabilidad se describe usando palabras clave y frases que representan las características del sistema con sus posibles detalles de configuración y relaciones. Pero esta representación puede resultar ambigua, ya que más de una opción puede ser seleccionada para una aplicación dada. Adicionalmente, una vez que la variabilidad adquiere mayor complejidad, este método falla rápidamente. Una manera clara de documentar la variabilidad sin tales ambigüedades, es listando explícitamente la variabilidad y describiendo la clase de variabilidad, removiendo la ambigüedad de una aproximación meramente lingüística.

El lenguaje de descripción de características.- Deursen and Klint [Deursen et al., 2002] proponen una lenguaje de descripción de características (del inglés Feature Description Language-FDL) para modelar características y restricciones. Ellos representan los puntos de variabilidad que contienen el conjunto de características variantes, y describen los tipos de restricciones utilizando palabras clave en su lenguaje de descripción de características. Pero mientras esta aproximación trabaja para pequeños problemas, puede ser difícil mantener una visión general de variabilidades y dependencias cuando el tamaño del modelo de características aumenta. Por esta razón los modelos de diagramas son generalmente preferibles para ser usados. No obstante, a veces la aproximación textual podría generarse de un diagrama del modelo de características y entonces tener una vista alternativa.

ii) Modelos gramaticales

Para obtener un mejor entendimiento de algo complejo es beneficioso mostrar el concepto utilizando diagramas. Modelar la variabilidad en forma de diagramas puede ayudar a expresar:

- características comunes y variables
- tipos de alternativas de la variabilidad: obligatoria, opcional y alternativa
- dependencias entre características
- un modelo coherente para diferentes usuarios (clientes e ingenieros de software)

Las aproximaciones que separan la variabilidad en vistas distintas están basadas en el primer metamodelo de variabilidad propuesto por [Bachmann et al., 2003] como la forma para separar la variabilidad de los artefactos afectados (e.g. componentes, clases, interfaces). Esta separación permite que la variabilidad por sí misma sea tratada como una nueva vista arquitectónica (i.e. la vista de variabilidad). Esta aproximación es genérica y puede por lo tanto ser usada para describir variabilidad en todas las clases de artefactos software. El metamodelo de Bachmann para capturar la variabilidad se presenta en la **figura 4.**

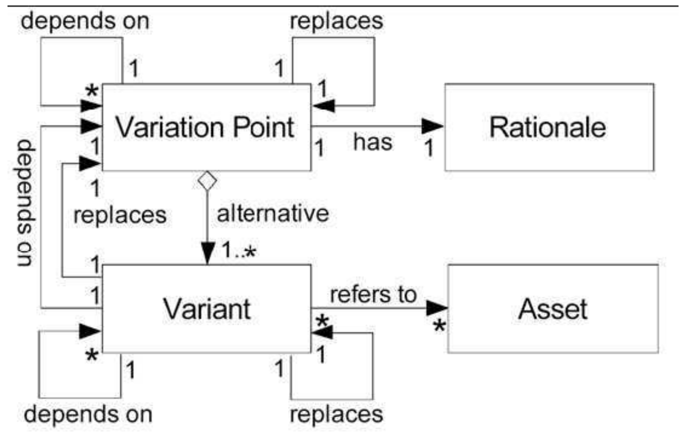


Figura 4 Metamodelo de variabilidad [Bachmann et al., 2003]

El metamodelo consiste en cuatro elementos principales: punto de variación, variante, activo y fundamento. Un punto de variación (*variation point*) describe un punto explícito donde el número de decisiones de ingeniería, i.e. variantes (*variants*), que puede tener un activo reutilizable (*asset*), i.e. un artefacto o un paquete de artefactos. El fundamento (*rationale*) explicita la intención del punto de variación, al ingeniero de software o al que configura el sistema. La relación de dependencia (*depends on*) denota que existe una dependencia entre: a) un punto de variación y otro punto de variación, b) una variante y un punto de variación, y c) entre una variante y otra variante. Similarmente la relación remplazar (*replace*) denota que: a) un punto de variación remplazará a otro punto de variación, b) una variante remplazará a un punto de variación, y c) una variante remplazará a otra variante.

En 1990, Kang et al. propusieron un modelo llamado modelo de características que permite representar todos los posibles productos que existen en una LPS en un solo modelo. Así pues, la primera notación del modelo de características fue introducida por el método FODA (Feature-Oriented Domain Analysis) [Kang et al., 1990].

FODA es una metodología de la ingeniería del dominio desarrollada por el Software Engineering Institute de la Carnegie Mellon University [Kang et al., 1990]. En FODA se enfoca al desarrollo de LPS a través de un proceso de ingeniería del dominio estructurado, basado en la notación de modelos de características. Los modelos de características modelan los conceptos comunes y variables y las propiedades de estructuras en el dominio de interés. A partir de entonces, muchas han sido las extensiones y mejoras que se han propuesto sobre dicho modelo que han intentado incrementar su capacidad expresiva, como puede observarse en los trabajos de [Griss et al., 1998].

Los modelos de características son la clave técnica de las líneas de producto para desarrollar software [Batory et al., 2006]. Son modelos formales que usan características para especificar productos.

Los modelos utilizados en el análisis de las LPS han de permitir el modelado de requisitos tanto comunes como variables en la línea de productos. Cada conjunto de características define un programa único en una LPS.

[Batory et al., 2006] definen un modelo de características de dos formas: como un conjunto de características organizadas jerárquicamente y como un lenguaje para especificar productos en una LPS. Utilizan un diagrama de características para representar gráficamente un modelo de características. Su trabajo señala que las relaciones entre un padre (o mezcla) de características y sus hijos (o subcaracterísticas) son categorizadas como:

- and (y).- todas las subcaracterísticas deben ser seleccionadas
- xor alternative (o).- solamente una subcaracterística puede ser seleccionada
- or (or)..- una o más pueden ser seleccionadas
- mandatory (obligatorio).- características que son requeridas
- optional (opcional).- características que son opcionales.

Su notación gráfica se muestra en la **figura 5**:

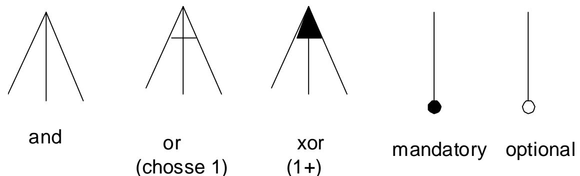


Figura 5 Notación gráfica utilizada en el modelo de características clásico

El modelado de características es una actividad principal de la ingeniería del dominio. Una característica (*feature*) es "una característica del producto que es usado para distinguir un producto de una familia de productos relacionados" [Batory et al., 2004].

El modelo de características identifica la LPS en términos de la variabilidad soportada. Entre las distintas variantes disponibles del dominio, el analista del dominio de la LPS debe seleccionar (incluir o excluir) las características de acuerdo a las alternativas que son presentadas.

2.4.5.2 Gestión de la variabilidad

La gestión de la variabilidad consiste en la habilidad para manejar eficientemente las diferencias o cambios en el desarrollo del software, considerando todas las fases de desarrollo, desde el análisis de requisitos hasta la implementación. El manejo de la variabilidad puede involucrar varios aspectos como los requisitos software, características, interfaz del usuario, o la plataforma de implementación.

El manejo de la variabilidad es de interés en cualquier aproximación de la ingeniería de LPS. Y cubre todo el ciclo de vida. La variabilidad es relevante para todos los *assets* del desarrollo de software. La variabilidad debe ser expresada en los activos (*product´s line core assets*), de tal forma que la variabilidad decore a la LPS con diferentes características.

Una técnica para el manejo de la variabilidad es la programación orientada a características (del inglés Feature Oriented Programming-FOP) que es un paradigma de las LPS donde las características son los bloques de los productos construidos, i.e. los componentes básicos reutilizables. A través de las características los diferentes productos pueden ser distinguidos y definidos en una LPS [Batory et al., 2004].

FOP es el estudio de la modularidad de características por lo que puede ser visto como una metodología modular, en donde las características son consideradas como ciudadanos de primera especie en el diseño. En FOP una característica es una parte relevante de un objeto o cosa [López-Herrejón, 2005].

FOP es una metodología de programación que sostiene que el crear varias características y conectarlas a través de procedimientos/métodos/funciones, enriqueciendo la funcionalidad principal, es la mejor manera de quitar la redundancia y mejorar la eficiencia. FOP enfoca principalmente las características de un sistema, en lugar de los objetos que lo componen (como lo sería en un lenguaje orientado a objetos).

2.4.6 Herramientas utilizadas en la línea de productos software

Las LPS requieren gestionar la variabilidad sobre todo el proceso de desarrollo de la línea de productos, involucrando los artefactos software en diferentes niveles de abstracción. Por ello, las herramientas para desarrollar los productos deben de soportar un proceso de desarrollo integral que permite crear artefactos software, asegurar propiedades, y generar pruebas, así como la calidad del producto.

En esta sección se limitó a considerar el conjunto de herramientas que han sido desarrollados expresamente para la ingeniería de la LPS;tomando en cuenta los principales intereses para el desarrollo y modelado de la LPS, que involucran artefactos software en todos los nivel de abstracción y todas las fases del ciclo de vida de la LPS.

En la **tabla 2** se presentan los criterios utilizados para evaluar las herramientas existentes y que son potencialmente relevantes para el desarrollo de la LPS, con respecto al proceso de desarrollo. Dichos criterios se basan en la funcionalidad principal de la herramienta en términos de la LPS y su gestión en el ciclo de vida del producto.

	Pure::variants	**Gears**	**Fmp2rsm**	**oAW**	**ProtoBOM**
COBERTURA DE LOS PROCESOS					
Definición de LPS	modelo de características	modelo de características	modelo de características	DSL	modelos conceptuales
Análisis/validación de LPS (espacio del dominio)	si, se comprueba compatibilidad entre el modelo de características y el modelo de la familia	si, la herramienta incluye un informe estadístico, el cual p.e. calcula el número de productos potenciales basados en el número de declaraciones y definiciones de caracterísitcas	si, se calculan un número de configuraciones representadas por un modelo de caraterísitcas, propagando elecciones de configuración, etc	no	no
Análisis/validación de productos (espacio de la aplicación)	si, se comprueba compatibilidad entre el modelo de características y el modelo de descripción de la variabilidad	si, se comprueba que la selección de características (en un modelo de características) de una instancia del producto sea la correcta	si, se verifican plantillas del modelo basaándose en características, con restricciones OCL bien formadas	si, se comprueba la validación del modelo con restricciones OCL	no
Ensamblado del producto	si	si	no	si	si
Prueba del producto	no	no	no	no	no
Ejecución del producto	no	no	no	no	si
Mantenimiento del producto	no	no	no	no	no
Soporte para dominios de aplicación específicos	no	no	no	no	si

	Pure::variants	Gears	Fmp2rsm	oAW	ProtoBOM
EXPRESIVIDAD DE EDITORES DEL MODELO DE CARACTERÍSTICAS					
Jerarquía de características	si	si	si, incluye la posibilidad de crear características clonadas	no relevante	si, incluye niveles en algunas características del dominio de aplicación
Selección de características	one-of, more-of, optional, mandatory	one-of, more-of, optional, mandatory	one-of, more-of, optional, mandatory	no relevante (no explícitas en el DSL)	and, or, optional, mandatory
Características con valores	si	si	si, incluye tipos de atributos	no relevante	si, incluye nombre, tipo y (en su caso, nivel) de atributos
Aserciones sobre valores de las características	si	si	si	no relevante	si
Representaciñon de las caracterísiticas	gráfico	textual	gráfico y textual	no relevante	gráfico y textual
Múltiples modelos de características	si	si	si, incluye restricciones entre ellos que permiten la especialización en escenas	no relevante (pero pueden usarse múltiples DSLs)	si, modelos conceptuales del dominio y del dominio de aplicación
Dependencias de los modelos de características	si	si	si, incluyen restricciones OCL	no relevante	si

	Pure::variants	**Gears**	**Fmp2rsm**	**oAW**	**ProtoBOM**
INGENIERÍA DEL PRODUCTO					
Soporte para manejar instancias del modelo de características	si	si	si	no	si
Soporte para la instanciación del producto	si	si	no, sólo modelo de instanciación	si (generación de código fuente desde el código DSL)	si (generación automática de código C# mediante un compilador de modelos)
Ambiente de ejecución	no	no	no	no	si, (generación automática del ejecutable sobre el middleware)
Editores para manejar dependencias entre modelos de características	si	si, es posible manejar las dependencias sobre los modelos de características de módulos mixin.	si	no	si
Generador de código	si	no	no	si	si
Destinos de implementación	tecnología agnóstica, y específica soportada para C/C++ y Java	tecnología agnóstica	no	tecnología agnóstica	tecnología agnóstica sobre plataformas destino PIM, y específica soportada para C#

Tabla 2 Comparación (a nivel funcional) de las herramientas para el desarrollo de LPS (Tabla adaptada del AMPLE D.3.1 project, [AMPLE,2007])

2.5 El modelo PRISMA

El modelo PRISMA (Plataforma OASIS para Modelos Arquitectónicos) [Pérez, 2006] surge para cubrir los complejos requisitos software a través de su potencia expresiva, facilidad de uso y novedad. PRISMA presenta propiedades y ventajas en la construcción de modelos arquitectónicos altamente reconfigurables y reutilizables dentro de un marco de calidad controlada.

El modelo arquitectónico PRISMA integra dos aproximaciones: el Desarrollo de Software Basado en Componentes-DSBC [Szyperski, 1998] y el Desarrollo de Software Orientado a Aspectos-DSOA [AOSD, 2001]. Esta integración se consigue definiendo los elementos arquitectónicos mediante aspectos, como son: coordinación, funcional, persistencia, distribución [Ali et al., 2005], presentación, etc. De esta forma, PRISMA, además de definir los elementos arquitectónicos básicos y especificar su sintaxis y semántica, también especifica los aspectos que cubren las propiedades necesarias de cada uno de ellos. Por ello un elemento arquitectónico de PRISMA puede ser analizado desde dos vistas diferentes: interna y externa (ver **figura 6**). La vista interna muestra al elemento arquitectónico como un prisma, de forma que cada lado del prisma corresponde a un aspecto. Mientras que la vista externa encapsula la funcionalidad del elemento arquitectónico teniendo sólo la visibilidad de los servicios que éste publica a través de sus puertos.

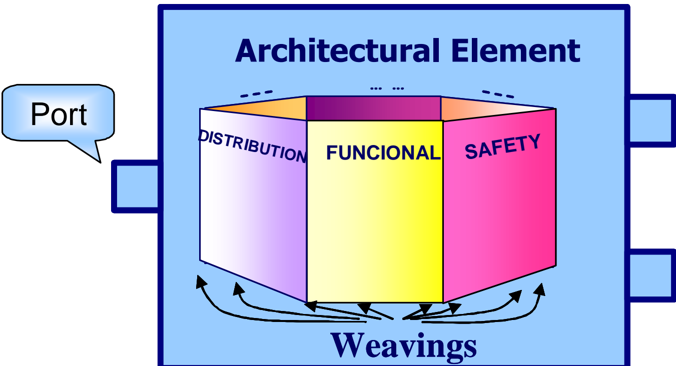


Figura 6 Vistas interna y externa de un elemento arquitectónico PRISMA [Pérez, 2006]

El modelo PRISMA consta de tres tipos de elementos arquitectónicos: componentes, conectores y sistemas. Un componente captura la funcionalidad del sistema, mientras que un conector actúa como coordinador entre otros elementos arquitectónicos. Un sistema es un elemento arquitectónico de mayor granularidad

que permite encapsular un conjunto de componentes, conectores y otros sistemas, correctamente conectados entre sí.

Un elemento arquitectónico PRISMA está formado por un conjunto de aspectos de diferente tipo, las relaciones de entretejido entre estos aspectos (*weavings*) y uno o más puertos.

Un aspecto PRISMA es un asunto de interés del sistema (*concern*) que es común y compartido por el conjunto de tipos del sistema (*crosscutting concerns*). Los tipos de aspectos (funcional, coordinación, distribución, etc.) que forman un elemento arquitectónico varían dependiendo del sistema software.

Los aspectos contienen los protocolos. El protocolo define un proceso que describe un conjunto de acciones cuya ocurrencia es posible. Está compuesto por un conjunto de procesos que establecen los servicios y transacciones posibles. También permite especificar las prioridades de ejecución de cada uno de los servicios implicados. En los conectores un protocolo modela el proceso global: coordina y sincroniza los servicios de los distintos *played_roles:* la coreografía.

Así pues, un *played_role* es una proyección del protocolo coreografía sobre el conjunto de servicios de una interfaz, que define el comportamiento parcial de un puerto con un papel determinado. Es una vista parcial del protocolo que tiene sentido por sí solo, un comportamiento específico y restrictivo que es posible asociarlo posteriormente a un puerto de un componente o conector. Un *played_role* permite a los componentes actuar de forma diferente en función de con qué otros componentes esté conectado.

Los *played_roles* y los protocolos se especifican usando pi-cálculo poliádico [Milner, 1991], el cual permite describir de forma sencilla la ejecución de procesos y servicios susceptibles de ejecutarse concurrentemente.

El *weaving* (entretejido) establece las sincronizaciones necesarias entre los aspectos que conforman un elemento arquitectónico. El *weaving* define que la ejecución de un servicio de un aspecto puede generar la invocación de un servicio de otro aspecto (*before, after,…*).

Los puertos representan los puntos de interacción entre los elementos arquitectónicos.

Existen dos tipos de relaciones para interconectar los elementos arquitectónicos: *attachments* y *bindings*. Los *attachments* establecen el canal de comunicación entre el puerto de un componente y el puerto de un conector., mientras que los *bindings* establecen la comunicación entre un puerto de un sistema y un puerto de un

elemento arquitectónico de los que encapsula. De este modo, los *bindings* permiten mantener un enlace entre los distintos niveles de granularidad de los elementos arquitectónicos que forman el modelo.

PRISMA especifica la arquitectura en dos niveles: el de definición de tipos y el de configuración. En el nivel de tipos se definen los tipos de la arquitectura: interfaces, aspectos, componentes, conectores y sistemas, los cuales son guardados en una librería para su reutilización. A nivel de configuración se especifican las topologías de las instancias de los modelos arquitectónicos de un determinado sistema software. Esta diferenciación proporciona importantes ventajas, ya que permite gestionar de forma independiente los tipos de elementos y las topologías específicas de cada sistema, obteniendo de este modo un mayor nivel de reutilización y un mejor mantenimiento de las librerías de tipos.

El metanivel del modelo PRISMA y las propiedades reflexivas de los lenguajes diseñados, dan soporte a la evolución de los elementos arquitectónicos y a la reconfiguración dinámica de la topología. Esta característica permite definir un metanivel por aspecto que reifica las propiedades que deseen ser evolucionadas basándose en la reflexión [Carsi, 1999]. Con ello, PRISMA da soporte a la evolución de sus modelos reduciendo el esfuerzo de mantenimiento de sus productos; la evolución proporciona la dinámica del tipo, i.e. la capacidad de cambio de su estructura y comportamiento basándose en la evolución del software que plantea la reflexión.

Los elementos arquitectónicos PRISMA se obtienen identificando escenas funcionales del sistema y asignando un conector a cada escena funcional, dependiendo de si la escena es simple o compleja, el elemento será un componente (componente simple) o un sistema (componente complejo), respectivamente. El concepto de escena aparece en el trabajo de [Noriega et al., 1998], y es definido con un enfoque PRISMA por [Pérez, 2003] al considerar que "Una escena se caracteriza por las tareas que se desempeñan en la actividad siguiendo un determinado protocolo, los actores que las realizan y el espacio virtual o físico donde se desarrolla"

PRISMA especifica la arquitectura a través de su propio Lenguaje de Descripción de Arquitecturas (LDA). El LDA de PRISMA está dividido en dos niveles de especificación: el nivel de definición de tipos y el nivel de configuración.

En el nivel de definición de tipos, PRISMA potencia la reutilización y combina el DSBC y el DSOA. Este nivel permite definir los tipos necesarios para especificar un sistema arquitectónico. Dichos tipos se guardan en una librería para posteriormente reutilizarlos en la definición de distintos modelos arquitectónicos.

Sus ciudadanos de primer orden son: interfaces, aspectos, componentes y conectores.

El nivel de configuración permite definir las instancias y especificar la topología del modelo arquitectónico. Para ello, en primer lugar se han de importar todos aquellos tipos (conectores, componentes y sistemas) definidos mediante el lenguaje de definición de tipos, que se necesiten para un determinado modelo arquitectónico. Después, se ha de definir el conjunto de instancias necesarias de cada uno de los tipos importados. Finalmente, se debe especificar la topología, interconectando adecuadamente las instancias del modelo.

Esta diferenciación proporciona importantes ventajas, ya que permite gestionar de forma independiente los tipos de elementos y las topologías específicas de cada sistema, obteniendo de este modo un mayor nivel de reutilización y un mejor mantenimiento de las librerías de tipos.

2.6 Los Sistemas Expertos

Los sistemas de soporte a la toma de decisiones (del inglés Decision Support Systems) son un bloque de toma de decisiones sustentado en base de datos, que puede ser usado por quienes toman las decisiones para apoyar el proceso de decidir.

De los diferentes tipos de sistemas de apoyo a las decisiones, los sistemas expertos (SE) dan mayor soporte en el proceso de toma de decisiones, permitiendo tener el conocimiento del experto capturado en una base (no estructurada) de conocimientos y utilizarlo cuando se requiera sin que esté él presente. Los sistemas expertos que están basados en reglas, contienen conocimientos predefinidos que se utilizan para tomar todas las decisiones. Para fines de esta tesis se han considerado los sistemas expertos basados en reglas.

Por ello, los SE empiezan a tener cada vez mayor auge, hasta el punto de ir suponiendo un punto de referencia importante en la toma de decisiones.

Druzdzel [Druzdzel et al., 2000] recomienda la técnica de los SE por las ventajas del acceso interactivo a grande volúmenes de datos e información para decidir entre alternativas recomendadas y justificadas para evitar los sesgos en las decisiones basadas únicamente en el juicio intuitivo humano.

Un SE es un programa software que permite simular el comportamiento de un especialista humano frente a un problema de su competencia en un determinado

campo. Estos sistemas intentan codificar los conocimientos y reglas de decisión de los especialistas, de manera que se pueden aprovechar estas pericias al tomar las decisiones. Estos sistemas están orientados esencialmente a ciertos tipos de trabajos limitados conceptualmente a una serie de acciones o decisiones.

El concepto de SE es extremadamente amplio y su definición varía dependiendo del punto de vista del autor. Diferentes definiciones de SE han sido propuestas hasta la fecha. Sin embargo un fundamento común puede ser extraído de todas ellas: un sistema experto es un sistema con pericia en la solución de problemas; esto es, un sistema que posee razonamientos, habilidades y conocimientos acerca de un dominio particular, para resolver los problemas de forma similar a la de un experto humano.

La arquitectura de estos sistemas se refiere a los módulos que forman parte del mismo, independiente del dominio. Específicamente se definen tres tipos de arquitecturas en general:

- Arquitecturas de primera generación, en donde el control y el conocimiento están centralizados.
- Arquitecturas de segunda generación, guiada principalmente por la filosofía de sistemas distribuidos. Se habla de agentes inteligentes, en donde cada uno de ellos presenta un comportamiento inteligente.
- Arquitecturas de tercera generación, en donde la idea básica es la reutilización de muchos de los componentes del sistema. Una arquitectura de este tipo, es la que se propone en esta tesis, ya que los SE pueden considerarse como una tercera generación de Sistemas de Información.

A principios de la década de los 90`s los sistemas expertos se emplean con bastante éxito en problemas de diagnóstico de fallos, interpretación de datos, predicción de comportamientos, planeación de producción, monitoreo de sistemas, el diseño, la depuración, la reparación, la instrucción y control de procesos, entre otros.

Algunos de los campos de aplicación de los sistemas expertos son: medicina, finanzas y gestión, educación, administración, militar, transportes, aeronáutica, agricultura, arqueología, derecho, geología, industria electrónica, informática y telecomunicaciones.

Las metodologías y aplicaciones desarrolladas de los sistemas expertos producen una amplia categoría de productos de investigación, ofreciendo sugerencias y soluciones a dichos sistemas en dominios específicos. Dichas metodologías y aplicaciones están diversificadas según los antecedentes de los autores, su experiencia, sus intereses de investigación, sus habilidades en la metodología utilizada y el dominio del problema. Un estudio realizado por [Liao, 2005] examina y

contempla las metodologías de los sistemas expertos, clasificándolos en 11 categorías.

Asimismo, los sistemas expertos han sido implementados en diversos paradigmas de programación, como la estructurada, la lógica y la orientada a objetos, entre otros, existiendo una tendencia en su desarrollo hacia lenguajes de cuarta generación y métodos de programación visual para dar un entorno y una estructura amigable de comunicación con el usuario [Liao, 2003].

Sin embargo, conforme a lo consultado en la literatura, la arquitectura de estos sistemas ha sido implementada con una estructura basada en componentes [Giarratano et al., 2004] sin considerar adicionalmente la orientación a aspectos, como lo contempla el modelo propuesto en este trabajo, que ha integrado estos dos enfoques.

La construcción de los sistemas expertos para resolver tareas de diagnóstico en el campo de la medicina es muy amplia, como puede observarse en [Liebewitz, 1998]. Por otra parte, el campo del diagnóstico abarca otras aplicaciones además de las médicas (si bien pueden ser estas últimas las más conocidas). En este caso se trata de fallos, averías o anomalías que se producen generalmente en un ente.

La necesidad del uso de estos sistemas se debe por un lado al aumento constante de la cantidad de conocimientos y datos a considerar en una decisión, y por otro lado a la capacidad de estos sistemas en manejar una gran cantidad de conocimientos y emular el razonamiento humano para tomar una decisión o llegar a una conclusión.

Una de las formas más utilizadas para representar el conocimiento es a través de un conjunto de reglas del tipo "si-entonces". En la parte del "si" están expresadas las premisas de alguna situación, en tanto que en la parte del "entonces" se encuentra la conclusión. La estructuración del conocimiento como sistemas de "si-entonces", permite ordenar el conocimiento como árboles virtuales, en los que la raíz se encuentra formada por las conclusiones terminales y las premisas son hojas de diferentes ramas. Los árboles de decisión, detallan una secuencia de pasos en los cuales se selecciona un camino a través de una cadena de eventos y acciones.

Una de las mejores formas de conocer el desempeño e importancia de los sistemas expertos es a través de las diferentes tareas genéricas que son capaces de ejecutar.

La amplia gama o tipología de sistemas expertos conocida hoy en día es consecuencia de la gran variedad de dominios de problemas explorados. Existen sistemas expertos para la interpretación, la predicción, el diagnóstico, el diseño, la

planeación, el monitoreo, la depuración, la reparación, la instrucción, el control y otros tipos de tareas. Una encuesta realizada por el Japan Information Processing Development Center [JIPDEC, 1989] en el Japón, sobre el uso de los sistemas expertos, demostró que las tareas que realizan éstos son las mostradas en **la figura 7**:

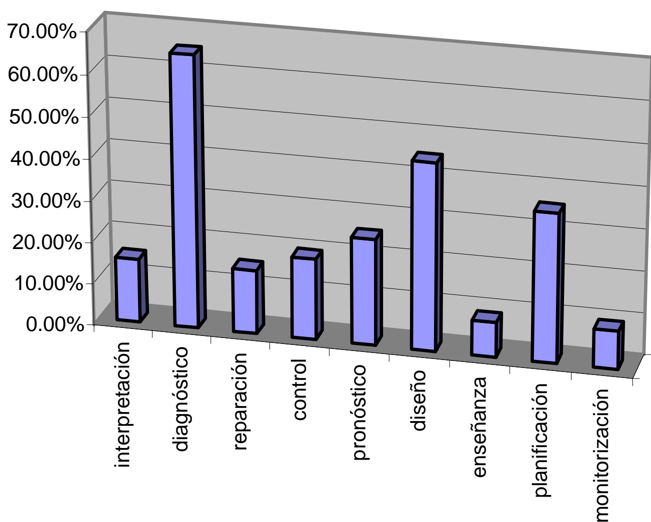


Figura 7 Tareas de los sistemas expertos [JIPDEC, 1989]

El porcentaje excede el 100% pues se admitían varias tareas para cada uno de los sistemas expertos. Es sin duda alguna, el diagnóstico la tarea más solicitada de los sistemas expertos.

Como puede observarse de la **figura 7**, la aplicación de los SE ha sido extensa en muchos ámbitos de la vida humana, sin embargo se ha mostrado un notable interés en los sistemas que realizan tareas de diagnóstico. Por ello, y con la finalidad de ilustrar en esta tesis nuestra aproximación, se han elegido a los SE que realizan tareas de tipo diagnóstico.

Un sistema experto para el diagnóstico es un sistema que infiere las razones de un buen o mal funcionamiento de un ente a partir de la interpretación de datos observados, los cuales pudieran ser ruidosos, inseguros o incompletos.

El dominio de sistemas expertos para el diagnóstico incluye sistemas para el diagnóstico médico, educativo, electrónico, mecánico y de software, entre otros. Los sistemas basados en reglas con encadenamiento hacia adelante o hacia atrás, o con los razonamientos oportunístico y diferencial, basado en casos, basado en

modelos y probabilístico, han resultado ser fuertes candidatos para problemas del tipo diagnóstico. Por otra parte, los sistemas basados en redes neuronales resultan muy útiles para problemas de diagnóstico, sobre todo cuando el diagnóstico es fuertemente dependiente del reconocimiento de patrones.

2.7 Conclusiones

Los espacios tecnológicos presentados en este capítulo ofrecen una funcionalidad conjunta y coordinada para la generación automática de aplicaciones. En particular, con la aproximación BOM se pretende mejorar el desarrollo de los SE de la siguiente manera:

- Usando las ventajas de los propios SE, al separar los procesos de inferencia de la información del conocimiento de un dominio específico, e incorporar varias estrategias de razonamiento para resolver un problema aplicando la manera más eficiente.
- Construyendo sistemas de una manera simple usando ontologías del diagnóstico y de los dominios de aplicación. De esta manera, se ofrece un acercamiento al lenguaje específico del dominio (DSL) del problema, facilitando la interacción con el usuario.
- Aplicando las técnicas de LPS al construir un diseño que compartan todos los miembros de una familia de programas. De esta manera, un diseño específico puede ser usado en diferentes productos, reduciendo los costes, los tiempos de producción, el esfuerzo y la complejidad.
- Construyendo las arquitecturas de la línea de productos en el marco de PRISMA, integrando componentes y aspectos reutilizables, facilitando su mantenimiento y complejidad.
- Aplicando técnicas de MDA para implementar los sistemas sobre diferentes plataformas, y transformarlos para obtener una aplicación ejecutable.
- Desarrollando sistemas independientes de plataforma, abordados desde la perspectiva del problema y no de la solución, lo cual provee generalidad en la aproximación desarrollada y aplicabilidad en diferentes dominios.

PARTE III. PRELIMINARES

ESTUDIO DE CAMPO

CAPÍTULO 3

LAS ARQUITECTURAS SOFTWARE DE LA LÍNEA DE PRODUCTOS DEFINIDA EN BOM

Los conceptos y principios fundamentales de la ciencia son invenciones libres del espíritu humano.
Albert Einstein (1879-1955). Científico estadounidense de origen alemán

3.1 Introdución

En este capítulo se presenta la arquitectura de la LPS considerada en la aproximación BOM.

La estructura de este capítulo es la siguiente: En la sección 2 se introduce la arquitectura de la LPS. En la sección 3 se presenta la arquitectura de los sistemas expertos. En la sección 4 se presentan la arquitectura genérica y las arquitecturas base de dichos sistemas consideradas por BOM. La sección 5 contiene las conclusiones de este capítulo.

3.2 La arquitectura de la Línea de Productos Software

Uno de los activos más importantes de una LPS es la arquitectura de dicha línea, debido a que ésta define cómo los elementos software se integran para crear un sistema y permite definir la funcionalidad básica de los productos que serán desarrollados.

En esta tesis la arquitectura de la LPS se corresponde con la arquitectura de los Sistemas Expertos (SE) o sistemas basados en el conocimiento. que realizan diagnóstico. En este contexto, BOM es un *framework* que genera SE de diagnóstico en un dominio particular.

La taxonomía de los sistemas software (SW) involucra la clasificación de dichos sistemas, como puede observarse en la **figura 8**. Dentro de esta taxonomía se encuentran los SE, i.e. los SE son una clase de sistemas software, por ello la relación entre ambos es del tipo *is_a*. Esta especialización de clases permite que la tipología de los SE pueda igualmente ser representado a través de una relación del tipo *is_a* con la clase SE (ver **figura 8**).

Esta tesis se enfocará en el dominio del diagnóstico, i.e. a los SE que realizan tareas de diagnóstico. Dicha elección se debe a que, como se mencionó en la sección 2.6, el dominio más socorrido por el mercado se lleva a cabo en esta temática, así como por mis conocimientos adquiridos previamente en el desarrollo de los SED. Sin embargo, para fortalecer dichos conocimientos, en la investigación realizada en esta tesis fue realizado un estudio de campo, y que se comentará en la sección 4.2.

Los casos de estudio realizados en el estudio de campo muestran que existen diferentes tipos de diagnóstico (diagnóstico médico, diagnóstico educativo, etc). Dentro de la taxonomía, esta tipología se sitúa como una especialización de los sistemas expertos de diagnóstico, por lo que entre ellos existe también una relación *is_a* (ver **figura 8**).

Finalmente, esta tipología es aplicada al espécimen, por ejemplo: para el diagnóstico médico podemos aplicarlo a detectar enfermedades infecciosas infantiles, enfermedades cardiovasculares, etc. Es decir, existe una instanciación, o sea el sistema que diagnostica las enfermedades infecciosas infantiles es una instancia de los SE de diagnóstico médico, por ejemplo. Por ello, y a diferencia de las anteriormente comentadas, la relación que exite entre ambos es *is_instante_of* (ver **figura 8**).

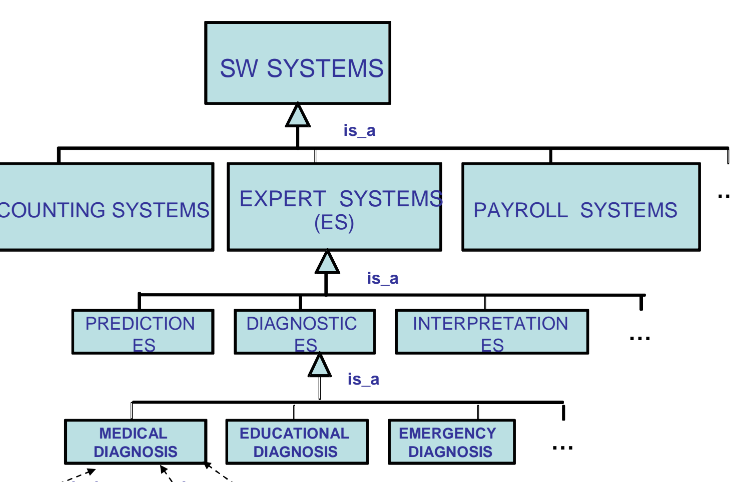


Figura 8 Taxonomía de los sistemas software

3.3 La arquitectura de los Sistemas Expertos

Como se muestra en la **figura 9**, la arquitectura típica de un SE [Turban et al., 2001] cuenta con varios módulos: la base de conocimientos, la memoria de trabajo, el mecanismo de inferencia, la interfaz de entrada/salida, el módulo de explicaciones y el módulo de aprendizaje. Sin embargo, no todos los SE son construidos con la totalidad de dichos módulos. La base de conocimientos, la memoria de trabajo, el mecanismo de inferencia y la interfaz de entrada/salida se encuentran en todo SE basado en reglas, mientras que el módulo de explicaciones pudiera no estar presente en alguno de ellos y el módulo de aprendizaje sólo está presente en unos cuantos.

La base de conocimientos contiene reglas, hechos e información acerca de un dominio especializado de conocimientos. Este conocimiento es utilizado por el mecanismo de inferencia para formular hipótesis. La cantidad y la calidad de los conocimientos contenido en la base de conocimientos determinan la bondad del sistema experto en la solución de problemas del dominio.

La memoria de trabajo es un almacén temporal de información dinámica. En ésta es almacenada, en forma de hechos, toda la información aportada por el usuario al sistema (datos iniciales y respuestas a preguntas formuladas), así como las conclusiones de todas las reglas disparadas en el transcurso del proceso de

inferencia. Cuando el proceso de solución de un problema particular ha concluido, el contenido de la memoria de trabajo es borrado o eliminado, de forma tal que esta memoria queda limpia antes de iniciar la solución de otro problema. La memoria de trabajo es tratada como un módulo independiente o bién es incorporada al módulo de la base de conocimientos.

En el mecanismo de inferencia, los procesos de inferencia de estos sistemas se llevan a cabo a través de estrategias de razonamiento como el encadenamiento de reglas hacia delante y hacia atrás o como resultado de una combinación de éstos.

La interfaz de entrada/salida permite la comunicación entre el usuario y el sistema. A través de ésta el usuario ofrece datos iniciales al sistema o responde preguntas formuladas por éste. La gran mayoría de las interfaces de comunicación conocidas establecen la comunicación usuario-sistema mediante simples menús de selección utilizando lenguajes restringidos, los cuales son aproximaciones cercanas al lenguaje cotidiano.

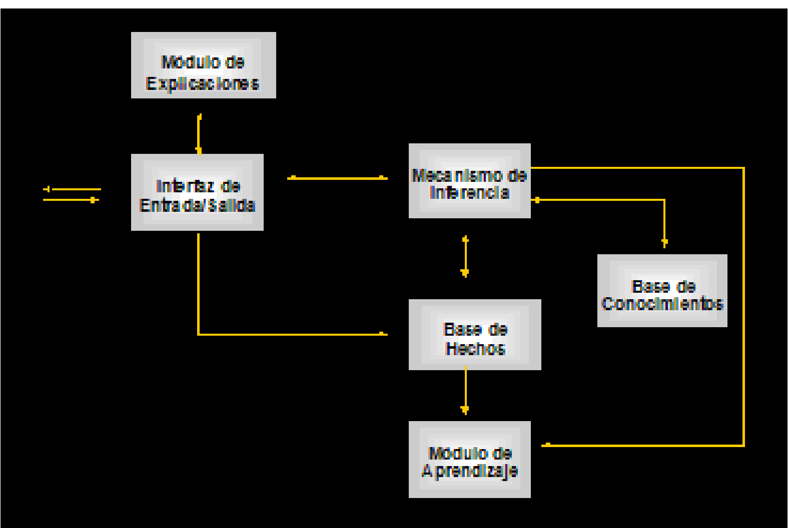


Figura 9 Arquitectura de un Sistema Experto

3.4 La arquitectura genérica y las arquitecturas base de los Sistemas Expertos de Diagnóstico en BOM

En las Líneas de Productos Software existen partes comunes y partes variables de los productos, en el caso de esta tesis, los sistemas expertos de diagnóstico (SED). La parte común está representada por la arquitectura genérica, la cual capta la

funcionalidad de dichos sistemas, i.e. la funcionalidad compartida de cada producto. La parte variable involucra las características particulares adicionales que definen los productos concretos, representados por las arquitecturas base.

Sin embargo los productos concretos pueden compartir características comunes, lo que conlleva a formar familias representadas por las arquitecturas base, como se muestra en la **figura 10**.

De esta manera, la arquitectura genérica es el representante canónico de las arquitecturas base (i.e. las familias) y una arquitectura base es el representante canónico de la familia de productos con la que se corresponde. Por lo tanto, la arquitectura genérica es el representante canónico de toda la LPS. Dicha situación es mostrada a través de la **figura 10**.

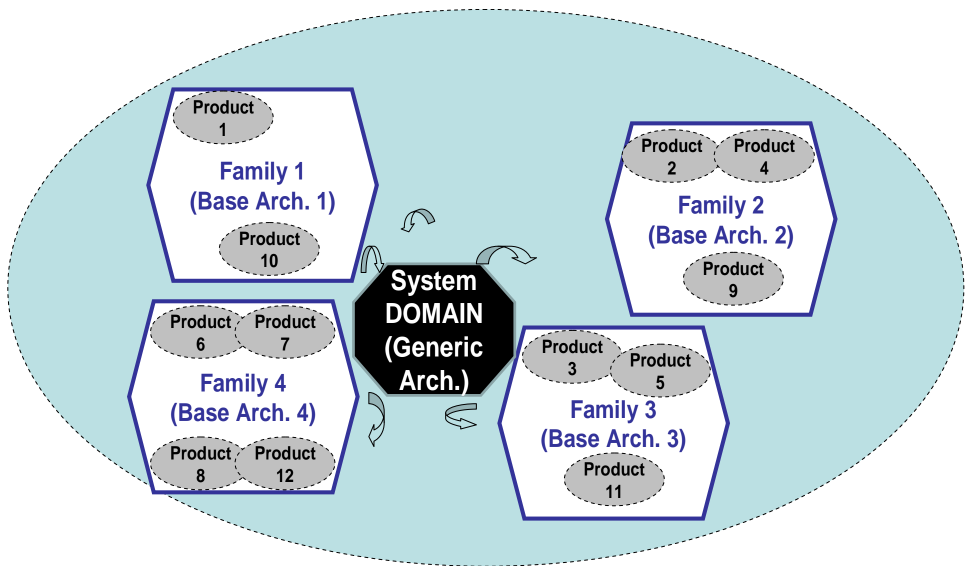


Figura 10 Metáfora visual de la relación que exite entre los representantes canónicos de la LPS

NOTA: En esta figura (y las siguientes) se muestran los términos en inglés.

Para conocer la arquitectura de los sistemas expertos de diagnóstico, se ha realizado un estudio de campo. Las fuentes en que se basa este estudio de campo contemplan desde bibliografía consultada en libros de la temática [Turban et al., 2001] y [Giarratano et. Al, 2004], estudios realizados en la metodología y tipología de los sistemas expertos [Liao, 2005], hasta consultas con profesores de asignaturas relacionadas con el conocimiento de los sistemas expertos, y mi propia experiencia en el desarrollo de los SED.

Con base en dicho estudio de campo se ha detectado que la arquitectura genérica que captura la funcionalidad de esos sistemas puede ser representada por tres módulos básicos, como se muestra en las **figuras 11 y 12.**

- el módulo *InferenceMotor* que establece el proceso de inferencia y toma decisiones,
- el módulo *KnowledgeBase* que contiene el conocimiento del dominio de aplicación. Dicho módulo integra el módulo de Base de Hechos o Memoria de Trabajo y el módulo de Base de Conocimientos
- el módulo *UserInterface* que establece la interacción hombre-máquina.

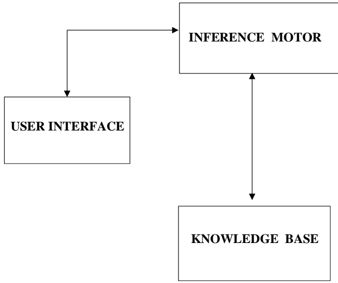


Figura 11 Módulos esenciales que integran un Sistema Experto de Diagnóstico

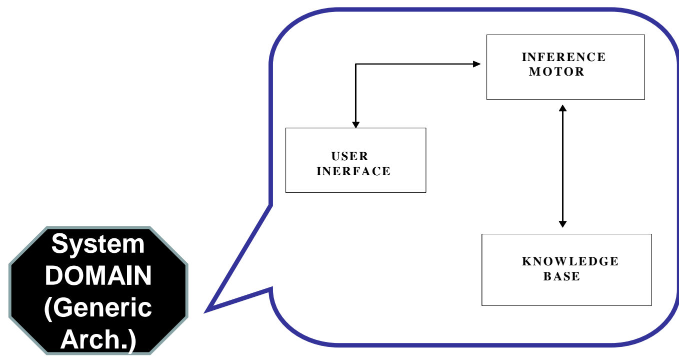


Figura 12 Arquitectura genérica en el dominio de los Sistemas Expertos de Diagnóstico

Sin embargo, la arquitectura software de la línea de productos en BOM (i.e. la arquitectura genérica de los SED) es implementada con arquitecturas tipo PRISMA, obteniendo las ventajas (reutilización, compilación, etc.) de los modelos PRISMA al incorporar las aproximaciones del desarrollo de software basado en componentes y el desarrollo de software orientado a aspectos

La elección de esta representación es debida a que la correspondencia entre ambos modelos tiene un pequeño *GAP* semántico. Como lo muestra (con flechas) la **figura 13**, en esta relación se corresponden uno a uno los módulos y los componentes *InferenceMotor*, *KnowledgeBase* y *UserInterface*. Pero para ser consistente con el metamodelo de PRISMA, es necesario incorporar un nuevo elemento arquitectónico: el conector, el cual establece la comunicación entre los componentes. Los *attachments* son las conexiones (canales de comunicación) entre componentes y conectores.

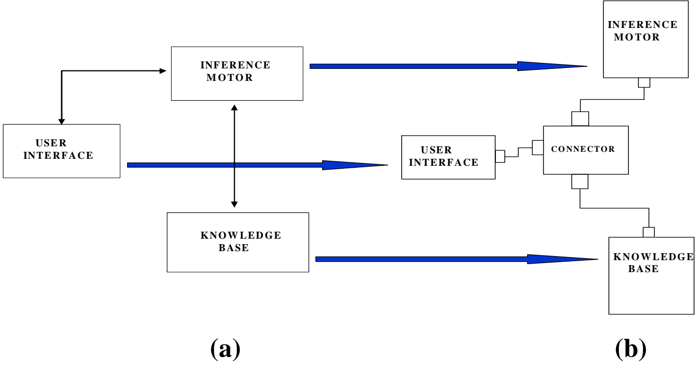
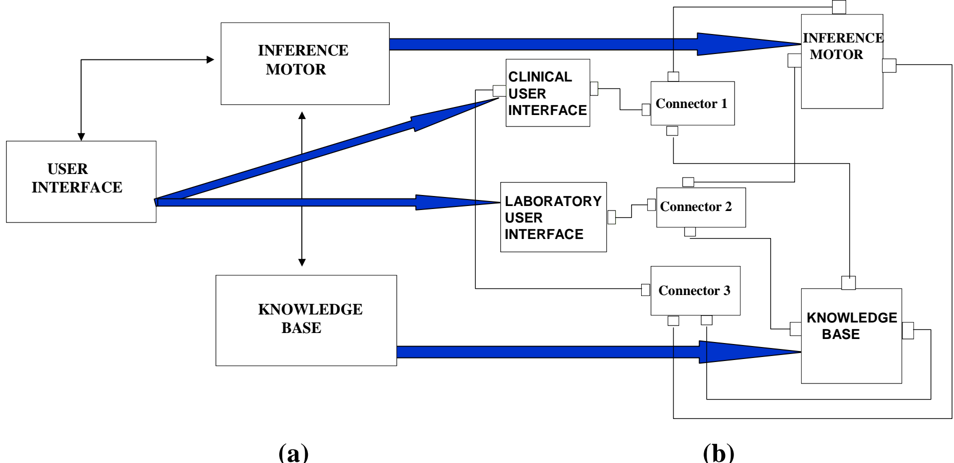


Figura 13 Correspondencia entre (a) la arquitectura genérica de un SED y (b) una arquitectura base

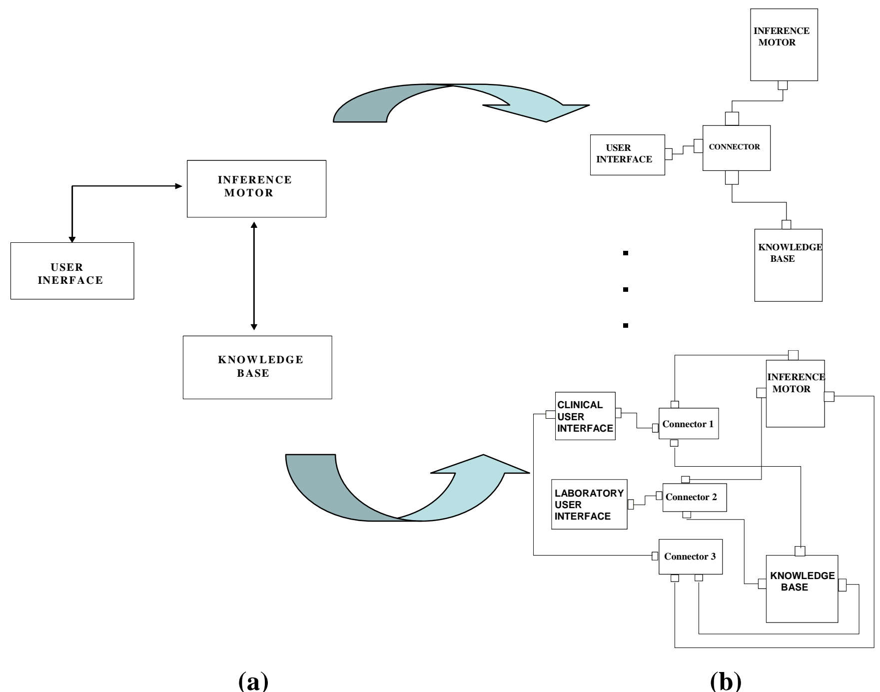
De esta manera, partiendo de la arquitectura genérica de la línea de productos se obtiene una arquitectura base. Pero en el proceso de desarrollo de una aplicación concreta (miembro de la línea de productos), ésta se deriva a partir de una arquitectura ligada al dominio: la arquitectura genérica, pero existen características particulares adicionales cuya presencia/ausencia define el producto concreto. Esta situación puede requerir eliminar o añadir componentes o relaciones entre ellos, desarrollar extensiones a los componentes existentes, configurar los componentes y desarrollar elementos software específicos para las arquitecturas base. Esto implica la creación de una arquitectura base específica, cada vez que se desarrolla un producto de la línea. En este contexto, en la **figura 13** se presenta la correspondencia entre la arquitectura genérica y una arquitectura base distinta a la de la **figura 14** asociada a un caso de estudio distinto.



(a) **(b)**

Figura 14 Correspondencia entre (a) la arquitectura genérica de un SED y (b) una arquitectura base

La **figura 15** muestra una metáfora visual de dos arquitecturas base distintas derivadas de la misma arquitectura genérica.



(a) **(b)**

Figura 15 Metáfora visual de (a) una única arquitectura genérica, y (b) dos arquitecturas base

La técnica de modelado para construir sistemas con arquitecturas base parte de la especificación de los requisitos funcionales que el producto final ha de satisfacer, modelados mediante un diagrama de casos de uso de UML. En particular, las variantes relevantes relacionadas con la variabilidad en la estructura de un sistema son: el número de casos de uso, el número de actores y el número de casos de uso a los que puede acceder un actor. Este punto será visto con detalle en el Capítulo 5 de esta tesis.

De lo anteriormente dicho, se puede deducir que la taxonomía de los SE desde el punto de vista de las LPS, se refleja en las relaciones *is-a* e *is_instance_of*. La variabilidad ligada a la semántica de la relación *is_a* se plasma en la transformación de la arquitectura genérica a las arquitecturas base; y la la semántica de la variabilidad ligada a la relación *is_instance_of* se plasma en la transformación de las arquitecturas base a las arquitecturas de los productos finales, como se muestra en la **figura 16**.

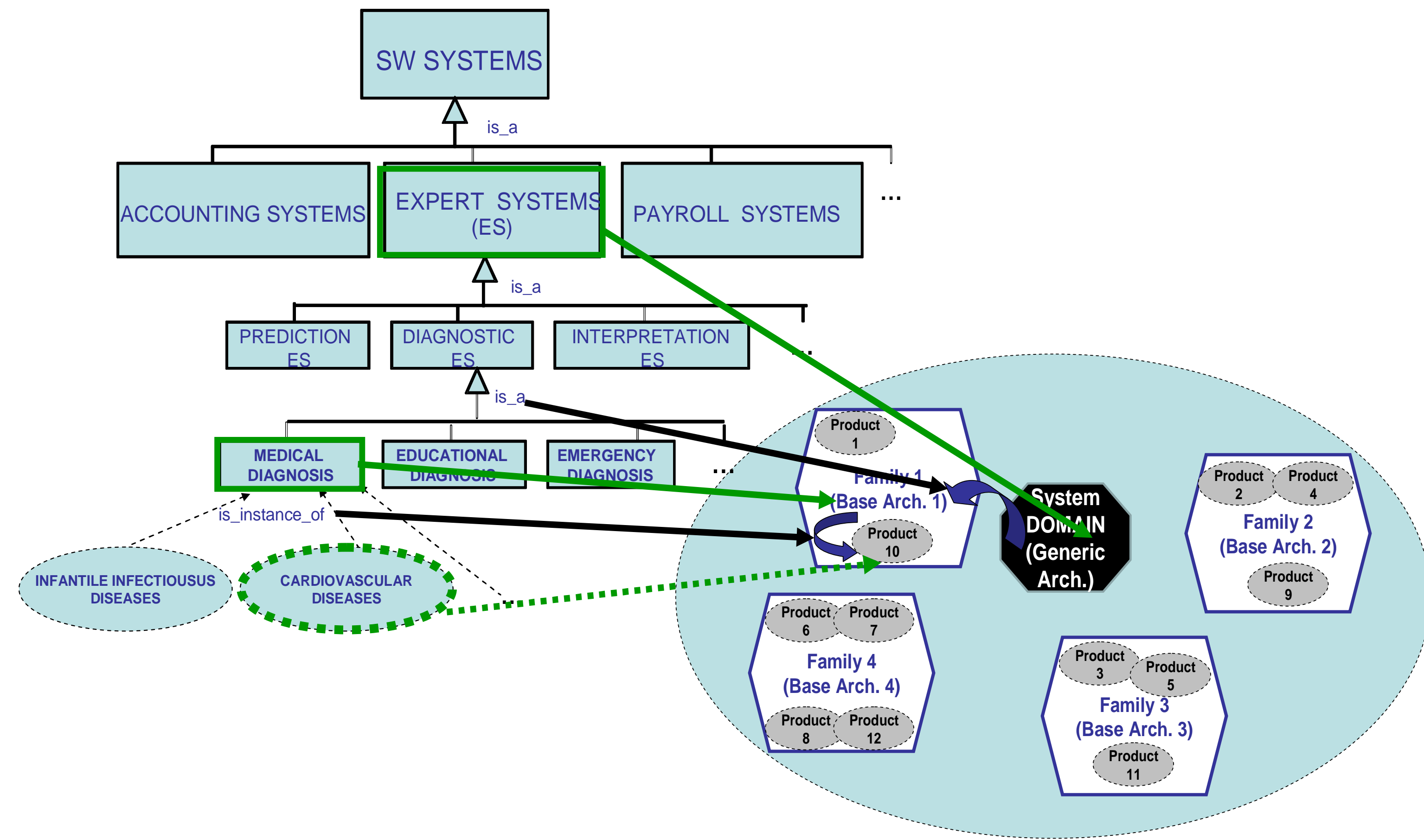


Figura 16 Mapping entre la taxonomía de los SED y las arquitecturas de la LPS

3.5 Conclusiones

Como se observa en este capítulo, la arquitectura base de los SED no es única, sino que existe variabilidad en dichas arquitecturas y por consiguiente en el funcionamiento de estos sistemas.

Asimismo se observa que la relación *is_a* (i.e. de herencia) soportada por la taxonomía de dichos sistemas, se corresponde con la funcionalidad de la arquitectura genérica de los SED, la cual es heredada por todos los miembros de la familia de productos software (i.e. la arquitectura base). Y la variabilidad ligada a la semántica de la relación *is_instance_of* se corresponde con la arquitectura del producto final específica del dominio de aplicación.

Para conocer la variabilidad, se ha realizado un estudio de campo sobre el comportamiento y la estructura de los sistemas expertos basados en reglas en un dominio específico. Para ello, se ha elegido el dominio del diagnóstico, realizándose también un estudio de campo en este dominio.

En el capítulo 4 se presenta el análisis del diagnóstico para mostrar la variabilidad del dominio. El capitulo 5 presenta el análisis de la arquitectura y comportamiento de los SED con el fin de mostrar la funcionalidad del dominio.

CAPÍTULO 4

EL DIAGNÓSTICO: VARIABILIDAD DEL DOMINIO

El experimentador que no sabe lo que está buscando no comprenderá lo que encuentra.
Claude Bernard (1813-1878) Fisiólogo francés

4.1 Introducción

En este capítulo se muestra el estudio de campo realizado en un dominio particular: los sistemas expertos de diagnóstico.

Se presentan varios casos de estudio que sirven de base para analizar las características del dominio del diagnóstico y de los dominios de aplicación.

Para los fines de esta tesis se han considerado los sistemas expertos basados en reglas que realizan tareas en un dominio específico. Y para conocer el diagnóstico, se ha realizado un estudio de campo a través de varios casos de estudio.

La estructura de este capítulo es la siguiente: En la sección 2 se presentan los casos de estudio que comprenden el estudio de campo, y se comentan los resultados del análisis realizado. En la sección 3 se presentan las características asociadas al diagnóstico. En la sección 4 se realizan las conclusiones de este capítulo.

4.2 Estudio de campo

Para conocer el dominio del diagnóstico, se ha realizado un estudio de campo a través de cinco casos de estudio (presentados en la siguiente sección de este capítulo):

- el diagnóstico médico para detectar la enfermedad que padece un paciente,
- el diagnóstico de víctimas en desastres para clasificar accidentados en desastres,

- el diagnóstico televisivo que indica si un video es considerado como adecuado o no para ser transmitido al aire,
- el diagnóstico de programas educativos para calificar la etapa de desarrollo de un programa educativo,
- el diagnóstico de candidato a beca para decidir si una beca es otorgada o no.

Con el fin de conocer los dominios de cada uno de estos casos de estudio, se llevaron a cabo reuniones con los expertos de dichos dominios de aplicación. De esta manera, yo misma con el rol de ingeniero del conocimiento, obtuve las variables observables, las reglas que relacionan dichas variables y los resultados del diagnóstico (objetivos) que se pueden obtener al final del proceso del diagnóstico.

Las fuentes en las que se basa este estudio de campo fueron las siguientes:

- Para el diagnóstico médico se obtuvo la información de un pediatra[1], particularizando en el caso de las enfermedades infecciosas infantiles.
- En el caso del diagnóstico de víctimas en desastres, la clasificación realizada a las víctimas en un desastre, fue dada por un paramédico y voluntario [2]de la Cruz Roja Mexicana.
- Para el diagnóstico televisivo, proporcionó la información un empleado[3] de la Red EDUSAT adscrita a la Dirección General de Televisión Educativa de la Secretaría de Educación Pública de México.
- El diagnóstico de programas educativos está enfocado a la evaluación de programas de posgrado que realiza el Consejo Nacional de Ciencia y Tecnología de México, a través de pares académicos, quienes realizaron una publicación de los criterios de evaluación y los resultados obtenidos [CONACYT, 1997]. Adicionalmente, se obtuvo la información del caso de los posgrados en Física en México, con el coordinador[4] del comité de física de la Dirección de Fortalecimiento del Posgrado Nacional de dicho organismo gubernamental, así como de mi propia experiencia al participar en los comités de evaluación de los programas de posgrado en física de México.
- Finalmente, en el diagnóstico de candidatos a beca, la información fue obtenida del actual responsable[5] del área de becas del Consejo Nacional de Ciencia y Tecnología de México., así como de mi propia experiencia cuando trabajé como responsable del área de becas tecnológicas del mencionado Consejo.

Al respecto es importante comentar que se consideró suficiente la información adquirida con estos cinco casos de estudio, para poder llevar a cabo un estudio de campo satisfactorio que conlleva identificar las características de la variabilidad

---

[1] Jessica de la Rosa
[2] Enrique Bastida Cabello
[3] Anónimo
[4] Fernando del Río Haza
5 Luis Gil Cisneros

asociada con el diagnóstico y que muestran la viabilidad de su aplicación en BOM. Además, pueden ser añadidos más casos de estudio ya que BOM es una aproximación reactiva, en la cual no se establece desde un principio el dominio de la línea de producto, sino que se irán incluyendo nuevos productos a medida que aparezca la necesidad de producirlos.

Del análisis realizado a través del estudio de campo se puede concluir lo siguiente:

El diagnóstico consiste en interpretar el estado de una entidad, o en su caso, identificar el problema o disfunción de una entidad, a través de sus propiedades (variables observables). Un proceso de diagnóstico es el conjunto de tareas encaminadas a la identificación de una anomalía o propiedad a partir de variables observables y razonamientos.

Las propiedades de una entidad pueden tener *n* niveles de abstracción. Dichas propiedades se relacionan interniveles a través de reglas deductivas. El objetivo del proceso del diagnóstico es obtener un resultado del diagnóstico o validación de una hipótesis.

Durante el proceso del diagnóstico, las propiedades relevantes con las que se analiza una entidad pueden ser siempre las mismas o bien pueden ser diferentes. Esto implica dos tipos de vistas sobre las entidades: la vista constante donde dichas propiedades son siempre las mismas en el proceso del diagnóstico, y la vista variable donde las propiedades difieren durante dicho proceso.

Dicha vista de las entidades está implícita en el número de hipótesis que se obtienen en el transcurso de la realización del diagnóstico. Si se tiene la misma vista de la entidad, i.e. las propiedades durante el proceso del diagnóstico no cambian, se llegará a una única hipótesis o resultado del diagnóstico; y por el contrario si la vista cambia, i.e. las propiedades durante el proceso del diagnóstico son diferentes, se puede llegar a más de una hipótesis, lo que implica que dichas hipótesis deberán ser validadas hasta obtener la hipótesis correcta que corresponde al resultado del diagnóstico.

Este comportamiento es representado a través de las estrategias de razonamiento que se aplican durante el proceso del diagnóstico. De esta forma, si existe una sola hipótesis será aplicado el razonamiento deductivo, ya que este tipo de estrategia deduce la hipótesis a partir de las propiedades de la entidad. Sin embargo, si estamos ante la presencia de varias hipótesis, es necesario realizar un razonamiento diferencial para que pueda elegirse la hipótesis correcta de entre todas las posibles, combinando con ello estrategias deductivas e inductivas.

En el proceso de desarrollo de una aplicación concreta (miembro de la línea de producto), ésta debe derivarse a partir de la arquitectura genérica ligada al dominio. En este proceso se deben seleccionar aquellas variantes que resultan apropiadas para los requisitos funcionales y no funcionales expresados por los usuarios del sistema. Las variantes seleccionadas deben ser tenidas en cuenta en función de los requisitos particulares de la aplicación.

Por otro lado, considerando el proceso de diagnóstico desde la perspectiva de un sistema software, se induce que además de estas características comentadas, se incluyan algunos de los requisitos de los usuarios finales del sistema en la variabilidad. Estas características son plasmadas en un diagrama de casos de uso, con lo cual incorporamos a la variabilidad del dominio, el número de casos de uso, el número de actores y el número de casos de uso a los que accede cada actor. La razón de ello es que se ha detectado una forma diferente de realizar el proceso del diagnóstico en cada caso de uso, y además que dicho proceso puede ser dividido en subprocesos que pueden ser invocados por diferentes actores (en caso de que existan).

Estos sistemas han sido desarrollados históricamente considerando la construcción de todos sus componentes en forma "*ad-hoc*" al caso de estudio (o aplicación del dominio), o bien, considerando un motor de inferencia que puede ser utilizado con diversas bases de conocimientos. Esto implica que existe variabilidad en la base de conocimientos, ya que las propiedades de las entidades a diagnosticar difieren de un caso al otro.

Sin embargo para llegar a una conclusión diagnóstica, el experto del dominio trata de aplicar todas aquellas estrategias de razonamiento que le permitan obtener un diagnóstico de la forma más eficiente. En otras palabras, el tipo de razonamiento que resulta más adecuado para solucionar una parte del proceso del diagnóstico, no tiene porque ser el más adecuado para resolver otra parte del mismo. Además, no todas las aplicaciones del dominio obtienen el diagnóstico aplicando la misma técnica de razonamiento. Por ello las estrategias de razonamiento del mecanismo de inferencia difieren de un caso de estudio a otro, lo que implica que existe variabilidad en el motor de inferencia.

La interfaz del usuario también involucra variabilidad, ya que un operario del sistema puede desempeñar más de un rol, i.e. un tipo de usuario puede solicitarle al sistema varias funcionalidades (p.e. varios casos de uso en un diagrama UML de casos de uso).

4.2.1 Casos de estudio

4.2.1.1 El diagnóstico médico

El tipo de diagnóstico es el *diagnóstico médico*, en el cual la entidad a diagnosticar es el paciente y el resultado del diagnóstico la enfermedad que padece el paciente. Las propiedades de las entidades cambian, son diferentes en cada hipótesis que puede resultar del proceso del diagnóstico.

En este tipo de diagnóstico se pueden considerar los siguientes casos de uso:

- caso de uso 1: realizar diagnóstico clínico
- caso de uso 2: realizar diagnóstico de laboratorio
- caso de uso 3: obtener resultados del diagnóstico (diagnóstico integral).

Los casos de uso son utilizados por dos usuarios finales distintos: el médico y el encargado del laboratorio. Los casos de uso 1 y 3 son realizados por el médico. El caso de uso 2 es realizado por el encargado del laboratorio.

Las propiedades son los signos y síntomas del paciente; los cuales están clasificados en dos niveles de abstracción: los de grano grueso y los de grano fino.
Las propiedades del nivel 0 son los signos y síntomas de grano grueso (p.e. tos y fiebre). Las propiedades del nivel 1 son los signos y síntomas de grano fino (p.e. tos seca y fiebre continua).

Las hipótesis del nivel 1 son los síndromes (p.e. ira), que son inferidos a través de las reglas del nivel 1. Las hipótesis del nivel 2 son las enfermedades (.p.e neumonía), i.e. el resultado del diagnóstico inferido a través de las reglas del último nivel.

El tipo de razonamiento aplicado es el diferencial. Al inicio se realiza un razonamiento deductivo el cual infiere varias hipótesis (i.e. posibles enfermedades), por lo que se invoca al razonamiento inductivo para poder diferenciar entre esas hipótesis, la hipótesis validada que es el resultado del diagnóstico (i.e. la enfermedad).

El escenario del proceso del diagnóstico médico es el siguiente: Con valores de los signos y síntomas de grano grueso, es inferido el síndrome. Esta parte del proceso se realiza con el razonamiento deductivo. Dicho síndrome da lugar (por deducción) a dos o más posibles enfermedades (posibles hipótesis). Estas hipótesis deben ser validadas, por lo que con los datos de los signos y síntomas de grano fino se infiere la enfermedad (hipótesis validada) que padece el paciente. Este última parte del proceso se realiza con el razonamiento inductivo.

En el grafo que a continuación se presenta, por simplicidad, sólo se han incluido cinco hipótesis (enfermedades) que pueden derivarse del proceso del diagnóstico médico.

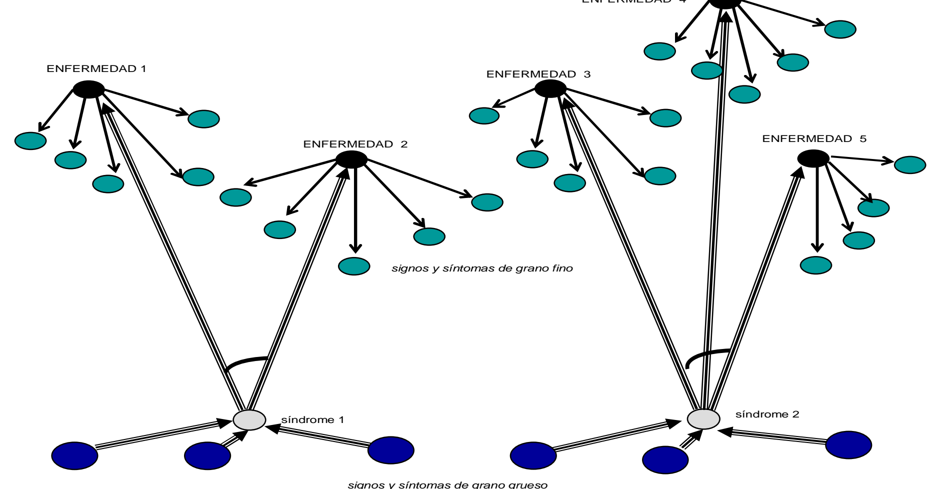


Figura 17 Grafo del diagnóstico médico

La notación utilizada tanto en este grafo como en los grafos del resto de los casos de estudio, son explicados en el apéndice A de esta tesis.

4.2.1.2 El diagnóstico de emergencias

El tipo de diagnóstico es el *diagnóstico de emergencias*, en el cual la entidad a diagnosticar es la víctima del desastre, y el resultado del diagnóstico es la indicación del color de la etiqueta que se le coloca a la víctima, la cual identifica el tipo de atención que deberá darse a la misma en dicha emergencia. Las propiedades de las entidades son diferentes en cada hipótesis que puede resultar del proceso del diagnóstico.

En este caso de estudio se cuenta solamente un caso de uso (indicar etiqueta) y un usuario final que desempeña un rol.

Las propiedades en este caso de estudio son los signos vitales que manifiesta la víctima.
Las propiedades del nivel 0 son la indicación de si una víctima es o no es ambulatoria. Las propiedades del nivel 1 son el resto de los signos vitales (p.e. perfusión).

Las hipótesis representan el estado de intervención de emergencia en la víctima (p.e inmediato), i.e. el resultado del diagnóstico inferido a través de las reglas del último nivel.

El tipo de razonamiento aplicado es el diferencial. Al inicio se realiza un razonamiento deductivo el cual infiere una hipótesis que será el resultado del diagnóstico, o bien pueden inferirse varias hipótesis, por lo que se invoca al razonamiento inductivo para poder diferenciar entre esas hipótesis, aquella hipótesis validada que es el resultado del diagnóstico.

El escenario para el proceso del diagnóstico de desastres es el siguiente: Con valores del signo vital de si es o no es ambulatoria la víctima, es inferida la hipótesis final (menor importancia) con lo cual termina el proceso del diagnóstico, o bien las posibles hipótesis (atención inmediata, no salvable, retardado). Esta parte del proceso se realiza con el razonamiento deductivo. En el caso de que se hayan generado varias hipótesis, éstas deben ser validadas por lo que con los datos del resto de los signos vitales de la víctima, se infiere la etiqueta que se colocará a la víctima del desastre (hipótesis validada). Este última parte del proceso se realiza con el razonamiento inductivo.

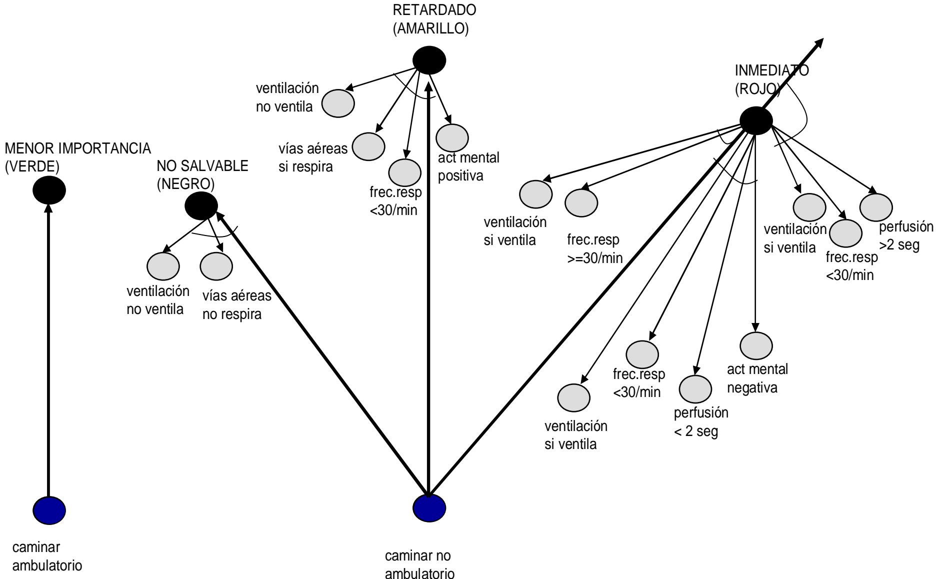


Figura 18 Grafo del diagnóstico en desastres

4.2.1.3 El diagnóstico educativo (caso: diagnóstico de candidatos a becas)

El tipo de diagnóstico es el *diagnóstico educativo*, en el cual la entidad a diagnosticar es un candidato a obtener una beca, y el resultado del diagnóstico es la decisión de si se le debe otorgar la beca a dicho candidato. Las propiedades con las que son consideradas las entidades son las mismas, por lo que se genera una sola hipótesis que resulta del proceso del diagnóstico.

Este caso de estudio cuenta con un caso de uso (realizar decisión de otorgamiento) y un usuario final que desempeña un solo rol.

Las propiedades en este caso de estudio son los requisitos que debe cumplir el candidato para ser beneficiado con una beca, y corresponden al nivel 0 (p.e. promedio de calificaciones).

La hipótesis es la indicación de si debe o no debe otorgarse la beca al candidato, i.e. el resultado del diagnóstico inferido a través de las reglas del nivel 1 (único nivel de reglas en este caso de estudio).

El tipo de razonamiento aplicado es el deductivo.

El escenario para el proceso del diagnóstico de becas es el siguiente: Con valores sobre los requisitos que debe cumplir un candidato a beca, es inferida la hipótesis, deduciéndose la decisión de otorgamiento de beca.

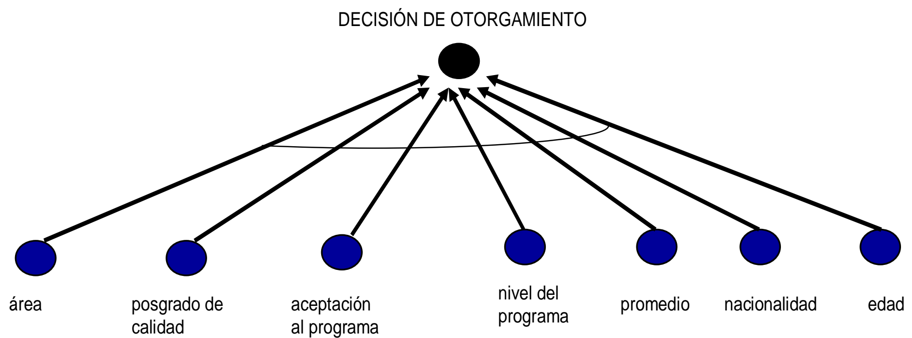


Figura 19 Grafo del diagnóstico de candidatos a beca

4.2.1.4 El diagnóstico de programas educativos (caso: diagnóstico de programas educativos)

El tipo de diagnóstico es el *diagnóstico educativo*, en el cual la entidad a diagnosticar es un programa educativo de postgrado, y el resultado del diagnóstico es el nivel de desarrollo que tiene el programa educativo. Las propiedades con las que son analizadas las entidades no cambian, por lo que se genera una sola hipótesis que resulta del proceso del diagnóstico.

En este caso, se ha considerado un caso de uso (obtener nivel de desarrollo) y un usuario final que desempeña un rol.

Las propiedades en este caso de estudio son los rubros y subrubros evaluados de un programa educativo. Las propiedades del nivel 0 son los subrubros (p.e. control de calidad de alumnos). Las propiedades del nivel 1 son los rubros cuyo valor es inferido al aplicarse las reglas del nivel 1 (p.e. plan de estudios).

La hipótesis es la calificación que se le otorga al plan de estudio según su estado de desarrollo, i.e. el resultado del diagnóstico inferido a través de las reglas del nivel 2.

El tipo de razonamiento aplicado es el deductivo.

El escenario para el proceso del diagnóstico educativo es el siguiente: Con valores de los subrubros involucrados en la evaluación del programa educativo, es inferido por deducción el valor de los rubros y posteriormente es inferida la hipótesis, deduciéndose la etapa de desarrollo del programa educativo.

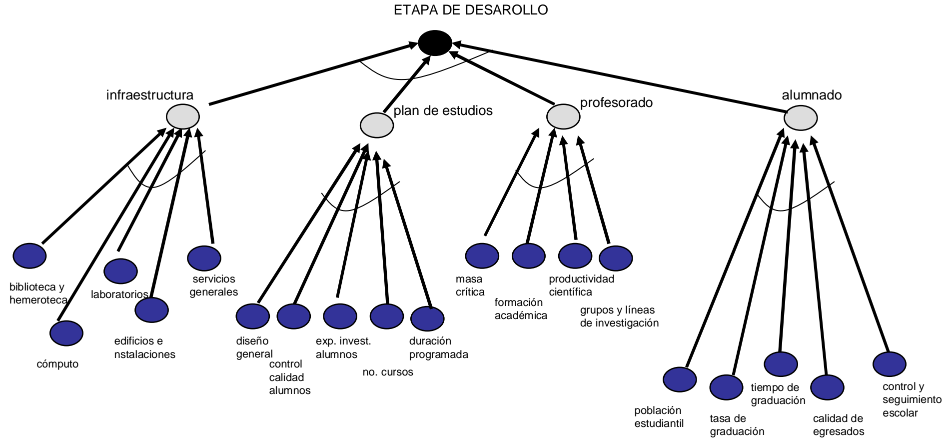


Figura 20 Grafo del diagnóstico de programas educativos

4.2.1.5 El diagnóstico televisivo

El tipo de diagnóstico es el *diagnóstico televisivo*, en el cual la entidad a diagnosticar es un video, y el resultado del diagnóstico es la calidad del video con el fin de tomar la decisión de si debe o no transmitirse al aire. Las propiedades de las entidades son las mismas, por lo que se genera una sola hipótesis que resulta del proceso del diagnóstico.

En este caso de estudio, se ha contemplado un caso de uso (obtener decisión de transmisión) y un usuario final que desempeña un rol.

Las propiedades en este caso de estudio son las características del video. Las propiedades del nivel 0 son los subrubros a evaluar del video (p.e. imágenes). Las propiedades del nivel 1 son los rubros que caracterizan un video, cuyo valor es inferido aplicando las reglas del nivel 1 (p.e. producción).

La hipótesis es el estado en el que se encuentra el video, i.e. el resultado del diagnóstico inferido a través de las reglas del nivel 2.

El tipo de razonamiento aplicado es el deductivo.

El escenario para el proceso del diagnóstico televisivo es el siguiente: Con los valores de las subrubros del video es inferido, por deducción, el valor de los rubros que caracterizan al video y finalmente es inferida la hipótesis, deduciendo el estado del video.

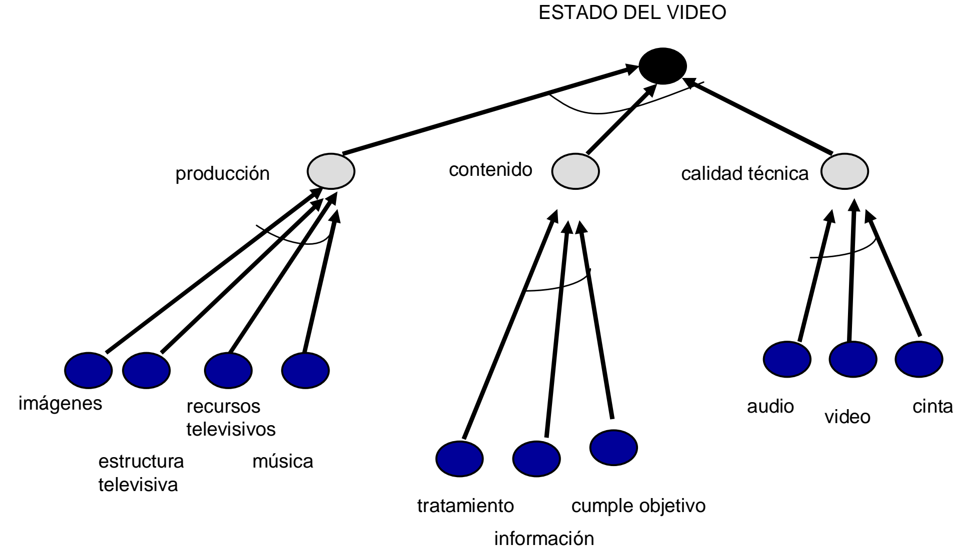


Figura 21 Grafo del diagnóstico televisivo

Finalmente, para una mayor comprensión, los casos de estudio del diagnóstico médico y el diagnóstico de programas educativos, son presentados con más detalle en los apéndices C y D de esta tesis, respectivamente.

La **tabla 3** muestra un resumen de las diferentes características de cada uno de los casos de estudio realizados:

	TYPE OF DIAGNOSIS entity/diagnosis	View	Hypotheses	Property levels	Reasoning	Use Cases	Actors	Use cases per actor
1	MEDICAL DIAGNOSIS patient/disease	Change Properties	5	2	Differential	3	2	1 1-2
2	DISASTER DIAGNOSIS victim/label	Change Properties	4	1	Differential	1	1	1
3	SCHORALSHIP DIAG. candidate/ give scholarship	Same Properties	1	1	Deductive	1	1	1
4	EDUCATIONAL DIAG. educative program/ developmental stage	Same Properties	1	2	Deductive	1	1	1
5	TV DIAGNOSIS video/transmit video	Same Properties	1	2	Deductive	1	1	1

Tabla 3 Características de los casos de estudio

4.3 Análisis de las características asociadas al diagnóstico

Basados en el estudio de campo descrito en la sección anterior, se puede concluir que existe variabilidad tanto en el proceso del diagnóstico como en los requisitos del usuario. Sin embargo al considerar los campos específicos del dominio de aplicación, surge otra variabilidad.

Dado que la variabilidad puede ser descrita en términos de sus características, en las siguientes secciones de este capítulo, se presenta el análisis de las características involucradas en el proceso del diagnóstico, los requisitos del usuario y el dominio de aplicación.

4.3.1 Análisis de las características involucradas en el proceso del diagnóstico

Se han observado cuatro características (fuentes o puntos de variabilidad) que están presentes en el proceso del diagnóstico, y que a continuación se enuncian:

- *vista de las entidades*.- una entidad puede ser considerada siempre con las mismas propiedades (misma vista), o bien una entidad puede tener diferentes propiedades (diferentes vistas) durante el proceso del diagnóstico,
- *nivel de las propiedades*.- las propiedades de las entidades pueden tener uno o varios niveles de abstracción. Las reglas que relacionan las propiedades de la entidad interniveles tienen *n-1* niveles, donde *n* es el nivel de las propiedades de la entidad,
- *número de hipótesis*.- es el objetivo o resultado del proceso del diagnóstico. Este proceso permite tener una o varias hipótesis candidatas, las cuales deberán ser validadas con el fin de seleccionar la hipótesis correcta.
- *tipo de razonamientos*.- los razonamientos o estrategias de razonamiento son la forma en que el mecanismo de inferencia realiza el diagnóstico, utilizando la información del dominio. Los razonamientos (en estos casos de estudio) pueden ser el deductivo, el inductivo y el diferencial (deductivo-inductivo).

A continuación se muestra un ejemplo de las características de esta variabilidad correspondientes al caso de estudio del diagnóstico médico:

```
Vista de la entidad: misma
Niveles de propiedades: 2
Número de hipótesis: 5 (i.e. >1)
Tipo de razonamiento: diferencial
```

4.3.2 Análisis de las características involucradas en los requisitos del usuario

Se han observado tres características (fuentes o puntos de variabilidad) relacionadas con los requisitos del usuario, y que a continuación se enuncian:

- *número de casos de uso*.- indica la división del sistema basada en la funcionalidad; i.e. las distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno (usuarios finales o actores),
- *número de actores*.- permite representar al número de usuarios finales del sistema,
- *número de casos de uso por actor*.- un actor puede acceder a uno o varios casos de uso.

A continuación se muestra un ejemplo de las características de esta variabilidad correspondientes al caso de estudio del diagnóstico médico:

Número de casos de uso: 3

Número de actores: 2

Número de casos de uso por actor: 1, 1-2

4.3.3 Análisis de las características del dominio

Este estudio de campo ha permitido considerar las características relacionadas con el dominio como las características que involucran tanto las del proceso del diagnóstico, como las de los requisitos del usuario. De esta manera, las características del dominio son representadas a través de un modelo de características que identifica la LPS en términos de la variabilidad.

Los casos de estudio realizados en esta tesis, se presentan en la **tabla 4** a través de un resumen de dichas características, así como sus grafos que muestran: el nivel de propiedades, el número de hipótesis, el tipo de razonamiento, y la vista de las propiedades de la entidad.

TYPE OF DIAGNOSIS ENTITY/RESULT OF DIAGNOSIS	FEATURES (1 st. VARIABILITY)	GRAPHS
Medical Diagnosis: *patient/disease*	change properties differential reasoning 5 hypotheses 2 property levels 3 use cases 2 users: (user A acess to 2 use cases, and user B acess to 1 use case)	
Disaster Diagnosis: *victim/label*	change properties differential reasoning 4 hypotheses 2 property levels 1 use case 1 user: (he access to a unique use case)	
Scholarship Diagnosis: *candidate/ give scholarship*	same properties deductive reasoning 1 hypothesis 1 property level 1 use case 1 user: (he access to a unique use case)	
Educational Diagnosis: *educational program/ developmental stage*	same properties deductive reasoning 1 hypothesis 2 property levels 1 use case 1 user: (he access to a unique use case)	
TV Diagnosis: *video/transmit video*	same properties deductive reasoning 1 hypothesis 2 property levels 1 use case 1 user: (he access to a unique use cases	

Tabla 4 Resumen de la variabilidad del dominio en los casos de estudio

La **figura 22** representa el modelo de características de la LPS usando una notación similar a [Czarnecki et al., 2005] y a [Batory et al., 2006], a excepción de intercambiar los conectivos "and" y "or" por así convenir a nuestros intereses (notación aplicada a los grafos de las reglas o relaciones entre las entidades de las propiedades). El modelo organiza las características en una composición jerárquica donde la opcionalidad y la obligatoriedad están presentes. Además de las relaciones jerárquicas, los modelos de características también permiten restricciones cruzadas en el árbol. Un ejemplo de una restricción cruzada en el árbol en nuestra línea de producto es que la entidad que tiene una vista en donde cambian sus propiedades implica obtener varias hipótesis y realizar un razonamiento diferencial.

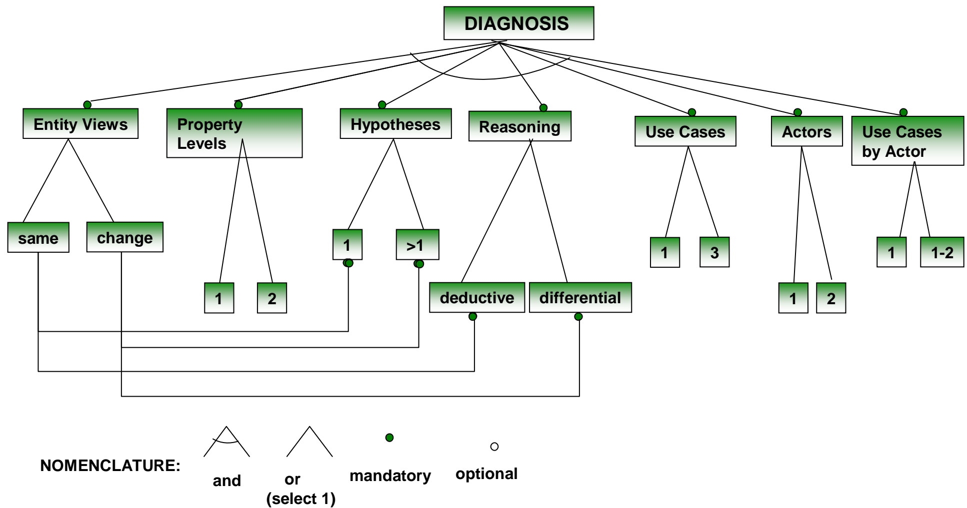


Figura 22 Modelo de características del dominio del diagnóstico

Podemos construir la especificación del modelo de características del dominio del diagnóstico, según la siguiente sintaxis:

```
Diagnosis:  entity_views   property_levels   hypotheses   reasoning
use_cases  actors  use_cases_by_actor;
    Entity_views: same   change
    Property_levels: 1   2
    Hypotheses:  1   >1
    Reazoning:  deductive   differential
    Use_cases:  1  3
    Actors:  1  2
    Use_cases_by_actor:  1  1-2

// cross Restrictions of the tree
Same   implies  1 hypothesis    deductive;
Change implies >1 hyphoteses    differential;
```

Las opciones de variabilidad (características de la LPS) observadas en el modelo de características del diagnóstico, son plasmadas en un árbol de decisión. (Ver **figura 23**) a razón de una por nodo.

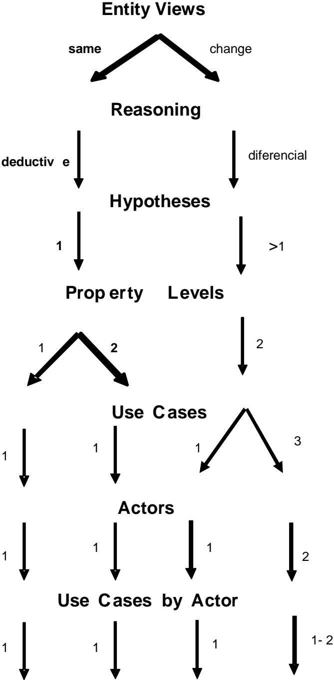


Figura 23 Árbol de decisión de las características de la LPS

Así mismo la instancia concreta de estas características (las variantes) es introducida de acuerdo con su modelo conceptual. La **figura 24** muestra el modelo conceptual del dominio del diagnóstico en el cual se especifica la ontología del diagnóstico, involucrando características y particularidades del diagnóstico en el que se desenvolverá el sistema que se desea construir. Los conceptos mostrados en el modelo conceptual de esta figura, a través de un modelo de clases UML, son los necesarios para el modelado de un SE desde una aproximación orientada al diagnóstico. Dichos conceptos se corresponden con la perspectiva CIM del diagnóstico y describen elementos propios del diagnóstico.

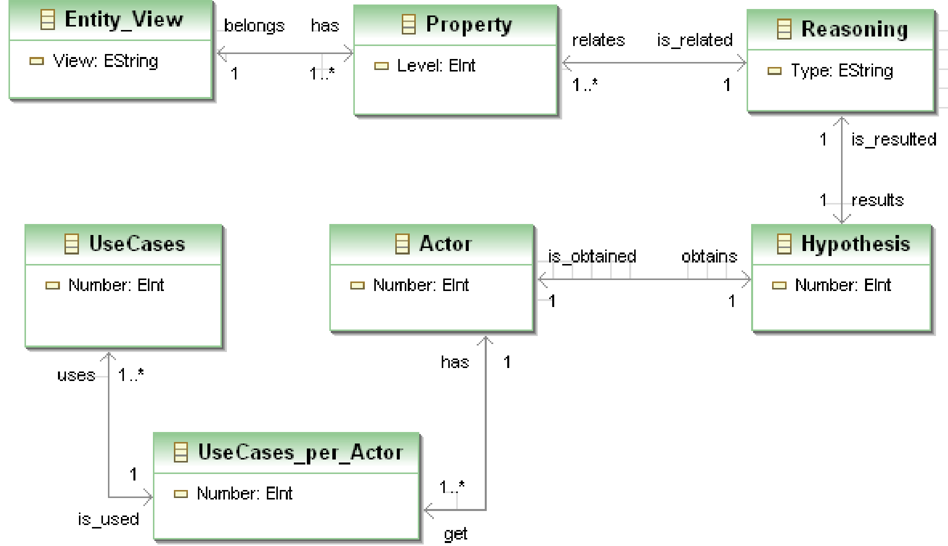


Figura 24 Modelo Conceptual del dominio del diagnóstico

Cada una de las características asociadas al dominio del diagnóstico involucradas en el modelo de características, el árbol de decisión y el modelo conceptual del dominio, son comentadas en el apéndice A de esta tesis.

4.3.4 Análisis de las características del dominio de aplicación

Algunas de las características seleccionadas del modelo de características deben de ser explicitadas de acuerdo al caso de estudio en el que se realiza la aplicación, conformando la ontología del dominio específico de aplicación. Dichas características son enunciadas a continuación:

- *nombre, tipo y nivel de las propiedades.-* por nivel de abstracción, las propiedades de la entidad a diagnosticar adquieren un nombre y un tipo,
- *reglas por niveles .-* las reglas que relacionan las propiedades de la entidad adquieren el nombre y tipo de las propiedades que están presentes en los antecedentes y el consecuente de cada una de las reglas,
- *nombre, tipo y nivel de las hipótesis.-* se adquiere el nombre y el tipo de las hipótesis a validar, por niveles, que produce el proceso del diagnóstico.

A continuación se muestra un ejemplo de las propiedades, reglas e hipótesis del caso de estudio del diagnóstico médico:

Propiedades del nivel 0: tos, fiebre

Propiedades del nivel 1: tos seca, fiebre continua

Hipótesis de nivel 1: ira, parotiditis

Hipótesis de nivel 2: neumonía, bronquitis, paperas

Regla de nivel 1:

```
IF (fiebre=true and tos=true and dificultad_respiratoria=tue)
THEN síndrome=parotiditis
```

Regla de nivel 2:

```
IF (fiebre_continua=true and tos_seca=true and
parótidas_anormales= true  and       )
THEN  enfermedad=paperas
```

La **figura 25** muestra (la vista de la variabilidad del dominio de aplicación) en el modelo conceptual del dominio de aplicación.

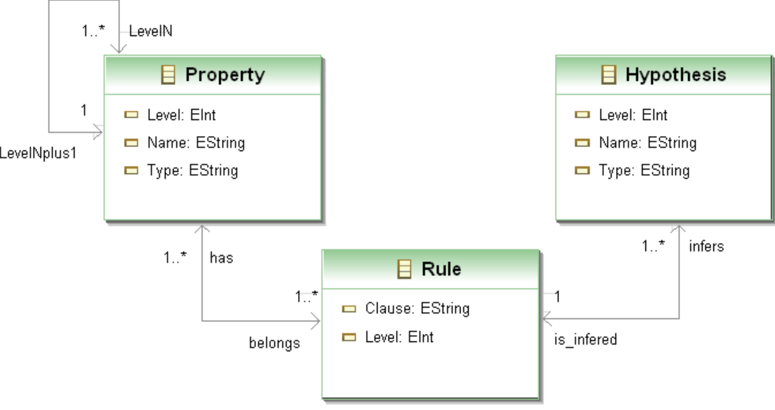


Figura 25 Modelo conceptual del dominio de aplicación

4.4 Conclusiones

El estudio de la variabilidad en el diagnóstico realizado en este capítulo mostró que las características observadas en el proceso del diagnóstico y los requisitos del usuario pueden ser consideradas como puntos o fuentes de una primera variabilidad, y que adicionalmente deben de ser consideradas las características del campo de aplicación, a través de una segunda variabilidad.

Estas consideraciones sirven como introducción para analizar la variabilidad de los SED en el siguiente capítulo.

CAPÍTULO 5

LOS SISTEMAS EXPERTOS DE DIAGNÓSTICO: FUNCIONALIDAD DEL DOMINIO

Si hay un secreto del buen éxito reside en la capacidad para apreciar el punto de vista del prójimo y ver las cosas desde ese punto de vista así como del propio.
Henry Ford (1863-1947). Industrial estadounidense

5.1 Introducción

En este capítulo se presenta el estudio de campo realizado en los sistemas expertos basados en reglas que realizan tareas de diagnóstico.

La estructura de este capítulo presenta en la sección 2 la variabilidad en la estructura de los SED, apoyándose en una técnica de modelado. En la sección 3 se presenta la variabilidad en el comportamiento de los SED. En la sección 4 se realizan las conclusiones de este capítulo.

5.2 Variabilidad en la estructura de los sistemas expertos de diagnóstico

Para mostrar que los elementos arquitectónicos de un SE varían en su estructura, se han modelado los requisitos funcionales de esta clase de sistemas, aplicando una técnica de modelado para construir sistemas con arquitecturas PRISMA, desarrollada ex profeso para esta tesis.

La técnica de modelado para construir sistemas con arquitecturas base tipo PRISMA, parte de la especificación de los requisitos funcionales que el producto final ha de satisfacer, modelados mediante un diagrama de casos de uso de UML. En particular, las variantes relevantes relacionadas con la variabilidad en la

estructura de un sistema son: el número de casos de uso, el número de actores y el número de casos de uso a los que puede acceder un actor.

Este análisis muestra que la estructura de los elementos arquitectónicos varía según el número de casos de uso considerados, así como el número de actores y de los casos de uso a los que accede un actor.

Los casos de uso constituyen una división del sistema basada en la funcionalidad. El diagrama de casos de uso muestra las distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno (usuario).

La obtención de la arquitectura base partiendo de la arquitectura genérica tiene en cuenta las particularidades (o "*features*") del caso de diagnóstico considerado. Los elementos arquitectónicos de la arquitectura base se obtienen identificando escenas funcionales del sistema y asignando cada elemento a una escena funcional, dependiendo de si la escena es simple o compleja, el elemento será un componente (componente simple) o un sistema (componente complejo), respectivamente.

El concepto de escena con un enfoque PRISMA [Pérez, 2003b] considera que "una escena se caracteriza por las tareas que se desempeñan en la actividad siguiendo un determinado protocolo, los actores que las realizan y el espacio virtual o físico donde se desarrolla"

Tomando en cuenta lo anteriormente expuesto, se presenta paso a paso la técnica para modelar arquitecturas base, utilizando el caso de estudio de un sistema experto que realiza el diagnóstico médico:

1.- *Especificar los requisitos funcionales* (en un alto nivel de abstracción) mediante un diagrama UML de casos de uso. En el ejemplo del diagnóstico médico (**figura 26**), se tienen tres casos de uso y dos actores, en donde un actor accede a dos casos de uso y el otro actor accede a un sólo caso de uso.

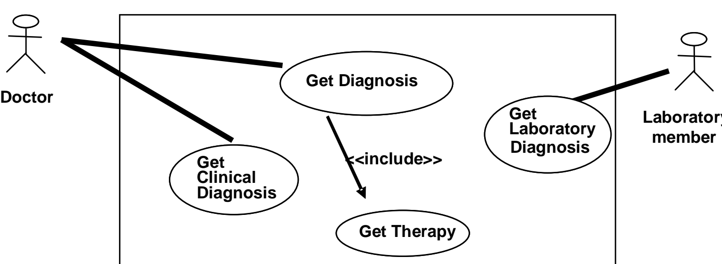
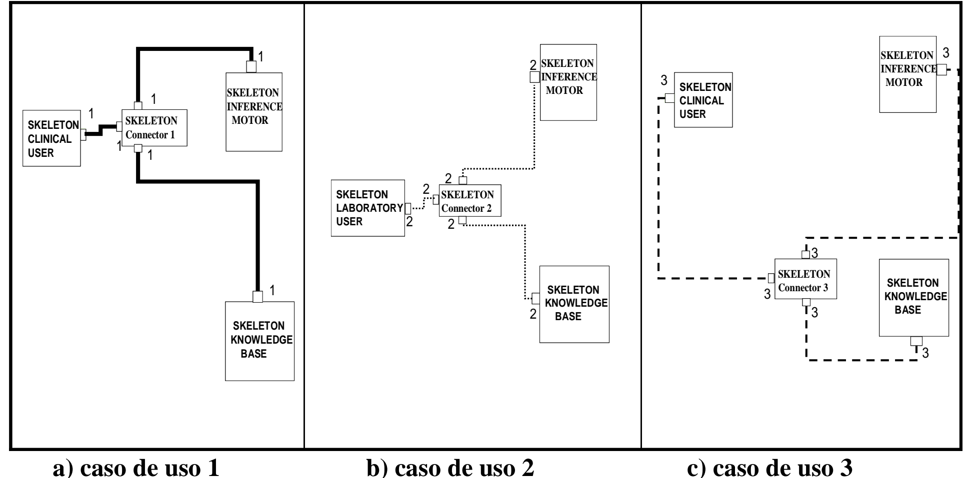


Figura 26 Diagrama UML-casos de uso del diagnóstico médico

2.- *Configurar un modelo arquitectónico, por cada caso de uso.* En el caso de estudio contemplado, se cuenta con tres casos de uso, por lo que se construyen tres modelos arquitectónicos.



a) caso de uso 1 **b) caso de uso 2** **c) caso de uso 3**

Figura 27 Modelos arquitectónicos correspondientes a cada caso de uso

3.- *Construir el modelo arquitectónico final*.- Los elementos arquitectónicos definidos en cada caso de uso son modificados para obtener los elementos arquitectónicos del modelo final, a través del siguiente criterio:

- *conectores*.- conservar los conectores, i.e. el número de conectores del modelo arquitectónico final es igual al número de casos de uso.
- *componentes*.- unir los componentes, i.e. obtener un componente por módulo y unir cada puerto del componente a un conector (por medio de un *attachment*). El número de puertos de cada componente del modelo final será igual al número de casos de uso donde fue utilizado dicho componente.

En el ejemplo que nos ocupa, el modelo arquitectónico final presenta la vista de la **figura 28**.

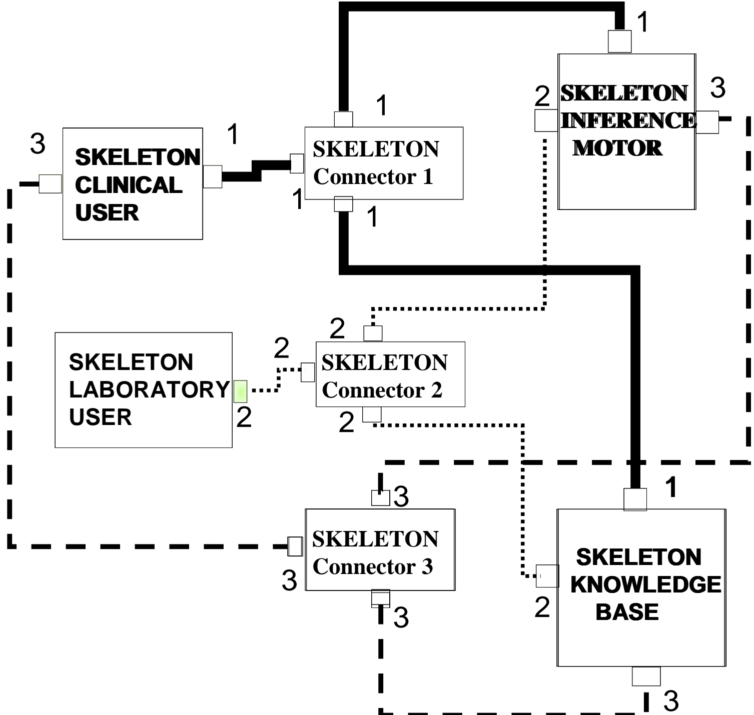


Figura 28 Modelo arquitectónico de un sistema de diagnóstico médico

En este ejemplo, el modelo arquitectónico cuenta con los siguientes elementos:

i) el componente InferenceMotor (con tres puertos)
ii) el componente KnowledegeBase (con tres puertos)
iii) el componente ClinicalUser (con dos puertos)
iv) el componente LaboratoryUser (con un puerto)
v) el conector Coordinator 1
vi) el conector Coordinator 2
vii) el conector Coordinator 3

De manera general se puede concluir que los elementos del modelo arquitectónico base tienen las siguientes características:

- *conector Coordinator.*- por casa caso de uso, hay un conector Coordinator que une a todos los componentes de ese caso de uso,
- *componente InferenceMotor* - el número de puertos del componente InferenceMotor igual al número de casos de uso,
- *componente KnowledgeBase.*- el número de puertos del componente KnowledegeBase es igual al número de casos de uso,
- *componente UserInterface.*- el número de componentes UserInterface es igual al número de actores del diagrama de casos de uso. Así mismo el número de

puertos de un componente UserInterface es igual al número de casos de uso a los que puede acceder el actor (representado por dicho componente).

5.3 Variabilidad en el comportamiento de los sistemas expertos de diagnóstico

El estudio de campo realizado en esta tesis, muestra que el comportamiento de los elementos arquitectónicos varía de acuerdo con el dominio de aplicación e implícitamente con el tipo de razonamiento usado al simular los razonamientos seguidos por los humanos para realizar un diagnóstico. Dicho comportamiento se basa en la estrategia de razonamiento que es aplicada por el componente InferenceMotor para realizar el proceso del diagnóstico. Es decir, si se aplica el razonamiento deductivo o el razonamiento diferencial, los procesos de inferencia son distintos, como se muestra en la siguiente sección.

5.3.1 Procesos de inferencia

En el componente InferenceMotor, a través del protocolo de su aspecto funcional, es especificado el proceso de decisión como una simulación de los razonamientos realizados por expertos para tomar una decisión (por ejemplo al realizar un diagnóstico). Dichos razonamientos son representados en forma de procesos. De esta manera, la variabilidad existente entre los diversos componentes InferenceMotor se corresponde con los diversos procesos de decisión.

Conforme al análisis de campo realizado en nuestra investigación, nuestra LPS contempla dos tipos de procesos de decisión: el estático y el dinámico. El estático se corresponde con el hecho de que la vista de la entidad a diagnosticar es la misma durante dicho proceso. Mientras que el dinámico se corresponde con el hecho de que la vista de la entidad a diagnosticar es diferente durante dicho proceso.

La variabilidad en el comportamiento entre los elementos arquitectónicos de un SE varía de acuerdo a la estrategia de razonamiento aplicada. Como se muestra en las **figuras 29 y 32**, dichas estrategias se corresponden con la vista de la entidad a diagnosticar durante el proceso de diagnóstico, el número de niveles de las propiedades de esa entidad y el número de hipótesis resultantes del proceso del diagnóstico.

El mecanismo de inferencia para la toma de decisión (i.e. los procesos estático y dinámico) fue modelado de varias formas:

i) a través de un diagrama de grafos
ii) utilizando el lenguaje Business Process Management Notation (BPMN)
iii) en un diagrama de transición de estados (en UML).

Así mismo se presenta la especificación de dichos procesos mediante el ADL de PRISMA usando un álgebra de procesos: el pi-calculo [Puhlmann, 2006].

5.3.1.1 Proceso de inferencia estático

En el proceso estático, una entidad siempre tiene las mismas propiedades durante el proceso del diagnóstico (i.e. la misma vista), y sólo una hipótesis (meta u objetivo del diagnóstico).

En la **figura 29** se presenta una metáfora visual del proceso de inferencia estático a través de un grafo, mostrando los niveles i de las propiedades Pi (color azul) y la hipótesis Hi (color negro), así como el razonamiento deductivo (flechas hacia arriba) de dicho proceso. Las reglas de inferencia presentan la forma:

$$p_{j,i} : P_i \leftarrow p_{k,\, i-1}{}^* : P_{i-1} \quad \text{and} \quad h_{j,i} : H_i \leftarrow p_{k,i-1}{}^* : P_i$$

donde i = 0, 1 ,..., n (n = número de niveles), j,k :Nat y suponen una iteración sobre las propiedades de nivel i-1 usadas en la obtención de la propiedad j de nivel i (en el caso de k) y lo mismo pero con la hipótesis, única en este caso, de nivel i. Por ejemplo, en la **figura 29** se tiene:

$$P0, \quad P1 \leftarrow P0, \quad P2 \leftarrow P1, \quad H \leftarrow P2.$$

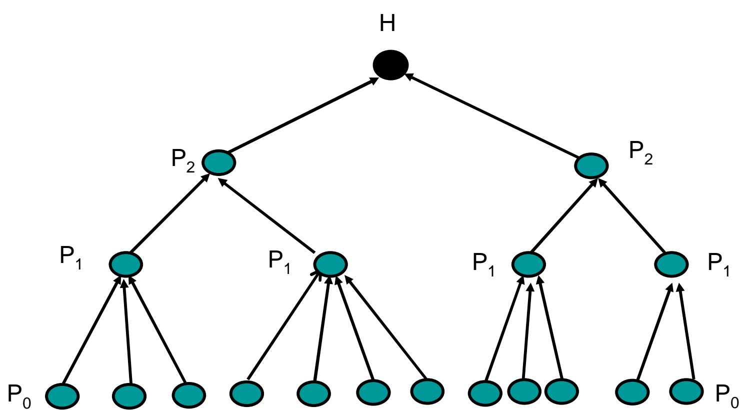


Figura 29 Grafo del proceso de inferencia estático

Las **figuras 30 y 31** muestran el modelado del proceso estático a través del BPMN y el diagrama de transición de estados en UML, respectivamente.

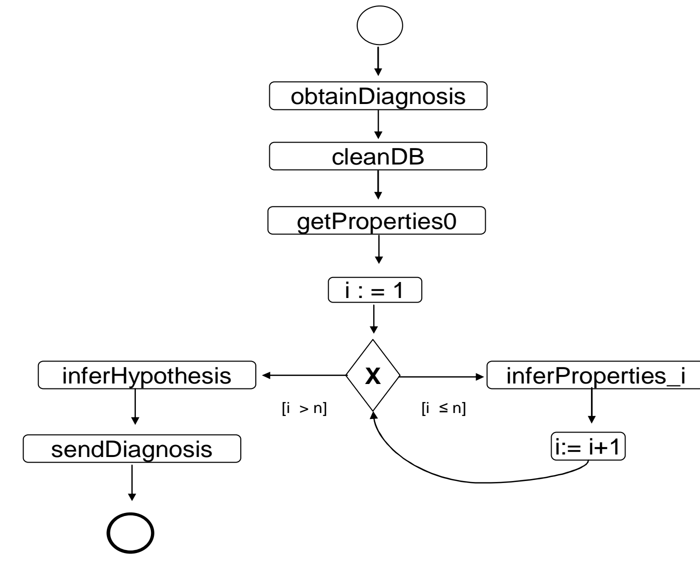


Figura 30 Modelado del proceso de inferencia estático en BPMN

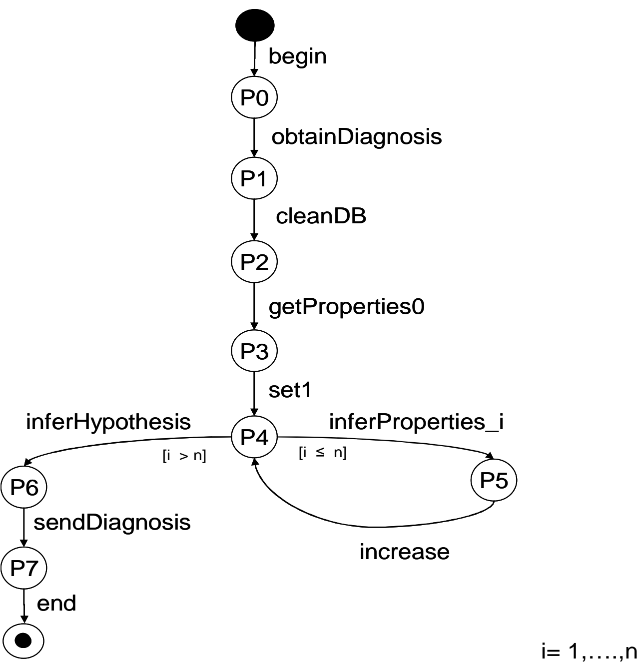


Figura 31 Diagrama de Transición de estados del proceso de inferencia estático

La especificación en el LDA de PRISMA de la **figura 31,** necesaria para que se incorpore en los activos de la Baseline (como protocolo del aspecto funcional del componente Motor de Inferencia), es la siguiente:

```
Protocols

    FINFERENCE::= begin ( ) → P0

    P0::= ( PROCESS_obtainDiagnosis ? (DIAGNOSIS) → P1

    P1::= cleanDB !  ( ) → P2

    P2::= ( PROCESS_getProperties0 ! (PROPERTIES0)
            →
            PROCESS_getProperties0 ? (PROPERTIES0) ) → P3

    P3::= set1 ( ) → P4

    P4::= ( {i<n} PROCESS_inferPropertiesI ! (PROPERTIES_0, PROPERTIES_I)
            →
            PROCESS_inferPropertiesI ? (PROPERTIES_0,  PROPERTIES_I ) ) → P5
        +
            ( {i=n} PROCESS_inferHypothesis ! (PROPERTIES_I, HYPOTHESIS)
            →
             PROCESS _inferHypothesis ? (PROPERTIES_I, HYPOTHESIS) )→ P6

    P5::= increase ( )  → P4

    P6::= PROCESS_sendDiagnosis ! (DIAGNOSIS) → P7

    P7::=  end;
```

5.3.1.2 Proceso de inferencia dinámico

En el proceso dinámico, una entidad durante el proceso del diagnóstico tiene diferentes propiedades (i.e. diferentes vistas), y existen diferentes hipótesis candidatas que deben ser evaluadas con el fin de seleccionar la hipótesis válida (meta u objetivo del diagnóstico).

En la **figura 32** se presenta una metáfora visual del proceso de inferencia estático a través de un grafo, mostrando los niveles de las propiedades (color azul), varias hipótesis (color negro), así como los razonamientos deductivo (flechas hacia arriba) e inductivo (flechas hacia abajo) en dicho proceso. Las reglas de inferencia presentan la forma:

$$h_{j,i+1} : H_{i+1} \leftarrow p_{k,i}{}^{*} : P_i \ \wedge \ h_{l,i}{}^{*} : H_i$$

donde i = 0, 1 ,..., n (n = número de niveles), j,k,l:Nat. Por ejemplo, en la **figura 32** se tiene:

$$H1 \leftarrow P0, \quad H2 \leftarrow P1 \wedge H1, \quad H3 \leftarrow P2 \wedge H2.$$

Figura 32 Grafo del proceso de inferencia dinámico

Las **figuras 33 y 34** muestran el modelado del proceso dinámico usando BPMN y el diagrama de transición de estados en UML, respectivamente.

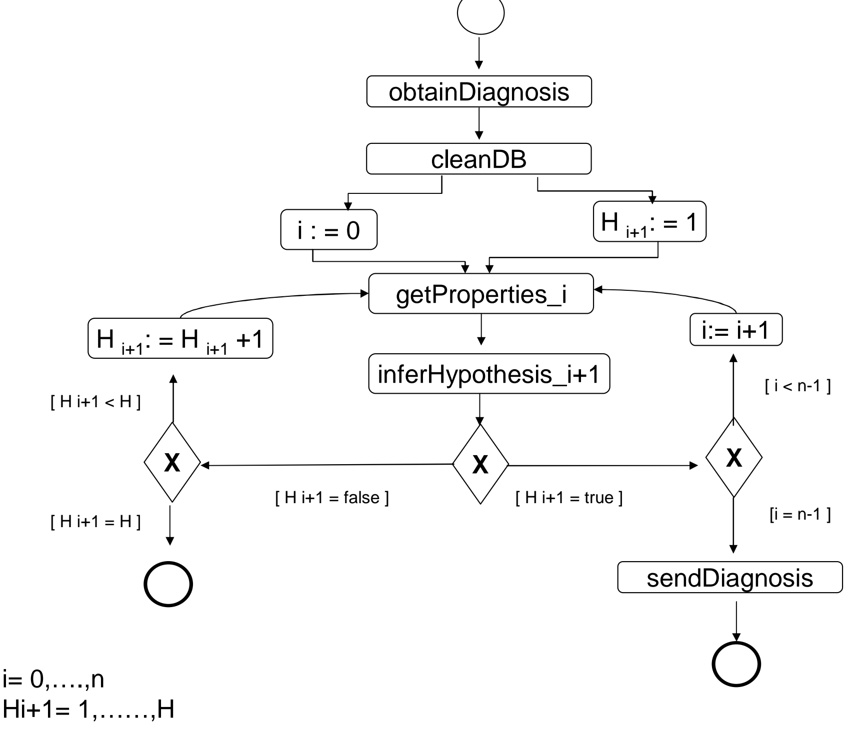


Figura 33 Modelado del proceso de inferencia dinámico en BPMN

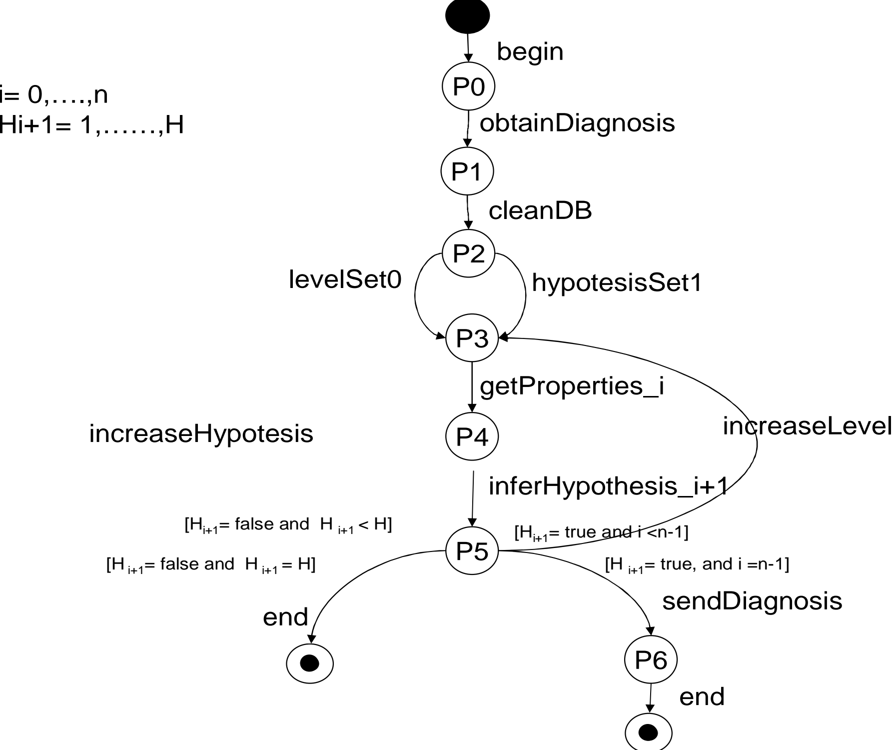


Figura 34 Diagrama de Transición de estados del proceso de inferencia dinámico

La especificación en el LDA de PRISMA de la **figura 34,** necesaria para que se incorpore en los activos de la Baseline (como protocolo del aspecto funcional del componente Motor de Inferencia), es la siguiente:

```
Protocols

    FINFERENCE::= begin ( ) → P0

    P0::= ( PROCESS_obtainDiagnosis ? (DIAGNOSIS) → P1

    P1::= cleanDB !  ( ) → P2

    P2::= ( setLevel0 ( )  |  setHypotesis1 ( ) ) → P3

    P3::= ( PROCESS_getProperties ! (PROPERTIES_I)
            →
            PROCESS_getProperties ? (PROPERTIES_I) ) → P4

    P4::= ( PROCESS_inferHypothesis ! (PROPERTIES_I, HYPOTHESIS_i+1)
            →
             PROCESS _inferHypothesis ? (PROPERTIES_I, HYPOTHESIS_I+1) )→ P5

    P5::= ( {Hi+1=true and i=n-1} PROCESS_sendDiagnosis ! (DIAGNOSIS)
                                  →
                                  PROCESS_sendDiagnosis ? (DIAGNOSIS) ) → P6
            +
             {Hi+1=true and i<n-1} increseLevel ( ) → P3
            +
             {Hi+1=false and Hi+1<H} increseHypotesis ( ) → P3
            +
             {Hi+1=false and Hi+1=H} end

    P6::=  end;
```

5.4 Conclusiones

El estudio de campo realizado en los SED, muestra que la arquitectura base de estos sistemas no es única, sino que varía en su estructura y comportamiento.

Asimismo, con el modelado de los requisitos y la funcionalidad de los SE, se mostró la variabilidad en la estructura y en el comportamiento de dichos sistemas, respectivamente. Considerando las características que representan dichas variabilidades se puede concluir que:

- la variabilidad en la estructura de un SED depende del número de casos de uso, del número de actores y casos de uso por actor, i.e. la variabilidad en los requisitos del usuario final
- la variabilidad en el comportamiento de un SED está relacionada con la estrategia de razonamiento que se aplique, la vista de la entidad durante el proceso del diagnóstico y el número y nivel de las hipótesis que intervienen en dicho proceso, i.e. la variabilidad en el proceso del diagnóstico.

Con el análisis de los SED realizados en este capítulo, se identificó y definió la variabilidad en este tipo de sistemas, lo cual permite presentar (en el próximo capítulo) como es gestionada la variabilidad en BOM.

PARTE IV. APROXIMACIÓN: BASELINE-ORIENTED MODELING

LA CREACIÓN (MICHELANGELO DI LODOVICO BUONARROTI SIMONI)
(1475-1564)

CAPÍTULO 6

GESTIÓN DE LA VARIABILIDAD EN BOM

La ciencia es la progresiva aproximación del hombre al mundo real.
Max Planck (1858-1947) Físico alemán

6.1 Introducción

En este capítulo se muestra la gestión de la variabilidad a través de la aproximación BOM.

La estructura de este capítulo es el siguiente. En la sección 2 se comenta la forma en que BOM gestiona la variabilidad. En la sección 3 se ilustra ampliamente el manejo de la primera variabilidad y en la sección 4 el de la segunda variabilidad. En la sección 5 se presentan las conclusiones de este capítulo.

6.2 La gestión de la variabilidad en BOM

La variabilidad entre los productos de una LPS puede ser expresada en términos de características [Kang et al., 1998], [Bosch, 2000]. Pero el tratamiento de la variabilidad implica por un lado el manejo de las características del dominio plasmadas en un Modelo de Características, y por otro lado el que la variabilidad debe ser captada en activos (product´s line core *assets*). Además la variabilidad del dominio de aplicación plasmada en sus características (*features*) deben decorar a un Esqueleto Arquitectura Base para obtener el producto final.

Así pués, el manejo de la variabilidad solamente a través de un Modelo de Características y la decoración con sus diferentes *features*, no resuelve la complejidad de los sistemas antes enunciados. Esto obliga a gestionar la variabilidad además sobre las arquitecturas base que comparten una arquitectura genérica.

Por ello se ha tratado la variabilidad en dos fases. La variabilidad inicial es tratada a través de puntos de variabilidad plasmados en un árbol de decisión, el cual selecciona la arquitectura base correcta. La 2ª variabilidad es manejada decorando esas arquitecturas base con las características del dominio de aplicación.

De esta manera, las características que definen la línea de productos software (LPS) implican dos tipos de variabilidad ortogonales. La primera variabilidad es dada por el propio proceso del diagnóstico (i.e. son las características del dominio del diagnóstico) y los requisitos del usuario final, lo que implica una arquitectura base específica. La segunda variabilidad corresponde al campo de aplicación del diagnóstico (por ejemplo, las características del diagnostico de programas educativos, diagnóstico televisivo, diagnóstico médico, etc.).

BOM es una aproximación donde la especificación de la variabilidad y la funcionalidad se manejan en modelos separados: representados a través de modelos conceptuales, que conforman a sus respectivos metamodelos.

La aproximación utilizada en BOM, modela la variabilidad distinguiendo un modelo de variabilidad y un modelo funcional (la arquitectura genérica como modelo independiente de la variabilidad). Pero, como ya fue comentado en el capítulo inmediato anterior en esta tesis, en los sistemas expertos, el modelo funcional no es único, lo que provoca trabajar con diversos modelos del sistema. Por esta razón fue necesario dividir esta situación en dos partes:

- i) (primera variabilidad) : modelar el sistema con: el modelo de variabilidad (variabilidad del dominio) y el modelo de funcionalidad (i.e. una única arquitectura genérica de los SE), y obtener (a través de técnicas de árbol de decisión) con estos datos uno de los modelos de arquitectura base.
- ii) (segunda variabilidad) : modelar el sistema con: el modelo de variabilidad (variabilidad del dominio de aplicación) y el modelo de funcionalidad (i.e. la arquitectura base seleccionada anteriormente de la Baseline) con el fin de obtener (a través de decorar a las arquitecturas base con las características del dominio de aplicación) un modelo de arquitectura PRISMA como producto final de la LPS (i.e. un SE).

Por ello, la aproximación BOM contempla dos LPS que se corresponden con las dos variabilidades contempladas:

- la LPS1, i.e la LPS de arquitecturas base, y
- la LPS2, i.e. la LPS de arquitecturas PRISMA.

Además, siguiendo los resultados del estudio de campo realizado, se puede concluir que la variabilidad gestionada en BOM incluye dos tipos de variabilidad ortogonal, como lo muestra la **figura 35**:

- La primera variabilidad (V1) es la relacionada con el proceso del diagnóstico y los requisitos del usuario, la cual determina una arquitectura base específica. Las características que se corresponden con el proceso de diagnóstico son las relacionadas con el comportamiento de los SE (i.e. vista de la entidad, nivel de las propiedades, número de hipótesis, tipo de razonamiento). Las características que se corresponden con los requisitos del usuario son las relacionadas con la estructura de los SE (i.e. número de casos de uso, número de actores, número de casos de uso por actor).

- La segunda variabilidad (V2) es la relacionada con el dominio de aplicación específico, la cual determina el producto final de la LPS. Las características del dominio de aplicación son las propiedades del nivel *n*, las reglas de inferencia del nivel *n-1* y las hipótesis.

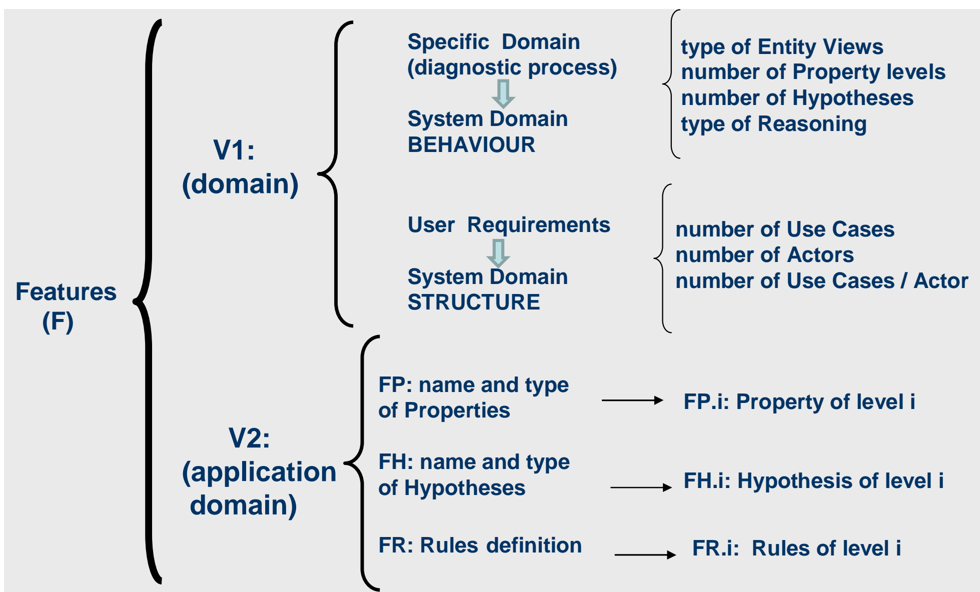


Figura 35 Clasificación de las características de la variabilidad gestionada por BOM

6.3 La primera variabilidad en BOM

El primer tipo de variabilidad involucra una familia de arquitecturas base, a la que se denomina LPS1. Esta LPS1 es almacenada en la Baseline, por ello la Baseline es en sí una LPS formada por todas las arquitecturas base como activos reutilizables o *assets*.

La variabilidad inicial representa la variabilidad del dominio (propiedades de la entidad que participan en el proceso del diagnóstico, niveles de dichas propiedades, tipos de razonamiento y número de hipótesis), y de la variabilidad en los requisitos del usuario final del sistema (número de casos de uso, número de actores, número de casos de uso por actor).

Las características de esta primer variabilidad están presentes en el Modelo de Características de la línea de productos software de diagnóstico (ver **figura 22**), las cuales son plasmadas en un Árbol de Decisión y captadas como instancias en un Modelo Conceptual del Dominio, representando todos los valores de la variabilidad. El Modelo de Características, el Árbol de Decisión y el Modelo Conceptual del Dominio, como mecanismos de variabilidad, son constructores que realizan la variabilidad en el nivel de artefacto (en este caso, de la arquitectura base).

El árbol de decisión permite seleccionar los *assets* necesarios para construir el modelo de las arquitecturas base. Las hojas del árbol de decisión apuntan hacia dichos *assets*.

La estructura de los elementos arquitectónicos de un SE involucra las características sobre tres puntos de variabilidad en el árbol de decisión: número de casos de uso, número de actores y número de casos de uso por actor. Estas características son utilizadas para seleccionar los activos que ayudarán a configurar las arquitecturas base.

El comportamiento de los elementos arquitectónicos de un SE involucra las características representadas por los siguientes cuatro puntos de variabilidad en el árbol de decisión: las vistas de las entidades, los niveles de las propiedades, el número de hipótesis y el tipo de razonamientos. Estas características están relacionadas con los servicios y sus evaluaciones, los protocolos y los *played_roles*, de los aspectos que contienen los elementos arquitectónicos PRISMA, i.e. la información de estado, de los procesos y de los roles, respectivamente. En consecuencia, y para optimizar el proceso de inserción de las características, en lugar de repetir estas características en cada uno de los tipos PRISMA, se fijan en los esqueletos.

El manejo de la variabilidad del dominio puede ser visto como una transformación que recibe de entrada el modelo de la variabilidad del dominio y el modelo funcional de la arquitectura genérica, produciendo como salida el correspondiente modelo de la arquitectura base.

6.4 La segunda variabilidad en BOM

El segundo tipo de variabilidad involucra las LPS de la aplicación en un campo específico. Esta variabilidad permite a las arquitecturas base ser enriquecidas o decoradas con las características del dominio de aplicación.

En el proceso del manejo de esta variabilidad, las variantes del dominio de aplicación específico son dadas como instancias de un modelo conceptual del dominio de aplicación. Dichas variantes o características son insertadas (por medio de transformaciones) en los esqueletos de las arquitecturas base con el fin de generar los artefactos software: tipos PRISMA.

Las características que son usadas para rellenar o decorar los esqueletos son: el nivel, nombre y tipo de las propiedades de la entidad a diagnosticar, el nivel, nombre y tipo de las hipótesis, y las reglas que relacionan a ambas características.

En la gestión de la variabilidad del dominio de aplicación, una LPS puede ser vista como un conjunto de modelos compartiendo un conjunto común de características manejables, que son desarrolladas usando un conjunto de activos.

A continuación se presenta la especificación de la gestión de las características del dominio de la aplicación (i.e. de la segunda variabilidad).

De acuerdo con Trujillo [Trujillo, 2007], en esta tesis se define un modelo de la línea de producto software de diagnóstico como:

```
M-X = {FP.i, FH.i, FR.i},
```

donde M-X es el modelo del dominio de aplicación específica X y el resto de elementos son las características de dicha línea de producto, donde i= 0,1,…….,n.

Así mismo, el diseño de un programa se define como:

```
progLPS-X = FP.i FH.i FR.i
```

lo que significa que el programa LPS-X tiene las características: FP.i, FH.i, FR.i: Features de las Propiedades, Hipótesis y Reglas respectivamente.

Por ello, el conjunto de programas que puede ser creado desde un modelo es una línea de producto. Es decir, el conjunto de programas que se corresponden con los elementos arquitectónicos que son creados desde el modelo M-DM del diagnóstico médico, es la línea de producto del caso médico.

Por ejemplo: para el caso de estudio del diagnóstico médico, se tiene:

Características de las propiedades de nivel 0:

FP.0 = tos, fiebre, dificultad respiratoria, ...

Características de las propiedades de nivel 1:

FP.1 = tos seca, tos con flema, fiebre continua, dificultad respiratoria grave, ...

Características de las hipótesis de nivel 1:

FH.1 = ira, parotiditis,....

Características de las hipótesis de nivel 2:

FH.2 = bronquiolitis, neumonía, crup espasmódico, paperas, parotiditis bacteriana, ...

Características de las reglas de nivel 1:

FR.1 =

{fiebre = true **and** tos = true **and** dificultad respiratoria=true} síndrome="ira"

{fiebre = true **and** dolor a masticación = true **and** parótidas anormales=true} síndrome="parotiditis"

Características de las reglas de nivel 2:

FR.2 =

{síndrome="ira" } enfermedad**=** **"**bronquiolitis"

{síndrome="ira" } enfermedad**=** **"**neumonía"

{síndrome="ira" } enfermedad**=** **"**crup espasmódico"

{síndrome=" parotiditis" } enfermedad**=** "paperas"

{síndrome=" parotiditis" } enfermedad**=** "parotiditis bacteriana"

....

Características de las reglas de nivel 3:

FR.3 =

{ fiebre continua=true **and** fiebre mayor a 38 =true **and** dolor y crecimiento de parótidas =true **and** dolor a masticación espontáneo agudo=true } enfermedad= "paperas"

....

El Modelado Orientado a Características (FOM) ofrece un conjunto de operaciones, donde cada operación implementa una característica. Las características pueden ser distinguidas como constantes o funciones [Trujillo, 2006]. De esta manera, las características consideradas como constantes representan modelos base (en nuestro caso, los esqueletos Arquitecturas Base). Los esqueletos son decorados o rellenados con las características del dominio de aplicación, para crear los tipos PRISMA respectivos. Las características son modeladas como funciones, representando refinamientos de modelos. Estos modelos son extensiones del modelo base de entrada. Por ejemplo:

$$\text{Fx.i} \bullet \text{E-MI-DM}_j,$$

que significa: "agregar la característica Fx.i al modelo base E-MI-DM", donde • denota la aplicación de la función, j=0,1,…,n, y E-MI-DMo es el Esqueleto del Motor de Inferencia correspondiente al caso del Diagnóstico Médico.

El proceso de decoración con características es representado por el siguiente algoritmo:

$$\text{E-MI-DM}_1 = \text{FP.0} \bullet \text{E-MI-DM}_0$$
$$\text{E-MI-DM}_2 = \text{FP.1} \bullet \text{E-MI-DM}_1$$
$$\text{E-MI-DM}_3 = \text{FH.1} \bullet \text{E-MI-DM}_2$$
$$\text{E-MI-DM}_4 = \text{FH.2} \bullet \text{E-MI-DM}_3$$
$$\text{E-MI-DM}_5 = \text{FR.1} \bullet \text{E-MI-DM}_4$$
$$\text{E-MI-DM}_6 = \text{FR.2} \bullet \text{E-MI-DM}_5$$
$$\text{E-MI-DM}_7 = \text{FR.3} \bullet \text{E-MI-DM}_6$$

Donde <Fx.i> son las características a rellenar, dadas por:

FP.0 = Características de las propiedades de nivel 0,
FP.1 = Características de las propiedades de nivel 1,
FH.1 = Características de las hipótesis de nivel 1,
FH.2 = Características de las hipótesis de nivel 2,
FR.1 = Características de las reglas de nivel 1,
FR.2 = Características de las reglas de nivel 2,
FR.3 = Características de las reglas de nivel 3,

Para clarificar el algoritmo anterior, en la **figura 36** se presenta una metáfora visual del proceso de inserción de las características de un esqueleto seleccionado de la Baseline (E-MI-DMo) para crear su correspondiente tipo PRISMA.

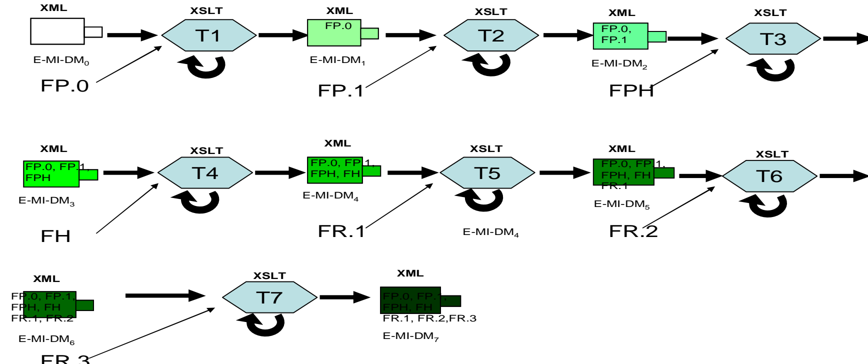


Figura 36 Proceso de inserción de las características en un esqueleto para formar su tipo PRISMA

La **figura 37** muestra el proceso desde la inserción de características en los esqueletos, hasta la configuración de la arquitectura del modelo específico de la LPS.

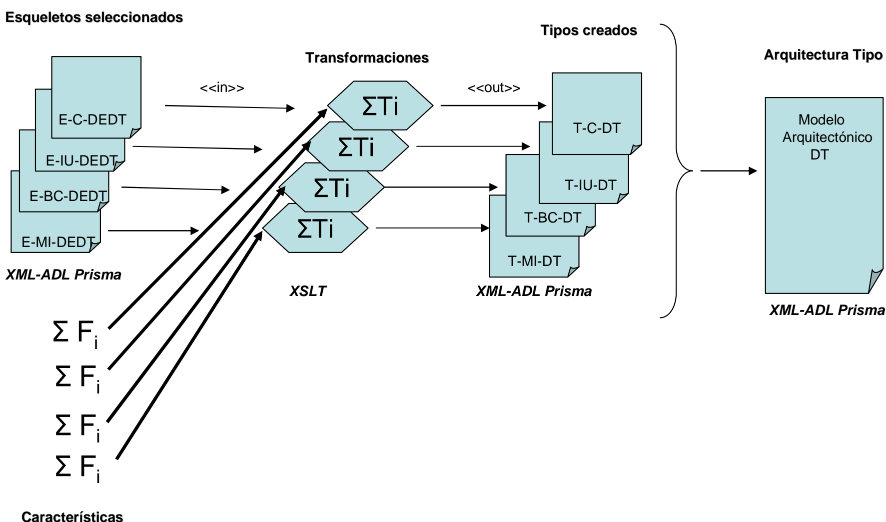


Figura 37 Proceso desde la inserción de características en los esqueletos, hasta la configuración de la arquitectura

Como un ejemplo, se presenta en la **tabla 5**, el esqueleto aspecto funcional del componente KnowledgeBase y su respectivo tipo PRISMA, del caso de estudio del diagnóstico médico. Ambos son especificados en el LDA de PRISMA. En negrita se ponen los huecos que son rellenados con las características del dominio de aplicación.

Functional aspect of the Knowledge Base of a skeleton	**Functional aspect of the Knowledge Base of a PRISMA type**
Functional Aspect FBaseMD using IDomainMD Attributes Variable **< FP.0 >**; **< FP.1 >**; **< FPL.1 >**; Derived **< FH.1 >**, derivation **< FR.1 >** **< FH.2 >**, derivation **< FR.2 >**, **< FH.2 >**, derivation **< FR.3 >**; Services ... End_Functional Aspect FBaseDM	Functional Aspect FBaseMD using IDomainMD Attributes Variables **cough**: bool; **fever**: bool; **respiratory_dificulty**: bool; ... **dry_cough**: bool; **constant_fever**: bool; **grave_respiratory_dificulty**:bool; ... **positive_hematic_biometric**:bool; ... Derived syndrome: string; derivation **{cough=true and fever=true and respiratory_dificulty=true}** **syndrome="warth"**; posible_disease: string; derivation **{syndrome="warth"}** **posible_disease="pneumonia";** **{syndrome="warth"}** **posible_disease="crup";** **{syndrome="warth"}** **posible_disease="bronchilitis"**; disease: string; derivation **{constant_fever=true and** **greater_39_fever=true and** **2_3_days_fever=true and** **phlegmatic_cough=true and** **frequent_cough=true and** **positive_hematic_biometric=true)}** **disease="pneumonia";** ... Services ... End_Functional Aspect FBaseDM

Tabla 5 Ejemplo de un aspecto esqueleto y su correspondiente tipo PRISMA

Es importante mencionar que una arquitectura base puede ser instanciada a una o más arquitecturas PRISMA (i.e. a varios productos de la LPS). Como ejemplo de ello tenemos los casos de estudio del diagnóstico de programas educativos y el diagnóstico televisivo realizados en el estudio de campo de esta tesis. Estos casos comparten la misma arquitectura base esqueleto, pero cada uno de ellos origina diferentes arquitecturas finales PRISMA, debido a que en cada una fueron añadidas o insertadas diferentes propiedades del dominio de aplicación. La **figura 38** muestra una metáfora visual de ello.

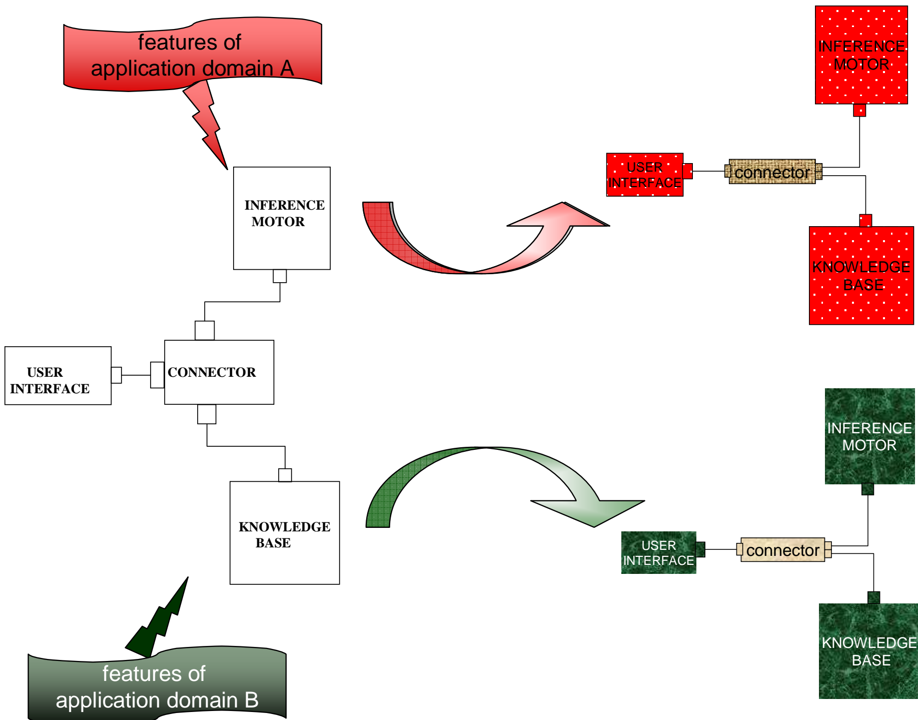


Figura 38 Metáfora visual de una arquitectura base y dos arquitecturas PRISMA

Para mostrar lo anteriormente dicho, en la **tabla 6** se presenta el esqueleto del aspecto funcional del componente KnowledgeBase y sus correspondientes tipos PRISMA de los dos casos de estudio. Igualmente al caso del diagnóstico médico (usado como ejemplo en esta tesis), ambos son especificados en el LDA de PRISMA. Los huecos ("feature holders") que son rellenados con las características del dominio de aplicación, son resaltados con negrita.

Functional aspect of the Knowledge Base of a skeleton	*Functional aspect of the Knowledge Base of a PRISMA type (case study: Educational Program Diagnosis)*	*Functional aspect of the Knowledge Base of a PRISMA type (case study: TV Diagnosis)*
Functional Aspect **FBaseDPEDT** using IDomainDPEDT	Functional Aspect **FBaseDPEDT** using IDomainDPEDT	Functional Aspect **FBaseDPEDT** using IDomainDPEDT
Attributes Variables **< FP.0 >,**	Attributes Variables **pictures:**string, **TVstructure:**string; **music:**string, **audio:**string, ...	Attributes Variables **laboratories:**string, **library:**string, **criticalMass:**string, **scientificProductivity:**string; ...
Deriveds **< FP.1 >,** **< FH >**	Deriveds **production**:string, **content**:string **technicalQuality**:string; **stateVideo**:string;	Deriveds **infrastructure**:string, **faculty**:string **develpmentStage**:string; ...
Derivations **< FR.1 >** **< FR.2 >**	Derivations **{pictures="proper" and TVstructure="authorized" and music="good"} production:="good"** **...** **{production="good" and content ="good" and technicalQuality="good"} sateVideo:="VideoOK"**	Derivations **{laboratories="good" and library=" good"} infrastructure:=" good"** **{critical_mass="good" and scientificProductivity =" good"} faculty:= "good"** **{infrastructure="good" and faculty= "good"}develpmentStage:="consolidated;**
....	......	......
Services …….	Services …….	Services …….
Played_Roles	Played_Roles	Played_Roles ....
Protocols …….	Protocols …….	Protocols …….
End_Functional Aspect F**BaseDPEDT**	End_Functional Aspect **FBaseDPEDT**	End_Functional Aspect **FBaseDPEDT**

Tabla 6 Ejemplo de un aspecto esqueleto y dos aspectos tipo PRISMA (casos: Diagnóstico de programas educativos y Diagnóstico televisivo)

6.5 Conclusiones

La forma de gestionar en dos etapas la variabilidad en BOM es la principal aportación de esta tesis en el ámbito de las LPS.

Dicha iniciativa fue resultado de observar que el manejo de la variabilidad con un solo modelo de características y el tratamiento de la variabilidad por decoración con diferentes caracterísiticas, no solucionaban la complejidad del problema del tratamiento de la variabilidad. Esto provocó gestionar la variabilidad sobre las arquitecturas base que comparten una arquitectura genérica.

De esta manera, en una primera etapa (con la variabilidad inicial) se selecciona la arquitectura base correcta, y en una segunda etapa (con la segunda variabilidad) se decoran esas arquitecturas base con las características del dominio de aplicación.

Por ello, se puede concluir que BOM maneja dos tipos de variabilidad, que se corresponden con el desarrollo de dos LPS:

i) La LPS de las arquitecturas base, denominada SPL1, la cual se corresponde con la relación *is_a* en la taxonomía de los SED, y que conforman familias de productos que comparten características comunes en el domino.

ii) La LPS de las arquitecturas PRISMA, denominada SPL2, la cual se corresponde con la relación *is_instance_of* en la taxonomía de los SED, y que conforman los productos finales específicos del dominio de la aplicación, (i.e. son las instancias de las arquitecturas base).

La gestión de la variabilidad es pues, la base para el desarrollo de los SED que se presentará en los capítulos 7 y 8 siguientes. En dichos capítulos se exhibirán dos aproximaciones de BOM (la aproximación BOM-EAGER y la aproximación BOM-LAZY). El tamaño de la Baseline determinará la aproximación más adecuada como una posible solución al desarrollo de la LPS, de la siguiente manera:

i) si la Baseline es pequeña (dominios de cardinalidad baja), es conveniente elegir a la aproximación BOM-EAGER, aplicando técnicas de árbol de decisión (para la primera variabilidad) y técnicas de FOM (para la segunda variabilidad). En BOM-EAGER, la Baseline es construida previamente (en la fase de la ingeniería del dominio), i.e. la Baseline es explícita y se accede a ella para elegir los "assets" en el Plan de Producción de la LPS (en la fase de la ingeniería de la aplicación).

ii) si la Baseline es grande (dominios de cardinalidad alta), es conveniente elegir la aproximación BOM-LAZY, aplicando técnicas de transformación de modelos (con QVT-Relations). En BOM-LAZY, la

Baseline se construye en el momento de la primera transformación (correspondiente a la primera variabilidad), i.e. la Baseline es implícita y se realiza un cálculo para construir los "assets" en el Plan de Producción de la LPS. BOM-LAZY es una solución computacional, y la Baseline es construida en la fase de la ingeniería de la aplicación cuando se ejecuta el Plan de Producción.

Cabe señalar que ambas aproximaciones fueron implementadas *exprofeso* para la investigación. Para la aproximación BOM-EAGER fue desarrollado un prototipo y para la aproximación BOM-LAZY fue implementada la transformación T1 utilizando QVT-Relations. Ambas se presentan en la parte VI (Implementación) de esta tesis.

CAPÍTULO 7

APROXIMACIÓN BOM-EAGER: DESARROLLO DE LA LPS MEDIANTE TÉCNICAS "AD-HOC" UTILIZADAS EN BOM PARA EL TRATAMIENTO DE LA VARIABILIDAD

La técnica es el esfuerzo para ahorrar esfuerzo
José Ortega y Gasset (1883-1955). Filósofo español

7.1 Introducción

En este capítulo se describe el desarrollo de la LPS, a través de las técnicas *ad-hoc* utilizadas por BOM para el tratamiento de la variabilidad.

La estructura de este capítulo es la siguiente: En la sección 2 se presenta la técnica utilizada en BOM para el tratamiento de la 1ª variabilidad (de la arquitectura genérica a las arquitecturas base). En la sección 3 se presenta la técnica utilizada en BOM para el tratamiento de la 2ª variabilidad (de la arquitectura base a la arquitectura PRISMA). En la sección 4 se describe el desarrollo de la arquitectura de los sistemas expertos a través de la aproximación BOM. La sección 5 presenta las conclusiones de este capítulo.

7.2 Tratamiento de la primera variabilidad: de la arquitectura genérica a las arquitecturas base.

La variabilidad inicial o primera variabilidad (relacionada con las características del dominio) es representada y manejada a través de dos modelos separados:

- El primer modelo corresponde al modelo funcional del dominio, representado por la arquitectura genérica de nuestra LPS. Esta arquitectura genérica es compartida por varias arquitecturas base (o plantillas de arquitecturas base), que representan la LPS1.

- El otro modelo corresponde al modelo de la variabilidad del dominio, representado por el MCD, que es plasmado en un árbol de decisión. El DSL se utiliza para manejar la variabilidad tanto a nivel de definición para crear el árbol de decisión, como para usar dicho árbol para elegir (por medio de puntos de variabilidad y la generación de variantes) las diferentes arquitecturas base de los dominios específicos.

De esta manera, la primera variabilidad implica la LPS de los esqueletos de las arquitecturas base. i.e. la LPS1. Esta LPS se encuentra almacenada en la Baseline. La Baseline es en sí una LPS, conformada por todos los *assets* necesarios para construir dichas arquitecturas base esqueleto.

Es interesante notar que una arquitectura base (activo reutilizable) puede ser visto como la constante de una LPS en sí misma, sólo que el producto que sale de la misma es una instancia específica del activo, en vez de una aplicación completa.

Para poder obtener las arquitecturas base como resultado de la combinación del modelo funcional (arquitectura genérica del SED) y del modelo de la variabilidad (MCD) en dos vistas separadas, se debe realizar una transformación de modelos (que llamamos T1).

Cabe aquí mencionar que la arquitectura genérica (arquitectura de los SED) conforma al metamodelo de módulos, el MCD conforma al metamodelo de clases de UML, y las arquitecturas base conforman el metamodelo de componentes-conectores (explícitamente el metamodelo Esqueleto, i.e. el metamodelo de PRISMA con huecos),

Dado que el número de las arquitectura base es limitado (esto no implica que no pueden agregarse más arquitecturas base, ya que BOM es una aproximación reactiva de LPS), en BOM, la transformación T1 es realizada como una función ya calculada a través de una tabla (representada por la Baseline). T1 es una función (1 a n) con dominio y codominio limitado, en donde el cálculo se realiza una sola vez, y el resultado es dado para todas las variantes. De esta forma, dadas las variantes se implementa T1 como un acceso a la tabla.

Esta tabla es la Baseline (la cual contiene los esqueleto de las arquitecturas base). La Baseline contiene de forma implícita la transformación T1. Con esta función se generan, partiendo de una única arquitectura genérica, las distintas arquitecturas base.

Pero, de éstas *n* arquitecturas base, debe ser seleccionada solamente una, adecuada al caso específico del dominio. Por ello, en BOM, la variabilidad del MCD ha sido plasmada en un árbol de decisión. De esta manera, con técnicas de árbol de

decisión puede ser seleccionada la arquitectura base del caso específico, a través de la introducción de los datos de la variabilidad del dominio expresados como variantes de los puntos de variabilidad (características del dominio).

Esta función recibe como dato sólo la variabilidad plasmada en el árbol de decisión (ya que la arquitectura genérica es fija). La función de transformación T1 es:

T1 (modelo_arqGenérica, modelo_variabilidad1) =

modelo_arqBase: modelo_LPS1

Con BOM se obtienen ventajas al tratar separadamente los modelos funcional y de variabilidad, permitiendo convertir una arquitectura genérica en varias arquitecturas base y modelar la variabilidad independientemente.

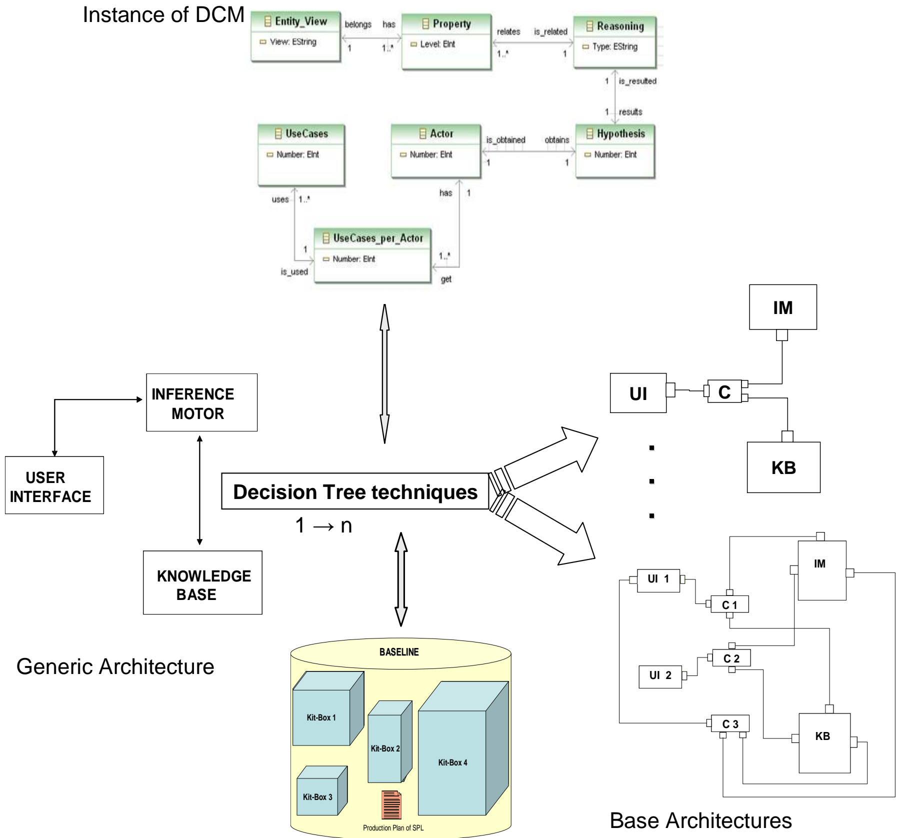


Figura 39 Tratamiento de la variabilidad V1 a través de BOM

7.3 Tratamiento de la segunda variabilidad: de la arquitectura base a la arquitectura PRISMA.

La segunda variabilidad (relacionada con las características del dominio de la aplicación) también es representada y manejada con dos modelos distintos:

- El primer modelo, corresponde a un modelo conceptual funcional del dominio de la aplicación, representado por una arquitectura base de la LPS1.
- El segundo modelo corresponde al modelo conceptual de la variabilidad del dominio de aplicación, representado por MCDA.

De esta manera, la segunda variabilidad implica la LPS de la aplicación en un campo específico, i.e. la LPS2. Esta variabilidad permite a las arquitecturas base de nuestra LPS ser enriquecidas o decoradas con las "*features*" del dominio de aplicación.

En BOM, se modela explícitamente la variabilidad del dominio de aplicación, ajeno a las arquitecturas base esqueleto. El DSL se utiliza para manejar la variabilidad tanto a nivel de definición para crear el MCDA, como para usar dicho MCDA (por medio de las características del dominio de aplicación) decorando las arquitectura base con dichas "*features*" para obtener una aplicación específica, i.e. los productos de la LPS2.

Esto se puede realizar mediante una transformación T2, que consiste en tomar los dos modelos (el modelo de arquitectura base seleccionado y el modelo de variabilidad del dominio de aplicación) y obtener un modelo (el modelo de la arquitectura PRISMA).

Para la transformación T2, la aproximación de BOM, implementa una función con la que se obtiene la arquitectura PRISMA, recibiendo como dato sólo la variabilidad del dominio de aplicación (ya que la arquitectura base es fija). i.e

T2 (modelo_arqBase, modelo_variabilidad2) =
modelo_arqPRISMA : modelo_LPS2

En BOM se obtienen ventajas al tratar separadamente los modelos funcional y de variabilidad, permitiendo convertir una arquitectura base en una arquitectura PRISMA, y modelar la variabilidad independientemente.

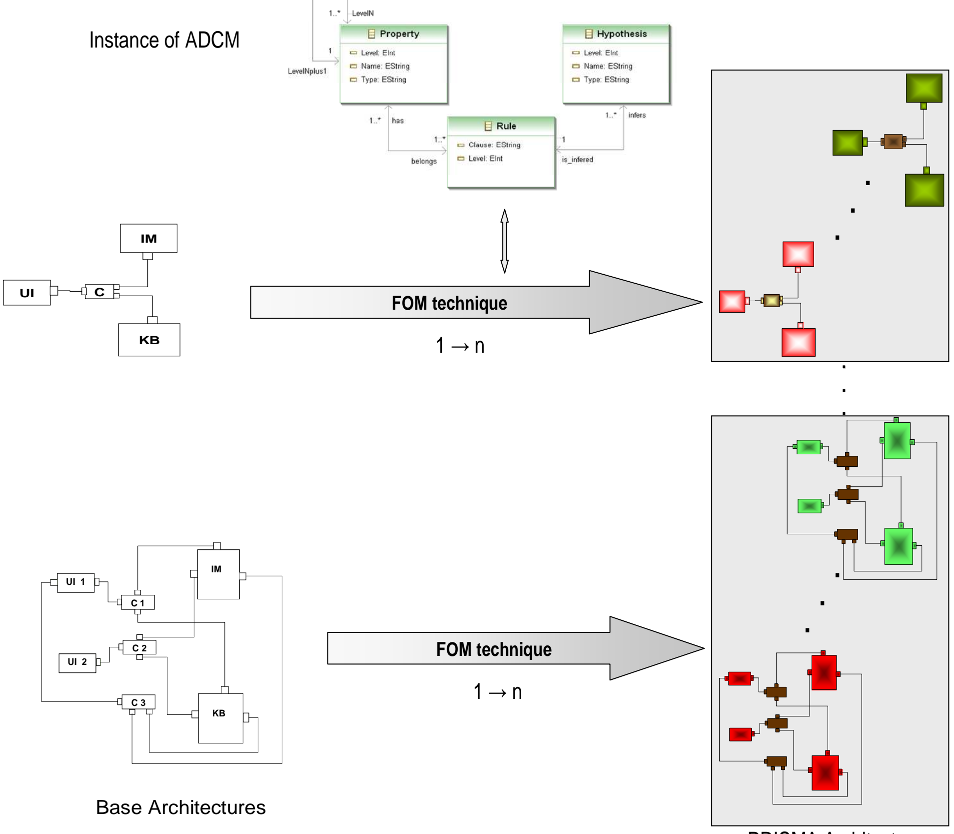


Figura 40 Tratamiento de la variabilidad V2 a través de BOM

7.4 Desarrollo de Sistemas Expertos de Diagnóstico en BOM

Un resumen de las anteriores secciones 2 y 3 de este capítulo, se muestra en la **figura 41** a través de una metáfora visual sobre el desarrollo de los SED a través de BOM.

En dicha figura se observa un primer paso donde a partir de la arquitectura genérica se obtiene la arquitectura base seleccionada (por técnicas de árbol de decisión), a través de una instancia del modelo MCD que captura la variabilidad V1 del dominio. Un segundo paso es realizado para decorar la arquitectura base y obtener la arquitectura PRISMA (por técnicas FOM, utilizando transformaciones

con plantillas XSLT sobre documentos XML), a través de una instancia del modelo MCDA que captura la variabilidad V2 del dominio de aplicación.

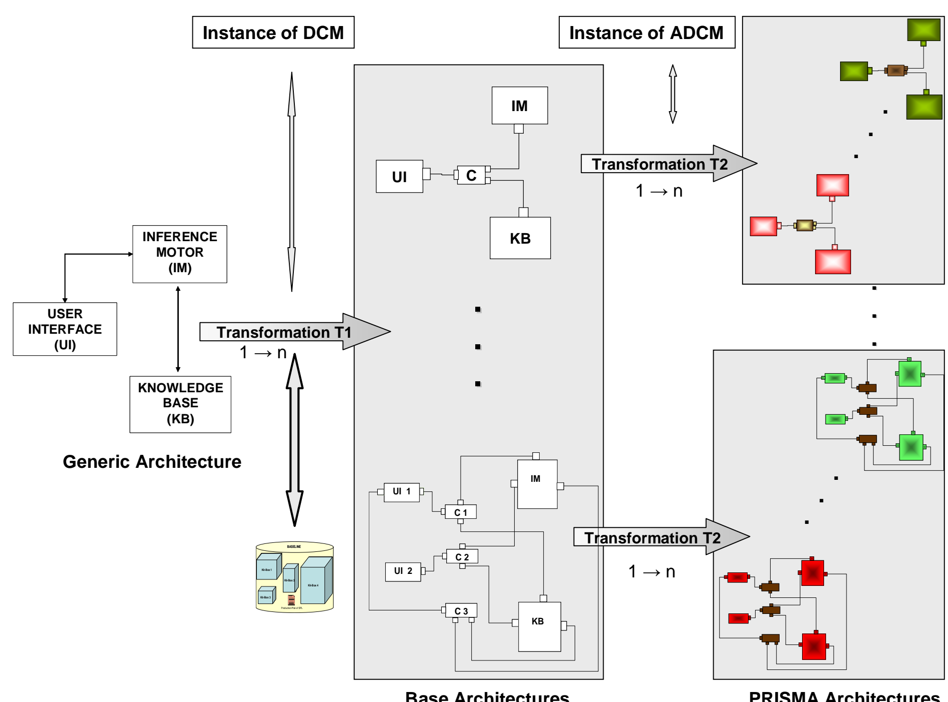
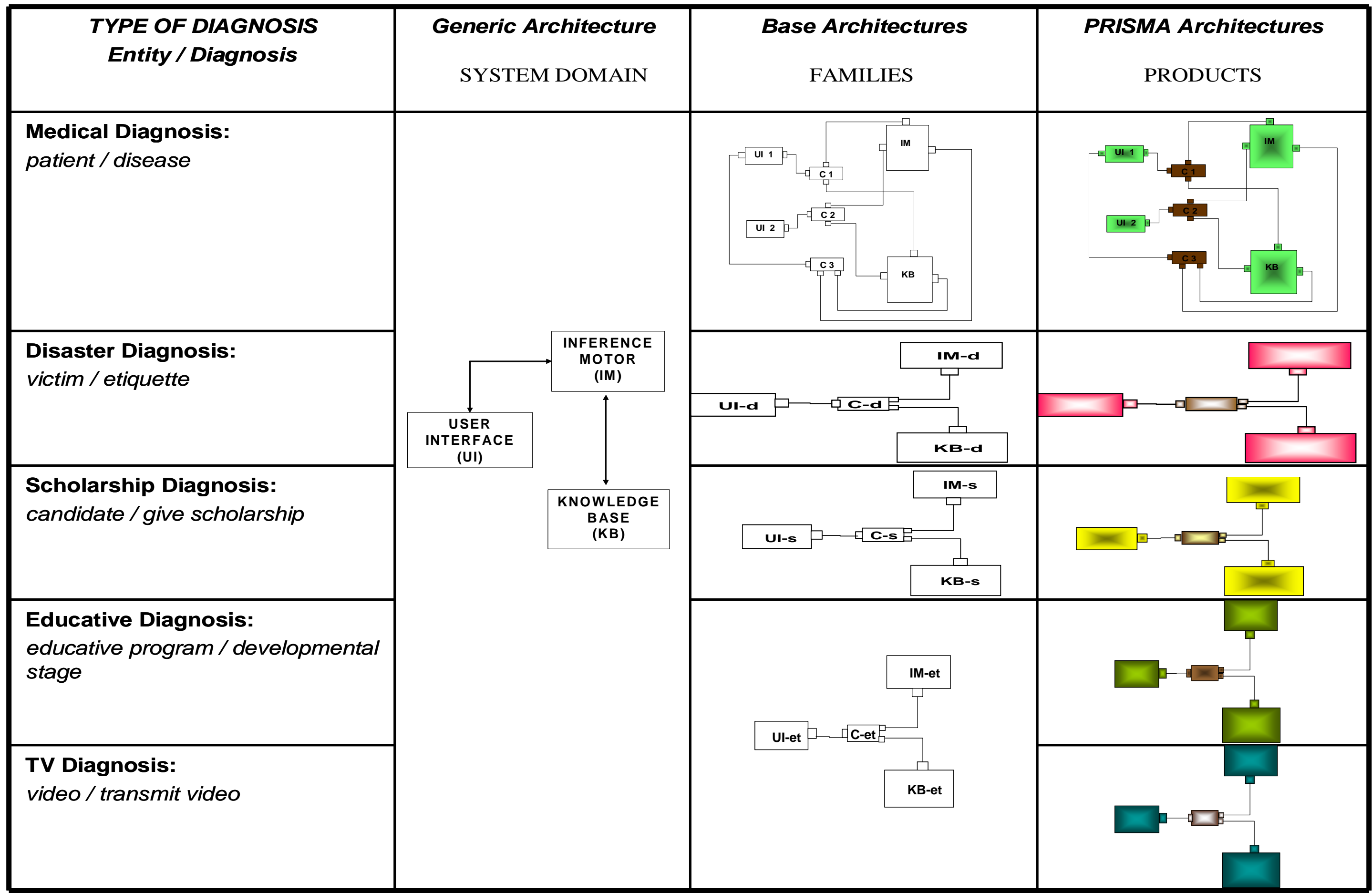


Figura 41 Desarrollo de los SED en BOM

Cabe aquí enfatizar que el modelo de arquitecturas base conforma el metamodelo Esqueleto, los modelos de variabilidad MCD y MCDA conforman al metamodelo de clases de UML, y el modelo de arquitecturas PRISMA conforma el metamodelo PRISMA.

La **tabla 7** muestra una metáfora visual de las arquitecturas (genérica, base y PRISMA) de cada uno de los casos de estudio.



TYPE OF DIAGNOSIS ***Entity / Diagnosis***	***Generic Architecture*** SYSTEM DOMAIN	***Base Architectures*** FAMILIES	***PRISMA Architectures*** PRODUCTS
Medical Diagnosis: *patient / disease*	USER INTERFACE (UI); INFERENCE MOTOR (IM); KNOWLEDGE BASE (KB)	UI 1, UI 2, C 1, C 2, C 3, IM, KB	UI 1, UI 2, C 1, C 2, C 3, IM, KB
Disaster Diagnosis: *victim / etiquette*		IM-d, UI-d, C-d, KB-d	
Scholarship Diagnosis: *candidate / give scholarship*		IM-s, UI-s, C-s, KB-s	
Educative Diagnosis: *educative program / developmental stage*		IM-et, UI-et, C-et, KB-et	
TV Diagnosis: *video / transmit video*			

Tabla 7 Metáfora visual de la variabilidad reflejada en las arquitecturas de los SED de los casos de estudio

El usuario (ingeniero de la aplicación) para obtener un producto de la LPS (i.e. un SED), necesita introducir en BOM los datos de la variabilidad, en dos pasos:

- En un primer paso, el ingeniero introduce las características de la variabilidad del dominio (i.e. V1) por medio de una instancia del MCD. De esta forma, BOM seleccionará una arquitectura base "ad hoc" al caso de estudio.
- En el segundo paso, el ingeniero introduce las caracterísitcas de la variabilidad del dominio de aplicación (i.e. V2) por medio de una instancia del MCDA Con ello, BOM generará una arquitectura PRISMA como un producto de la LPS.

En BOM el entorno destino es el *framework* PRISMA. La compilación del modelo arquitectónico, generando automáticamente el código (en C#,.NET) se realiza a través de la herramienta PRISMA-MODEL-COMPILER. El sistema final como una aplicación ejecutable, una instancia de la LPS, se obtiene a través del Middleware PRISMANET.

Un ejemplo de la interacción de un SED con el usuario final (el cliente) se muestra en la **figura 42**. El ejemplo corresponde al caso de estudio del diagnóstico médico.

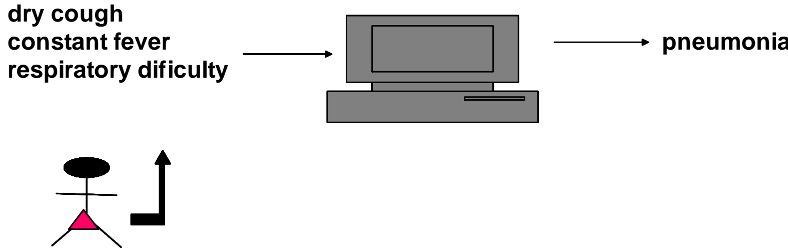


Figura 42 Información de entrada y salida de un SED desde el punto de vista del usuario final

7.5 Conclusiones

En BOM se obtienen ventajas al tratar separadamente los modelos funcional y de variabilidad, permitiendo convertir una arquitectura genérica en una arquitectura base PRISMA, y modelar la variabilidad independientemente. Con ello se realiza la transformación de modelos de una manera expresiva y fácil.

Asimismo, si la LPS es de cardinalidad baja, con la aproximación BOM-EAGER se obtienen ventajas al manejar la variabilidad por medio de técnicas de decisión (i.e. la técnica de árbol de decisión para obtener la arquitectura base) y la técnica FOM para obtener la arquitectura PRISMA como el producto final, en lugar de utilizar

aproximaciones más complejas (como la aproximación BOM-LAZY que utiliza QVT-Relations), como se verá en el próximo capítulo de esta tesis.

La externalización de la Baseline y su acceso mediante técnicas de árbol de decisión aparece como una solución mejor y ha sido la empleada en BOM para la realización de su primer prototipo: ProtoBOM (presentado en el capítulo 10 de esta tesis). El tratamiento de la segunda variabilidad V2 permite también una solución *ad-hoc* mediante técnicas basadas en XML y plantillas XSLT que ha sido igualmente utilizada dentro del marco de FOM a efectos de la decoración de los esqueletos de las arquitecturas base para producir las arquitecturas PRISMA (como instancias de las arquitecturas base).

Sin embargo, aunque la aproximación BOM-EAGER es menos compleja y es una solución adecuada en la transformación de modelos, si la LPS es de cardinalidad alta, es más conveniente utilizar la aproximación BOM-LAZY. Si se cuenta con una LPS muy grande, el árbol de decisión tendrá demasiadas ramas y no sería práctico construir todos los assets de la Baseline (como en la aproximación BOM-EAGER).

CAPÍTULO 8

APROXIMACIÓN BOM-LAZY: DESARROLLO DE LA LPS MEDIANTE TÉCNICAS DE TRANSFORMACIÓN DE MODELOS PARA EL TRATAMIENTO DE LA VARIABILIDAD

Ciencia es todo aquello sobre lo cual siempre cabe discusión.
José Ortega y Gasset (1883-1955) Filósofo español

8.1 Introducción

En este capítulo se realiza la especificación del desarrollo de la LPS a través de BOM, mediante técnicas de transformación de modelos para el tratamiento de la variabilidad.

La estructura de este capítulo es la siguiente: En la sección 2 se definen las vistas de los sistemas expertos en BOM. En la sección 3 se especifican los metamodelos de las vistas software. En la sección 4 se establecen las relaciones entre metamodelos y las transformaciones entre modelos. En la sección 5 se especifican las relaciones entre vistas software. En la sección 6 se establecen las transformaciones entre modelos. En la sección 7 se describe el proceso BOM para la transformación de modelos. La sección 8 presenta las conclusiones de este capítulo.

8.2 Vistas de los sistemas expertos de diagnóstico en BOM

En BOM el tratamiento de la variabilidad del dominio y la variabilidad de la funcionalidad de los sistemas es manejado en modelos independientes. Por ello, en BOM se han considerado dos clases de vistas de los sistemas expertos: la Vista de la Variabilidad del Sistema (VVS) y la Vista Funcional del Sistema (VFS).

La VVS se describe a través de dos modelos conceptuales de variabilidad: el MCD que captura la variabilidad del proceso del diagnóstico y de los requisitos del usuario (V1), y el MCDA que captura la variabilidad del dominio de aplicación (V2). EL MCD conforma al metamodelo de la variabilidad V1 (MM V1) y el MCDA conforma al metamodelo de la variabilidad V2 (MM V2). Ambos metamodelos son el metamodelo de clases de UML 2.0 [UML, 2005], pero otros metamodelos específicos de dominio pueden ser usados, permitiendo de esta forma lenguajes específicos de dominio para capturar la VVS.

La VFS de los sistemas expertos es dada en diferentes etapas del proceso de producción de la LPS, por medio de dos vistas, las cuales se plasman en tres modelos arquitectónicos.

- La vista modular: para el modelo de la arquitectura genérica, el cual conforma al metamodelo modular (MM MODULAR).
- La vista componente-conector (C-C): para los modelos de las arquitecturas base que conforman al metamodelo skeleton (MM SKELETON), y los modelos de las arquitecturas PRISMA que conforman al metamodelo PRISMA (MM PRISMA) para el producto final. El MM SKELETON es similar al MM PRISMA [Pérez, 2006], pero con huecos ("*feature*-holders": donde serán insertadas las características del dominio de aplicación).

Las vistas utilizadas en la VFS son las vistas denominadas por MDA como vistas de arquitecturas software (del inglés Software Architecture Views-SAV). En BOM se han seleccionado la vista modular y la vista componente-conector como las vistas de arquitectura software más apropiadas con la propuesta de esta tesis.

8.3 Metamodelos de las vistas software

En MDA, la OMG propone la Meta Object Facility (MOF) [MOF, 2006] para definir metamodelos. Dado que MOF provee un lenguaje para expresar metamodelos, en esta tesis se ha utilizado la notación gráfica de MOF para la especificación de los metamodelos.

Para la especificación de los metamodelos de las vistas modular y C-C varias propuestas [Bass et al., 2003] [IEEE-1471, 2000] [Krutchten, 1995] han sido analizadas. Sin embargo se han seleccionado la vistas descritas en [Bass et al., 2003] como las aproximaciones más apropiadas a esta tesis.

Especificación del metamodelo de la vista modular (MM MODULAR VIEW). La **figura 43** muestra el metamodelo de la vista modular. El principal elemento

considerado en esta vista es el módulo. Existe distinción ente un módulo simple y un módulo complejo, el cual es usado como un contenedor de módulos. La relación class representa los estilos presentados en [Bass et al., 2003] (*decomposition, uses and layers*), los cuales son tratados como relaciones. El tipo de relación distingue un estilo de otro, y las etiquetas en las asociaciones indican cómo la relación es hecha. Por ejemplo, en la **figura 43** la etiqueta "used" indica que el elemento de Module class será usado por otro elemento de Module class.

Especificación del metamodelo de la vista *C-C*: (MM *C-C* VIEW). El metamodelo de la vista *C-C* se muestra en la **figura 44**. Esta figura muestra Component class y Connector class como los principales elementos; pero ambos se derivan de un component class más genérico: Tcomponent. La manera en que esos elementos se relacionan se corresponde con sus estilos o relaciones *(Pipe-Filter, Client-Server, PeerToPeer, Publish-Suscribe, Shared-Data)* heredados de Relation class. Esta clase une los componentes por medio de la clase conector. Las interacciones entre los componentes de esta vista son refinados usando el metamodelo de clases de UML. Este metamodelo specifica cómo se relacionan los componentes (objetos).

Especificación del metamodelo de la vistas de la variabilidad del dominio (MMV1 y MMV2). El metamodelo de clases de UML [UML, 2005] es usado como ambos metamodelos de la variabilidad V1 y V2.

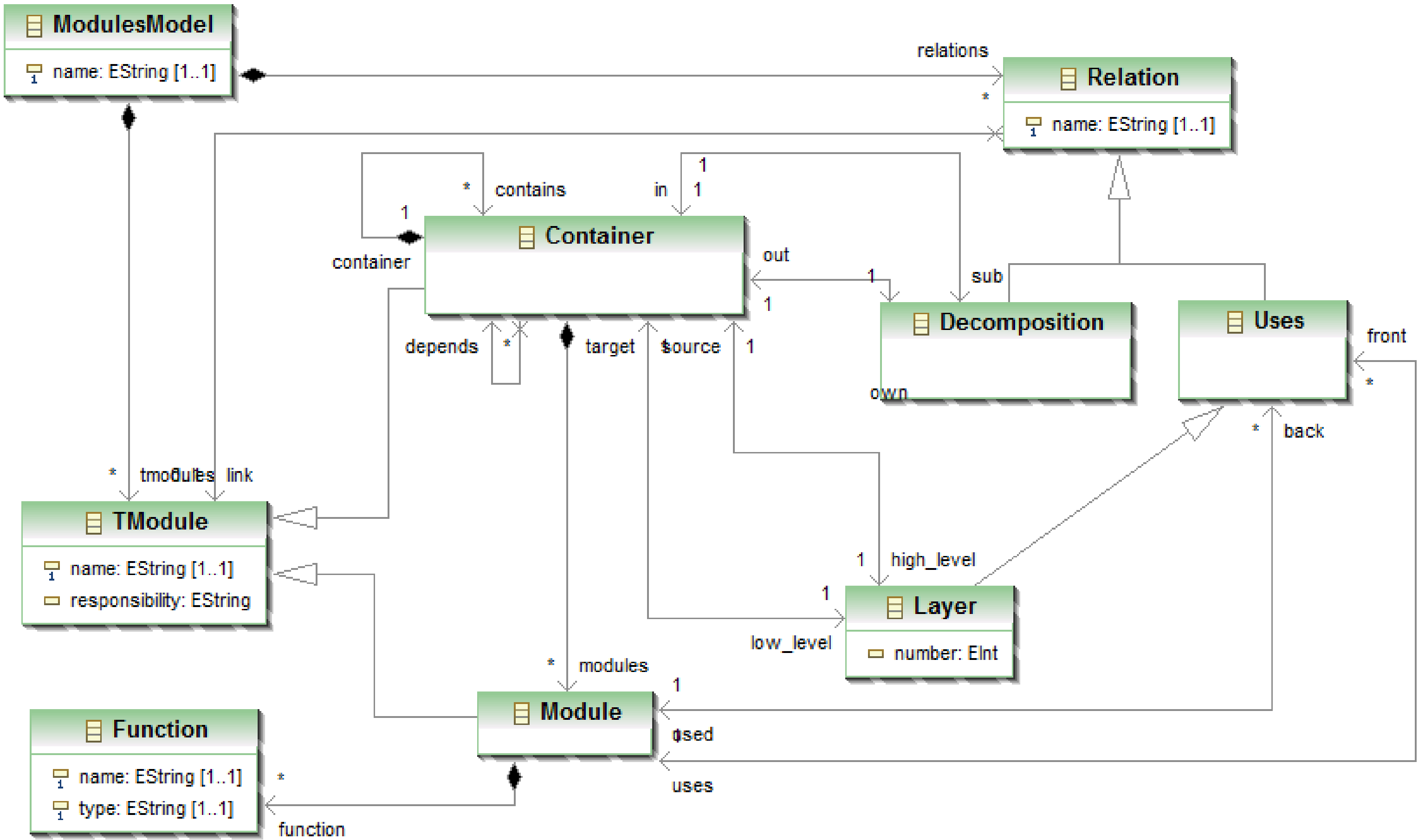


Figura 43 Vista del metamodelo modular (notación MOF)

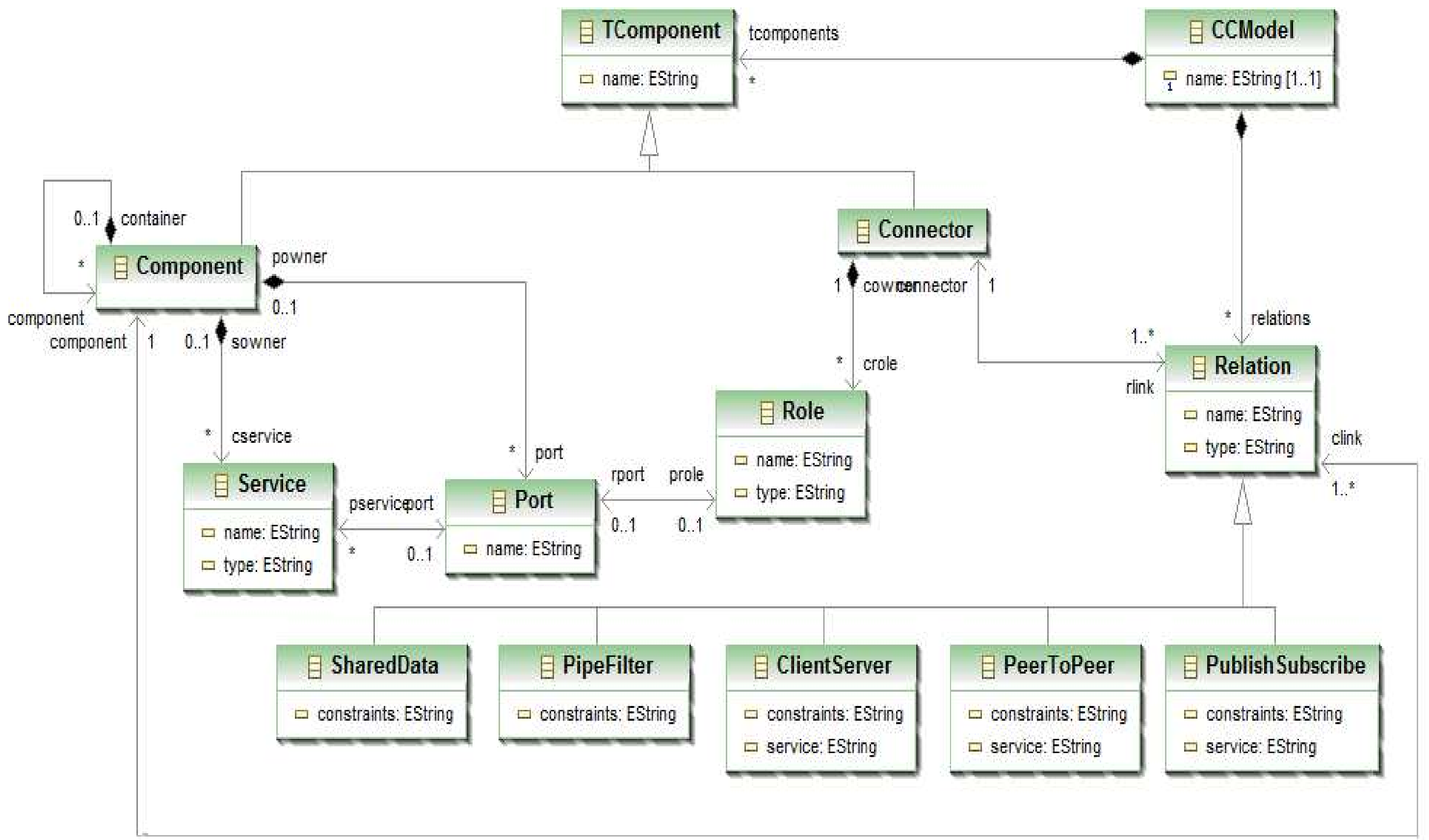


Figura 44 Vista del metamodelo C-C (notación MOF)

8.4 Relaciones entre metamodelos y transformaciones entre modelos

En esta sección se analizarán las relaciones entre las vistas funcionales del sistema (SFV). El objetivo principal de este análisis es establecer las reglas de correspondencia en el nivel de metamodelo y realizar la transformación en el nivel de modelo.

La **figura 45a** muestra un ejemplo de dos relaciones de correspondencia en el nivel de metamodelo. En este caso, R1 indica que cada elemento módulo de la vista modular («Modular View») se corresponde con un elemento componente de la vista C-C («C-C View»), y R2 indica que un «Uses» de «Modular View» se corresponde con un elemento conector de la («C-C View». Estas relaciones son usadas para realizar la transformación a nivel de modelo, como se muestra en la **figura 45b**.

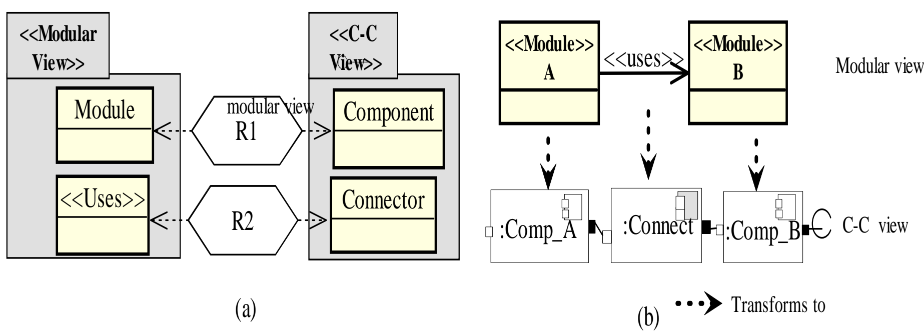


Figura 45 Ejemplo de: (a) Relaciones entre dos vistas en el nivel de metamodelo, (b) Transformación de un modelo en otro modelo en el nivel de modelo [Limón et al., 2007]

Antes de definir las relaciones, debe elegirse un lenguaje para representarlas correctamente. Para el caso específico del MDA, la OMG propone Meta Object Facility (MOF) para expresar metamodelos, y Query/Views/Transformations (QVT) [QVT, 2005] para establecer las relaciones entre esos metamodelos Específicamente, el lenguaje QVT-Relations es usado para describir dichas relaciones.

Inicialmente debemos dar los metamodelos fuente y destino. La correspondencia entre cada elemento de los metamodelos debe ser definida tomando en cuenta la manera en que sus reglas son representadas a través de QVT. En este caso, los metamodelos fuente son el metamodelo modular y el metamodelo de la variabilidad

V1, y el metamodelo destino es el metamodelo esqueleto. Las reglas consideradas en las relaciones de los elementos de los metamodelos modular y de variabilidad V1, son del tipo check only (para verificar los elementos), en su parte izquierda y del tipo enforce (para crear los elementos del metamodelo esqueleto) en su parte derecha.

Nótese que no todos los elementos entre los modelos considerados tienen relaciones de correspondencia. La **tabla 8** muestra algunas de las relaciones de correspondencia identificadas, con su nombre, tipo (de acuerdo a QVT) y los elementos implicados (clases).

Relation Identification $R_1 = R_2 X R_1$	Type	Classes involved by view	
		Modular view	C-C view
moduleToComponet	top	Module	Component
functionToService	-	Module, Function, Type	Component, Service, Port
rUseModToConnector	top	Uses, Module, Function	Connector, Role, Component, Port, Service
rCompositionModToComp	top	Composition, module	Composition component

Tabla 8 Algunas relaciones identificadas en el metamodelo modular y el metamodelo C-C [Limón et al., 2007]

Para el caso de BOM, han sido identificados dos tipos de relaciones entre las vistas funcionales del sistema:

- i) las relaciones entre las vistas modular y C-C,
- ii) las relaciones entre vistas C-C

De las vistas que definen la variabilidad de la LPS, un modelo (MCD) captura la variabilidad del dominio: V1 (p.e. el diagnóstico), y otro modelo (MCDA) captura la variabilidad del dominio de aplicación:V2 (p.e. diagnóstico médico, diagnóstico educativo, etc.).

Las relaciones i) y ii) son creadas en el marco de MDA como se muestra en la **figura 46**. MOF es usado para especificarse así mismo (nivel M3) y para especificar los metamodelos de las vistas y sus interacciones (nivel M2). También las relaciones R1 y R2 se establecen en el nivel M2, donde R1 se relaciona con i), y R2 se relaciona con ii). Sus respectivos perfiles son:

$$R1 \subseteq_{def} \text{MM MODULAR X MM V1} \rightarrow \text{MM SKELETON};$$

$$R2 \subseteq_{def} \text{MM SKELETON X MM V2} \rightarrow \text{MM PRISMA};$$

Un nivel más bajo (nivel M1) es donde las transformaciones T1 y T2 son aplicadas. Con la transformación T1, un primer modelo es obtenido: el modelo esqueleto (correspondiente al modelo de la arquitectura base), usando como fuente la vista del modelo modular (correspondiente al modelo de la arquitectura genérica) y el modelo conceptual de la variabilidad del dominio (MCD):V1. Esta primera versión es refinada obteniendo la vista *C-C* (correspondiente al modelo de la arquitectura PRISMA) a través de la transformación T2, usando para ello el modelo conceptual de la variabilidad del dominio de aplicación (MCDA):V2.

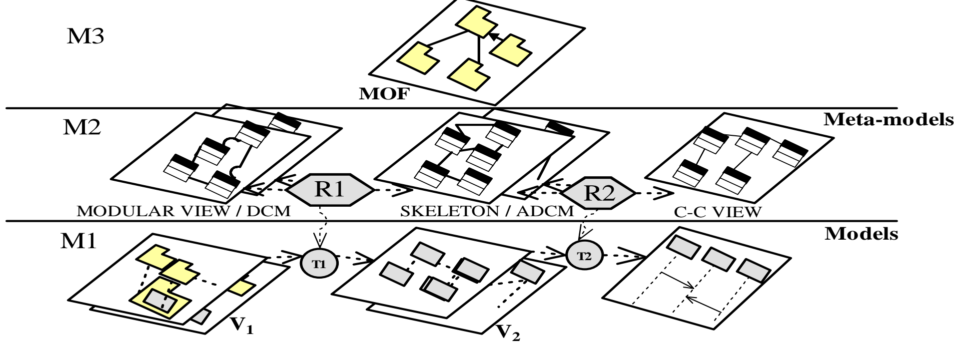


Figura 46 Relaciones entre metamodelos y transformaciones entre modelos, en el escenario de LPS [Limón et al., 2007]

8.5 Relaciones entre vistas software

En esta sección, las relaciones entre las vistas software mostradas en la **tabla 8** son especificadas en QVT-Relations. El código y los programas son mostrados en la **figura 47**.

Con el fin de ganar claridad, en esta sección será omitido el prefijo "esqueleto", i.e. "componente esqueleto" será mencionado como "componente" y el "conector esqueleto" será mencionado como "conector".

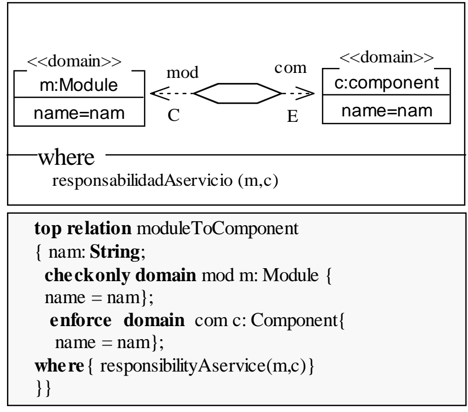
Relación *moduleToComponent*. Esta relación transforma cada uno de los módulos en un componente. En la figura 40a se muestran dos tipos de prefijos: "*checkonly*" y

"*enforce*". El objeto del módulo dominio es del tipo "*checkonly*", y en contraste, el objeto del componente dominio es del tipo "*enforce*". Esto crea un objeto de la clase componente que es relacionado con la clase módulo. La cláusula "*where*" indica una llamada a la relación "*functionToService*", la cual relaciona un objeto de la clase función con un objeto de la clase servicio. El diagrama y el correspondiente código son mostrados en la **figura 47a**.

Relación *functionToService*. Esta relación implica que una función de la clase módulo generará un servicio de la clase componente.. En la cláusula "*where*", se obtiene el nombre del puerto, llamando a la función "*typePort*". Cuando la relación es ejecutada, serán creadas sólo las clases que están en el metamodelo fuente (la clase "*ServiceToPort*"). El diagrama y el correspondiente código son mostrados en la **figura 47b**.

Relación *rUseModToConnector*. La relación 'uses' se transforma de la metaclase modular a la metaclase conector para realizar la unión entre un conector y dos componentes en la metaclase *C-C*, como lo muestra la **figura 47c**. La cardinalidad de objetos de esta relación es 1 a n, porque una relación de dos componentes se establece a través de un conector. La **figura 47c** muestra parte del código de cómo las relaciones son invocadas por la cláusula "*where*" para crear las relaciones entre módulo, componente y "*functionToService*".

Relación *rCompositionModToComp*. El conjunto de módulos también generará un conjunto de componentes. Por lo tanto, cuando un componente (contenedor) es creado, un subordinado es creado dentro de él. Esto se muestra en la **figura 47d.**



(a)

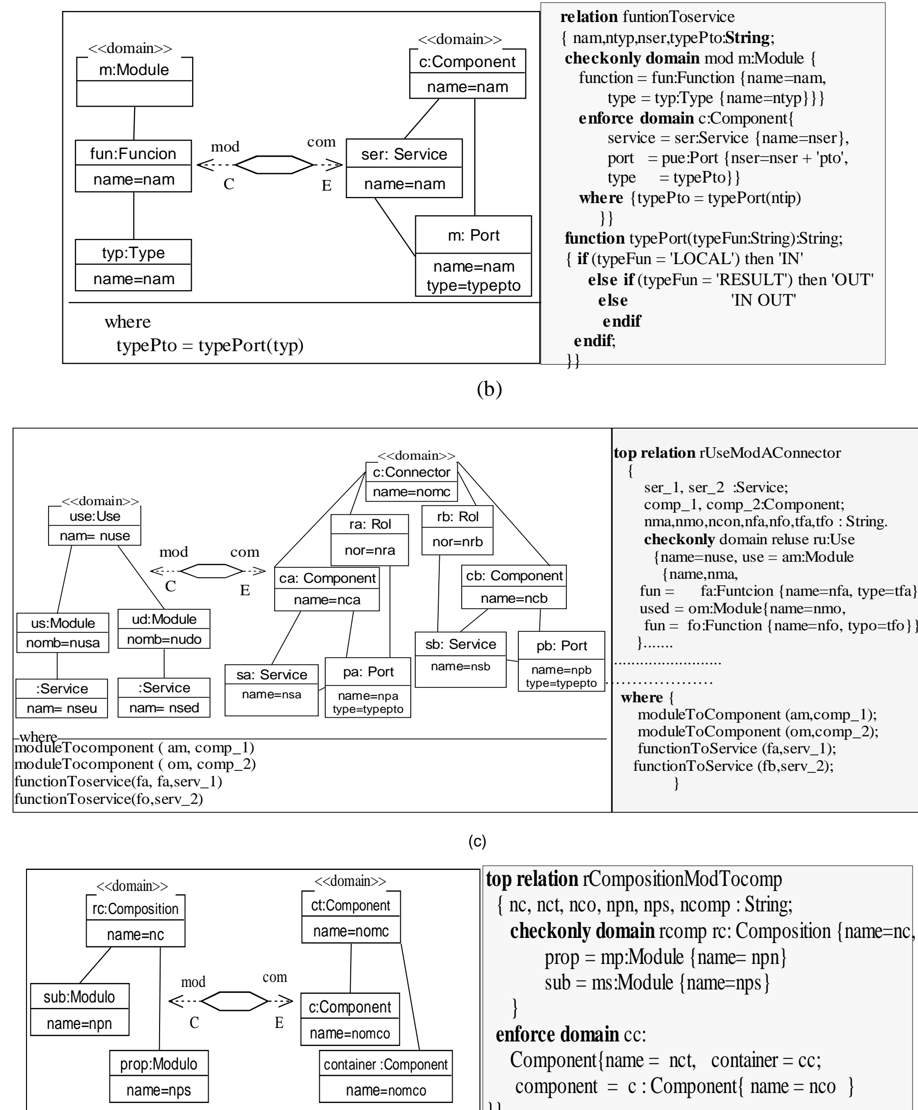


Figura 47 Diagramas y códigos de las relaciones: (a) moduleToComponent, (b) functionToService, (c) rUseModToConnector, (d) rCompositionModToComp [Limón et al., 2007]

8.6 Transformaciones entre modelos

La **figura 48** muestra las transformaciones involucradas en la construcción de las arquitecturas de la LPS. Esta figura ilustra cómo las transformaciones se realizan a nivel de modelo y son aplicadas en el proceso de transformación. Las transformaciones T1 y T2 son ejecutadas a nivel de modelo (nivel M1 en MOF), y son definidas en el nivel de metamodelo (nivel M2 en MOF).

En la transformación T1 se obtiene un primer modelo (modelo esqueleto) usando como fuentes al modelo modular y al modelo MCD. En la transformación T2 usando como fuentes al modelo esqueleto y al modelo MCDA, se obtiene un segundo modelo (modelo PRISMA) como un refinamiento.

Como un paso previo a la transformación T1, el modelo arquitectónico modular debe ser diseñado. Este modelo está constituido por tres módulos: *InferenceMotor, KnowledgeBase* y *UserInterface*. Dichos módulos son unidos entre sí a través de las relaciones de dependencia. A continuación, la transformación T1 es ejecutada produciendo el modelo esqueleto (modelo destino), utilizando como fuentes al modelo MCD y el modelo modular. Como muestran las **figuras 48 y 49**, un componente esqueleto es producido por cada módulo, y cada relación de dependencia genera un conector esqueleto. Las relaciones *moduleToComponent* y *rUseModToConnector* son las reglas aplicadas para generar el modelo esqueleto (usando los metamodelos como plantillas).

Las transformaciones T1 y T2 de modelos son ejecutados por medio de transformaciones QVT-Relations. En la transformación T1, las QVT-Relations deben tomar en cuenta la arquitectura genérica, la instancia del MCD y la configuración de la arquitecutura base. En la transformación T2, las QVT-Relations deben tomar en cuenta la configuración de la arquitectura base, la instancia del MCDA y la configuración de la arquitectura PRISMA.

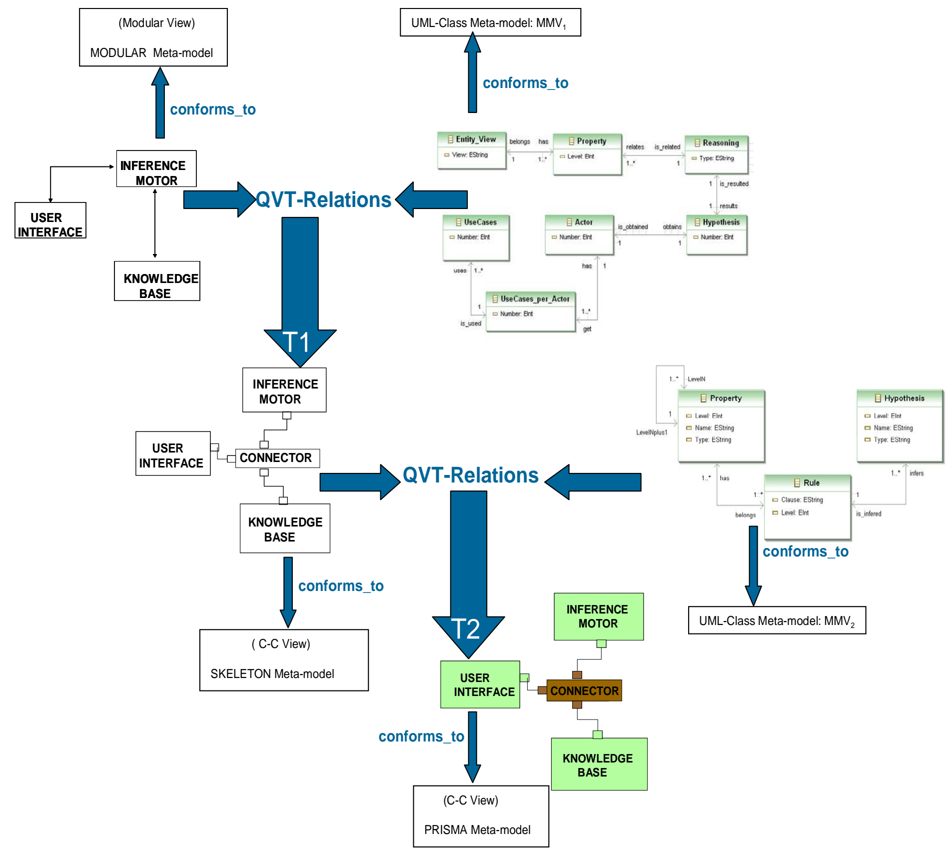


Figura 48 Las transformaciones T1 y T2 de modelos

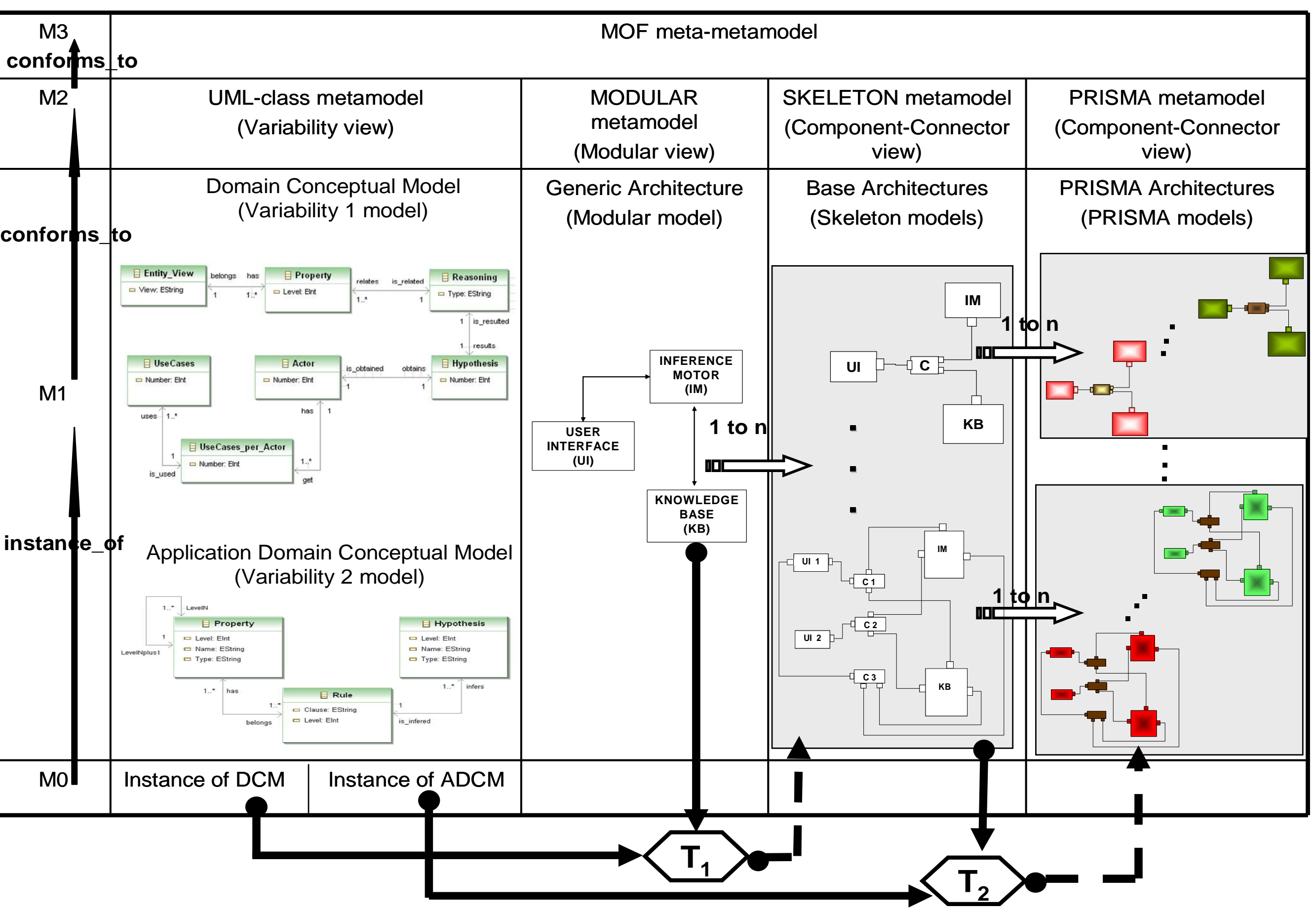


Figura 49 Las transformaciones T1 y T2 de modelos (expresados en niveles de MOF)

8.7 El proceso BOM de transformación de modelos

En esta sección se presenta un proceso que permite el diseño de los sistemas expertos. Las tareas y elementos involucrados en este proceso son mostradas en la **figura 50**.

Las especificaciones de los metamodelo de la vista modular, *C-C*, y del diagrama de clases de UML, son los primeros elementos que son requeridos para inicializar este proceso. A continuación, las relaciones de correspondencia entre la vista modular y la vista *C-C*, y el diagrama de clase de UML, son identificadas y especificadas por medio del lenguaje QVT-Relations. Todas estas tareas son ejecutadas solamente una vez. La tarea de diseñar los modelos, puede ser realizada tantas veces como sea requerida.

La vista del modelo modular, el MCD y las relaciones establecidas entre las vistas del metamodelo de vista modular y el metamodelo de vista *C-C* son usadas como fuentes en la transformación T1, la cual produce la vista del modelo esqueleto. A continuación, una segunda transformación T2 es aplicada para refinar al modelo esqueleto, tomando como fuentes el MCDA y dicho modelo para producir la vista del modelo PRISMA.

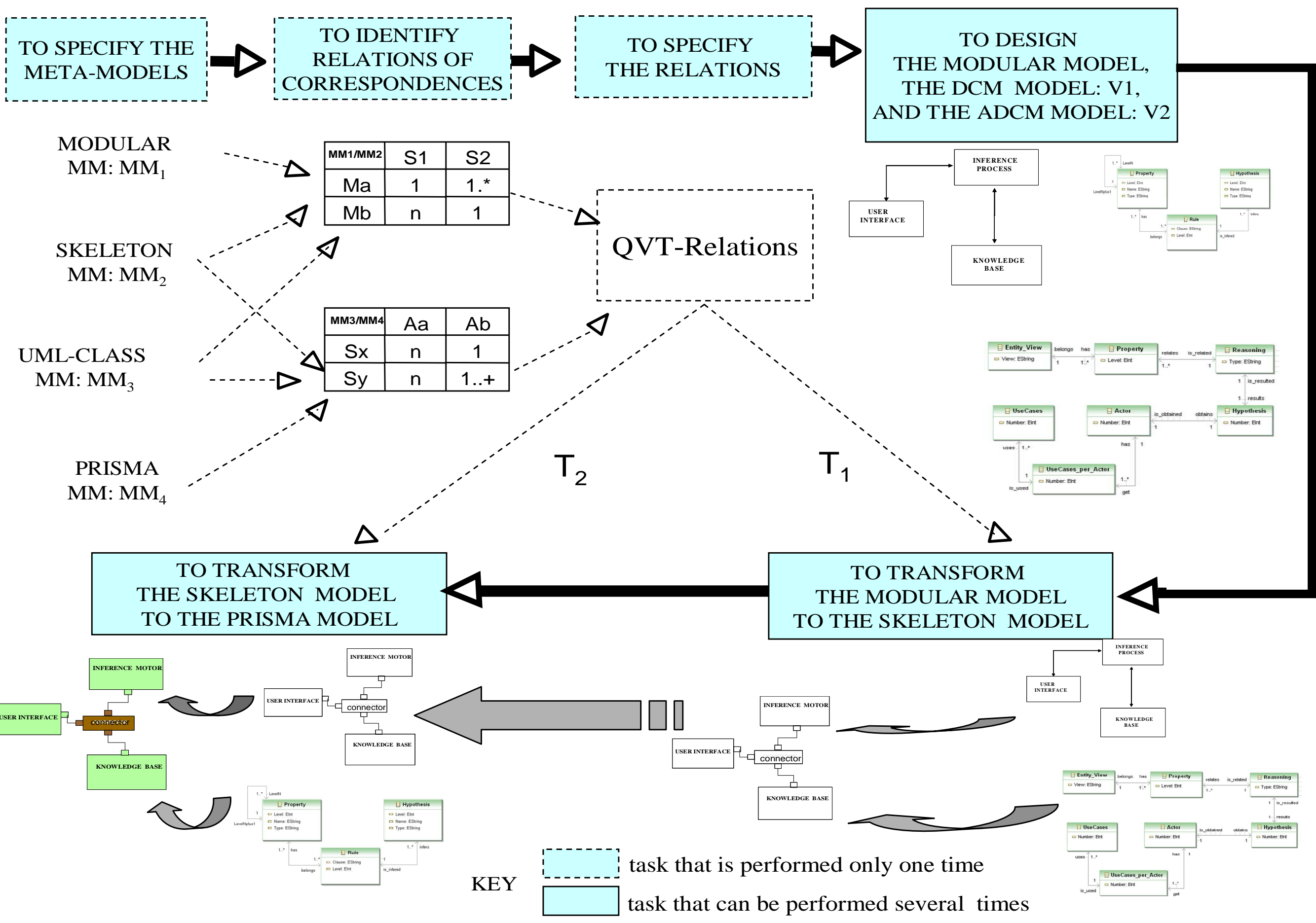


Figura 50 Tareas y elementos involucrados en el proceso de transformación de los modelos (figura adaptada de [Limón et al., 2007])

8.8 Conclusiones

La especificación presentada en este capítulo, para el desarrollo de la LPS por medio de QVT-Relations como técnica de transformación de modelos para el tratamiento de la variabilidad, muestra que esta aproximación (BOM-LAZY) realiza el proceso de gestión de la variabilidad en el dominio de los sistemas software de manera más compleja que la aproximación BOM-EAGER.

El número de reglas QVT se multiplica: una al menos por cada camino que une la raiz con una hoja del árbol de decisión y por cada artefacto de la vista modular, a efectos de considerar la variabilidad de las arquitecturas base al tratar la variabilidad V1. Así mismo por cada característica o conjunto de características de la aplicación es necesario introducir una regla QVT. La idea que subyace en el fondo es que la Baseline aparece plasmada en el código de las QVT-Relations complicando innecesariamente el proceso.

El uso de transformaciones con QVT-Relations debe tomar en cuenta al modelo de la variabilidad y al modelo arquitectónico. En la 1ª transformación, a la arquitectura genérica y al modelo conceptual de la variabilidad del dominio, y en la 2ª transformación a las arquitecturas base y al modelo conceptual de la variabilidad del dominio de aplicación, lo cual implica una solución computacional y algorítmica, obteniendo reglas de transformación complejas.

Una solución *ad-hoc* a la complejidad del manejo de la variabilidad con QVT, fue presentada en el capítulo anterior, a través de las técnicas utilizadas en la aproximación BOM-EAGER.

Sin embargo, si la LPS es de cardinalidad alta, resulta más conveniente aplicar la aproximación BOM-LAZY presentada en este capítulo, dado que el problema de la complejidad (con el uso de QVT-Relations) sería menor que el construir todos y cada uno de los assets contenidos en la Baseline (como la aproximación BOM-EAGER), así como el que se incrementaría notablemente el número de ramas del árbol de decisión para poder acceder a la gran cantidad de productos de la LPS que se desarrolla.

CAPÍTULO 9

MODELADO DEL PROCESO DE DESARROLLO DE LA LPS EN BOM

En el punto donde se detiene la ciencia, empieza la imaginación.
Jules de Gaultier (1858-1942) Filósofo francés.

9.1 Introducción

Este capítulo presenta detalladamente el modelado del desarrollo de la LPS, clasificándolo en dos partes: la ingeniería del dominio y la ingeniería de la aplicación. Dichos procesos son modelados haciendo uso de dos estándares de la OMG: Reusable Asset Specification [RAS, 2005] para definir como los *assets* son manipulados y reutilizados; y Software Process Engineering Metamodel [SPEM, 2006] que es un lenguaje de modelado de procesos de desarrollo de software.

La estructura del capítulo es la siguiente: En la sección 2 se describe el modelado del proceso de creación de la LPS de diagnóstico. En la sección 3 se describe el modelado de la Baseline de la LPS en la etapa de la ingeniería del dominio. En la sección 4 se describe el Plan de Producción de la LPS en la etapa de la ingeniería de la aplicación. En la sección 5 se presentan las conclusiones de este capítulo.

9.2 Modelado del proceso del desarrollo de software para la creación de la LPS

El modelado del proceso de creación de la LPS de diagnóstico, define el conjunto de recursos o activos principales (*core assets*) para crear una familia de productos software, basado en dos estándares de la OMG: la especificación de recursos reutilizables (del inglés Reusable Asset Specification-RAS) para definir de manera estándar toda la información asociada a un activo o recurso reutilizable, a través de la cual es posible manipularlo y reutilizarlo, y el metamodelo de ingeniería de procesos software (del inglés Software Process Engineering Metamodel-SPEM)

como lenguaje que modela procesos de desarrollo de software usando una terminología común y estándar.

SPEM permite el modelado de muchos aspectos y problemas del proceso de desarrollo. Aunque el metamodelo de SPEM es muy extenso, el trabajo de la tesis se ha centrado en el modelado de tareas, utilizando relaciones de secuencia pero sin prioridad entre las mismas. Las tareas, ejecutadas por roles, consumen artefactos de entrada y producen artefactos de salida. Una tarea puede tener asociada elementos guía que ayuden al rol a desempeñarla.

La **tabla 9** presenta los iconos estándar de SPEM utilizados en esta tesis, así como su significado.

ICONO	SIGNIFICADO
	Actividad
	Rol del ingeniero
	Guía
	Proceso
	Producto de trabajo
	Modelo en UML (especialización de producto de trabajo)
	Documento (especialización de producto de trabajo)

Tabla 9 Iconos estándar de SPEM utilizados en el modelado de la LPS

Asimismo, en la **tabla 10** se presentan los iconos SPEM utilizados *exprofeso* en esta tesis:

ICONO	SIGNIFICADO
	esqueleto (aspecto)
	aspecto tipo PRISMA
	interfaz tipo PRISMA
	componente tipo PRISMA
	conector tipo PRISMA
	esqueleto aspecto-proceso inserción de características
	híbrido empaquetado
MI 3 U 3 C 3 BC 3	modelo arquitectóinico tipo PRISMA
	configuración del modelo arquitectónico
	caja (kit-box)

	baseline

Tabla 10 Iconos de SPEM creados *exprofeso* para el modelado de la LPS

Existen algunas herramientas para el modelado en SPEM, por ejemplo la que incorpora Visual Studio, pero estas herramientas que soportan SPEM no son ejecutables sino gráficas, por lo que la expresividad de los iconos utilizados pueden ser realizados con otro tipo de herramientas. En el caso de esta tesis, a lo largo de la investigación y publicación de resultados, se realizaron en power point, varios iconos (como especializaciones del icono producto del trabajo) elaborados ex profeso para dibujar los "assets", la Baseline y el Plan de Producción. Por ello, se aprovechó el esfuerzo y tiempo invertidos, y dado que la expresividad en power point de la vista final es la misma, se decidió continuar, hasta el final de esa manera. Además, fue utilizado dicho trabajo para insertar animación en las presentaciones de los congresos en los que fue presentada la tesis.

Tomando como referencia la aproximación para producir LPS de Clements et al. [Clements et al., 2001] en la que se clasifican los procesos de desarrollo de las LPS, se describen a continuación las tareas realizadas en dichos procesos. Cabe señalar que en este trabajo se ha introducido un cambio importante que constituye una de las aportaciones.de la tesis: la Baseline contiene, como un activo más, el plan de producción de la LPS. De esta forma se incrementa notablemente la automatización de la producción de un producto de la LPS. Por ello, se fusionarán las tareas de desarrollo de los componentes básicos reutilizables y del plan de producción en una única tarea (desarrollo de componentes básicos reutilizables).

1).-Ingeniería del dominio

i) análisis del dominio

ii) desarrollo de los componentes básicos reutilizables
tareas:

- crear modelo de características
- crear árbol de decisión
- crear procesos de selección de activos
- crear modelo conceptual del dominio
- crear modelo conceptual del dominio de aplicación
- crear esqueletos
- crear artefactos tipo PRISMA

- crear proceso principal de inserción de características del dominio de aplicación
- crear procesos de inserción de las características del dominio de aplicación
- crear esqueletos aspecto-proceso
- crear híbridos empaquetados
- crear configuración del modelo arquitectónico
- crear modelos RAS de activos
- crear cajas (o kit-boxes)
- crear el Plan de Producción de la LPS
- crear la Baseline

2.- Ingeniería de la aplicación
i) caracterización del producto
tareas:

- obtener características del dominio
- seleccionar activos
- desempaquetar caja
- obtener características del dominio de aplicación
- crear aspectos tipo PRISMA

ii) síntesis del producto
tareas:

- construir especificación PRISMA

iii) construcción del producto
tareas:

- compilar modelo arquitectónico (generar código)
- crear sistema ejecutable

De manera general se puede concluir que el ingeniero de dominio, es quien crea el plan de producción y los artefactos software que son utilizados para llevar a cabo sus tareas; y el ingeniero de aplicación, es quien ejecuta el plan de producción de la LPS.

Comparando las actividades que deberán realizar los analistas de BOM, en sus roles de ingeniero de dominio y/o ingeniero de aplicación, se observa que el trabajo del ingeniero de dominio es más complejo, al tener que diseñar y modelar con detalle modelos, recursos reutilizables y procesos adecuados para generar cada uno de los productos de la línea, así como especificar al ingeniero de aplicación como obtener una solución concreta a través de lenguajes específicos de dominio (los Modelos Conceptuales de la variabilidad).

En cambio el trabajo del ingeniero de aplicación se vuelve mucho más sencillo, trabajando a mayor nivel de abstracción sin preocuparse de las tareas de configuración de los componentes básicos reutilizables, dirigiendo sus esfuerzos en la especificación de las características deseadas para el producto a generar. El ingeniero de la aplicación sólo debe de realizar la tarea de configuración; éste será el ejecutor del Plan de Producción, que previamente había sido diseñado por el ingeniero de dominio. Con esta aproximación el ingeniero de la aplicación es obligado a seguir el plan predefinido por el ingeniero de dominio.

9.3 Ingeniería del dominio: creación de la Baseline

En esta primera fase se crean todos los activos y se describe el proceso de configuración de activos así como el plan de producción de la familia de productos. Pero una LPS requiere almacenar sus activos en un repositorio. En esta tesis, todos los activos son depositados en la Baseline, por ello se presenta toda la fase de la ingeniería del dominio como la fase donde se construye la Baseline.

La Baseline es un repositorio especializado que almacena activos de software y facilita su recuperación y mantenimiento. Su objetivo es asegurar la disponibilidad de los activos para apoyar el desarrollo de productos de la LPS.

En la Baseline se plasman tanto los activos como el conocimiento necesario, para que BOM operacionalice lo almacenado con el fin de obtener el producto final, lo cual incrementa la automatización. Es importante señalar que tanto para crear la Baseline como para explotarla, se utilizan modelos. En la Baseline se materializa la variabilidad del dominio, de forma que la Baseline es en sí una LPS.

Es importante aclarar que el ingeniero del dominio no aborda la creación directa de la LPS de productos (esa actividad la desempeña el ingeniero de aplicación), sino la generación de familias de activos. A medida que el ingeniero del dominio va encontrando similitudes (a través de los puntos de variabilidad) entre los activos que va creando en los diferentes dominios específicos, formará familias de activos organizándolos en forma de LPS. De esta forma el ingeniero del dominio creará una LPS para generar la familia de productos.

Cuando el ingeniero del dominio desee crear un activo, inspeccionará la Baseline en busca de uno que cubra sus requerimientos. Si ninguno de los miembros de la familia satisface sus necesidades, deberá crear un activo por su cuenta, a partir del miembro que más se adecué a ellas. Cuando termine la tarea de creación, deberá realizar la ingeniería del dominio para incluir la nueva variante en la familia de productos y almacenarla así en la Baseline. A partir de este momento, el proceso de configuración del activo permitirá generar la nueva variante de activos, para ser

aplicada siempre que se repitan las condiciones que originaron su aparición. Es importante considerar que el ingeniero del dominio, deberá tener un buen conocimiento del dominio, con el fin de seguir patrones y establecer similitudes.

A continuación se listan todos los artefactos software creados por el analista que desempeña el rol de ingeniero del dominio, con el fin de obtener la Baseline como un artefacto más. Se muestran en SPEM las tareas que implican la creación de la Baseline y los activos implicados. Asimismo se comenta brevemente su uso en la ingeniería de la aplicación.

Las tareas que estan presentes en la primera fase de la ingeniería del dominio son realizadas en dos partes: análisis del dominio, y desarrollo de los componentes básicos reutilizables y del plan de producción.

9.3.1 Análisis del dominio

En el análisis del dominio se estudia la variabilidad del dominio (en este caso el del diagnóstico), en términos de sus características. [Arango y Prieto-Díaz, 1991] mencionan que el análisis del dominio es donde se adquiriere y se modela el conocimiento sobre el dominio, realizando una búsqueda de elementos comunes y diferencias en la familia de sistemas que lo conforman. Esta etapa es muy importante para desarrollar una línea de productos, ya que los dominios son analizados y la información sobre éstos capturada y organizada en modelos del dominio y en elementos software reutilizables ("*assets*"). Cabe mencionar que el análisis del dominio ya fue realizado en los capítulos 4 y 5 de esta tesis.

9.3.2 Desarrollo de los componentes básicos reutilizables

Durante el desarrollo de los componentes básicos reutilizables (*assets*), se conciben, diseñan e implementan los *assets*. Esto no sólo involucra el desarrollo de la funcionalidad del dominio, sino también define cómo los *assets* deben ser extendidos. Los *assets* correspondientes a esta etapa son:

1.- **El Modelo de Características**.- El modelo de características identifica la LPS en términos de la variabilidad del dominio. Las características de este modelo son plasmadas en un árbol de decisión (ver asset 2).

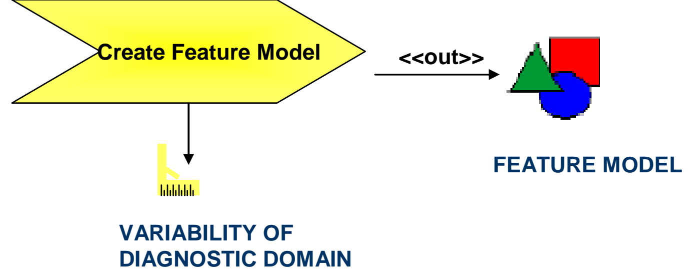


Figura 51 Creación del Modelo de Características (notación SPEM)

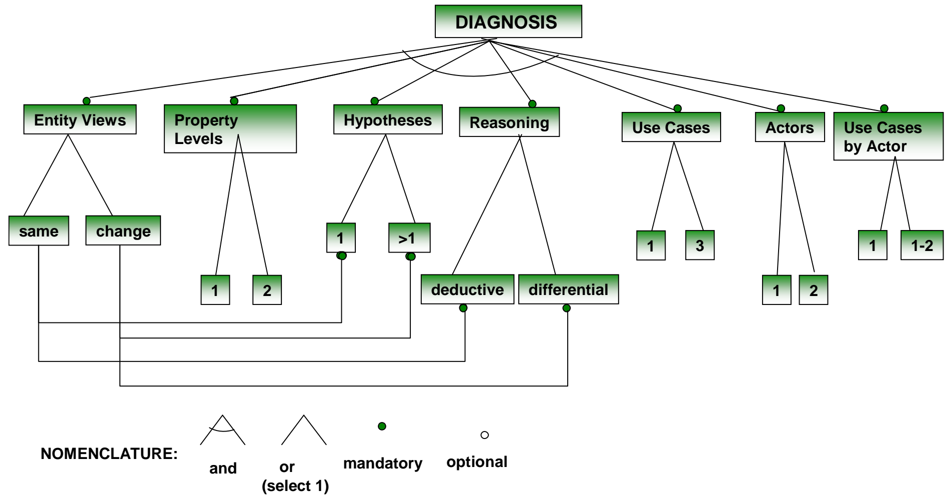


Figura 52 Modelo de Características

2.- **El Árbol de Decisión**.- Las características observadas en el modelo de características son plasmadas en el árbol de decisión. Los puntos de variabilidad que conforman la primera variabilidad están representados en los nodos del árbol de decisión, y las hojas del árbol representan las familias de *assets* de la LPS. Las rutas del árbol de decisión son plasmadas en el proceso de selección de *assets* (ver asset 3).

Figura 53 Creación del Árbol de Decisión (notación SPEM)

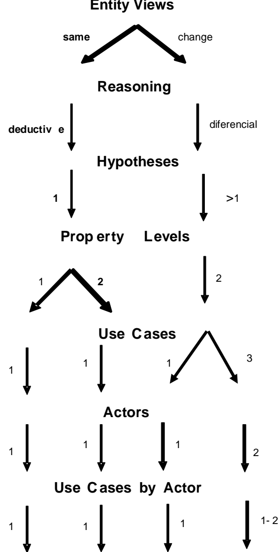


Figura 54 Árbol de Decisión

Los puntos de variabilidad (VP) plasmados en el árbol de decisión son representados mediante sus variantes:

Entity Views = {same, change}	VP1 = {V11,V12}
Reasoning={deductive, differential}	VP2 = {V21,V22}
Hypotheses = {1, >1}	VP3 = {V31, V32,}
Property Levels = {1,2}	VP4 = {V41, V42, V43}
Use Cases = {1,3}	VP5={V51,V52,V53,V54,V45}
Actors = {1,2}	VP6 = {V61, V62, V63, V64}
Use Cases per Actor = {1, 1-2}	VP7 = {V71,V72, V73, V74}

Tabla 11 Puntos de variabilidad y sus variantes

3.- **El Proceso de Selección de *Assets***.- Para crear este proceso, el ingeniero del dominio se apoya con el árbol de decisión y la información del contenido de la Baseline. Este proceso computa las trayectorias del árbol de decisión para seleccionar *assets*, de acuerdo con los puntos de variabilidad seleccionados del

dominio. Cada hoja del árbol representa la familia de *assets* seleccionada y está contenida en una caja (Kit Box).

La información del contenido de la Baseline está representada por: Baseline = {caja1, caja2, caja3, caja4, Plan de Producción}, donde caja i corresponde a la familia i de *assets* (empaquetados en una caja)

Este proceso es ejecutado por BOM, en la fase de la ingeniería de la aplicación, cuando el ingeniero configura el MCD, aportando al sistema la información del dominio específico (i.e. las variantes de los puntos de variabilidad del dominio).

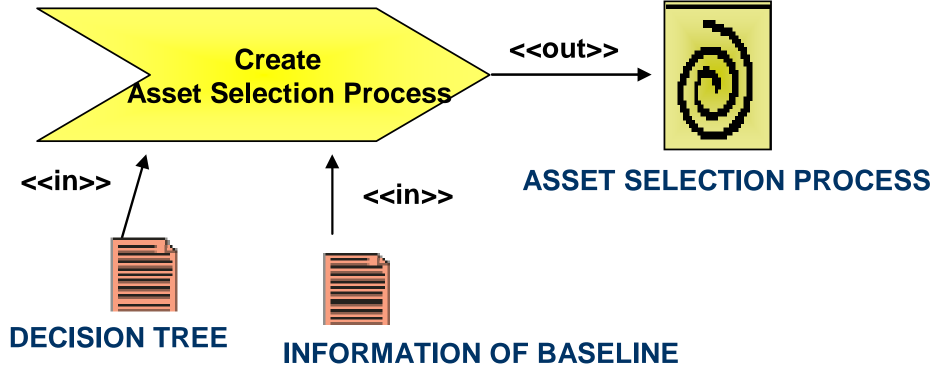


Figura 55 Creación del Proceso de selección de *assets* (notación SPEM)

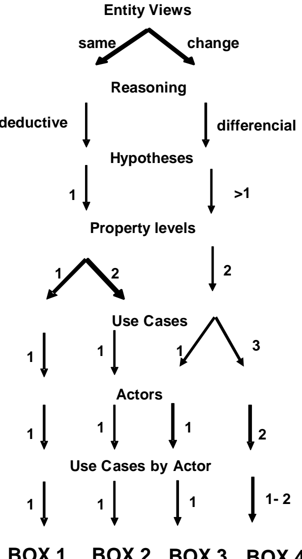


Figura 56 Proceso de selección de *assets*

El proceso de selección de *assets* está representado por las trayectorias que recorren el árbol de decisión, conforme a las variantes de los puntos de variabilidad elegidos por el ingeniero de la aplicación. Cada trayectoria sigue un camino desde la raíz del árbol de decisión hasta sus hojas que apuntan a la caja seleccionada. En esta investigación se contemplan las siguientes trayectorias:

Path 1= [V11,V21,V31,V41,V51,V61,V71]⇒ Kit-Box1
Path 2= [V11,V21,V31,V42,V52,V62,V72]⇒ Kit-Box2
Path 3= [V12,V22,V32,V43,V53,V63,V73]⇒ Kit-Box3
Path 4= [V12,V22,V32,V43,V54,V64,V74]⇒ Kit-Box4

Tabla 12 Trayectorias del árbol de decisión

4.- **El Modelo Conceptual de Dominio (MCD)**.- las características como puntos de variabilidad presentes en el modelo de características y el árbol de decisión, están presentes en el modelo conceptual del dominio, el cual captura la variabilidad del dominio. En la ingeniería de la aplicación, este artefacto es usado cuando el ingeniero introduce en el sistema y como una instancia de dicho modelo las variantes (la información del dominio específico) a través de BOM, con el fin de seleccionar las cajas contenidas en la Baseline.

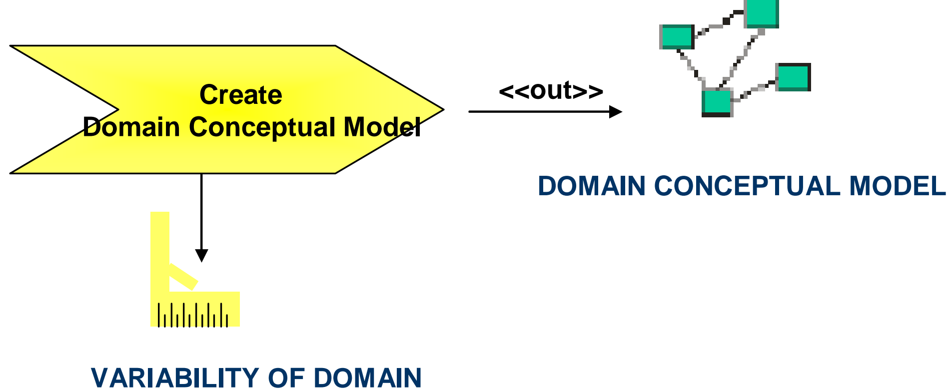


Figura 57 Creación del Modelo Conceptual de Dominio (notación SPEM)

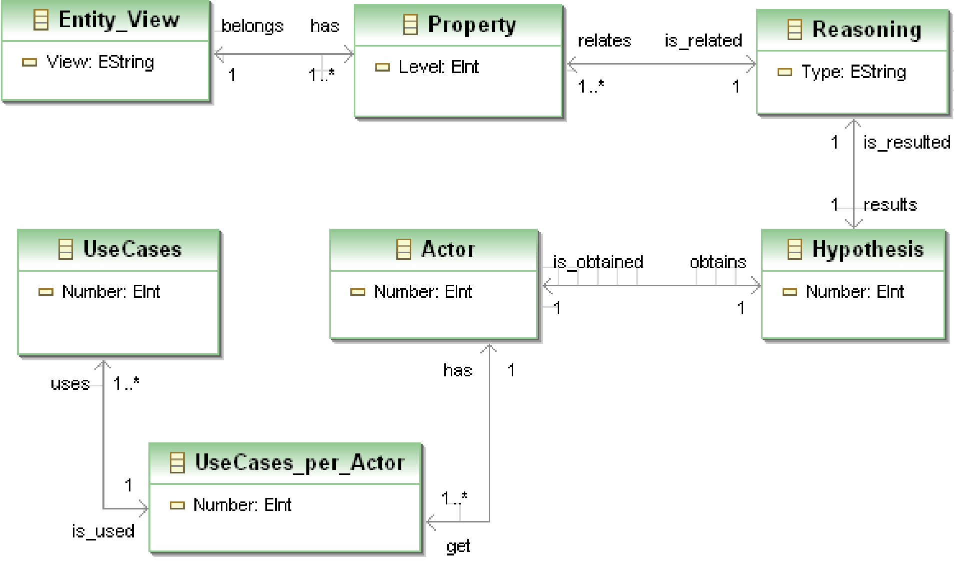


Figura 58 Modelo Conceptual del Dominio del Diagnóstico

5.- **El Modelo Conceptual del Dominio de Aplicación (MCDA)**.- Este modelo captura la variabilidad del dominio de aplicación. Con este artefacto software, BOM obtiene del ingeniero de la aplicación, las características de un dominio de aplicación específico, con el fin de decorar las arquitecturas base de ese dominio de aplicación.

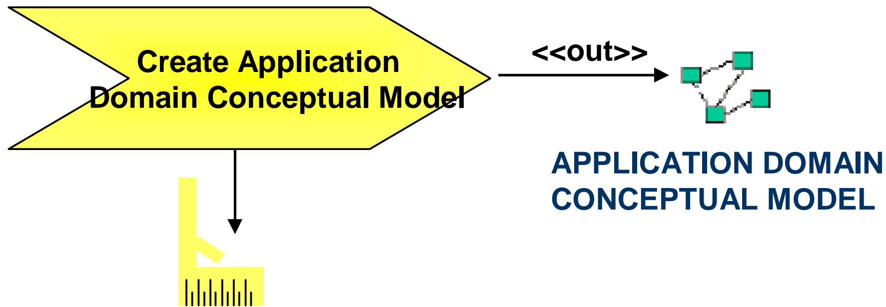


Figura 59 Creación del Modelo Conceptual del Dominio de Aplicación (notación SPEM)

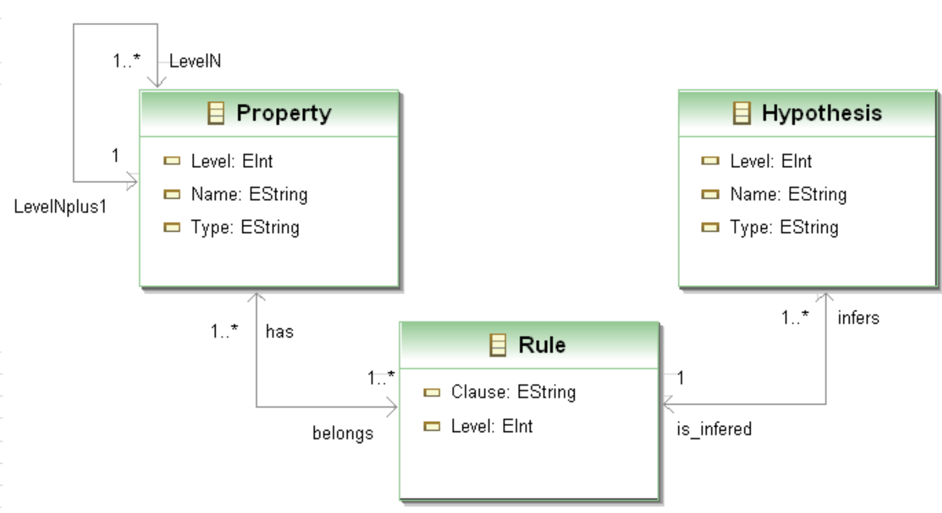


Figura 60 Modelo Conceptual del Dominio de Aplicación

6.- **Los Esqueletos o Plantillas de *Assets*.**- Los esqueletos de nuestra LPS son representados por medio de los aspectos (i.e. un aspecto PRISMA pero con huecos). En esta tesis, sólo hemos considerado un aspecto para cada elemento arquitectónico (puede existir más de un aspecto en un elemento arquitectónico). Los aspectos considerados en esta tesis son el aspecto funcional de los componentes y el aspecto de coordinación de los conectores. Los aspectos esqueletos son utilizados en la etapa de la ingeniería de la aplicación cuando BOM los recupera de la Baseline y los rellena con las características del dominio de aplicación.

Estas características son dadas de entrada al sistema por el ingeniero usando una instancia del MCDA. Los aspectos esqueleto rellenados o decorados son los aspectos tipo PRISMA, que son usados para configurar el modelo arquitectónico de un SE.

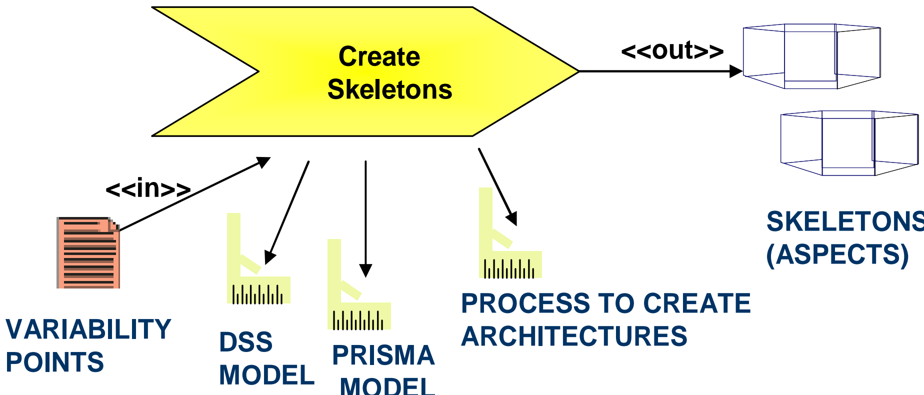


Figura 61 Creación de los (aspectos) esqueletos (notación SPEM)

Se considera a un esqueleto como un tipo PRISMA con huecos, dichos huecos serán rellenados con las "*features*" del dominio de aplicación (i.e. propiedades de la entidad del dominio, las hipótesis y las reglas, que se corresponden con los atributos del aspecto tipo PRISMA). Es lógico considerar que los esqueletos contienen *n* huecos, dependiendo de la cantidad de "*features*". De esta manera, si un esqueleto no tiene huecos (llamado esqueleto degenerado) es porque no tendrá ninguna *feature* que insertar (i.e. no tendrá atributos cuando se convierta en tipo PRISMA). Un ejemplo de esqueleto degenerado es el correspondiente al aspecto de coordinación de un conector (ver apéndices C y D).

Esta situación hace que el aspecto de coordinación siempre será un tipo PRISMA y no un esqueleto, sin embargo para utilizar una nomenclatura unificada, dicho artefacto software es tratado como esqueleto (i.e. esqueleto degenerado).

En PRISMA el conjunto de aspectos que conforman un elemento arquitectónico es visto como una cara de un prisma, de modo que dicho elemento tendrá tantos aspectos como caras de un prisma. Aunque nosotros consideramos sólo un aspecto por elemento, en la metáfora visual rellenamos todo el prisma. Una metáfora visual de un esqueleto es

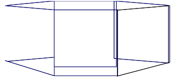


Figura 62 Metáfora visual de un esqueleto (aspecto)

7.- **Los tipos PRISMA (Intefaces, Elementos Arquitectónicos y Modelo Arquitectónico).-** Una interfaz contiene el conjunto de servicios que se reciben/envían a través de un puerto de un elemento arquitectónico. Las interfaces no involucran "*features*", por ello no son consideradas como esqueletos, sino tipos PRISMA. Una metáfora visual de una interfaz es mostrada en la **figura 63**.

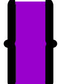


Figura 63 Metáfora visual de una interfaz tipo PRISMA

En PRISMA los elementos arquitectónicos importan los aspectos que lo conforman. Los cuatro elementos arquitectónicos PRISMA básicos de nuestra LPS son:

- *El componente InferenceMotor-* que establece el control del sistema y toma decisiones. Por lo que, este componente establece la estrategia general de la solución para tomar una decisión (por ejemplo obtener un diagnóstico). Este componente tiene un aspecto funcional.
- *El componente KnowledgeBase-* que contiene el conocimiento del dominio del caso de estudio en forma de reglas de inferencia (cláusulas de Horn con cabeza) y hechos (información que permanece constante). Este componente tiene un aspecto funcional.
- *El componente UserInteface-* que establece la interacción hombre-máquina, permitiendo la comunicación entre los usuarios y el sistema. A través de este componente el usuario ofrece los datos al sistema o responde a preguntas formuladas por éste. Este componente tiene un aspecto funcional.
- *El conector Coordinator-* que contiene la coreografía del proceso del dominio (por ejemplo el proceso del diagnóstico), y sincroniza los servicios que son enviados/recibidos por los componentes que une. Este conector tiene un aspecto de coordinación.

La **figura 64** muestra una metáfora visual de un componente y un conector tipos PRISMA:

(a) (b)

Figura 64 Metáfora visual de un componente (a) y un conector (b) tipos PRISMA

Las interfaces, los elementos arquitectónicos y la configuración no contienen los huecos semánticos, i.e. no conllevan "*features*". Por ello, estos artefactos software

no son considerados esqueletos sino tipos PRISMA. Estos artefactos son almacenados en la Baseline.

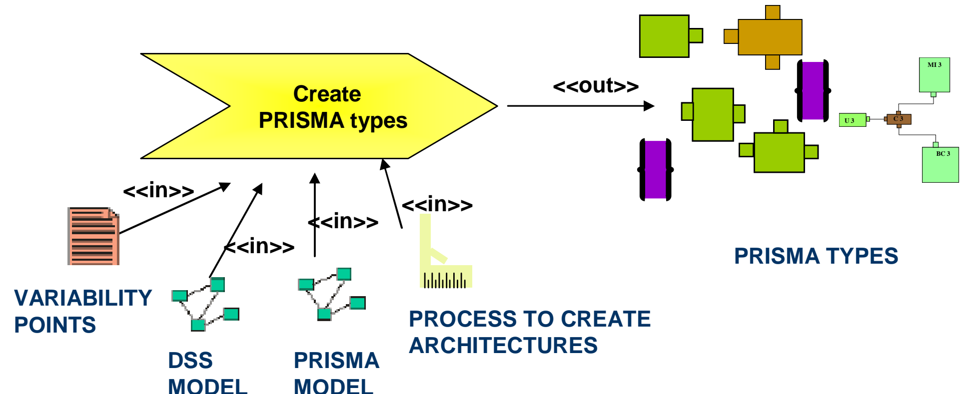


Figura 65 Creación de los artefactos tipo PRISMA (notación SPEM)

8.- **La configuración del Modelo Arquitectónico**.- Este asset captura la forma en que es configurada una arquitectura PRISMA, por medio de la instanciación de los artefactos software tipo PRISMA que la conforman y las conexiones entre ellos. Este asset es almacenado en la Baseline. En la ingeniería del dominio, el modelo de configuración y los artefactos tipo PRISMA son tomados por BOM con el fin de compilar dicho modelo, obteniendo el código.

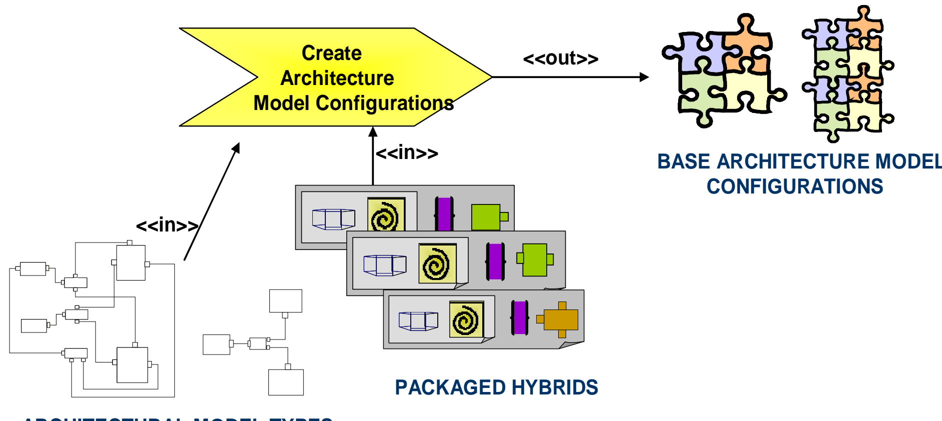


Figura 66 Creación de las configuraciones de los modelos arquitectónicos (notación SPEM)

9.- **El Proceso Principal de Inserción de Características**.- Éste es el proceso general de inserción de las características del dominio de aplicación, el cual invoca a los procesos individuales de inserción de características asociados a los esqueletos (aspectos).

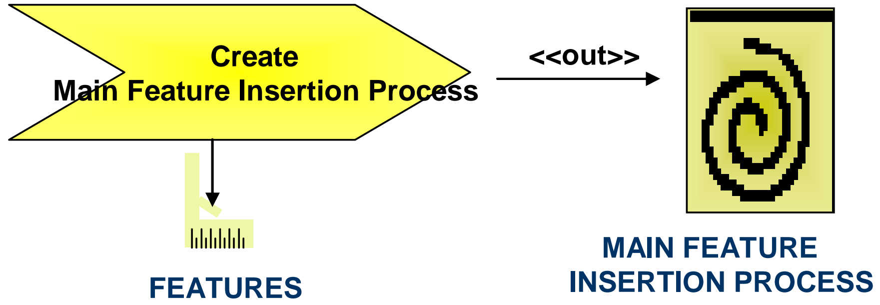


Figura 67 Creación del Proceso principal de inserción de características (notación SPEM)

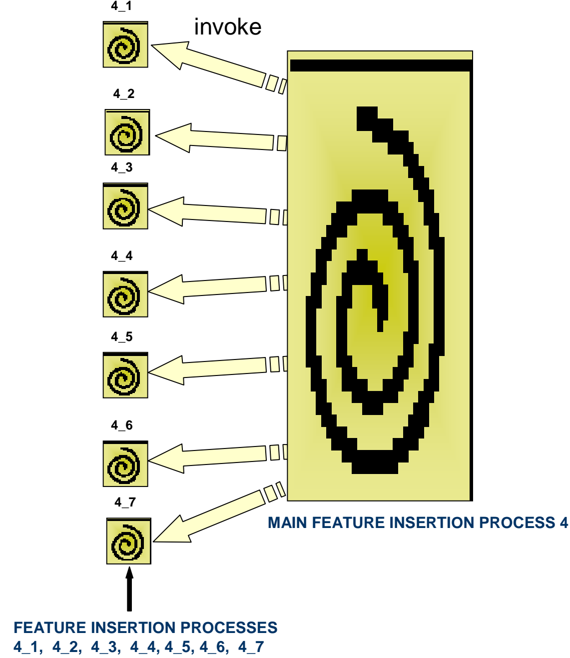


Figura 68 Proceso principal de inserción de características

10.- **Los Procesos de Inserción de Características**.-Cada uno de estos procesos es usado para insertar las características del dominio de aplicación en los esqueletos (aspectos). Estos procesos son invocados por el proceso principal de inserción de características.

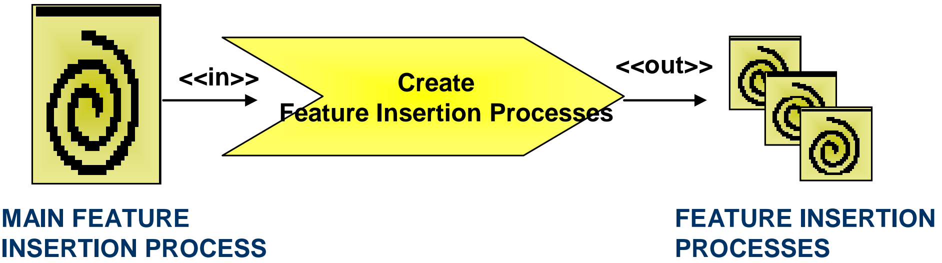


Figura 69 Creación de los Procesos de inserción de características (notación SPEM)

11.- **Los Esqueletos Aspecto-Proceso**.- Cada uno de estos *assets* está formado por un esqueleto aspecto y su correspondiente proceso de inserción de características. Estos *assets* son ejecutados en la ingeniería de aplicación, para decorar los esqueltos y convertirlos en tipos PRISMA.

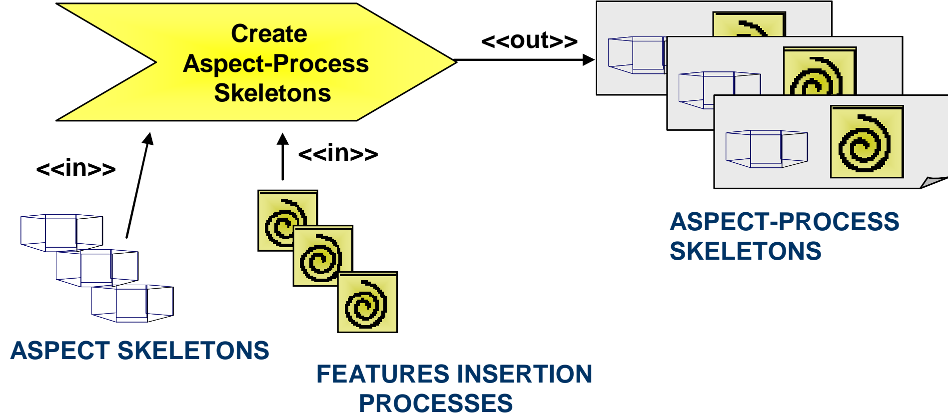


Figura 70 Creación de los esqueletos aspecto-proceso (notación SPEM)

12.- **Los Híbridos Empaquetados**.- Un híbrido empaquetado está compuesto por un esqueleto aspecto-proceso y sus respectivas interfaces y elemento arquitectónico tipo PRISMA, que son empaquetados en un nuevo asset. Estos *assets* son depositados en la Baseline. En el dominio de aplicación, los híbridos empaquetados

seleccionados son desempaquetados para que cada uno de los elementos que lo integran puedan ser utilizados.

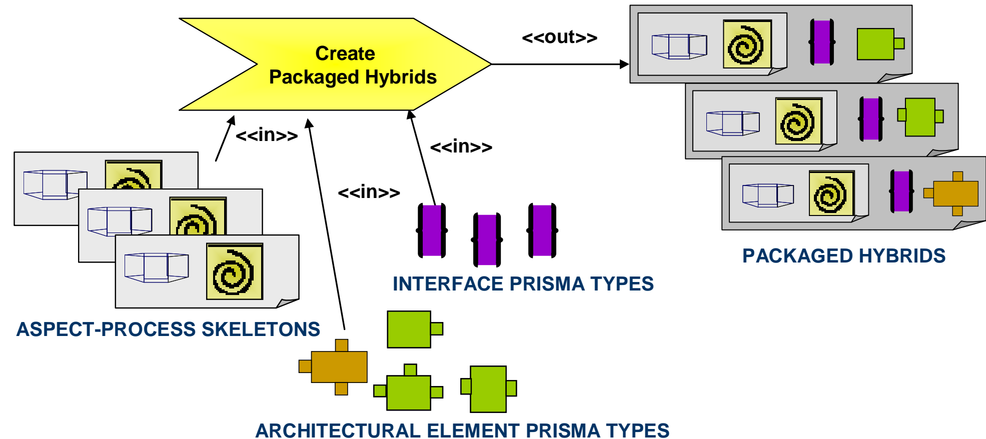


Figura 71 Creación de los Híbridos empaquetados (notación SPEM)

13.- **Los Modelos RAS de los *Assets*.**- Un modelo RAS almacena la información de cada uno de los *assets* que están contenidos en una caja: identificador ID de cada asset, clasificación de los *assets*, descripción de los *assets* y los puntos de variabilidad del dominio asociados a esos *assets*.

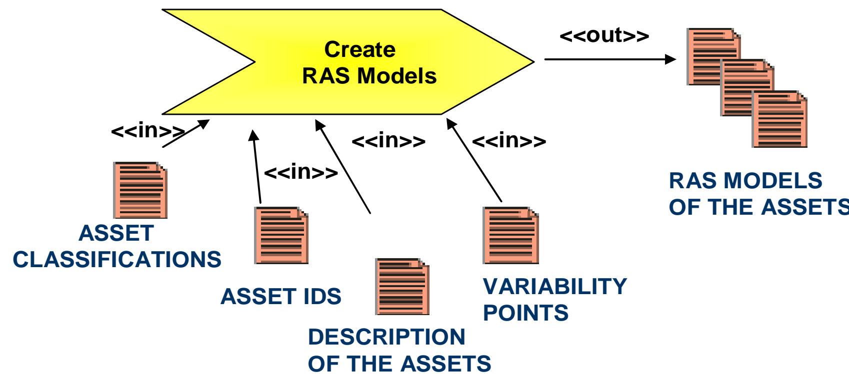


Figura 72 Creación de los modelos RAS (notación SPEM)

14.- **Las Cajas.**- Los *assets* seleccionados por el ingeniero del dominio para formar familias de LPS con características comunes (de la primera variabilidad), son

empaquetados como un asset. Dichos *assets* empaquetados conforman cajas cuyo contenido (activos y conocimiento de "cómo hacer") es construido en esa misma fase.

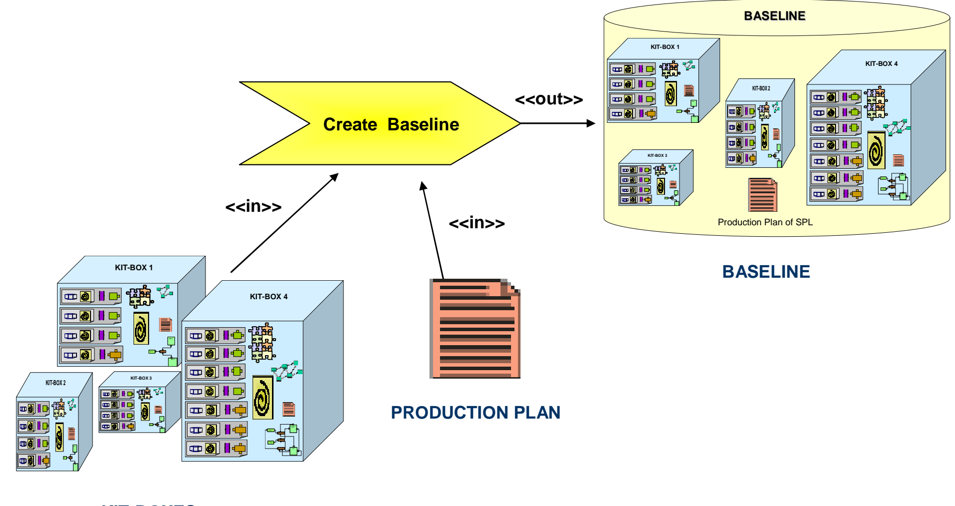


Figura 73 Creación de las cajas (notación SPEM)

La **figura 74** muestra una metáfora visual de una caja y su contenido.

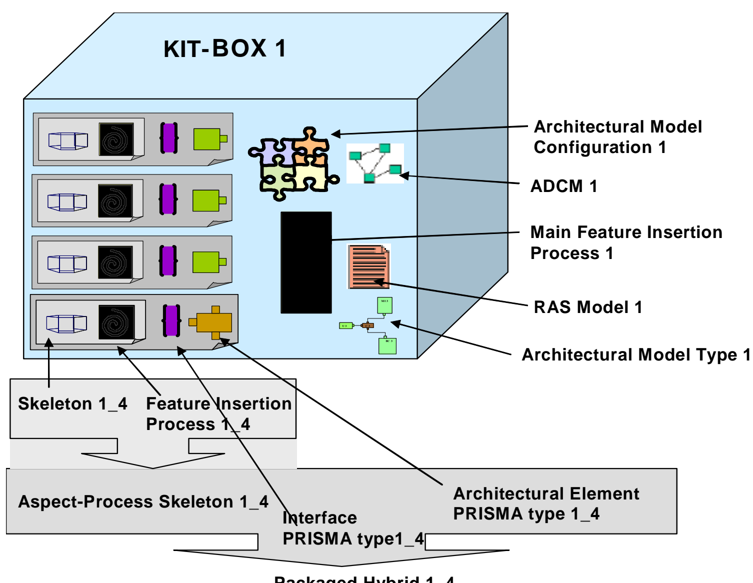


Figura 74 Ejemplo de una caja (kit-box)

15.- **El Plan de Producción de la LPS**.- este proceso muestra el ciclo de vida de producción de nuestra LPS. Dicho proceso, en la etapa de la ingeniería del dominio,

se especifica en SPEM, y es depositado en la Baseline como un activo. En la ingeniería de aplicación, el Plan de Producción es realizado para generar un SE como producto de la LPS.

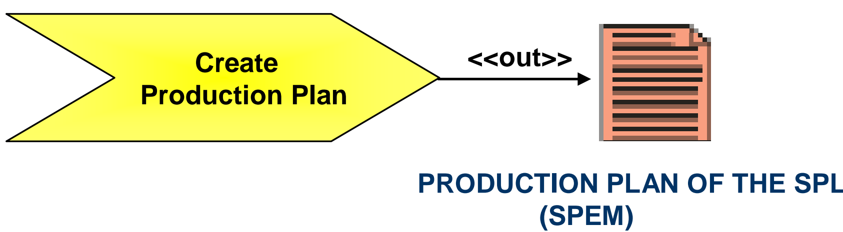


Figura 75 Creación del Plan de Producción (notación SPEM)

El Planeamiento de la producción implica la capacidad de fábrica de la LPS. En esta etapa se define cómo los productos software individuales son creados. Para ello es creado el plan de producción de la familia de productos de la LPS, que es descrita en el punto 9.4 de esta tesis, ya que la realización de dicha producción es parte de la ingeniería de la aplicación.

El ingeniero del dominio describirá el proceso de configuración del activo como el plan de producción de la LPS. La **figura 76** muestra en SPEM el proceso a seguir en la ingeniería de la aplicación para el plan de producción de la LPS.

Cabe señalar que este proceso de configuración del activo (que es el plan de producción de una línea de producto) es automático y obligatorio. Con ello, se obliga al ingeniero de la aplicación a utilizar las variantes ofrecidas por el activo reutilizable. De esta manera, el ingeniero de la aplicación no puede crear cualquier sistema de diagnóstico, sino que se le obliga a seguir el plan, pudiendo seleccionar únicamente una de las variantes de la línea seleccionando las características que lo identifican. Con ello se obtienen dos ventajas: la primera es obligar a seguir patrones y buenas prácticas de modelado que el ingeniero del dominio realizando esta tarea ha ido estableciendo progresivamente mientras creaba el activo, y la segunda es centralizar el conocimiento para ayudar a nuevos ingenieros de la aplicación a aprender a realizar dicha tarea.

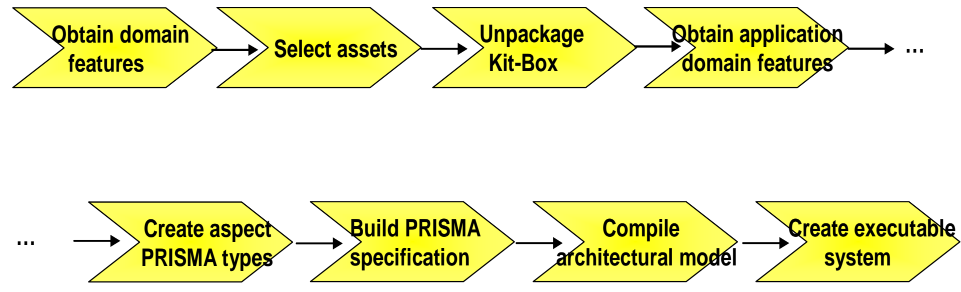


Figura 76 El plan de producción de la LPS en BOM

16.- **La Baseline**.- La Baseline es el último asset que se construye en la ingeniería del dominio, dado que es el repositorio de todos los *assets* necesarios para obtener un producto de la LPS.

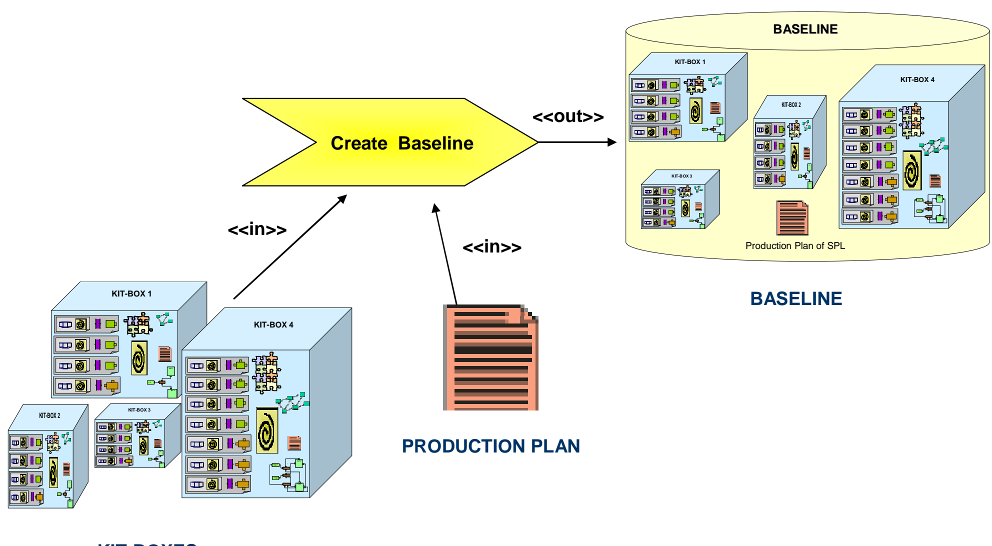


Figura 77 Creación de la Baseline (notación SPEM)

La Baseline está formada por un conjunto de *assets* empaquetados formando cajas. Además de estas cajas, la Baseline contiene el Plan de Producción de la LPS como un *assets* más. Dicho contenido implica que nuestra Baseline es un repositorio tanto de todos los *assets* como del conocimiento necesario para producir la LPS. Lo cual es una novedad en las LPS y una de las aportaciones de este trabajo.

La Baseline es utilizada en la ingeniería de la aplicación para elegir de este repositorio los *assets* (cajas o kit-boxes) correspondientes al dominio específico. La Baseline está representado por:

Baseline = {Kit-Box1, Kit-Box2, Kit-Box3, Kit-Box4,Production Plan}

Una metáfora visual de la Baseline se presenta en la **figura 78.**

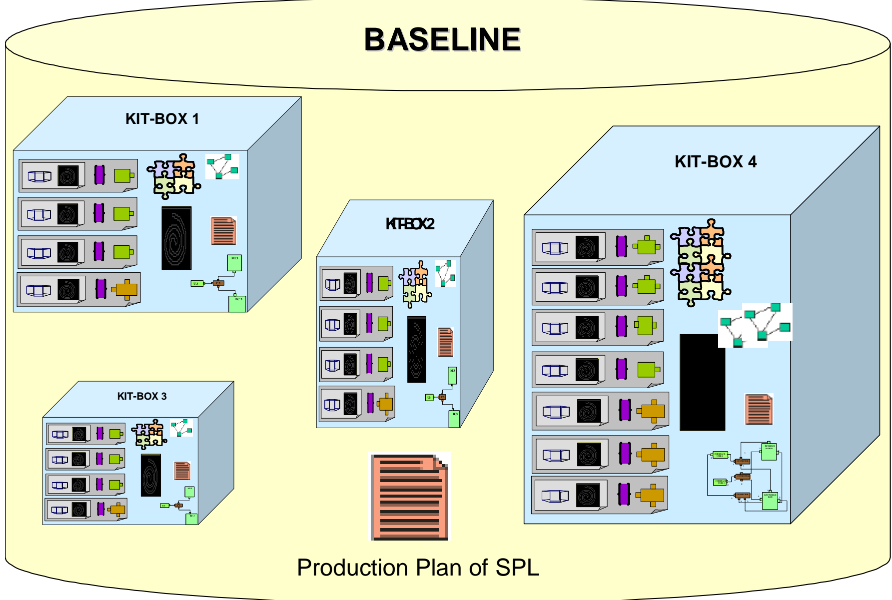


Figura 78 La Baseline

9.4 Ingeniería de la aplicación: ejecución del Plan de Producción

En esta segunda fase, las diversas tareas del proceso de desarrollo del software para el plan de producción de la LPS son ejecutadas por el rol del ingeniero de aplicación. Es importante hacer notar que este plan de producción será el proceso de configuración del activo del diagnóstico específico que está siendo desarrollado.

Las LPS proponen la reutilización del mismo conjunto de activos principales para generar una línea de productos [Clements et al., 2001]. En esta estrategia es de vital importancia la capacidad de los activos de ser configurados por el ingeniero de

la aplicación. El proceso de configuración de activos para generar una aplicación concreta puede ser modelado a través de SPEM.

En la fase de la ingeniería de la aplicación se realiza el plan de producción, el cual fue creado en la fase de la ingeniería del dominio como un asset.

Dado que, como resultado de esta investigación, se proponen dos aproximaciones para el desarrollo de la LPS, a continuación se presenta (en las secciones 9.4.1 y 9.4.2 de esta tesis) el plan de producción de la LPS desde la perspectiva de cada una de las aproximaciones.

9.4.1 Plan de producción de la LPS en la aproximación BOM-EAGER

En la aproximación BOM-EAGER, la Baseline es utilizada en la fase de la ingeniería de la aplicación, como uno de los recursos de las tareas del plan de producción de la LPS, para seleccionar los *assets* contenidos en ella.

El plan de producción describe el proceso a través de tareas o actividades que se realizan para obtener un producto final de la LPS. Dicho proceso se muestra en la **figura 79**, haciendo uso de la notación del estándar SPEM.

El plan de producción es seguido a través de BOM que interacciona con el usuario (ingeniero de aplicación) de forma que solamente introduce en dos ocasiones los datos correspondientes al dominio específico. Estos datos son los puntos de variabilidad (características de la primera variabilidad) y las características del dominio de aplicación (características de la segunda variabilidad).

En la **figura 79** se muestran las 8 tareas involucradas en el plan de producción, todas ellas realizadas por BOM a través de la interacción con el usuario que desempeña el rol de ingeniero de aplicación.

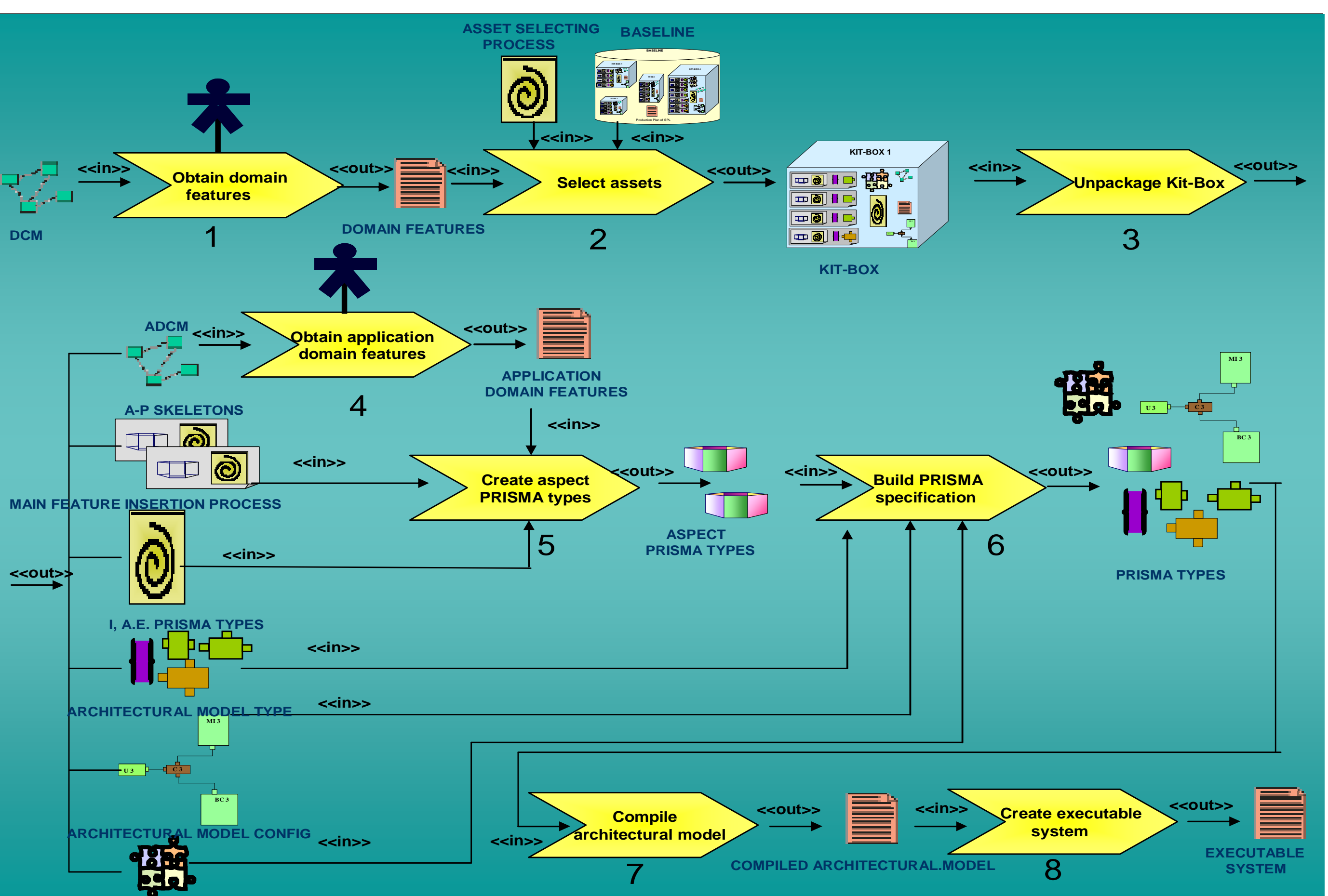


Figura 79 Plan de Producción de la aproximación BOM-EAGER (notación SPEM)

A continuación se describen y se modelan en SPEM cada una de estas tareas, enfatizando en la tarea donde es utilizada la Baseline. Estas tareas son realizadas en tres partes: la caracterización del producto, la síntesis del producto y la construcción del producto.

9.4.1.1 Caracterización del producto

En la caracterización del producto se eligen las características que diferencian un producto seleccionado. Este proceso involucra tres tareas ejecutadas a través del rol del ingeniero de aplicación: crear la configuración de las características del dominio, seleccionar los *assets*, configurar características del dominio de aplicación y crear los tipos.

Cabe señalar que el proceso de caracterización del producto, tiene como punto de partida un modelo conceptual de diagnóstico (i.e. dominio del diagnóstico) y un modelo conceptual del dominio específico (i.e. dominio de la aplicación). Se aplica una secuencia de transformaciones automáticas de dichos modelos a una arquitectura PRISMA para la LPS. La transformación automática entre los modelos es esencialmente una transformación dirigida por la selección de características efectuada por el ingeniero de aplicación, configurando un modelo conceptual de características del diagnóstico y un modelo conceptual del dominio, que a su vez (por las relaciones de trazabilidad) generarán el modelo arquitectónico de la aplicación. Las variantes seleccionadas lo son en función de los requisitos particulares de la aplicación, que deben ser tomadas en cuenta por el ingeniero.

Tarea 1. **Obtener las Características del Dominio**.- BOM obtiene (del ingeniero de aplicación usando *check boxes* y *pull-down* menus) las características del dominio expresadas en el modelo conceptual del dominio, y que son consideradas como puntos de variabilidad de la primera variabilidad.

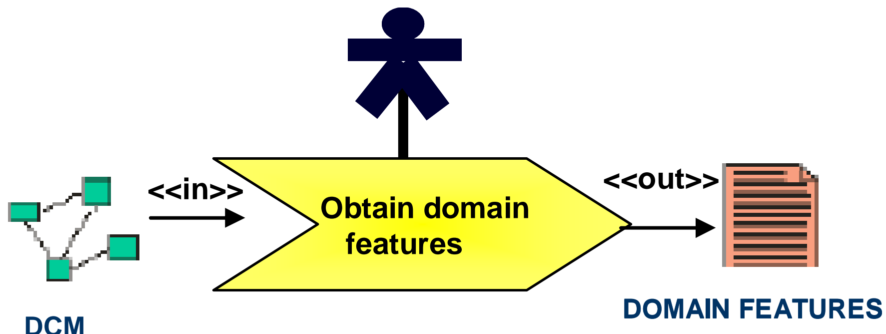


Figura 80 Proceso en SPEM para obtener las características del dominio

Tarea 2. **Seleccionar los *Assets* (Kit-Boxes).-** Con los puntos de variabilidad seleccionados por el ingeniero, BOM recupera de la Baseline la kit-box correspondiente. En esta actividad mostrada en la **figura 75** son consumidos de entrada los siguientes artefactos software:

a) El Proceso de Selección de *Assets*.- BOM aplica el *Assets* Selecting Process para seleccionar los *assets* correspondientes al dominio de aplicación,
b) Las Características del Dominio, i.e. la información dada por el ingeniero de aplicación sobre las características del dominio de aplicación a través de una instancia del MCD,
c) La Baseline.- que contiene los kit-box *assets* correspondientes a un dominio específico,

y de salida, son producidos los artefactos software seleccionados con el fin de obtener un producto de la LPS, i.e. la caja que corresponde al dominio específico.

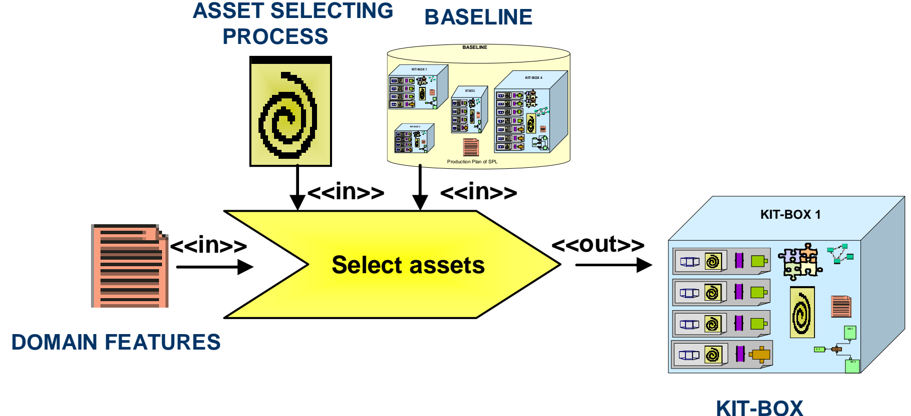


Figura 81 Proceso en SPEM para seleccionar asset

Tarea 3. **Desempaquetar la Caja**.- La caja que fue seleccionada por el ingeniero a través de BOM, debe ser desempaquetada para poder utilizar cada uno de los *assets* de su contenido en forma individual.

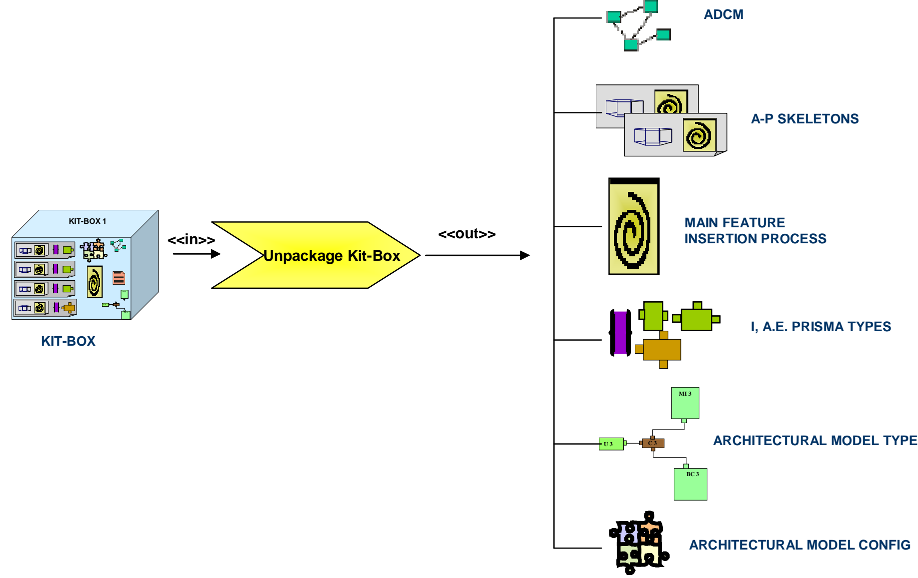


Figura 82 Proceso en SPEM para desempaquetar una caja

Tarea 4. **Obtener Características del Dominio de Aplicación**.- BOM obtiene (del ingeniero de aplicación) las características del dominio de la aplicación consideradas como características de la segunda variabilidad.

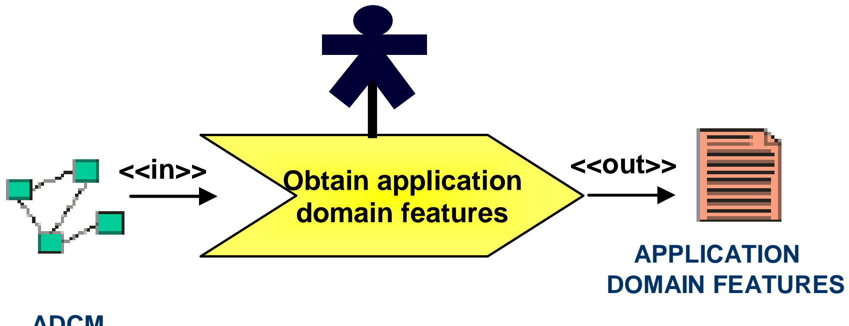


Figura 83 Proceso en SPEM para obtener las características del dominio de aplicación

Tarea 5. **Crear los aspectos tipo PRISMA**.- BOM rellena los esqueletos (aspectos) con los datos de las características del dominio de aplicación del caso de estudio

que fueron definidas por el ingeniero de la aplicación, creando los aspectos tipo PRISMA.

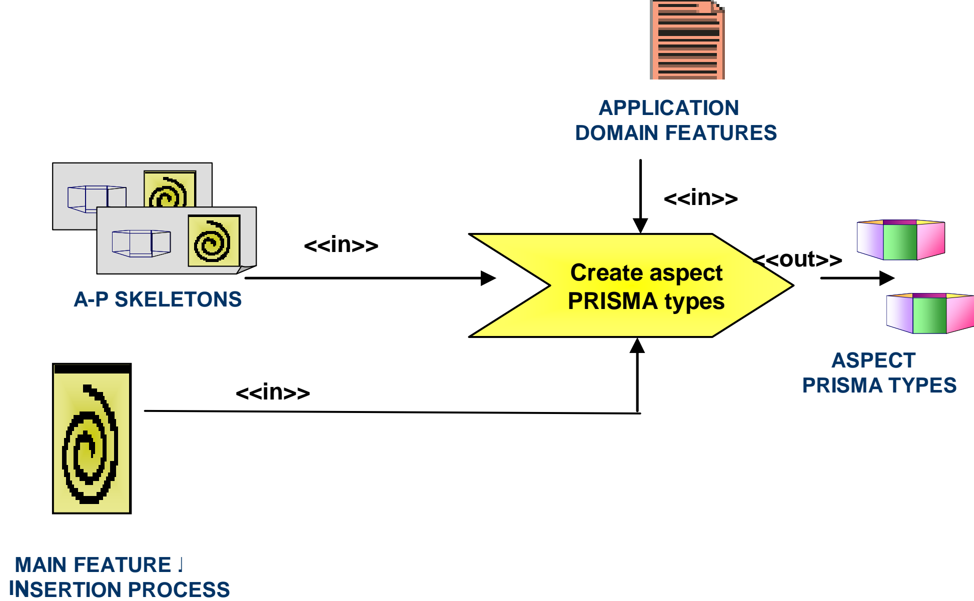


Figura 84 Proceso en SPEM para obtener los aspectos tipo PRISMA

9.4.1.2 Síntesis del producto

Un producto está compuesto por activos. La síntesis del producto reúne los activos para obtener la materia prima. En este proceso se ejecuta la tarea "construir especificación PRISMA" a través del rol del ingeniero de aplicación.

Tarea 6. **Construir especificación PRISMA**.- BOM toma los aspectos tipo PRISMA y los aglutina con otros *assets* desempaquetados en la tarea 3: los artefactos tipo PRISMA (interfaces, elementos arquitectónicos, modelo arquitectónico), y la configuración del modelo arquitectónico, con el fin de reunir a todos los *assets* necesarios para construir la especificación del modelo arquitectónico en PRISMA, y que serán utilizados en la siguiente tarea.

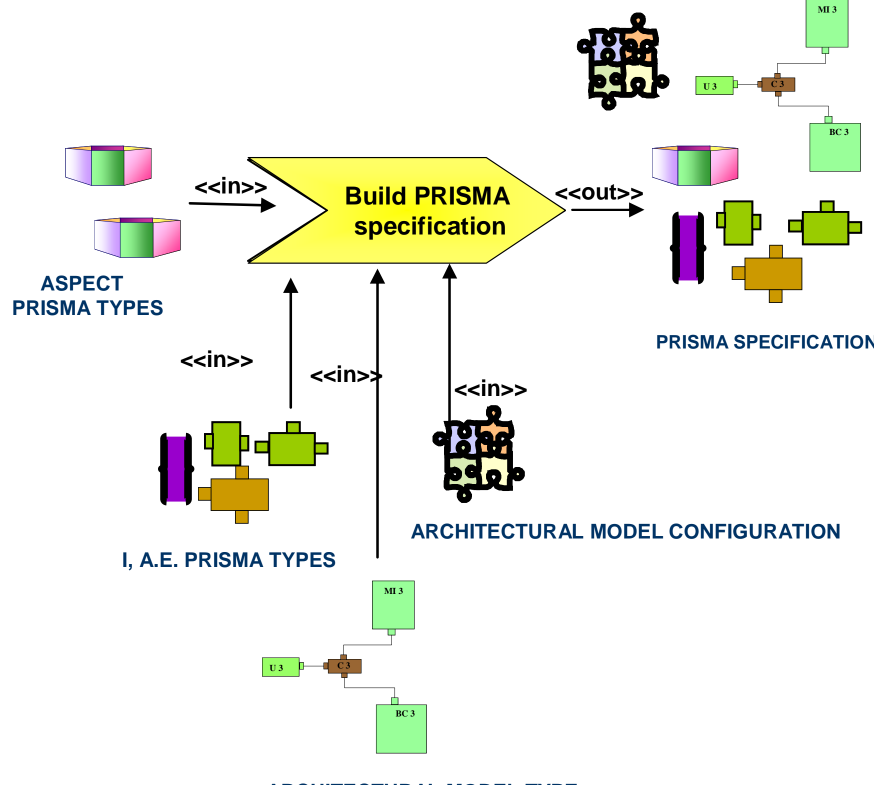


Figura 85 Proceso en SPEM para construir la especificación PRISMA

Cabe señalar en esta sección que con la herramienta PRISMA-CASE [Cabedo et al., 2005], [Pérez et. al, 2005a], los *assets* (artefactos software tipo PRISMA y la configuración del modelo arquitectónico) son creados al modelar el sistema a través de una metáfora visual. Es decir, es equivalente la obtención de los *assets* a través de BOM, a la generación de dichos *assets* mediante el modelado de la arquitectura del sistema con PRISMA-CASE. La **figura 85** muestra el modelo del sistema del diagnóstico de programas educativos.

Como puede observarse en dicha figura, con la aproximación BOM es más sencillo especificar un modelo arquitectónico, obteniendo los mismos resultados (*assets* en el LDA de PRISMA).

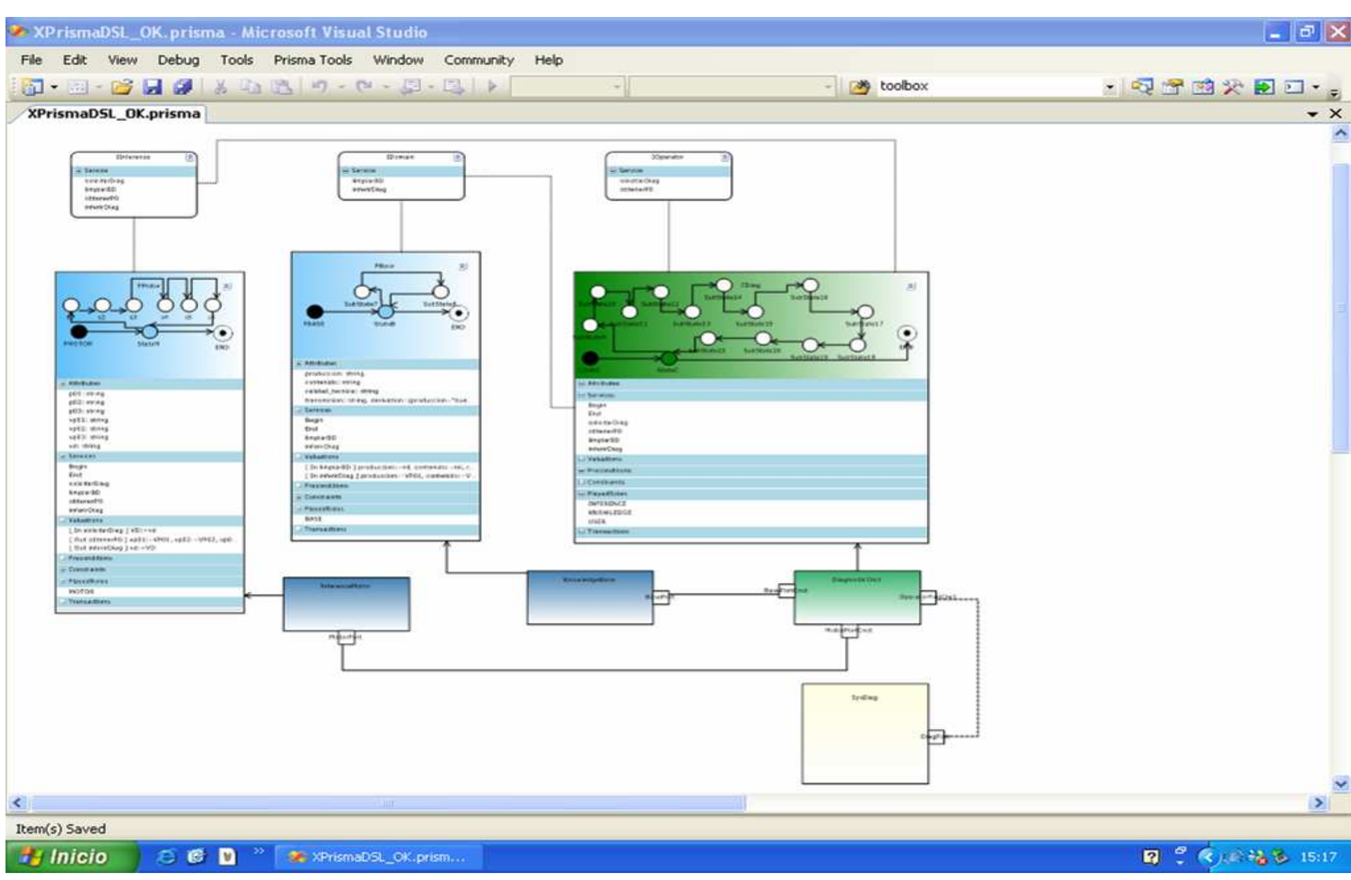


Figura 86 Metáfora visual del sistema del diagnóstico de programas educativos (realizada con la herramienta PRISMA-CASE)

9.4.1.3 Construcción del producto

La construcción del producto procesa la materia prima siguiendo el proceso de construcción (compilar, generar código) para obtener el producto final.

Las tareas que se realizan en esta etapa de la construcción del producto, a través del rol del ingeniero de aplicación son: compilar modelo y crear sistema ejecutable.

El entorno destino de BOM es el *framework* PRISMA. Por ello, las tareas de compilar y crear ejecutable son realizadas sobre las herramientas PRISMA-MODEL-COMPILER [Cabedo et al., 2005] y el Middleware PRISMA-NET [Costa et al., 2005], [Pérez et. al, 2005a] respectivamente, como se comenta a continuación.

Tarea 7. **Compilar el Modelo Arquitectónico**.- Los artefactos tipo PRISMA y la configuración del modelo arquitectónico, son incorporados por medio de BOM a la herramienta PRISMA-MODEL-COMPILER para compilar el modelo arquitectónico y generar automáticamente el código (en C#,.NET) .

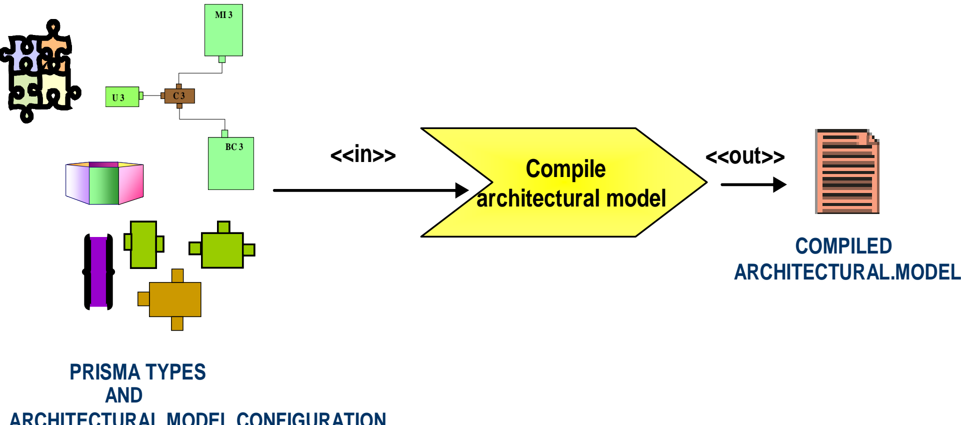


Figura 87 Proceso en SPEM para compilar el modelo arquitectónico PRISMA

Tarea 8. **Crear el sistema ejecutable**- BOM, a través del Middleware PRISMANET, obtiene del modelo arquitectónico compilado, el sistema final como una aplicación ejecutable, i.e. una instancia de nuestra LPS.

Figura 88 Proceso en SPEM para crear el sistema ejecutable

9.4.2 Plan de producción de la LPS en la aproximación BOM-LAZY

En la aproximación BOM-LAZY, la Baseline es calculada en la fase de la ingeniería de la aplicación, i.e. la Baseline es implícita.

El plan de producción describe el proceso a través de 6 tareas o actividades que se realizan para obtener un producto final de la LPS. Dicho proceso se muestra en la **figura 88**, haciendo uso de la notación del estándar SPEM.

El plan de producción es seguido a través de BOM que interacciona con el usuario que desempeña el rol de ingeniero de aplicación, de forma que solamente introduce en dos ocasiones los datos correspondientes al dominio específico. Estos datos son los puntos de variabilidad (características de la primera variabilidad) y las características del dominio de aplicación (características de la segunda variabilidad).

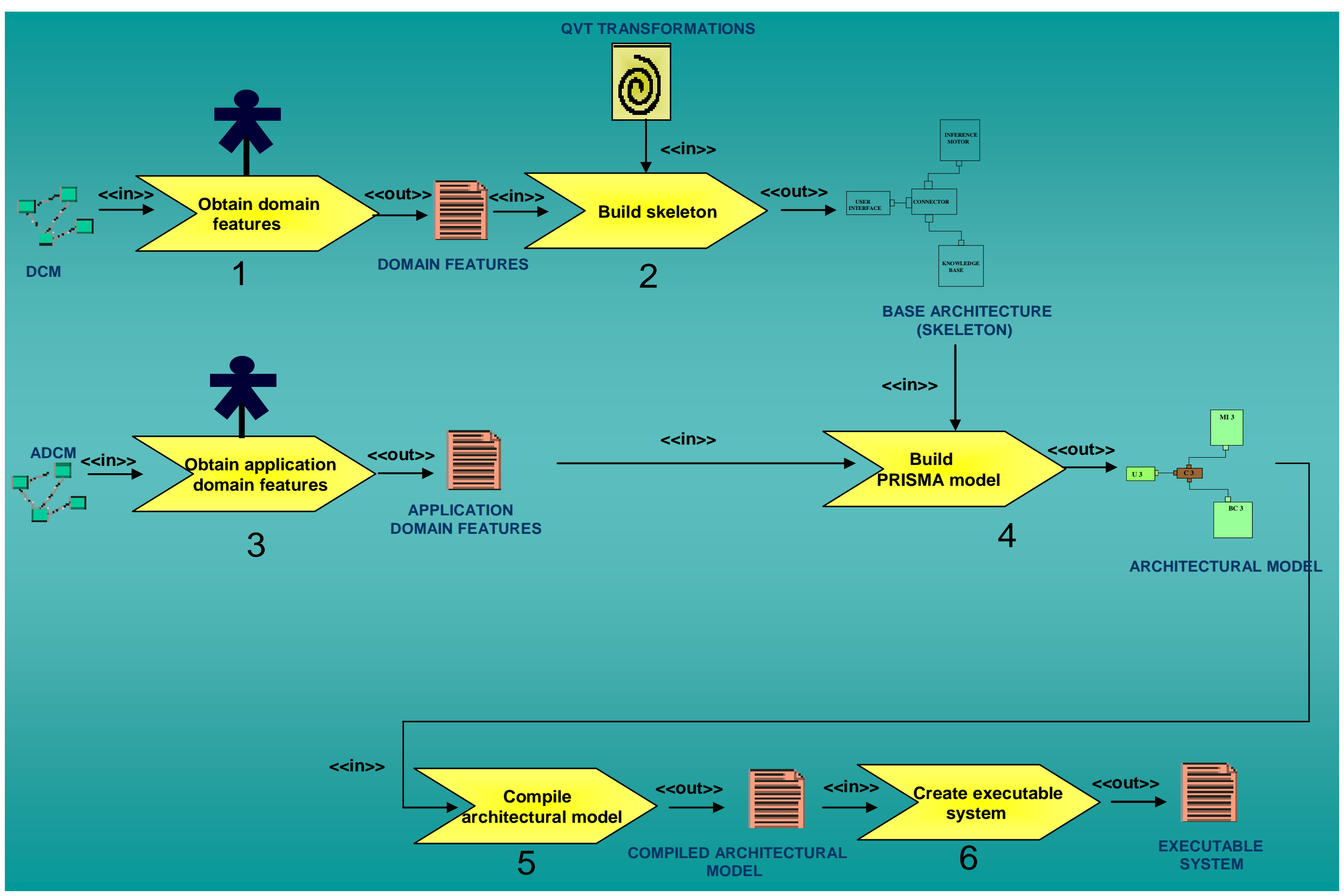


Figura 89 Plan de Producción de la aproximación BOM-LAZY (notación SPEM)

9.4.3 Ejecución del sistema (producto final)

Para clarificar la ejecución del sistema experto (producto final) por el usuario final, se ha utilizado la notación en SPEM. Para realizar la tarea "ejecutar sistema", se consumen de entrada el sistema ejecutable correspondiente al modelo del sistema diagnóstico del caso específico (o sea el producto final de la LPS), y los valores de las propiedades de la entidad a diagnosticar (introducidos a través de la IGU), produciendo como salida el resultado del diagnóstico de dicha entidad (en la IGU).

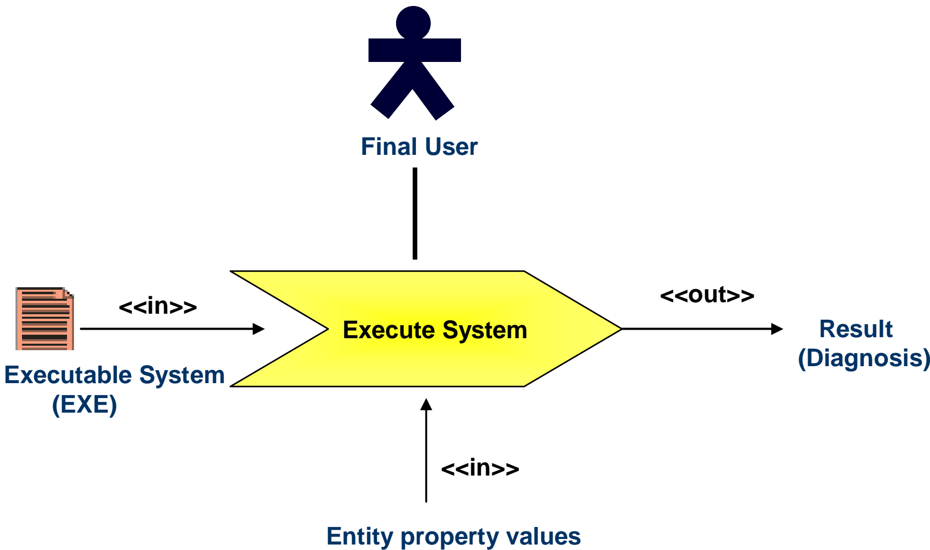


Figura 90 Ejecución de un sistema de la LPS (con notación en SPEM)

9.5 Conclusiones

Cualquiera de las dos aproximaciones de BOM (i.e. BOM-EAGER o BOM-LAZY) muestra que BOM es una solución genérica aplicable a otros dominios y a diferentes tipos de sistemas software, ya que se mantiene el mismo plan de producción de la LPS.

Además, en la ingeniería del dominio pueden ser creados los mismos *assets* pero con las características *ad-hoc* al dominio, o bién construir otros *assets* que se correspondan con el dominio en el que se desarrolla la LPS.

PARTE V. IMPLEMENTACIÓN

LOGOTIPO DE PROTOBOM[6]

[6] Creado por María Gómez Lacruz *exprofeso* para esta tesis

CAPÍTULO 10

IMPLEMENTACIÓN DE LA APROXIMACIÓN BOM-EAGER: PROTOBOM

Sírvete de lo aparente como indicio de lo inaparente.
Solón de Atenas (640 A.C.- 558 A.C.). Legislador griego

10.1 Introducción

En este capítulo se presenta un prototipo de la aproximación BOM-EAGER: ProtoBOM [Cabello et al., 2008]. ProtoBOM es una herramienta que genera aplicaciones basadas en LPS.

La estructura de este capítulo es la siguiente: En la sección 2 se comentan las características principales de ProtoBOM. En la sección 3 se describe el uso de ProtoBOM en la ingeniería del dominio. En la sección 4 se describe el uso de ProtoBOM en la ingeniería de la aplicación. La sección 5 presenta las conclusiones de este capítulo.

10.2 Características de ProtoBOM

ProtoBOM es una aplicación para generar sistemas sobre el sistema operativo Windows, desarrollada empleando el lenguaje de programación orientada a objetos C#, .NET.

ProtoBOM utiliza un servicio Web para acceder a la base de datos (Baseline). ProtoBOM no requiere de algún hardware específico.

ProtoBOM puede ser utilizado por dos tipos de usuario, o en otras palabras, por dos usuarios con diferentes roles: el ingenierio del dominio y el ingeniero de la aplicación. El ingeniero del dominio crea todos los artefactos software necesarios

para desarrollar un producto de la LPS. El ingeniero de la aplicación ejecuta el plan de producción, generando un producto de la LPS, i.e. un SED.

El uso de ProtoBOM (por el ingeniero del dominio) implica:

- construir el modelo de características,
- construir el árbol de decisión
- construir las trayectorias de dicho árbol para seleccionar *assets*.
- empaquetar todos los *assets* uno a uno hasta conformar la Baseline

ProtoBOM integra el uso de varias herramientas, ofreciendo una herramienta única para construir aplicaciones de una manera simple. Algunas de esas herramientas fueron desarrollas "exprofeso" para ProtoBOM y otras ya habían sido desarrolladas "a priori" para otros ámbitos.

Las herramientas realizadas "exprofeso" para este prototipo permiten:

- ingresar las características del dominio (puntos de variabilidad) y con ello seleccionar una kit-box de la Baseline
- ingresar las características del dominio de aplicación, y con ello crear los aspectos tipos PRISMA, a través de la inserción de características del dominio de aplicación sobre los aspectos esqueletos.
- reunir la información necesaria para que las herramientas PRISMA puedan generar el código y crear el ejecutable del sistema.
- integrar todas las herramientas mencionadas en una sola:

Con las herramientas realizadas "a priori", ProtoBOM puede:

- utilizar el compilador de modelos PRISMA-MODEL-COMPILER para generar automáticamente el código C# del modelo arquitectónico final.
- crear el sistema ejecutable (i.e. una instancia de la LPS) sobre el middleware PRISMA-NET.

Para capturar los requisitos funcionales de ProtoBOM, ha sido usado el diagrama de casos de uso de UML. En las **figuras 91 y 92** se presentan los casos de uso a los que accede el ingeniero del dominio y el ingeniero de la aplicación, respectivamente.

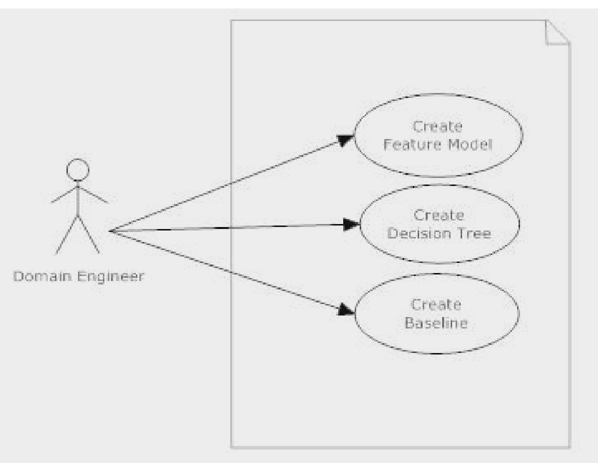


Figura 91 Casos de uso en la ingeniería del dominio

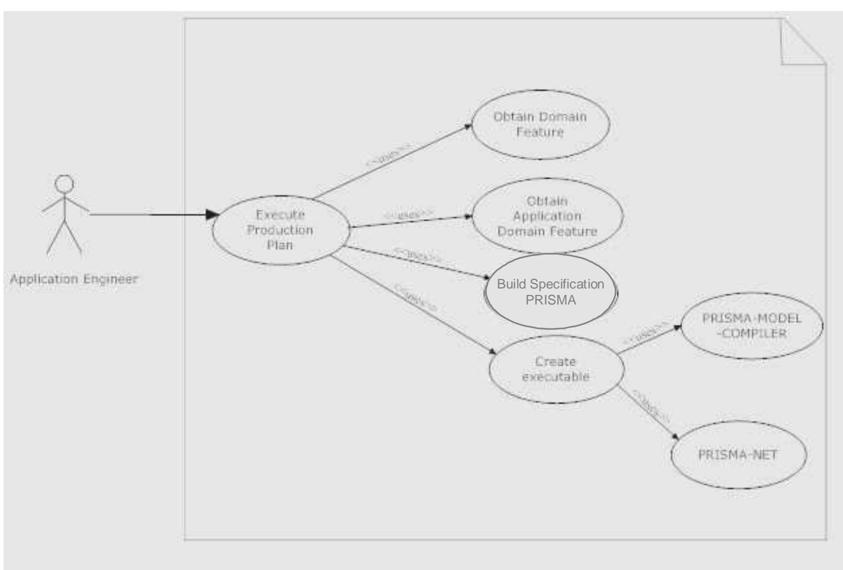


Figura 92 Casos de uso en la ingeniería de la aplicación

La IGU inicial de ProtoBOM permite seleccionar al usuario, conforme al rol que desempeña, las actividades de la ingeniería del dominio o de la ingeniería de la aplicación, como lo muestra la **figura 93**.

Figura 93 IGU inicial de ProtoBOM

10.3 ProtoBOM en la ingeniería del dominio.

Para crear la Baseline, el ingeniero del dominio deberá empaquetar y depositar cada uno de los *assets* (previamente creados), lo que implica que en la Baseline se irán depositando una a una las cajas (con sus *assets*) y el Plan de Producción. La **figura 94** muestra la respectiva IGU, la cual contiene dos pestañas: "Add Box" y "Add Prod Plan".

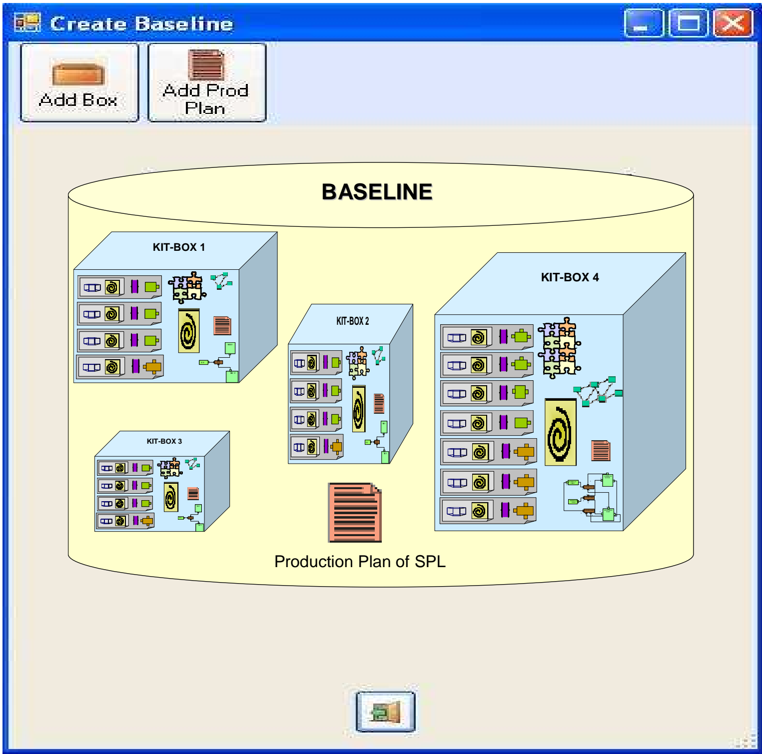


Figura 94 IGU para crear la Baseline

La pestaña "Add Box" permite insertar (una a una) las cajas en la Baseline. La **figura 95** muestra la creación de la caja 1 (vacía).

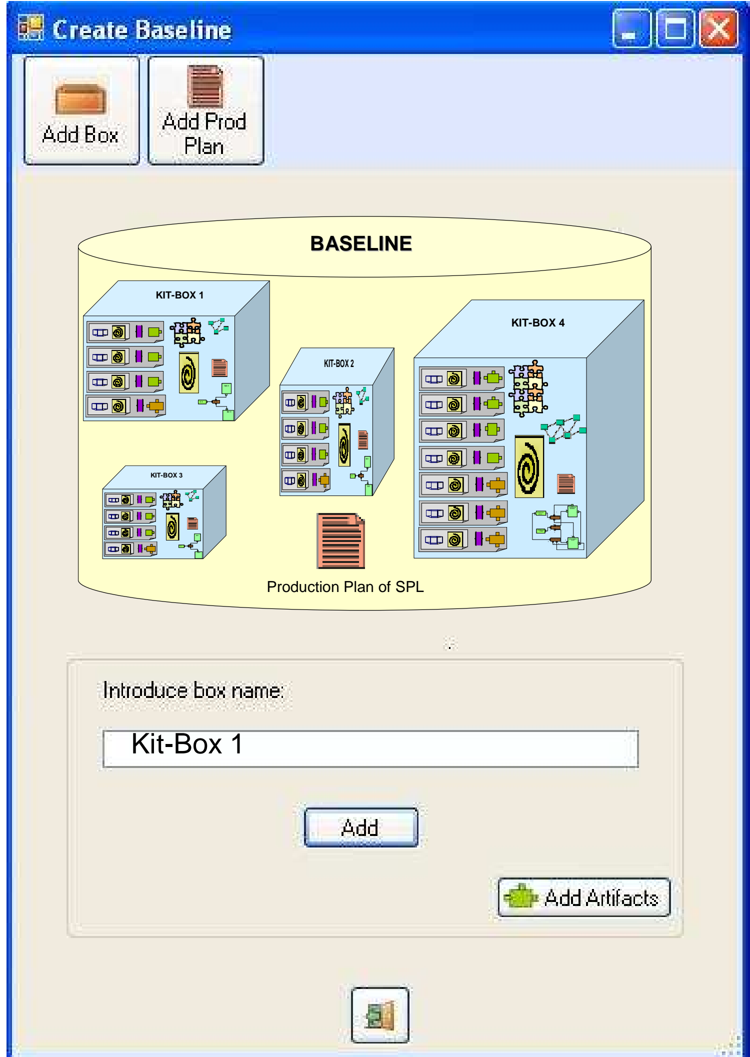


Figura 95 IGU para empaquetar una caja a la Baseline

Para "llenar" la caja (i.e. depositar todos los *assets* de su contenido) se utiliza la pestaña "Add Artifact". Al ser seleccionada esta pestaña, BOM presentará una IGU, en la que mostrará por default el nombre de la caja del paso anterior, asimismo mostrará una serie de iconos, que servirán para darle el nombre del asset y cargarlo a su caja (seleccionándolo desde el Explorer), de modo que se irán a agregando uno a uno los *assets* correspondientes. Un ejemplo de depósito del asset "Base Architectural Model Configuration" se muestra en la **figura 96**.

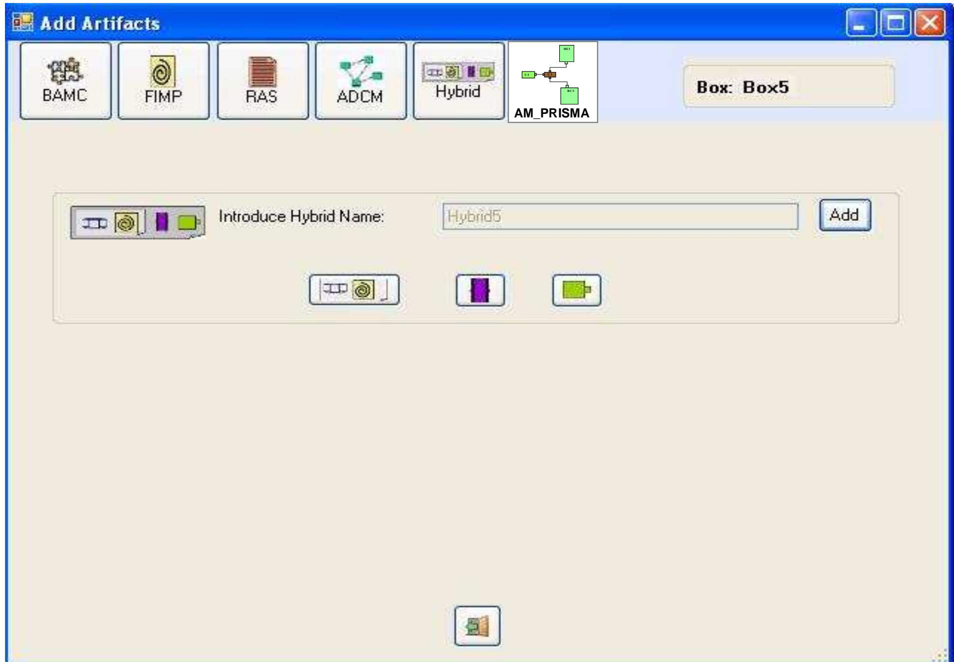


Figura 96 IGU para empaquetar artefactos software en una caja

Pero el depósito de algunos *assets* en la caja no se efectúa de forma directa, sino que conlleva un proceso previo de empaquetamiento, conformando un nuevo asset que será insertado en la caja. El mismo proceso comentado para el asset "Feature Insertion Process" se realiza en estos casos. La IGU de para empaquetar el asset "Aspect-Process Skeleton" se muestra en la **figura 97**:

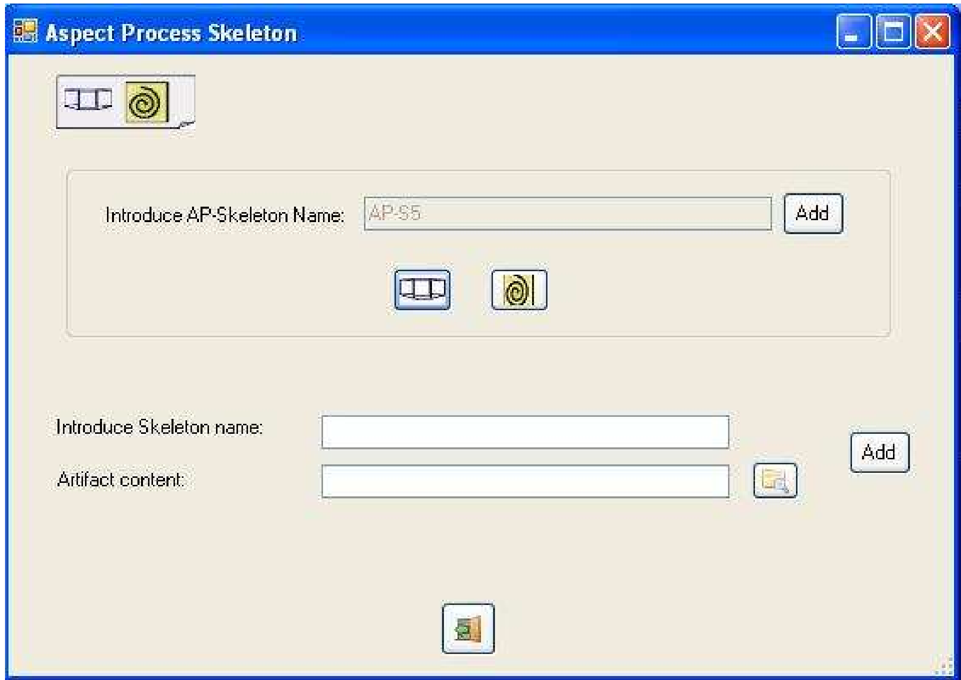


Figura 97 IGU para empaquetar un aspecto-proceso esqueleto

Este proceso de empaquetamiento y depósito es realizado tantas veces como sea necesario. Por ejemplo, para construir el asset "Hybrid" se deberá realizar el proceso ejemplificado anteriormente pero con lo *assets* "Aspect-Process Skeleton", "Interface" y "Architectural Element". Nótese que el asset "Aspect-Process Skeleton" debe haberse empaquetado previamente, para ser empaquetado como un item en el nuevo asset empaquetado. La IGU correspondiente a este caso muestra la vista de la **figura 98**.

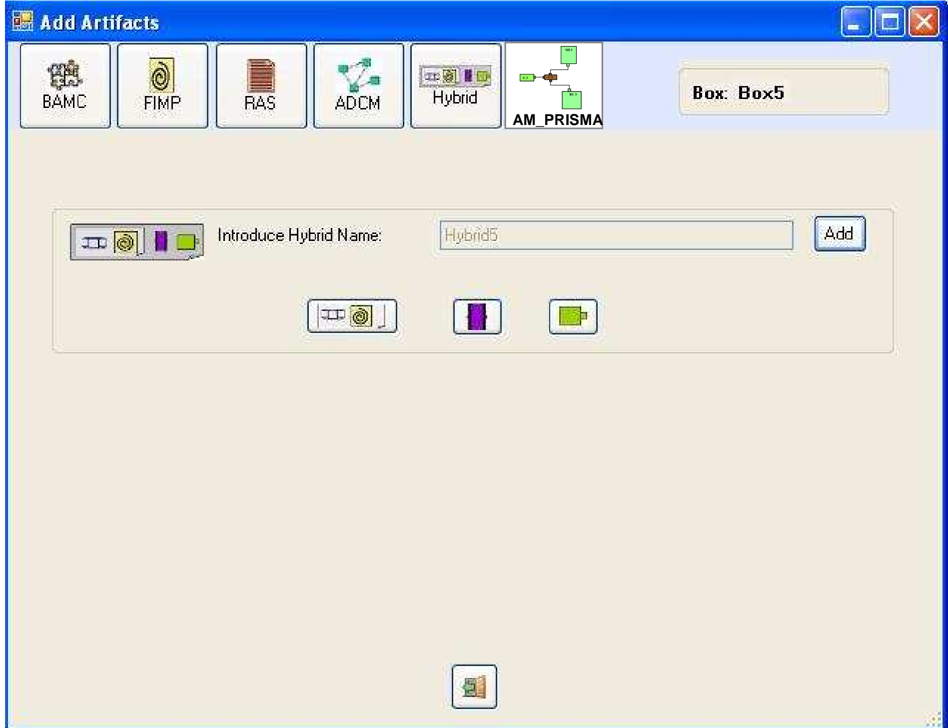


Figura 98 IGU para empaquetar un híbrido

La pestaña "Add Prod Plan" permite insertar el Plan de Producción a la Baseline, como se muestra en la **figura 99**.

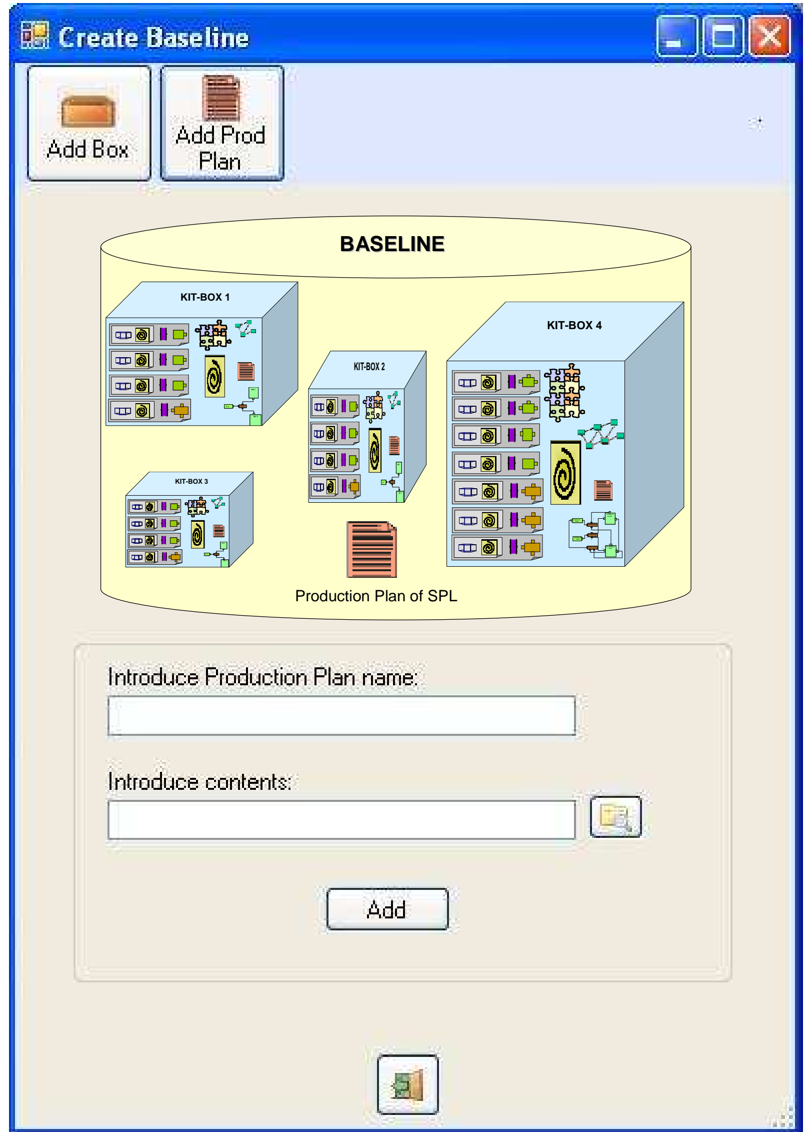


Figura 99 IGU para empaquetar el Plan de Producción en la Baseline

ProtoBOM ofrece (como un valor agregado) al ingeniero del dominio, además de construir la Baseline (como tarea específica de esta etapa de desarrollo de la LPS), el crear el Modelo de Características, el Árbol de Decisión y las trayectorias de dicho árbol cuyas hojas apunten al conjunto de *assets* seleccionados (i.e la caja seleccionada). Las IGU correspondientes a dichas actividades son mostradas en las **figuras 100 y 101**.

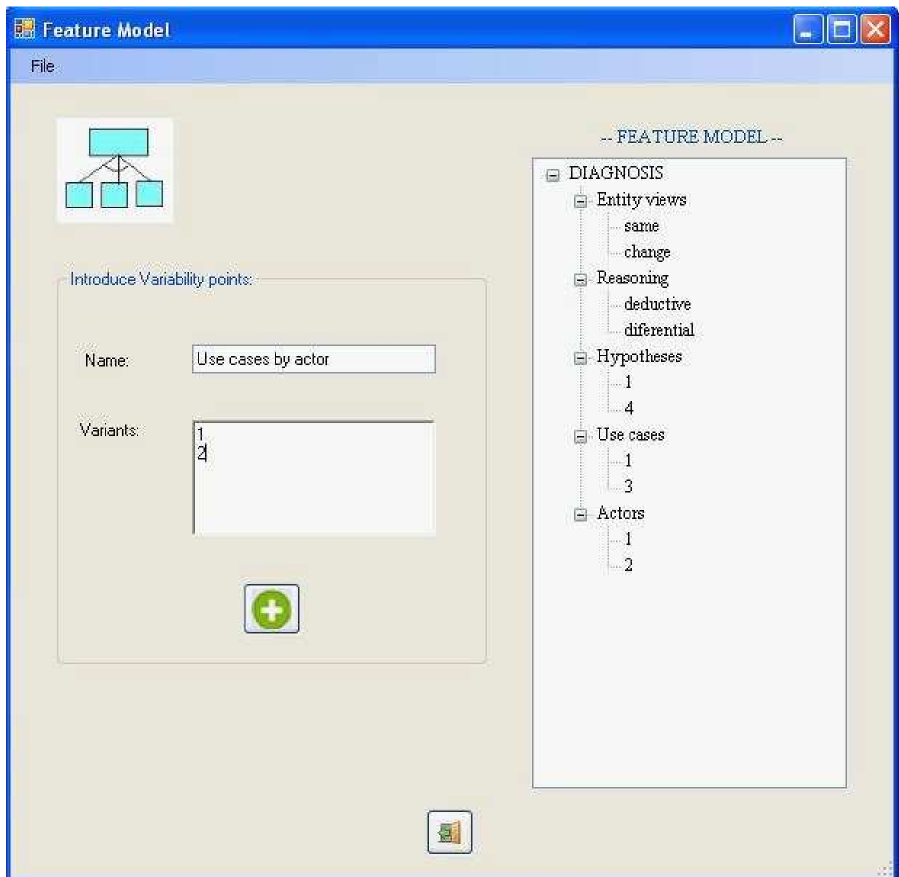


Figura 100 IGU para crear el Modelo de Características

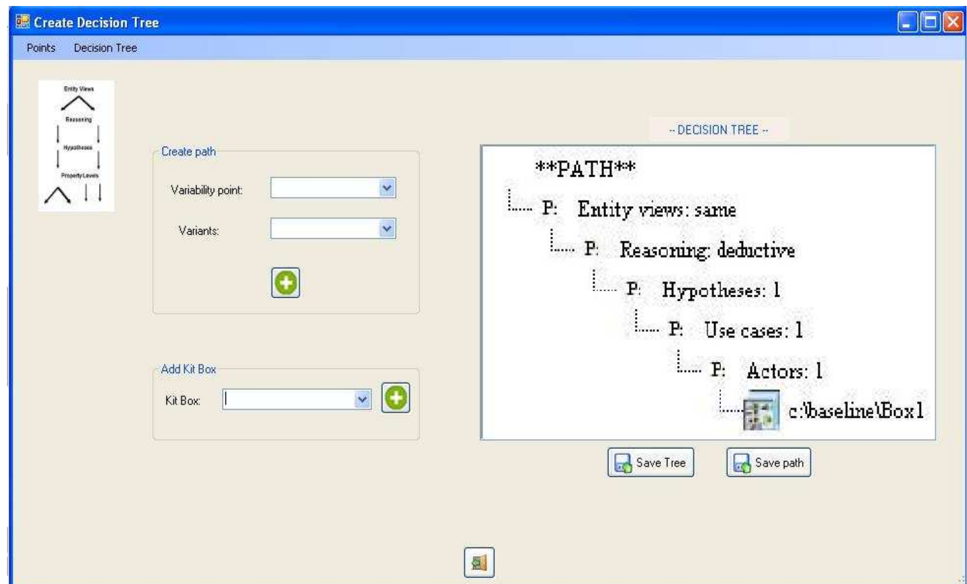


Figura 101 IGU para crear el Árbol de Decisión

10.4 ProtoBOM en la ingeniería de la aplicación

El ingeniero de la aplicación podrá obtener una aplicación ejecutable de un producto final de la LPS (i.e. un SE) al realizar el Plan de Producción.

El Plan de Producción se muestra mediante ocho tareas o actividades representadas con la notación del estándar SPEM, a través de la IGU de la **figura 102**.

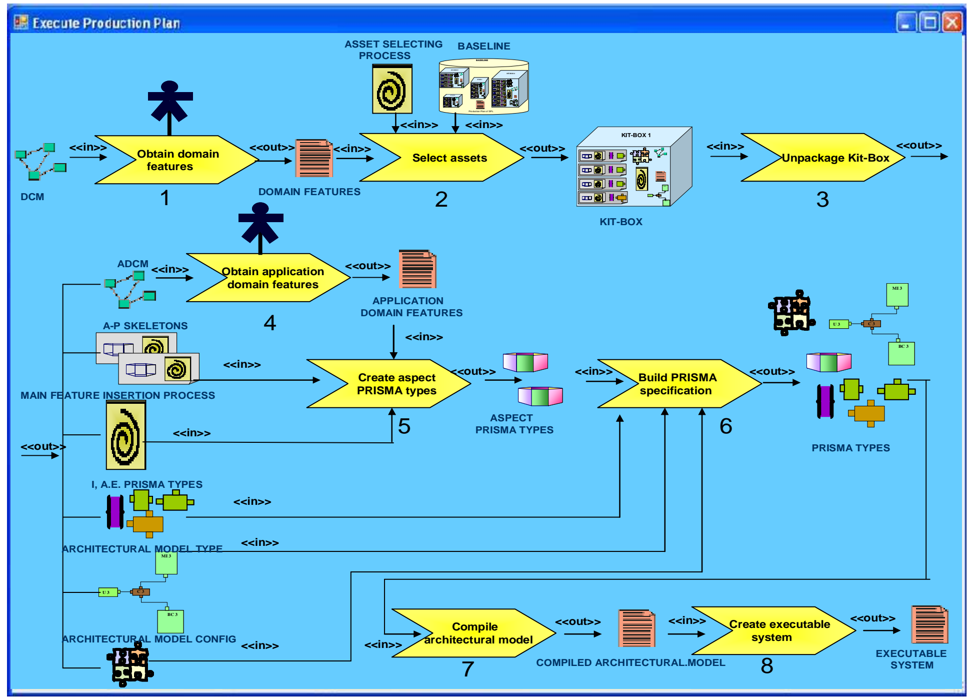


Figura 102 IGU para la ejecución del Plan de Producción

De las ocho tareas que contempla el Plan de Producción, ProtoBOM pide la intervención del usuario (con el rol del ingeniero de la aplicación) en sólo dos de esas tareas: la tarea "obtener características del dominio" y la tarea "obtener características del dominio de aplicación" (mostradas en la **figura 102** con el icono).

El plan de producción comienza cuando ProtoBOM obtiene (del ingeniero de aplicación) las características del dominio consideradas como puntos de variabilidad. El stakeholder elegirá la trayectoria ad hoc a las características del dominio elegidas por él, con lo que ProtoBOM seleccionará la correspondiente caja de la Baseline y mostrará la ubicación y el nombre de la caja correspondiente a esos puntos de variabilidad. La **figura 103** muestra la IGU.

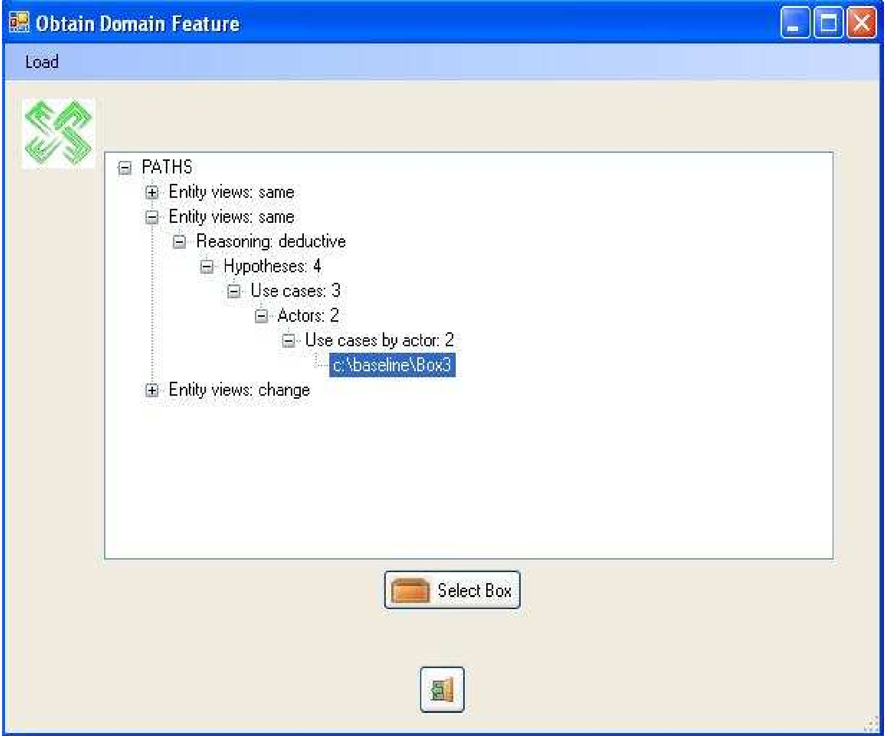


Figura 103 IGU para obtener las características del dominio

La caja seleccionada por ProtoBOM, es desempaquetada para poder utilizar cada uno de los *assets* de su contenido en forma individual.

Un asset recuperado de la caja es el "Aplication Domain Conceptual Model", el cual es utilizado para que ProtoBOM obtenga (del ingeniero de aplicación) las características del dominio de la aplicación consideradas como "*features*" de la segunda variabilidad. La **figura 104** muestra la IGU para obtener las características del dominio de aplicación, y las **figuras 105, 106 y 107** muestran respectivamente las IGU para obtener las propiedades, las hipótesis y el conjunto de reglas de derivación de dicho dominio de aplicación.

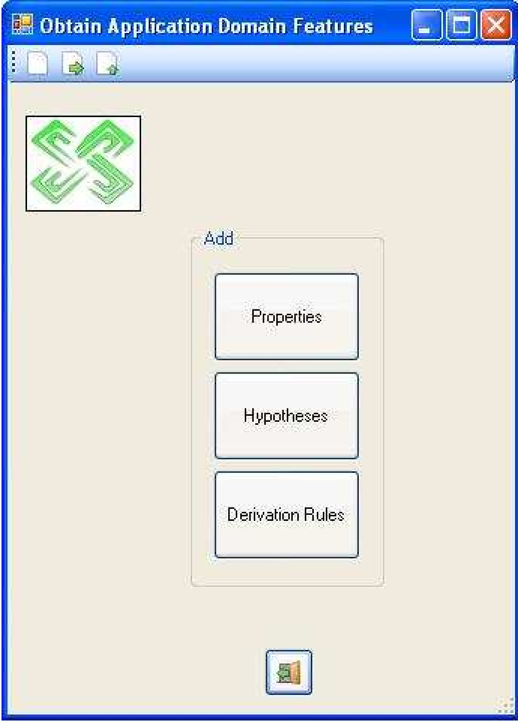


Figura 104 IGU para obtener las características del dominio de aplicación

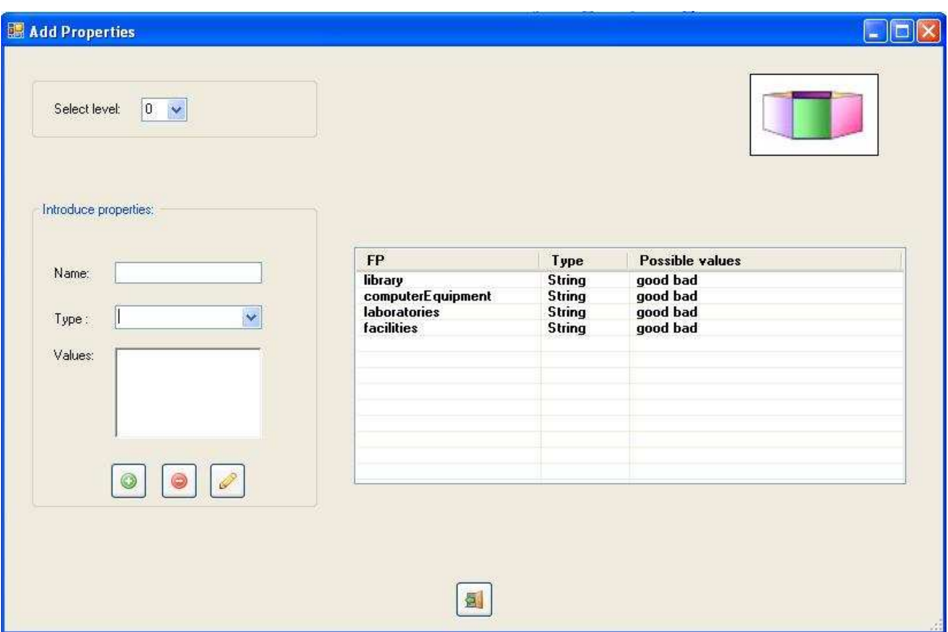


Figura 105 IGU para obtener las propiedades del dominio de aplicación

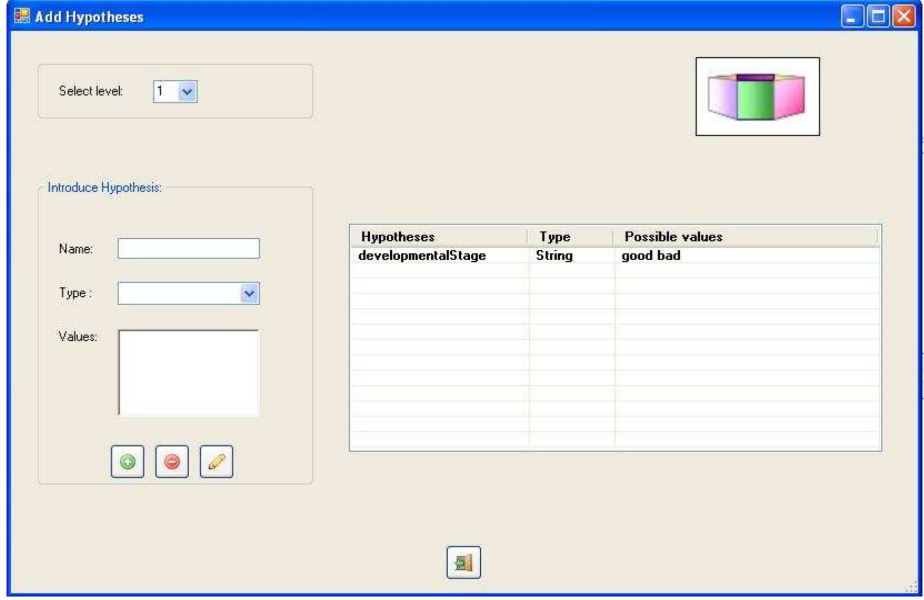


Figura 106 IGU para obtener las hipótesis del dominio de aplicación

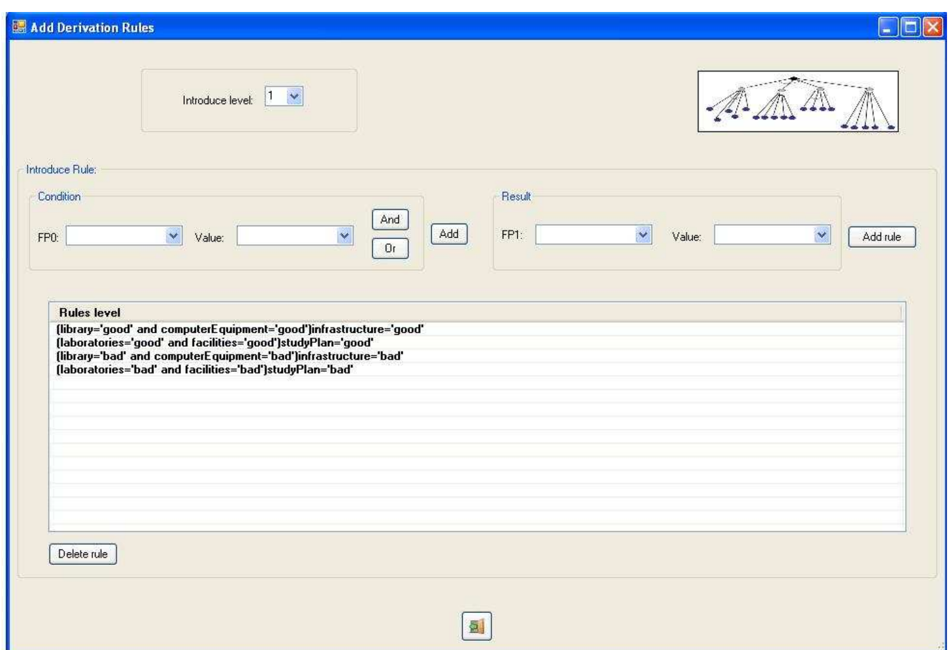


Figura 107 IGU para obtener las reglasde derivación del dominio de aplicación

Otros *assets* recuperados de la caja son los "Packaged Hybrids". Estos *assets* a su vez son desempaquetados (esqueletos aspecto, proceso de inserción de *features*, interfaces y elementos arquitectónicos) para obtener los artefactos tipo PRISMA en dos pasos (ver **figura 108**):

i) En el primer paso, ProtoBOM aplica el "Feature Insertion Process" para rellenar o decorar los "Aspect Skeletons" con los datos de las características específicas del caso de estudio (que fueron introducidas por el ingeniero), creando los aspectos tipo PRISMA.

ii) En el segundo paso, ProtoBOM toma los otros artefactos software desempaquetados: artefactos tipo PRISMA ("Interfaces", "Architectural Elements" y "Architectural Model") y la configuración del modelo arquitectónico ("Architectural Model Configuration"), con el fin de construir la especificación PRISMA.

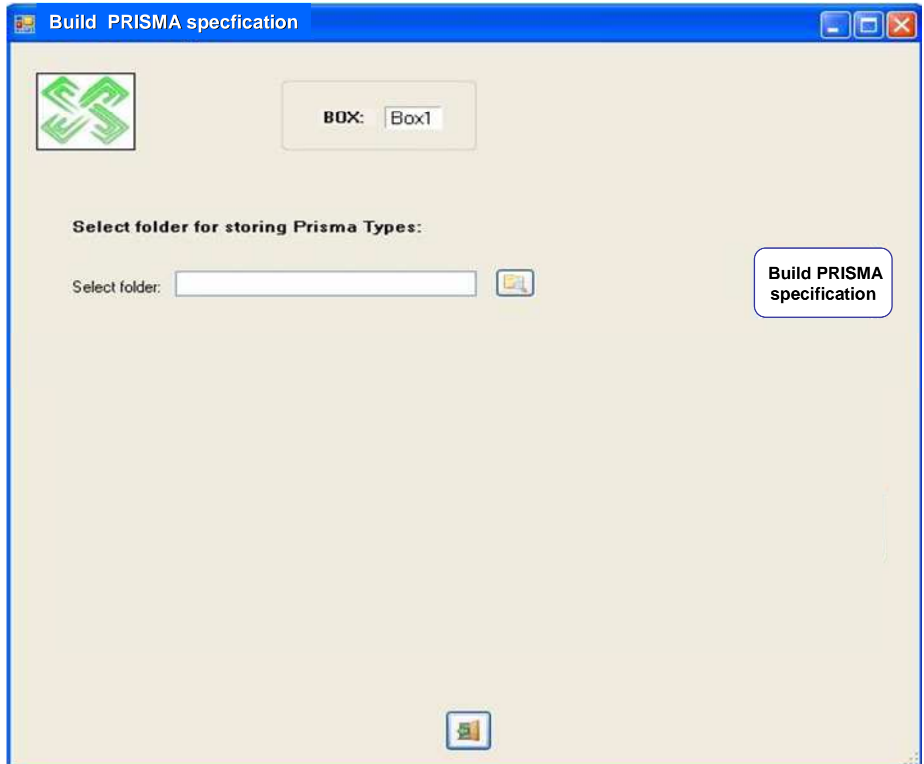


Figura 108 IGU para construir la especificación PRISMA

Como lo muestra la **figura 109**, ProtoBOM permite al ingeniero empaquetar (en una sola carpeta) todos los artefactos PRISMA de la tarea anterior, con el fin de obtener toda la información del modelo arquitectónico final. Dicho modelo (como un

documento XML) será la entrada a las herramientas PRISMA para obtener una aplicación ejecutable.

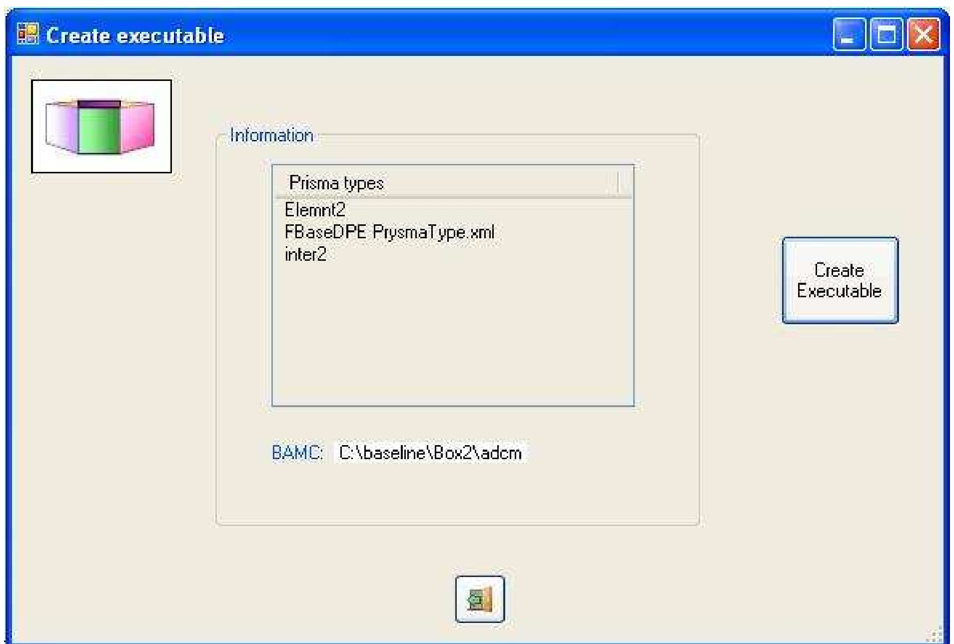


Figura 109 IGU para ingresar la especificación PRISMA a las herramientas PRISMA y crear el ejecutable del sistema final

De esta manera, con el compilador de modelos PRISMA-MODEL-COMPILER se generará automáticamente el código (en.NET, C#) del modelo arquitectónico de la tarea anterior. Finalmente, con el Middleware PRISMANET se creará el sistema final como una aplicación ejecutable.

El producto final (i.e. el sistema ejecutable) de nuestra LPS, tiene la IGU que se muestra en las **figuras 110 y 111**. La **figura 110** corresponde al caso de estudio del diagnóstico de programas educativos y **la figura 111** corresponde al caso de estudio del diagnóstico médico. Una vista más completa de las IGUs de los sistemas finales de estos casos de estudio, es presentado en los apéndices C y D de esta tesis, respectivamente.

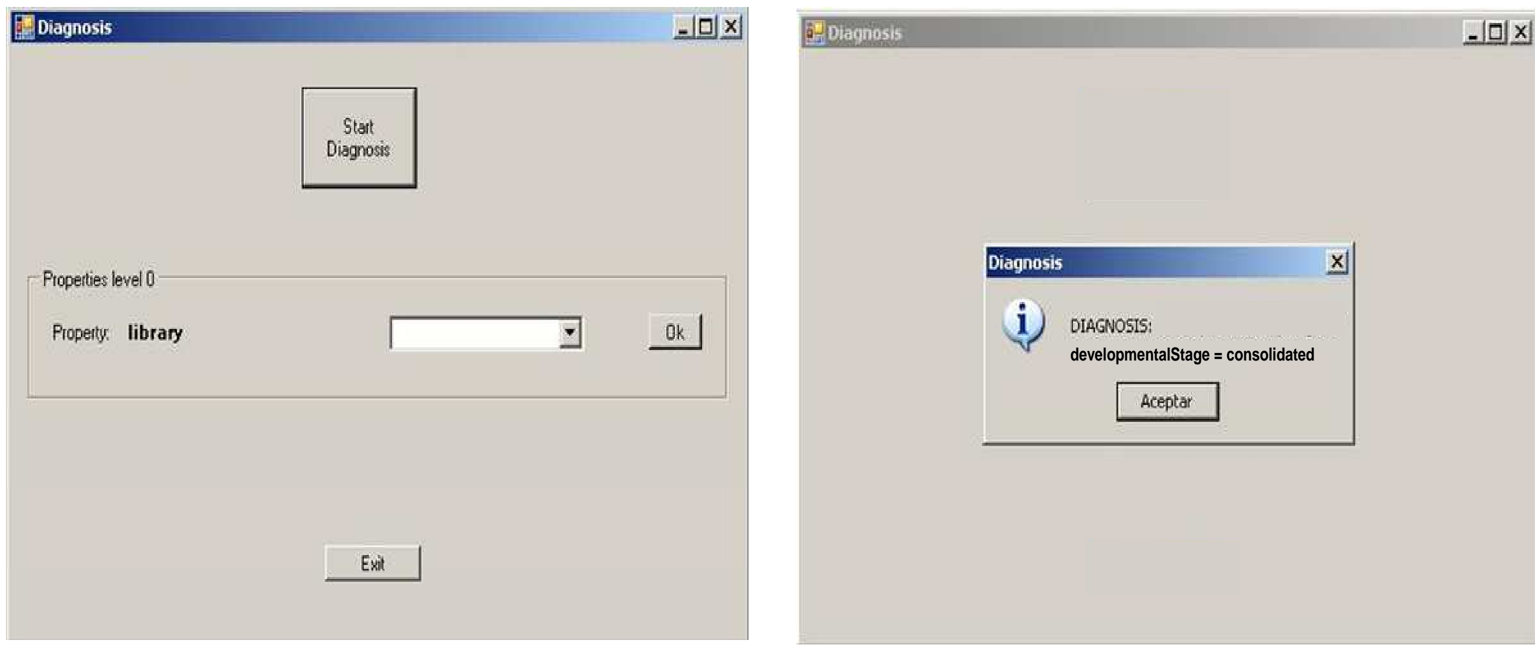


Figura 110 IGUs inicial y final del sistema ejecutable (caso diagnóstico de programas educativos)

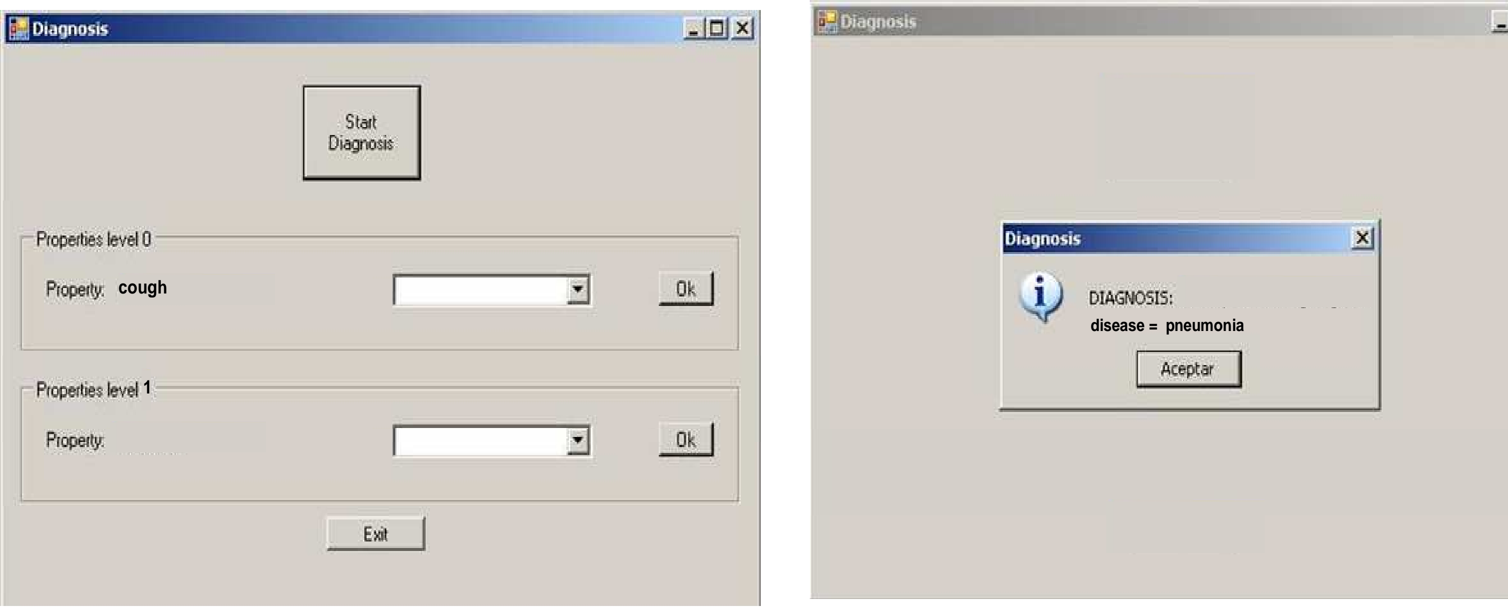


Figura 111 IGUs inicial y final del sistema ejecutable (caso diagnóstico médico)

10.5 Conclusiones

ProtoBOM es una herramienta que ofrece las siguientes ventajas:

- construir aplicaciones de sistemas expertos, de una forma sencilla y con una interfaz amigable e intuitiva, utlizando un lenguaje cercano al dominio, sin necesidad del conocimiento de lenguajes de programación.
- reutilizar *assets* y
- obtener una aplicación distribuida (al utilizar servicios Web para acceder a la Baseline).

Con ello se obtiene una reducción en los costes y el "time to market", así como ofrecer una herramienta para el desarrollo de aplicaciones software de una manera rápida y sencilla, minimizando esfuerzos en la producción del software.

CAPÍTULO 11

IMPLEMENTACIÓN DE LA APROXIMACIÓN BOM-LAZY

La imaginación gobierna al mundo.
Sir Winston Leonard Spencer Churchill (1874-1965)
Estadista, historiador, escritor y orador británico

11.1 Introducción

En este capítulo se presenta parcialmente la aproximación BOM-LAZY mediante técnicas de transformación de modelos, implementadas en QVT-Relations.

La estructura de este capítulo es la siguiente: En la sección 2 se comentan las características principales de la implementación parcial de BOM-LAZY en QVT. En la sección 3 se describe brevemente el desarrollo de la herramienta. La sección 4 presenta las conclusiones de este capítulo.

11.2 Características de la implementación BOM-LAZY en QVT

La arquitectura de la implementación de la aproximación BOM-LAZY es mostrado en la **figura 112**.

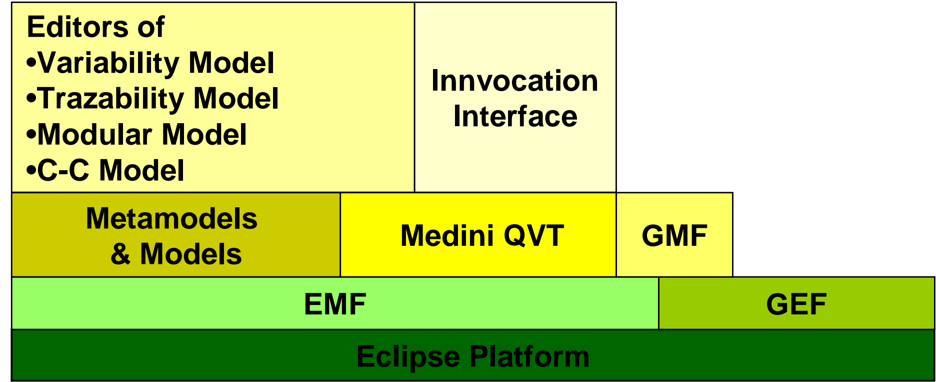


Figura 112 Arquitectura de la implementación BOM-LAZY

Esta herramienta realiza la transformación de modelos de vista modular a modelos de vista componente-conector, que se corresponden con la transformación T1 propuesta en BOM para el manejo de la primera variabilidad (variabilidad del dominio).

La transformación se efectuó utilizando QVT-Relations. La implementación de dicha transformación se realizó en Eclipse 3.3, empleando el *framework* de modelado EMF 2.3 para la definición de los metamodelos, modelos e instancias de los modelos.

Como motor de transformaciones se ha empleado medini QVT [MediniQVT, 2007], un motor de transformaciones QVT-Relations de código abierto. Sobre dicho motor se implementó una interfaz de invocaciones específicas que permite ejecutar transformaciones con *n* dominios distintos, tal y como se muestra en la **figura 113**.

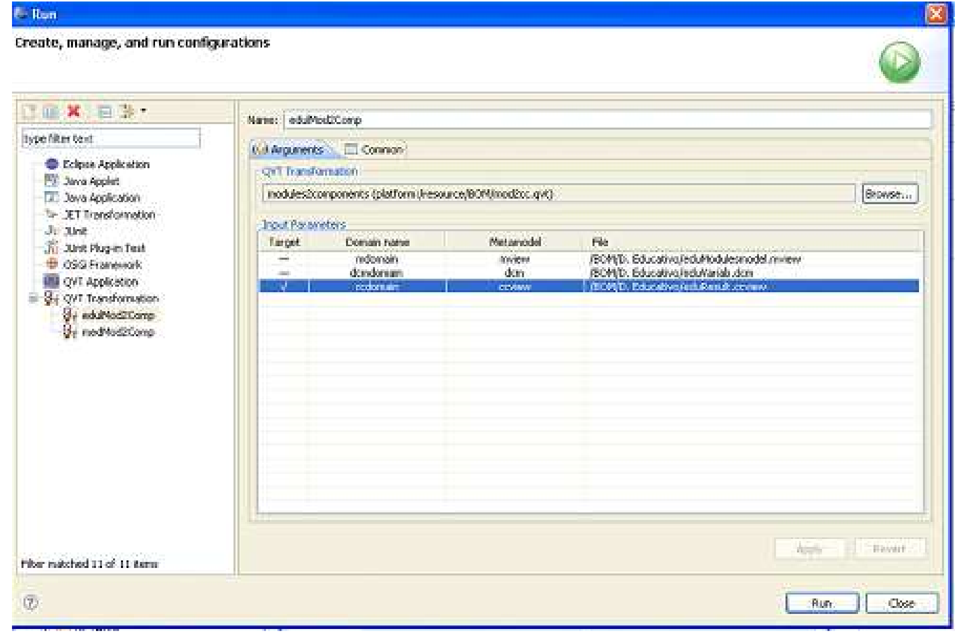


Figura 113 **IGU de las invocaciones para ejecutar las transformaciones**

11.3 Desarrollo de la herramienta

Para el desarrollo de la herramienta que implementa la transformación T1 de la aproximación BOM-LAZY se realizaron las siguientes acciones:

- construir el metamodelo modular (ver **figura 114**),
- construir el metamodelo de componentes-conectores (ver **figura 115**),
- construir el modelo modular (ver **figura 116**)
- construir el modelo de variabilidad MCD (ver **figura 117**),
- codificar las QVT-Relations (ver **figura 118**),
- definir las instancias del modelo de la variabilidad del dominio, de cada caso de estudio (ver **figuras 123 y 127**),

La transformación T1 (del modelo modular al de componente-conector) se define usando QVT-Relations, tomando en cuenta la arquitectura genérica, la instancia del MCD y la configuración de la arquitecutura base. Al ejecutarse dichas transformaciones, son generados:

- el modelo específico de componentes-conectores. En las **figuras 120 y 124** se muestan los modelos de los casos de estudio del diagnóstico de programas educativos, y del diagnóstico médico respectivamente,

- las trazas entre el modelo modular y el modelo de componentes-conectores, de cada caso de estudio, y que son mostrados en las **figuras 122 y 126**.

Metamodelo Modular

Figura 114 IGU del meta modelo Modular

Metamodelo Componente-Conector

Figura 115 IGU del metamodelo Componente-Conector

Modelo Modular de los Sistemas Expertos de Diagnóstico

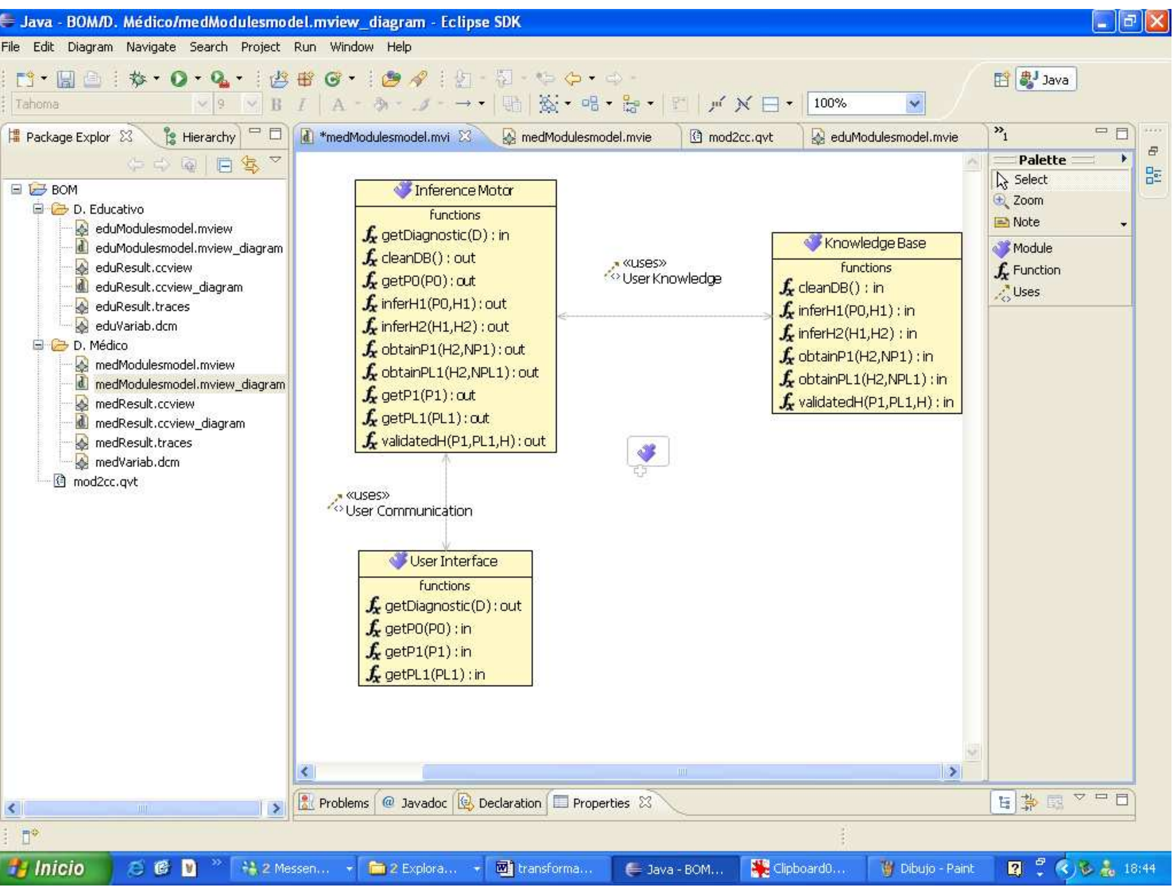


Figura 116 IGU del modelo modular de los SED

Árbol correspondiente al Modelo Modular de los Sistemas Expertos de Diagnóstico

Figura 117 IGU del árbol del modelo modular de los SE

Implementación de las QVT-Relations

(El código completo de las QVT-Relations se presenta en el apéndice E de la tesis)

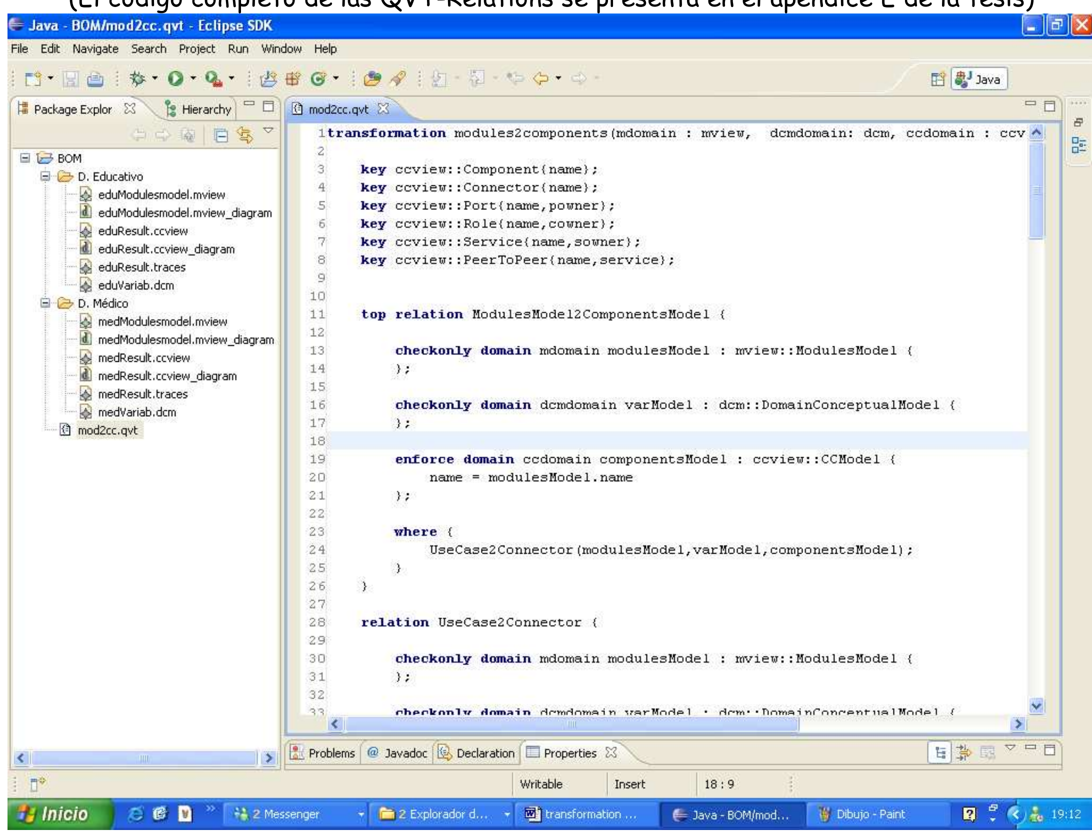


Figura 118 IGU de las QVT-Relations Caso:Diagnóstico de programas educativos

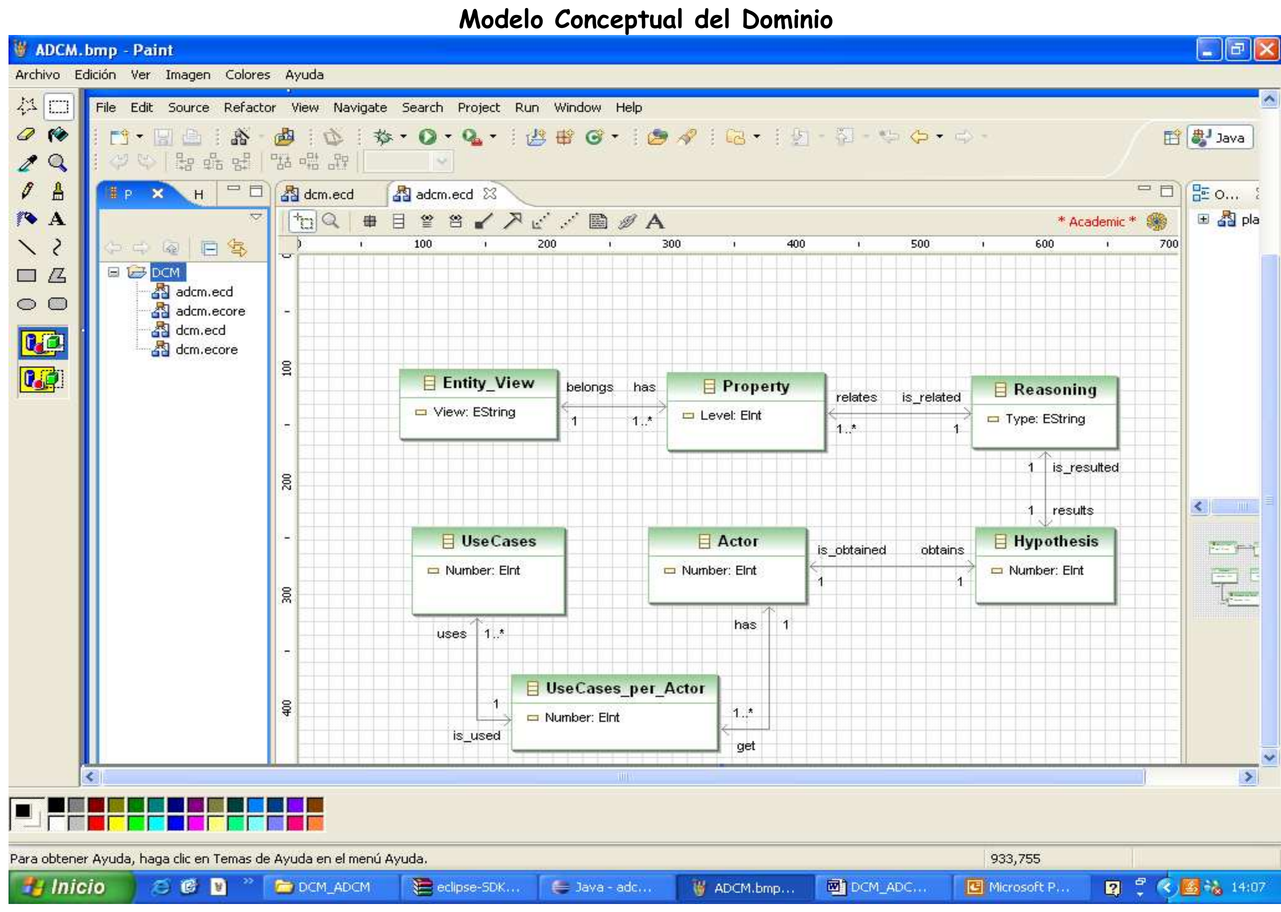


Figura 119 IGU del Modelo Conceptual del Dominio del Diagnóstico

Caso:Diagnóstico de programas educativos
Modelo Componente-Conector

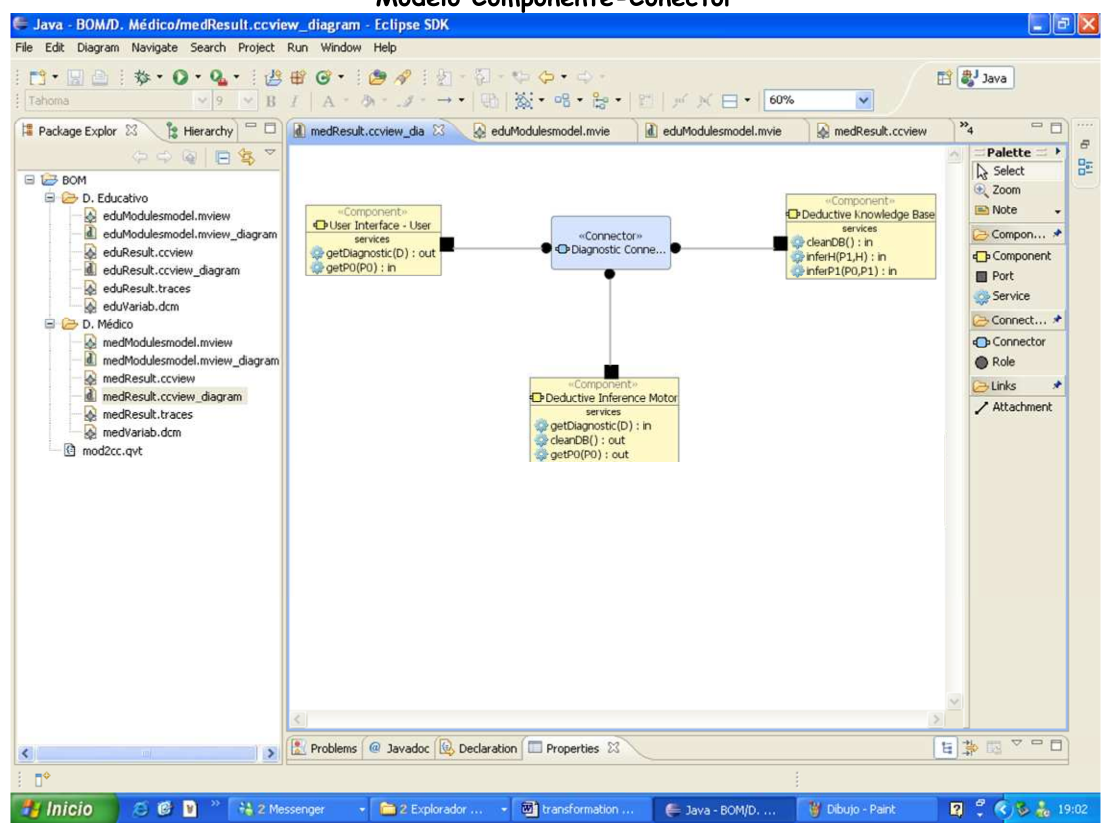


Figura 120 IGU del modelo C-C del SE del diagnóstico de programas educativos

Caso:Diagnóstico de programas educativos

Árbol correspondiente al Modelo Componente-Conector

Figura 121 IGU del árbol del modelo C-C del SED del diagnóstico de programas educativos

Caso:Diagnóstico de programas educativos

Trazas entre los Modelos Modular y Componente-Conector

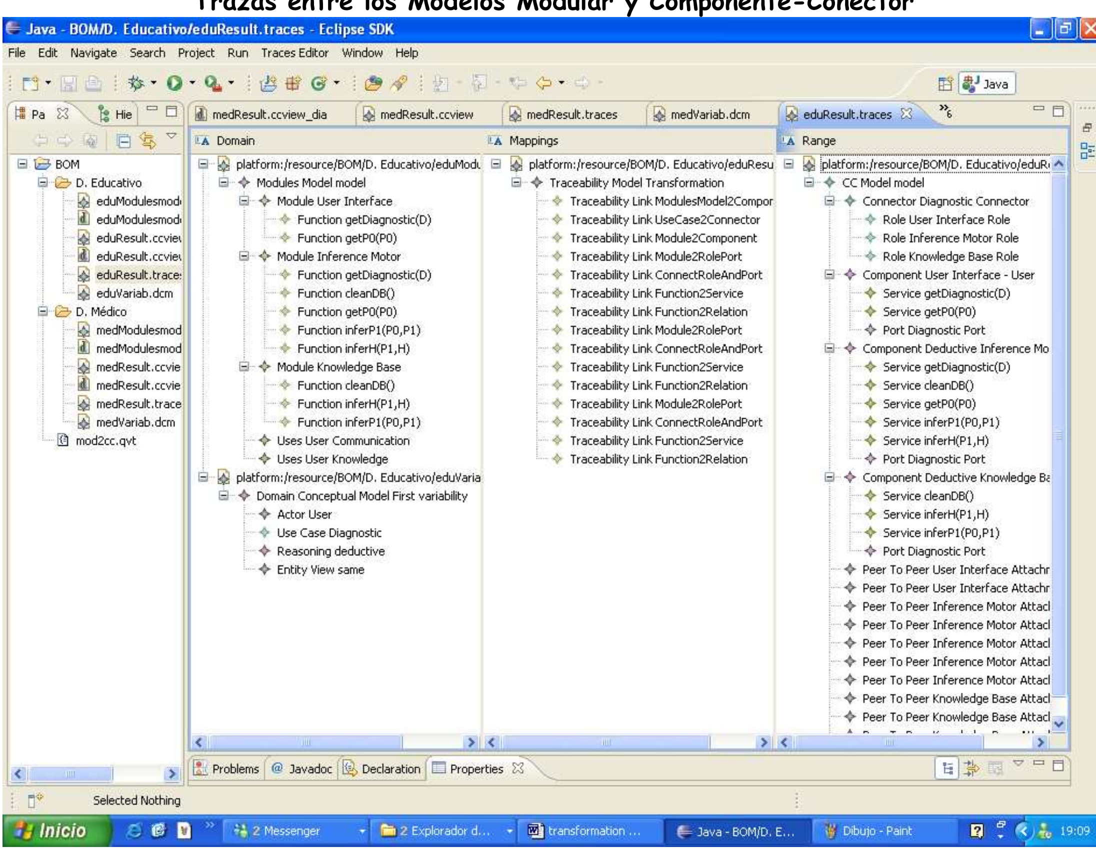


Figura 122 IGU de las trazas entre los modelos modular y C-C del SED del diagnóstico de programas educativos

Caso:Diagnóstico de programas educativos

Instancia del Modelo Conceptual del Dominio

Figura 123 IGU de la instancia del DCM del diagnóstico de programas educativos

Caso:Diagnóstico médico

Modelo Componente-Conector

Figura 124 IGU del árbol del modelo C-C del SED del diagnóstico médico

Caso:Diagnóstico médico

Árbol correspondiente al Modelo Componente-Conector

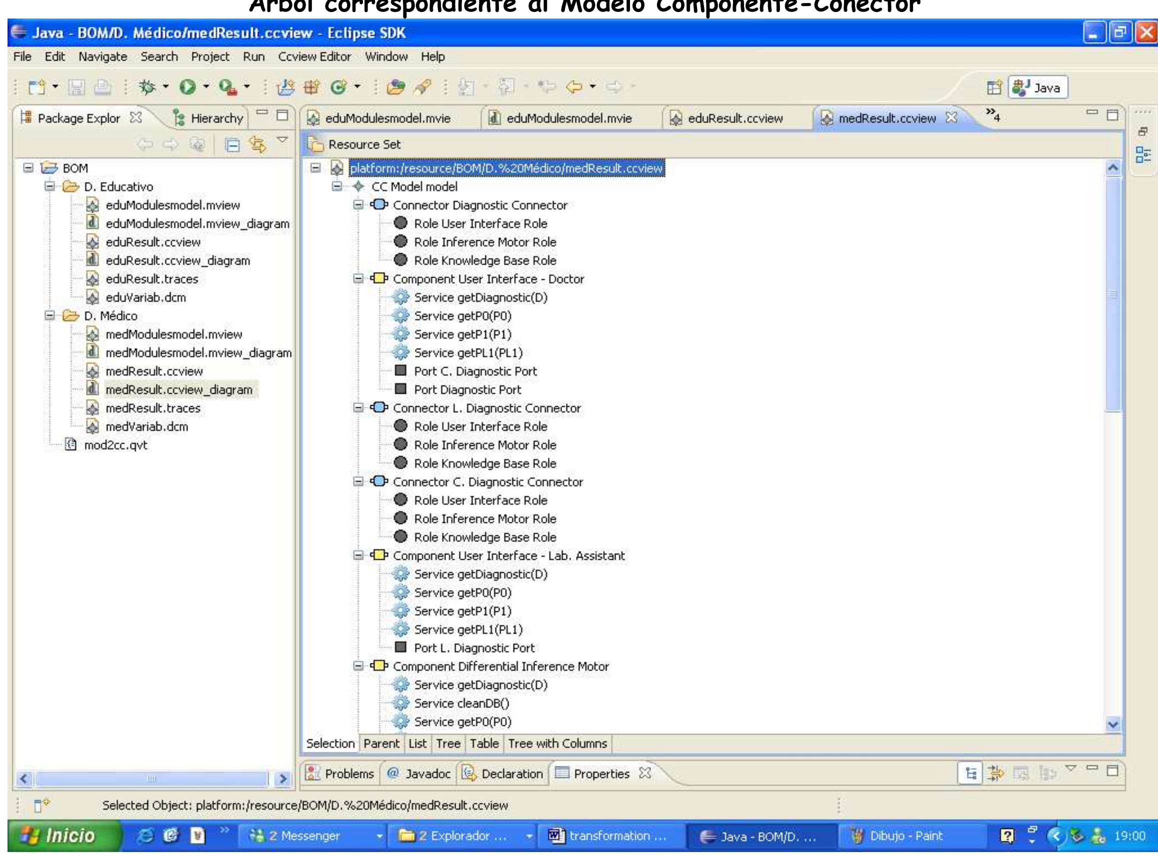


Figura 125 IGU del árbol del modelo C-C del SED del diagnóstico médico

Caso:Diagnóstico médico

Trazas entre los Modelos Modular y Componente-Conector

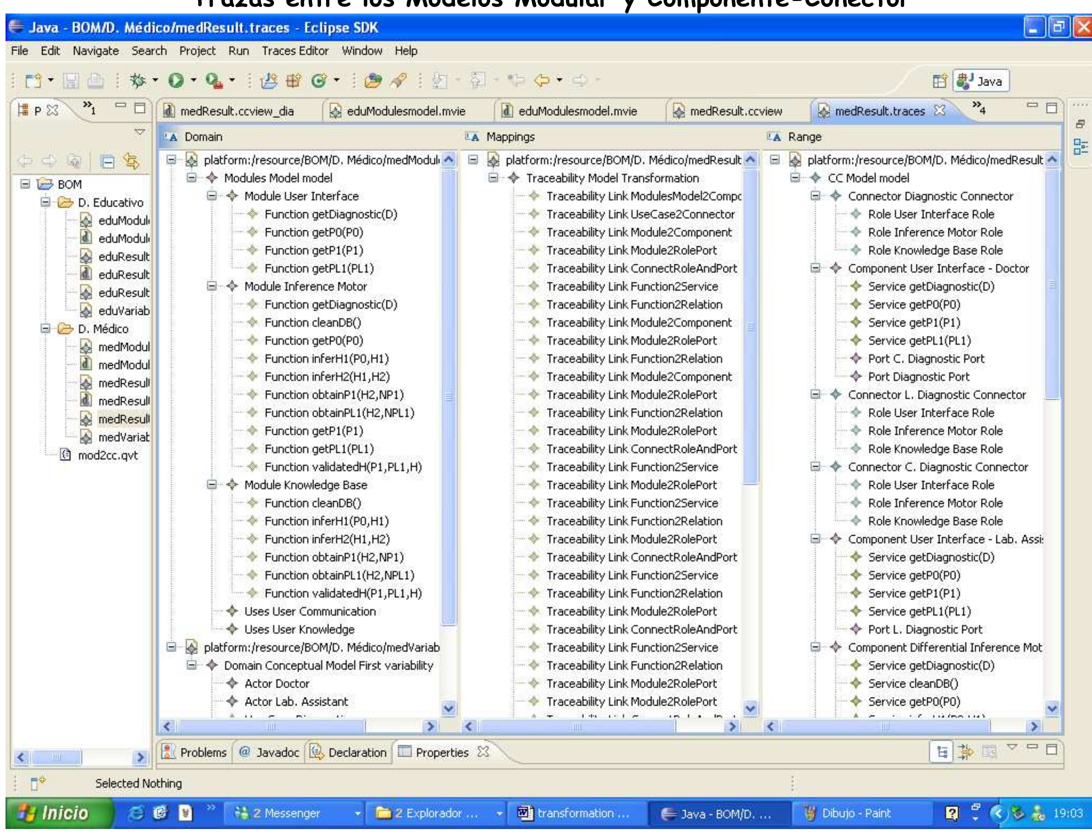


Figura 126 IGU de las trazas entre los modelos modular y C-C del SED del diagnóstico médico

Caso:Diagnóstico médico

Intancia del Modelo Conceptual de Dominio

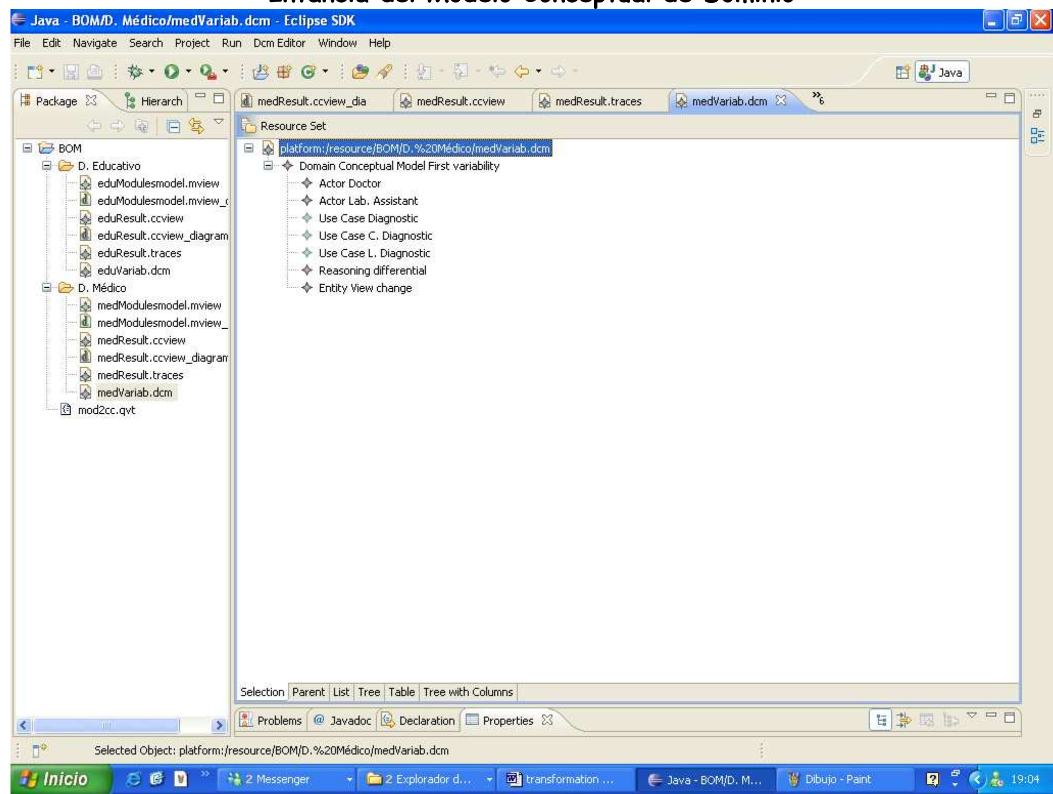


Figura 127 IGU de la instancia del DCM del diagnóstico médico

11.4 Conclusiones

Con la implementación de la transformación T1 usando QVT-Relations, es posible dar cuenta de la complejidad presentada en los capítulos 7 y 8 de esta tesis, para llevar a cabo la transformación de modelos.

Además, el haber realizado esta implementación permite mostrar la aproximación BOM-LAZY y comprobar que es posible su realización.

PARTE VI. CONCLUSIONES

CAPÍTULO 12

CONCLUSIONES Y TRABAJO FUTURO

La ciencia avanza a pasos, no a saltos.
Thomas Macaulay (1800-1859). Historiador y político británico.

En este capítulo se presentan las conclusiones más importantes derivadas de esta investigación, así como algunas posibles ideas para realizar en el futuro, con el fin de enriquecer BOM.

12.1 Conclusiones generales

Esta tesis presenta una aproximación denominada Baseline-Oriented Modeling (BOM): un *framework* que genera aplicaciones en un dominio específico con arquitecturas PRISMA utilizando técnicas de MDA y LPS.

Para ilustrar BOM, se eligió al dominio de los Sistemas Expertos de diagnóstico.

Conforme a un análisis de campo realizado, se concluyó que esta clase de sistemas es complejo dado que varían tanto en su comportamiento como en su estructura final. Esta situación produce varias arquitecturas base en la LPS, las cuales comparten una sola arquitectura genérica. Además existe otra variabilidad que decora diferentemente a las arquitecturas base con las características del dominio de aplicación.

En consecuencia, en BOM se realizó un tratamiento de la variabilidad en dos niveles, asociados a dos tipos de LPS. La primera variabilidad implica la LPS1 formada por todas las arquitecturas base que comparten la arquitectura de los SE. La segunda variabilidad implica las LPS2 de la aplicación en un campo específico, i.e. los SE como productos finales de la LPS.

En este contexto, mediante técnicas ad-hoc utlizadas en BOM fueron gestionadas dichas variabilidades. BOM captura las variabilidades del dominio y del dominio de aplicación en instancias de dos modelos conceptuales separados. Las técnicas de árbol de decisión y de FOM explotan dicha información en las fases de la ingeniería del dominio y de la ingeniería de la aplicación, con el fin de obtener una aplicación específica por medio de técnicas de LPS y de transformación de modelos (i.e. MDA). Los modelos finales son compilados y es generado el código objeto (en NET, C#) obteniendo una aplicación ejecutable.

Asimismo se modeló la LPS del *framework* BOM en dos fases: la ingeniería del dominio (donde se construyen todos los *assets* y el conocimiento de cómo construir un SE, depositados en la Baseline) y la ingeniería de la aplicación (donde se realiza el plan de producción de la LPS).

Pero el tamaño de la Baseline determina la conveniencia de utilizar dos aproximaciones diferentes. Si la Baseline es pequeña, entonces es conveniente utilizar la aproximación BOM-EAGER, que utiliza técnicas de árbol de decisión para seleccionar de la Baseline los assets. Si la Baseline es grande, es mejor utilizar la aproximación BOM-LAZY, que utiliza técnicas de transformación de modelos para construir los productos de la LPS.

La aproximación BOM pretende mejorar el desarrollo de los SE de la siguiente manera:

- Usando las ventajas de los SE, al separar los procesos de inferencia de la información del conocimiento de un dominio específico, y contemplar varias estrategias de razonamiento para resolver un problema aplicando la manera más eficiente.
- Construyendo sistemas de una manera simple usando ontologías del diagnóstico y de los dominios de aplicación. De esta manera, se ofrece un acercamiento al lenguaje específico del dominio (DSL) del problema, facilitando la interacción con el usuario.
- Aplicando las técnicas de LPS al construir un diseño que compartan todos los miembros de una familia de programas. De esta manera, un diseño específico puede ser usado en diferentes productos, reduciendo los costos, los tiempos de producción, el esfuerzo y la complejidad.
- Construyendo las arquitecturas de la línea de productos en el marco de PRISMA, integrando componentes y aspectos reutilizables, facilitando su mantenimiento y complejidad.
- Aplicando técnicas de MDA para implementar los sistemas sobre diferentes plataformas, y transformarlos para obtener una aplicación ejecutable.

- Desarrollando sistemas independientes de plataforma, abordados desde la perspectiva del problema y no de la solución, lo cual provee generalidad en la aproximación desarrollada y aplicabilidad en diferentes dominios.
- Ofreciendo una funcionalidad conjunta y coordinada de todos los espacios tecnológicos contemplados en BOM.

Con base en lo anteriormente expuesto, se puede considerar que sí es factible generar de forma automática aplicaiones, basadas en líneas de producto, utilizando técnicas de MDA (hipótesis planteada en esta tesis).

Se puede concluir que las características de BOM, son las siguientes:

- La variabilidad es manejada a nivel de modelos (no a nivel de programas).
- La variabilidad de los sistemas es modelada usando modelos de la varibilidad independientes de sus modelos funcionales. Los DSLs para expresar la variabilidad son adecuados al dominio, en lugar de agregarlos a los modelos funcionales (UML, ADLs).
- La variabilidad es manejada por dos tipos de variabilidad ortogonales: una dada por las características del dominio (diagnóstico), y otra dada por las características del dominio de la aplicación.
- Varios espacios tecnológicos son integrados para cubrir la complejidad del problema. Estas son las tendencias actuales de la ingeniería de software.
- BOM implementa una aproximación genérica para el desarrollo de la LPS, que puede ser aplicada en diferentes dominios, dominios de aplicación, tipos de sistemas software y plataformas destino (PSM), ya que se mantiene el mismo plan de producción de la LPS. Sin embargo, en la fase de la ingeniería del dominio deberán ser creados los mismos *assets* pero con las particularidades *ad-hoc* al caso. Asimismo, algunos artefactos software pueden ser reutilizados o bien modificados para dominios que comparten algunas características comunes.

Dichas características permiten que al utilizar BOM se cuente con las siguientes ventajas:

- **Generalidad**.- BOM es una aproximación genérica que puede ser utilizada para desarrollar varios tipos de sistemas, en diversos dominios.
- **Abstracción**.- La semántica de BOM contempla altos niveles de abstracción, utilizando modelos.
- **Amigabilidad**.- El uso de la herramienta BOM por el desarrollador de las aplicaciones es muy sencilla, dado que solamente debe de ingresar en BOM los datos de la variabilidad del dominio y del dominio de aplicación, para obtener una aplicación ejecutable.
- **Automatización**.- Con BOM se obtiene una aplicación ejecuatable de forma automática.
- **Estandarización**.- BOM se basa en varios estándares de la OMG.

Finalmente, se concluye que BOM podrá ser una solución genérica aplicable a diversos tipos de sistemas software, si se satisfacen las siguientes condiciones:

- En el dominio existe una arquitectura genérica que se pueda captar a través de un modelo modular.
- Dicha arquitectura tiene variantes expresadas como puntos de variabilidad.
- Podemos construir una Baseline que almacene activos y el "saber hacer"
- La obtención de un producto se pueda realizar por selección de características plasmadas en un modelo de características.
- Las arquitectura base pueden ser especificadas "formalmente", i.e. con una semántica y una sintaxis formales.

12.2 Contribuciones

Las principales contribuciones de esta tesis en los espacios tecnológicos contemplados en BOM son:

- **MDA**
    - Proponer una aproximacion dirigida por modelos orientada a la variabilidad para el desarrollo de sistemas software.

- **Línea de Productos Software**
    - Ofrecer un marco conceptual para identificar y definir la variabilidad en la fase de la ingeniería del dominio, y para gestionar e implementar la variabilidad en la ingeniería de la aplicación.
    - Manejar la variabilidad a través de dos tipos de variabilidad ortogonales: la del dominio y la del dominio de aplicación.
    - Presentar un plan de producción genérico para diversos tipos de sistemas software.
    - Implementar una herramienta CASE (prototipo) para el desarrollo de aplicaciones de una LPS durante todo su ciclo de vida.
    - Implementar la gestión de la variabilidad mediante QVT-Relations como técnica de transformación de modelos.

- **Sistemas Expertos**
    - Generar SED de una manera sencilla con solo introducir la información de de la variabilidad, utilizando DSL.
    - Ofrecer una LPS para SED, reduciendo complejidad, costes, tiempo y esfuerzos.

- **Herramientas de PRISMA**
    - Incorporar directamente al compilador de modelos PRISMA MODEL COMPILER el modelo arquitectónico (especificado en el

LDA de PRISMA sobre documentos XML) para generar el código que será ejecutado sobre el Middleware PRISMA-NET.

Relacionado y derivado de este trabajo de investigación, se han realizado las siguientes publicaciones:

1. **Cabello Ma. Eugenia**, Andrade María, Edwards Arthur, Flores Carlos Alberto, Macías Salvador, Cortés Miguel Ángel, Cruz Miguel Ángel y Magaña Héctor Homero. "SETI: Sistema Experto para la Tutoría Individualizada". En memorias del XVI Congreso Nacional y III Congreso Internacional de Informática y Computación de la ANIEI-CNCIIC´04, Tepic, Nayarit, México, Oct. 2004, 6 págs. ISBN: 970-36-0155-3.

2. **Cabello Ma. Eugenia**, Alí Nour, Pérez Jennifer, Ramos Isidro y Carsí José Ángel. "DIAGMED: Un modelo arquitectónico para el DIAGnóstico MÉDico". En memorias de las IV Jornadas de Trabajo DYNAMICA, Archena, Murcia, España, Nov. 2005, pp. 1-4.

3. **Cabello Ma. Eugenia**, Ramos Isidro, and Carsí José Ángel. "An architectural model for medical diagnosis". In Proceedings of the International Association for Development of the Information Society International Conference: Applied Computing - IADIS´06, San Sebastian, Spain, Feb. 2006, pp. 700-701, ISBN 972-8924-09-7. (poster).

4. **Cabello Ma. Eugenia**, Costa Cristóbal, Ramos Isidro, and Carsí José Ángel. "Generic architecture of an expert system for diagnosis". In Proceedings of the 10 th WSEAS International Conference on Computers, Athens, Greece, Jul. 2006, pp. 759-764, ISSN 1790-5117.

5. **Cabello Ma. Eugenia,** Ramos Isidro y Carsí José Ángel. "Uso de un modelo arquitectónico de componentes, aspectos y reflexión en sistemas de diagnóstico". Informe técnico DSIC-II/11/06, Universidad Politécnica de Valencia, España, Jul. 2006, pages 241.

6. **Cabello Ma. Eugenia**, Costa Cristóbal, Ramos Isidro, and Carsí José Ángel. "Component-Based and Aspect-Oriented Architectural Model of a Diagnostic Expert System". Journal of the WSEAS Transactions on Information Science and Applications, Issue 10, Volume 3, Oct. 2006, pp. 1901-1908, ISSN 1790-0832, ISBN 960-8457-47-5.

7. **Cabello Ma. Eugenia**, Costa Cristóbal y Ramos Isidro. "Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de

diagnóstico". En memorias del XIX Congreso Nacional y V Congreso Internacional de Informática y Computación del ANIEI-CNCIIC ´06, Chiapas, México, Oct. 2006. ISBN 970-31-0751-6.

8. Limón Rogelio, **Cabello Ma. Eugenia**, and Ramos Isidro, "Establish Relations among Software Architecture Views through MDA for SPL". In Proceedings of the 14th International Congress on Computer Science Research-CIICC'07, Veracruz, México, pp- 175-187, Nov. 2007, ISBN 13 978-970-95771-0-5.

9. **Cabello Ma. Eugenia** and Ramos Isidro. "A Generic Solution for the Construction of Diagnostic Expert Systems Based on Product Lines". In Proceedings of the International Conference of Health Informatics-HEALTHINF´08, Madeira, Portugal, Vol. 2, pp. 237-246, Jan. 2008, ISBN 978-989-8111-16-6.

10. **Cabello Ma. Eugenia** and Ramos Isidro. "Variability Management in Product Lines for Decision Support Systems". In Proceedings of the 10 th. International Conference on Enterprise Information Systems-ICEIS´08, Barcelona, Spain, Vol. 2, pp. 49-56, Jun. 2008, ISBN: 978-989-8111-38-8.

11. **Cabello Ma. Eugenia,** Gómez María, LLavador Manuel, and Ramos Isidro. "ProtoBOM: a *framework* that semi-automatically generates Expert Systems based on Software Product Lines". Technical report: DSIC II/02/08, Universitat Politécnica de Valéncia, April 2008, pages 68.

12. **Cabello Ma. Eugenia** y Ramos Isidro. "Análisis y diseño de un generador automático de sistemas de diagnóstico basado en líneas de producto". Reporte técnico: DSIC II/07/08, Universitat Politécnica de Valéncia, Jul. 2008, pages 236.

13. **Cabello Ma. Eugenia** and Ramos Isidro. "Model-Driven Development of Decision Support Systems: tackling the variability problem". 17 th. International Conference of Information Systems Development-ISD´08, Paphos, Cyprus, Ago. 2008. In Information Systems Development: Towards a Service Provision Society, LNCS, Springer-Verlag.

14. **Cabello Ma. Eugenia** and Ramos Isidro. "Expert Systems development through Software Product Lines techniques". 17 th. International Conference of Information Systems Development-ISD´08, Paphos, Cypros, Ago. 2008. In Information Systems Development: Towards a Service Provision Society, LNCS, Springer-Verlag.

15. **Cabello Ma. Eugenia** and Ramos Isidro. "The Baseline: the milestone of Software Product Lines for Expert Systems automatic development". In proceedings of the 9 th. Mexican International Conference on Computer Science, Baja California, México, Oct. 2008, pp. 44-51, IEEE Computer Society, ISBN: 978-0-7695-3439-8, ISSN: 1550-4069.

16. **Cabello Ma. Eugenia,** Ramos Isidro, Gómez Abel, and Limón Rogelio. "Baseline-Oriented Modeling: an MDA approach based on Software Product Lines for Expert Systems development". 1st Asia Conference on Intelligent Information and Database Systems-ACIIDS´09, Vietnam, April 2009, IEEE Computer Society (aceptado).

12.3 Trabajo futuro

En el futuro, se tiene contemplado realizar las siguientes acciones:

En primer lugar, y por el hecho de haber establecido la aproximación reactiva sobre el uso de las LPS, se podrá extender el análisis del campo a otros dominios de aplicación, con el fin de incrementar la variabilidad y la Baseline. De esta manera, la LPS podrá ofrecer más productos.

Asimismo, serán contemplados otros dominios (como el de interpretación y el de predicción). Con dichos dominios será aplicado BOM, realizando un nuevo proceso de desarrollo de la LPS en la fase de ingeniería del dominio, y aplicando el mismo plan de producción realizado en esta tesis en la fase de la ingeniería de aplicación.

Adicionalmente, se tiene planteado utilizar el *framework* BOM en otros ámbitos diferentes a las arquitecturas de los SE, como las arquitecturas Pipeline (p.e. los compiladores), o bién las arquitecturas software orientadas a servicios (p.e. una LPS para gestionar agencias de viajes), así como contemplar LPS de actualidad como el software empotrado, entre otros.

En cuanto a la implementación, con ProtoBOM (prototipo de la aproximación BASE-EAGER) se pretende mejorar los DSL en los modelos conceptuales del dominio y del dominio de aplicación para que la interacción con los *stakeholders* sea más sencilla. Y con la aproximación BASE-LAZY, se realizará la implementación de la segunda transformación de modelos usando QVT-Relations.

Además, se pretende convertir el prototipo ProtoBOM en una herramienta (BOM tool), incorporando la obtención de las características de la 1ª. Y la 2ª. varibilidad a través de modelos de características al uso compilables, y realizar todo el proceso del plan de producción de forma automática.

Por otro lado, es necesario verificar BOM en casos de estudio de la vida real. Adicionalmente, el funcionamiento del proceso de producción de los productos de la LPS generados por BOM, será comparado con otras aproximaciones, siguiendo algún método sistemático de evaluación.

De este útimo punto es importante recalcar que la evaluación de BOM debe ser realizada sobre el proceso de producción de la LPS, no sobre los productos finales de la misma, dado que ello depende de la calidad de los *assets*. Por ejemplo, la calidad de un SE dependerá de la eficiencia de su motor de inferencia y de una buena base de conocimientos.

Por último, se tiene contemplado crear una herramienta que integre las aproximaciones BOM-EAGER y BOM-LAZY. El ingeniero de la aplicación podrá elegir desde el inicio, cuál de las dos aproximaciones va a utilizar. Asimismo dicha herramienta incorporará la verificación y será aplicada a casos de la vida real. Finalmente se plantea que la herramienta sea evaluada y validada con algún método sistemático en la Ingeniería del Software (i.e. "benchmarks").

BIBLIOGRAFÍA

BIBLIOGRAFÍA

[Ali et al., 2005] Ali N., Ramos I. and Carsí J. A., *Conceptual Model for Distributed Aspect-Oriented Software Architectures.* International Conference on Information Technology Coding and Computing, ITCC 2005, IEEE Computer Society, Las Vegas, NV, USA, March 2005.

[AMPLE, 2007] AMPLE: Aspect-Oriented, Model-Driven, Product Line Engineering. Specific Targeted Research Project: IST-33710. Project start date: 1 October 2006, duration: 3 years. http://www.ample-project.net

[Andrade et al., 1999] Andrade L. and Fiadeiro J., *Interconnecting Objects via Contracts*. OOPSLA´99.

[AOSD, 2001] Aspect Oriented Software Development, http://aosd.net. http://portal.acm.org/browse_dl.cfm?linked=1&part=magazine&idx=J79&coll=ACM&dl=ACM (October 2001 issue of Communications of the ACM).

[Arango et al., 1991] Arango G. y Prieto-Díaz R., *Domain Analysis Concepts and Research Directions. Domain Analysis and Software Systems Modeling.* R. Prieto-Díaz y G. Arango editores. Páginas 9-32. IEEE Computer Society Press., 1991.

[Ávila-García et al, 2006] Ávila-García, O., García A. E. Rebull V. S., y García J. L. R., *Integrando modelos de procesos y activos reutilizables en una herramienta MDA*. XI Jornadas de Ingeniería de Software y Bases de Datos JISBD'2006, Barcelona, España, Oct 2006, pp. 483-488.

[Atkinson et al., 2000] Atkinson C., Bayer J., and Muthig D. Component-based product line development:the KobrA Approach.. In Proceedings of the 1st

Software Product Line Conference (SPLC), P. Donohoe (editor), The International Series in Engineering and Computer Science 576, pp. 289-310, Denver (Colorado, USA), August 2000.

[Bachman et al., 2000] Bachman F., Bass L., Chastek G., Donohoe P. and Peruzzi F., *The Architecture Based Design Method.* Technical Report CMU/SEI-2000-TR-001, Carnegie Mellon University, USA, January 2000.

[Bachman et al., 2003] Bachman F., Goedicke M., Leite J., Nord R., Pohl K., Ramesh B., and Vilbig A. *A meta-model for representing variability in product family development*. In Proceedings in the 5th International Workshop on Product Family Engineering (PFE'5), pp. 66-80, 2003.

[Bass et al., 1997] Bass L., Clements P., Cohen S., Northrop L. M., and Withey J., *Product Line Practice Workshop Report*. Technical Report CMU/SEI-97-TR-003 (ESC-TR-97-003), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1997.

[Bass et al., 1998] Bass L., Chastek G., Clements P., Northrop L. M., Smith D., and Withey J., *Second Product Line Practice Workshop Report.* Technical Report CMU/SEI-98-TR-015 (ESC-TR-98-015), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1998.

[Bass et al., 2003] Bass L., Clements P., and Kazman R., *Software Architecture in Practice*. SEI Series in Software Engineering, 2nd edition. Addison-Wesley, ISBN 0-321-15495-9, 2003, pages 528.

[Batory et al., 2004] Batory D., Sarvela J.N., and Rauschmayer A., *Scaling Stepwise Refinament.* IEEE Transactions on Software Engineering (TSE), 30(6):3555-371, 2004

[Batory et al., 2006] Batory D., Benavides D., and Ruiz-Cortés A., *Automated Analyses of Feature Models: Challenges Ahead*. CACM on Software Product Lines, 2006

[Batory, 2004] Batory D., *Feature-oriented programming and the AHEAD tool suite*. In Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, May 2004, pp. 702 – 703, ISSN: 0270-5257, ISBN: 0-7695-2163-0.

[Bernstein et al., 2000] Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: *A Vision for Management of Complex Models*. Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00).

[Bezevin, 2005] Bezevin J. On the Unification Power of Models, Software and Systems Modeling, vol. 4(2), pp. 171-188, 2005.

[Bonczek et al., 1981] Bonczek, R. H., C.W. Holsapple, and A.B. Whinston. Foundations of Sistemas Expertos, New York: Academic Press, 1981.

[Boronat et al., 2004] Boronat, A., Pérez, J., Carsí, J. Á., and Ramos I., *Two experiencies in software dynamics*. Journal of Universal Science Computer. Special issue on Breakthroughs and Challenges in Software Engineering, Vol. 10, (Issue 4), April 2004.

[Bosch, 2000a] Bosch J., *Design & Use of Software Architectures: Adopting and Envolving a Product-Line Approach*. Addison-Wesley, 2000.

[Bosch, 2000b] Bosch J., *Product Line Architectures*. ObjectiveView, (4):13-18, http://www.ratio.co.uk

[BPMI, 2005] OMG-Business Process Management Initiative, http://www.bpmi.org. http://*www.omg.org/news/releases/pr2005/06-29-05.htm - 15k -*

[BPMN, 2006] OMG-Bussiness Process Management Notation, http://www.omg.org/issues/, OMG final adopted specification, feb 2006, dtc/06-01-01, http://www.bpmn.org/Documents/.

[Bracciali et al., 2002] Bracciali A., Brogi A. and Canal C., *Sistematic Component Adaptation*. In Proceedings of IDEAS, La Habana, Cuba, 2002.

[Brito et al., 2003] Brito I., and Moreira A., *Towards a Composition Process for Aspect-Oriented Requeriments*. Early Aspects 2003: Aspect-Oriented Requeriments Engineering and Architecture Design, Workshop of The 2nd. International Conference on Aspect_Oriented Software Development, Boston, USA, March 2003.

[Butler, 2003] Butler G., *Application Development Strategies. New Technologies and Methods for the Full Life Cicle*. 2003.

[Cabedo et al., 2005] Cabedo R., Pérez J., Carsí J.A. y Ramos I., *Modelado y Generación de Arquitecturas PRISMA con DSL Tools,* en Actas del IV Workshop DYNAMICA, Archena, Murcia, España, 2005.

[Cabello et al., 2008] Cabello M. E., Gómez M., LLavador M., and Ramos I. *ProtoBOM: a framework* that semi-automatically generates Expert Systems based on Software Product Lines". Technical report: DSIC II/02/08, Universitat Politécnica de Valéncia, april 2008, pages 68.

[Carsi, 1999] Carsí J A, *OASIS como Marco Conceptual para la Evolución de Software*. Tesis doctoral. Universidad Politécnica de Valencia, Valencia, España, 1999.

[Chastek et al., 2002] Chastek G. and McGregor J.D. *IGUdelines for Developing a Product Line Production Plan.* Technical Report, CMU/SEI, June 2002, CMU/SEI-2002-TR-06.

[Clauss et al., 2001a] Clauss M. *Modelling variability with UML.* Young Researchers Workshop, 3rd Int. Symposium on Generative and Component-Based Software Engineering (GCSE), Erfurt (Germany), 2001.

[Clauss et al., 2001b] Clauss M. *Generic Modelling using UML extensions for variability*. Workshop on Domain-Specific Visual Languages (DSVL), 16 Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 11-18, Tampa Bay , Florida, USA, 2001.

[Clements et al., 1998] Clements P., Northrop L. M., Bachmann F., Bass L., Bergey J., Chastek G., Cohen S., Donohoe P., Jones L., Krut R., Little R., Smith D., Tilley S., Weiderman N., Withey J., and Woods S., *A Framework for Software Product Line Practice – Version 1.0. Product Line Systems Program.* Software Engineering. Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1998.

[Clements et al., 2001] Clements P. and Northrop L.M., *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison Wesley, 2001.

[Clements et al., 2002a] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J. *Documenting Software Architecture, Views and Beyond.* Addison Wesley, Reading USA, 560 pages, 2002, ISBN-10: 0201703726, ISBN-13: 978-0201703726.

[Clements et al., 2002b] Clements P., and Krueger C.W. Being proactive pays off/Eliminating the adoption barrier. IEEE Software, pp. 28-31, Jul-August 2002.

[Cohen et al., 1995] Cohen S. G., Friedman S., Martin L., Solderitsch N., and Webster R., *Product Line Identification for ESC-Hanscom.* Special Report CMU/SEI-95-SR-024, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1995.

[Cohen et al., 1998] Cohen S. G., and Northrop L. M., *Object-Oriented Technology and Domain Analysis.* In Proceedings of the Fifth International Conference on Software Reuse, ICSR-5, páginas 86-93., Victoria, B.C., Canada. IEEE-CS, 2 - 5 June 1998.

[CONACYT, 1997] Revista Ciencia y Desarrollo. Consejo Nacional de Ciencia y Tecnología, México, Edición especial, 1987.

[Constantinides et al., 2000] Constantinides C.A., and Errad T., *On the Requeriments for Concurrent Software Architectures to Support Advanced Separation of Concerns*. In Proceedings of The OOPSLA 2000, Workshop on Advanced Separation of Concerns in Object-Oriented Systems, 2000.

[CORPOICA, 2006] Librería virtual de sistemas expertos.: http://www.corpoica.org.co/Libreria/software.asp?id_categoria=clase&id_producto=16&offsetr=1&index=1

[Costa et al., 2005] Costa C., Pérez J., Ali N., Carsí J.A. y Ramos I. *PRISMANETMiddelweare: Soporte de la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos.* Actas de las X Jornadas de Ingeniería de Software y Bases de Datos, JISBD, Granada, España, 2005.

[Czarnecki et al., 2000] Czarnecki K., and Eisenecker U., *Generative Programming: Methods, Tools, and Applications*. Addisson-Wesley (2000). ISBN 0-201-30977-7, pag. 267-304.

[Czarnecki et al., 2005] Czarnecki K. and Antkiewicz M., *Mapping Features to Models: A Template Approach Based on Superimposed Variants.* In 4th International Conference on Generative Programming and Component Engineering (GPCE 2005), Tallinn, Estonia, Sep.-Oct. 2005.

[Deelstra et al, 2003] Deelstra S., Sinnema M., Van Gurp J., and Bosch J., *Model Driven Architecture as Approach to Manage Variability in Software Product Families.* Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications (MDAAAFA 2003), pp. 109-114, CTIT Technical Report TR-CTIT-03-27, University of Twente, June 2003.

[Deursen et al., 2002] Deursen A. V. and Klint P. *Domain-specific language design requires feature descriptions.* Journal of Computing and Information Technology, 10(1), pp. 1-17, January-March 2002.

[Durzdel, 2000] Druzdzel M. J., and Flynn R.R. *Decision Support Systems.* In Encyclopedia of Library and Information Science, Vol. 67, Suppl. 30, pages 120-133, Allen Kent (ed.), Marcel Dekker, Inc., New York, 2000.

[Eclipse, 2007] http://www.eclipse.org

[Eeles, 2006] Eeles P. What is Software Architecture, IBM Developerworks http://www-128.ibm.com/developerworks/rational/library/feb06/eeles/

[France et al., 2001] France, R., and Bieman, J., *Multi-view Software Evolution: A UML based Framework for Evolving Object-Oriented Software*. Actas del International Conference on Software Maintenance (ICSM 2001), IEEE Computer Society, pp. 386-39.

[Fresnel et al, 1999] Fensel D., Benjamins R., and Motta E., *UPML: A Framework for Knowledge System Reuse*. In Proceeding of the International Joint Conference on AI (IJCAI-99), Stockholm, Sweden, 1999.

[García et al., 2002] Garcia F.J., Barras J.A., Laguna M.A. y Marques J.M., *Líneas de Producto, Componentes, Frameworks y Mecano*. Informe Técnico DPTOIA-IT-2002-04, Universidad de Valladolid, España, 2002.

[Garlan et al., 2001] Garlan D., Cheng S. and Kompanek A. J. *Reconciling the Needs of Architectural Description with Object Modeling Notations.* Science of Computer Programming Journal, Special UML Edition, Elsevier Science, 2001.

[Giarratano et al., 2004] Giarratano, J., and Riley, G., *Expert Systems: Principles and Programming*. Fourth Edition: (Hardcover), ISBN: 0534384471, 842 pages, 2004

[Gomma et al., 2006] Gomma H., Kerschberg L., Sugumaran V., Bosch C., Tavakoli I., and O'Hara L., *A Knowledge Based Software Engineering Environment for Reusable Software Requirements and Architectures*. Journal of Automated Software Engineering. Vol 3. Numbers 3-4, pages 285-307, August 2006.

[Gomma et al., 2007] Gomma H. and Shin M.E., *Automated Software Product Line Engineering and Product Derivation*. In Proceedings of the 40 th Hawaii

International Conference on System Sciences-HICSS, Waikoloa, Hawaii, USA, January 2007.

[Gomma, 2004] Gomma, H., *Designing Software Products Line with UML: From uses cases to pattern-based software architectures.* The Addison-Wesley Object Thecnology Series. (Hardcover) 736 pages. 2004. ISBN-10: 0201775956, ISBN-13: 978-0201775952.

[González-Baixauli et al., 2005] González-Baixauli B. y Laguna M. A., *MDA e Ingeniería de Requisitos para Líneas de Producto.* Taller sobre Desarrollo Dirigido por Modelos. MDA y Aplicaciones. (DSDM´05), Granada, España, 2005. pags10, http://ftp.informatik.rwthaachen.de/Publications/

[Greenfield et al., 2004] Greenfield J., Short K., Cook S, Kent S., and Crupi J., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[Griss et al., 1998] Griss M., Favaro J., and D´Allessandro M., *Integrating Feature Modeling with the RSEB.* In Proceedings of Fifth International Conference on Software Reuse (ICSR`5), Victoria, Canadá, June 1998, pp. 76-85.

[Hickman et al., 1989] Hickman F., Killin J., and Land L., *Analysis for Knowledge-Based Systems: A Practical IGUde to the KADS Methodology.* Ellis Horwood, England, 1989, pages 190.

[IEEE-1471, 2000] IEEE Standard Asotiation. IEEE Product No. SH94869-TBR, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000. http://standards.ieee.org/reading/ieee.std_public/description/se/1471-2000_desc.html.

[Jacobson et al., 1997] Jacobson I., Griss M., and Jonsson P., *Software Reuse. Architecture, Process and Organization for Business Success*. ACM Press. Addison Wesley Longman, 1997.

[Kang et al., 1990] Kang K. C., Cohen S. G., Hess J. A., Novak W. E., and Peterson A. S., *Feature-Oriented Domain Analysis (FODA). Feasibility Study.* Technical Report CMU/SEI-90-TR21 (ESD-90-TR-222), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, 1990.

[Kang et al., 1998a] Kang K., Kim S., Lee J., Kim K., and Shin E., *FORM: A Feature Oriented Reuse Method with Domain Specific Reference Architectures*. Annals of Software Engineering, Vol. 5, 1998, pp. 143-168.

[Kang et al., 1998b] Kang K. C., Kim S., Lee J., Kim K., and Shim E., *Feature-Oriented Software Engineering for the Electronic Bulletin Board System (EBBS) Domain*. In Proceedings of the Third World Conference on Integrated Design and Process Technology (IDPT'98), 1998.

[Kent, 2002] Kent S. "Model Driven Engineering", Integrated Formal Methods: Third International Conference, IFM 2002, Turku, Finland, May 15-17, 2002. LNCS Vol. 2335, Springer-Verlag, 2003, pp. 286 – 298.

[Kleppe et al., 2003] Kleppe A., Warmer J., and Bast W. *MDA Explained: The Model Driven Architecture™:* Practice and Promise, Addison Wesley Professional, ISBN 978-0-321-19442-8, 2003.

[Kiczales et al., 1997] Kiczales G. et al., *Aspect Oriented Programming* (ECOOP` 97), LNCS 124, Springer Verlag, 1997.

[Kiczales et al., 2001] Kiczales G., Hilsdale E., HuIGUn J., Kersten M., Plam J. and Griswold W., *An Overview of AspectJ.* Proceeding of the European Conference on Object-Oriented Programming, Springer Verlag, 2001.

[JIPDEC, 1989] Japan Information Processing Development Center: http://www.jipdec.or.jp/eng/. http://findarticles.com/p/articles/mi_m1052/is_/ai_8550825?tag=artBody;col1

[Klingler et al., 1996] Klingler C. D., and Solderitsch J., *DAGAR: A Process for Domain Architecture Definition and Asset Implementation.* In Proceedings of the Annual International Conference on ADA (TriAda'96), pages 231-245, December 3-7, 1996, Philadelphia, PA (USA). ACM, ACM Press., 1996.

[Knauber et al., 2001] Knauber P., and Succi G. *Perspectives on Software Product Lines*. ACM Software Engineering Notes, 26(2):29-33. March, 2001.

[Krueger, 1992] Krueger Ch.W., *Software Reuse*. ACM Computing Surveys, 24(2):131-183. 1992.

[Krueger, 2002] Krueger Ch.W., *Variation Management for Software Production Lines*, In Proceedings of SPLC 2002 , pg 37-48, 2002

[Krueger, 2006] Krueger, Ch.W. *New Methods in Software Product Line Development,* SPLC 2006, Austin, Texas, EUA, pp. 95-102, 2006.

[Krutchten, 1995] Krutchen P., The 4+1 View Model of Architecture. IEEE Software 12(6), pp.42-50, 1995.

[Kurtev, 2005] Kurtev I., Adaptability of Model Transformations. PhD. Thesis, University of Twente, The Netherlands, 2005.

[Lee et al., 2000] Lee K., Kang K. C., Chae W., and Choi B. W., *Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse*. Software: Practice and Experience, 30(9):1025-1046, 2000.

[Liao, 2003] Liao S.-H., *Knowledge Management Technologies and Applications-Literature Review from 1995-2002.* In Expert Systems with Applications, Vol. 25, Issue 2, 2003, pp. 155-164.

[Liao, 2005] Liao S.-H., *Expert Systems Methodologies and Applications- a Decade Review from 1995-2004*. In Expert Systems with Applications, Vol. 28, Issue 1, 2005, pp. 93-103.

[Liebewitz, 1998] Liebewitz J., *The Handbook of Applied Expert Systems*. CRC Press, 1998.

[Limón et al., 2006] Limón R., Ramos I., and Torres J.J. *Designing Aspectual Architecture Views*. In Aspect Oriented Software Development, Computational Science and its Applications, LNCS 3983, pp, 726-735, 2006.

[Limón et al., 2007] Limón R., Cabello M.E., and Ramos I. *Establishing Relations among Software Architecture Views through MDA for SPL.* In proceedings of the 14 th International Congress on Computer Science Research-CIICC´07, Veracruz, México, pp.175-187, 2007. ISBN 13978-970-95771-0-5.

[Linton et al., 1989] Linton M. A., Vlissides J. M., and Calder, P. R., *Composing User Interfaces with Interviews*. IEEE Computer, 22(2), 1989.

[Little, 1970] Little J.D.C., *Models and Managers: The Conceptual of a Decision Calculus*. Management Science, Vol. 16, No. 8, April 1970, pp. 466-485.

[López-Herrejón, 2005] López-Herrejón R.E., *Understanding Feature Modularity in Feature Oriented Programming, 2005,*

http://www.ecoop.org/phdoos/ecoop2005phd/RobertoErickLopezHerrejon.pdf

[Loques et al., 2000] Loques O., Sztajnberg A., Leite J., and Lobosco M., *On the Integration of Meta-level Programming and Configuration Programming*. In Reflextion and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato V. 1826, Lectures Notes in Computer Science, pp. 191-210, Springer-Verlag, Heidelberg, Germany, 2000.

[McIlroy, 1976] McIlroy M. D., *Mass-Produced Software Components. Software Engineering Concepts and Techniques;* 1968 NATO Conference on Software Engineering. J. M. Buxton, P. Naur y B. Randell editores. Páginas 88-98. Van Nostrand Reinhold, 1976.

[MDA, 2003] OMG-Model Driven Achitecture, MDA Guide Version 1.0.1, http://www.omg.org/docs/omg/03-06-01.pdf

[MediniQVT, 2007] http://projects.ikv.de/qvt

[Medvidovic et al., 2000] Medvidovic N. and Taylor R. N. A Classification and Comparison *Framework* for Software Architecture Description Languages". IEEE Transactions on Software Engineering 26(1) 70-93, January 2000.

[Medvidovic et al., 2002] Medvidovic N., and Taylor R.N., *A Classification and Comparison Framework for Software Architectures.* In Proceedings of IDEAS, La Habana, Cuba, 2002.

[Melnik et al., 2003] Melnik, S., Rahm E., and Bernstein P. A., *Rondo: A Programming Platform for Generic Model Management*. SIGMOD 2003, Extended ver. in Web Semantics, vol. 1, Num. 1.

[Mens et al., 2005] Mens T., Van Gorp P., A Taxonomy of Model Transformation. Proceedings of Workshop on Graph and Model Transformation (GraMoT ` 2005), 2005.

[Meyer et al., 1997] Meyer M., and Lehnerd A., *The Power of Product Platforms*. Free Press, 1997].

[Mili et al., 1995] Mili H., Mili F. and Mili A., *Reusing Software: Issues and Research Directions.* IEEE Transaactions on Software Engineeering, 21(6):528-562, 1995.

[Miller et al., 2001] Miller, J., and Mukerji, J., *Model Driven Architecture*. Document number ormsc/2001-07-01, http://www.omg.org/mda.

[ModelWare, 2006] SmartQVT: An Open Source Model Transformation Tool Implementing the MOF 2.0 QVT-Operational Language, http://smartqvt.elibel.tm.fr.

[MOF, 2006] OMG-Meta-Object Facility 2.0, http://www.omg.org/spec/MOF/2.0/PDF, http://www.omg.org/docs/ formal/06-01-01.pdf

[Monge et al., 2002] Monge R., Alves C. and Vallecillo A., *A Graphical Representation of COTS-Based Software Architecture*. In Proceedings of IDEAS, La Habana, Cuba, 2002.

[Myllymaki, 2001] Myllymaki T., *Variability Management in Software Product-Lines*. Software Systems Laboratory, Tampere University of Technology, 2001.

[Neighbors, 1984] Neighbors J. M., *The Draco Approach to Constructing Software from Reusable Components*. IEEE Transactions on Software Engineering, SE-10(5):564-574, 1984.

[.NET, 2005] Microsoft .NET Framework 2.0 / Visual Studio 2005. http://msdn.microsoft.com/en-us/library/zw4w595w(VS.80).aspx

[Noriega et al., 1998] Noriega P. y Sierra C., *Subastas y Sistemas Multiagente*. Revista Iberoamericana de Inteligencia Artificial, Número 6, pp. 68-84, 1998.

[Northrop, 2002] Northrop.L.M., *SET Software Product Line Tenets*. IEEE Software, 19(04):32-40, 2002.

[Northrop, 2003] Northrop.L.M., *Software Architecture in Practice*. SEI Series in Software Engineering, 2nd edition. Addison-Wesley, ISBN 0-321-15495-9, Oct 2003, Cap.14: *Software Product Lines: re-using architectural assets.*, pp. 352-368.

[OCL, 2003] OMG-Object Constraint Language, version 2.0. Document UML 2.0 OCL Final Adopted specification http://www.omg.org/cgi-bin/doc?ptc/2003-10-14

[Oliva et al., 1998] Oliva A., Garcia I.C., and Buzato L.E., *The Reflective Architecture of Guaraná*. Technical Report IC-98-14, Instituto de Computación, Universidad de Campiñas, 1998.

[OMG] Web page of The Object Management Group, http://www.omg.org

[Pérez et al., 2002]. Pérez J., Ramos I., Lorenzo A., Letelier P. and Jaén J., *PRISMA: Plataforma OASIS para Modelos Arquitectónicos.* Actas de las VII Jornadas de Ingeniería de Software y Bases de Datos, JISBD, Escorial (Madrid), España, 2002.

[Pérez et al., 2005a] Pérez J., Ali N., Costa C., Carsí J.A., and Ramos I., *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*. In proceedings of 3rd International conference on .NET Technologies, Pilsen, Czech Republic, pp. 97-108, 2005. ISBN 80-86943-01-1.

[Pérez et al., 2005b] Pérez J., Ali N., Carsí J.A, Ramos I., and Navarro E., *Designing Software Architectures with an Aspect-Oriented Language.* Journal on Aspect Orientation, Published by Houman Younessi, RISE ®ADVICE 2004 ISSN 1548-3851, Vol. 1.1. pags. 20, 2005

[Pérez, 2003] Pérez J., *OASIS como Soporte Formal para la Definición de Modelos Hipermedia Dinámicos, Distribuídos y Evolutivos*. Trabajo de investigación dentro del programa de doctorado de Programación Declarativa e Ingeniería de la Programación, Universidad Poltécnica de Valencia, España, 2003.

[Pérez, 2006] Pérez J., *PRISMA: Aspect-Oriented Software Architectures*. PhD. Thesis of Philosophy in Computer Science, Polytechnic University of Valencia, Spain, pages 397, Dec. 2006,.

[Pohl et al., 2005] Pohl K., Bockle G., and Van der Linden F., *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[Pree, 1994] Pree W., *Meta Patterns – A Means for Capturing the Essential of Reusable Object-Oriented Design*. In Proceedings of the 8th European Conference on Object-Oriented Programming. Bologna, Italy, 1994.

[Pree, 1995] Pree W., *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[Queralt et al, 2006] Queralt P., Hoyos L.,Boronat A., Carsí J.A, and Ramos I., *Un motor de transformación de modelos con soporte para el lenguaje QVT Relations.* Taller Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto con JISBD'06), Sitges, España, Octubre 2006.

[QVT, 2005] OMG-Query/View/Transformations: MOF 2.0 QVT Final Adopted Specification, http://www.omg.org/docs/ptc/05-11-01.pdf

[RAS, 2005] OMG- Reusable Asset Specification v2.2, http://www.omg.org/technology/documents/formal/ras.htm, http://www.omg.org/cgi-bin/doc?formal/2005-11-02

[Rashid et al., 2002] Rashid A., Sawyer P., Moreira A., Araujo J. and Esarly, *Aspects: a Model for Aspect-Oriented Requeriments Engineering.* IEEE Joint Conference on Requeriments Engineering, Essen, IEEE Computer Society, pp. 199-202, Germany, 2002.

[Reich et al., 2005] Reich Y. and Kapeliok., Sistemas Expertos. Vol. 41, Issue 1, p.1-19, Nov. 2005.

[Riebisch et al., 2003] Riebisch M., Streitferdt D., and Pshov I., *Modeling Variability for Object-Oriented Product Lines.* In Proceedings of the Workshop Modeling Variability for Object-Oriented Product Lines, ECOOP, 2003.

[Santos et al., 2007] Santos André l, Koskimies Kai and Lopes Antonia. Using Model-Driven Architecture for Variability Management in Software Product Lines. Ph Thesis Proposal Facultade de Ciencias de la Universidade de Lisboa, Portugal,

[Schmidt, 2006] Schmidt D.C., Model-Driven Engineering. IEEE Computer. 39 (2), pp.41-47, Feb. 2006.

[SEI, 2007] Software Engineering Institute. A Framework for Software Product Lines Practice v4.2, http://www.sei.cmu.edu

[Sendall et al., 2003] Sendall S. and Kozaczynski W., *Model Transformation – the Heart and Soul of Model-Driven Software Development.* IEEE Software, Special Issue on Model Driven Software Development, pages 42--45, Sept/Oct 2003.

[Shaw et al., 2006] Shaw M., Clements P. *The Golden Age of Software Architecture,* IEEE Software, Volume: 23 Issue: 1 , pp. 31-39, 2006.

[Simos et al., 1996] Simos M., Creps D., Klingler C., Levine L., and Allemang D., *Organization Domain Modeling (ODM) IGUdebook – Version 2.0.* Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, 9255 Wellington Road Manassas, VA 22110-4121, 1996.

[Simos, 1995] Simos M., *Organization Domain Modeling (ODM): Formalizating the Core Domain Modeling Life Cycle*. Symposium on Software Reusability. April, 1995.

[Sonnemann, 1995] Sonnemann R., *Exploratory Study of Software Reuse Factory*. PhD. thesis, George Mason University, Fairfax, Virginia, Spring 1995.

[SPEM, 2006] Software Process Engineering Metamodel, Version 2.0. Technical Report, 2006, http://www.omg.org/docs/ad/06-06-02.pdf

[SPL] Web page of The Software Product Lines, http://www.softwareproductlines.com

[Sprague, 1980] Sprague, R, H., Jr., A *Framework* for the Development of Expert Systems, Management Information Systems Quarterly, vol. 4, no. 4, Dec. 1980, pp. 1-26.

[Studer et al., 2000] Studer R., Decker S., and Fresel D., *Situation and Perspective of Knowledge Engineering* In J. Cuenca et al. Editors, Knowledge Engineering and Agent Technology, IOS Press, 2000, pages 16.

[Suvée et al., 2003] Suvée D., Vanderperren W., Jonckers V., *JAsCo: An Aspect-Oriented Aprroach Tailored for Component-Based Software Development.* 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 21-29, ISBN: 1-58113-660-9. Boston, Massachusetts, USA, March 2003.

[Svahnherg et al., 2000a] Svahnherg M., Van Gurp J., and Bosch J., *On the Notion of Variability in Software Product Lines.* Blekinge Institute of Technology Research Report 2000:2, ISSN: 1103-1581.

[Svahnherg et al., 2000b] Svahnherg M., and Bosch J., *Issues Conserning Variabiliaty in Software Product Lines. In Development and Evolution of Software Architectures for Product Familie.* Proceedings of International Workshop IW-SAPF3, Vol. 1409 of Lecture Notes in Computer Science, Springer, March 2000, pp. 146-157

[Szyperski, 1998] Szyperski C., *Component Software: beyond Object-Oriented Programming*. ACM Press and Adisson Wesley, New York, USA, 1998.

[Taenatzer et al., 2000] Taenatzer G., Nagl M., Schurr A., and Munch M., *AGG: A Tool Environment for Algebraic Graph Transformation. Applications of Graph Transformations with Industrial Relevance*. Ed. Springer Verlag, LNCS 1779, pp. 481-488.

[Turban et al., 2001] Turban, E., and Aronson, J.E. *Expert Systems and Intelligent Systems.* Prentice Hall, 2001, 865 pages, ISBN: 0-13-089465-6

[Trujillo, 2007] Trujillo S., *Feature Oriented Model Driven Product Lines.* PhD. Thesis, The University of the Basque Country, San Sebastian, Spain, March 2007, pages.175.

[UML, 2005] OMG-Unified Modeling Language version 2.0, http://www.omg.org/docs/formal/05-07-04.pdf.

[Van der Linden et al., 2007] Van der Linden F., Schmid K., and Rommes E. *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering.* Springer, 2007, pages 333, ISBN978-3-540-71436-1

[Van Group et al., 2002] Van Group J., and BoschJ., *Design Erosion: Problems and Causes*, Journal of Systems & Software, 61 (2), pp. 105-119, 2002.

[Vici et al., 1998] Vici A. D., and Argentieri N., *FODAcom: An Experience with Domain Analysis in Italian Telecom Industry.* In Proceedings of the Fifth International Conference on Software Reuse, ICSR-5, páginas 166-175. 2 - 5 June, 1998, Victoria, B.C., Canada. IEEE-CS, 1998.

[Warmer et al., 2004] Warmer, J., and Kleppe, A., *The Object Constraint Language, Second Edition, Getting Your Models Ready for MDA*. Addison-Wesley, 2004.

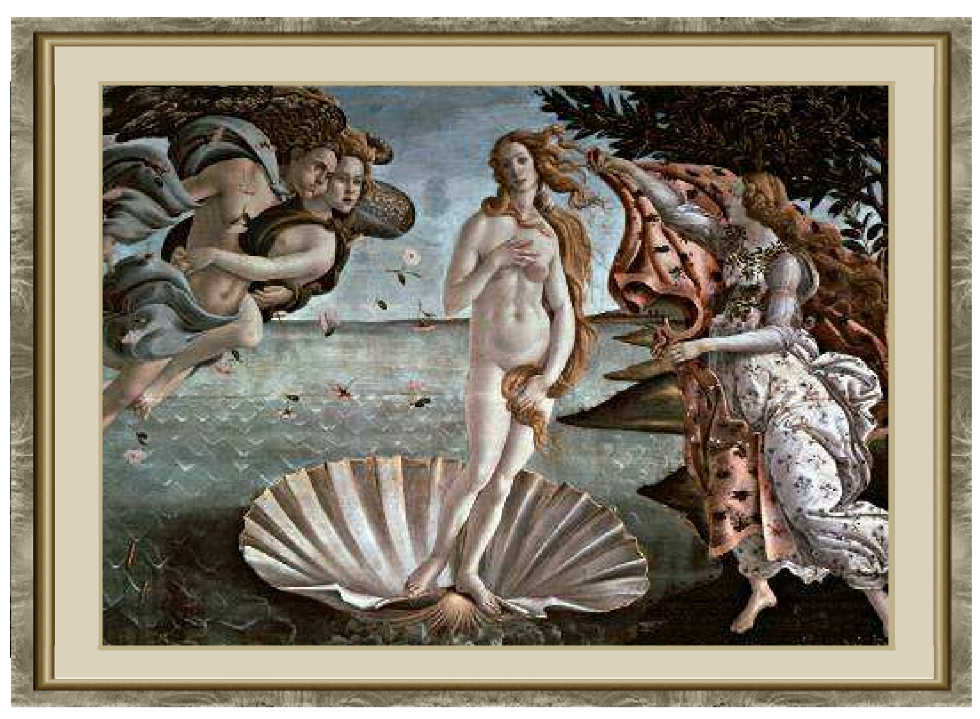
[White, 2004] White, S. Process modelling notations and workflow patterns. BPTrends, 2004.

[Wirfs-Brock et al., 1990] Wirfs-Brock R. J., and Johnson R.E., *Surveying current research in object-oriented design.* Communications of the ACM, 33(9):105-124, 1990.

[XML, 2006] W3C Consortium. Extensible Markup Language (XML), Version 1.0, 2006, http://www.w3.org/TR/2006/REC-xml-20060816/.61

[XSLT, 2005] W3C Consortium. XSL Transformations (XSLT), Version 2.0, 2005, http://www.w3.org/TR/xslt20/.60,61

APÉNDICES



EL NACIMIENTO DE VENUS (SANDRO BOTICCELLI)

(1480-1485)

APÉNDICE A

TERMINOLOGÍA DEL DIAGNÓSTICO

En este apéndice se presentan brevemente los conceptos que especifican la ontología del diagnóstico, incluyendo algunas características y particularidades de los sistemas de diagnóstico.

Definición de términos

Con el fin de clarificar la terminología utilizada en la variabilidad de la LPS, a continuación se definen y/o describen los conceptos manejados en esta tesis para abordar el tema de la variabilidad en el dominio del diagnóstico.

Cada una de las características de la LPS involucradas en el modelo de características, el árbol de decisión, el modelo conceptual del dominio y el modelo conceptual del dominio de aplicación, son las necesarias para el modelado de un sistema basado en el conocimiento desde una aproximación orientada al diagnóstico. Los conceptos que se corresponden con la perspectiva CIM del diagnóstico y describen elementos propios del diagnóstico, son comentados a continuación:

a) **Diagnóstico**: El diagnóstico consiste en interpretar el estado de una entidad, o en su caso, identificar el problema o disfunción de una entidad, a través de sus propiedades (variables observables).

b) **Proceso de diagnóstico**: Un proceso de diagnóstico es el conjunto de tareas encaminadas a la identificación de una cuestión a partir de datos observables y razonamientos.

c) **Coreografía del proceso de diagnóstico**: La coreografía del proceso de diagnóstico establece la sincronización entre el mecanismo de inferencia, la información del dominio y la interfaz del usuario.

d) **Entidad**: Una entidad es el sujeto al que se le realiza el diagnóstico.

e) **Hipótesis**: Una hipótesis es el objetivo o resultado del proceso de diagnóstico. Cuando se identifica el problema o disfunción que presenta una entidad, se obtiene una hipótesis validada. El diagnóstico realizado a una entidad puede llegar a tener una sola hipótesis o bien varias hipótesis, las cuales deberán ser validadas hasta obtener la correcta.

f) **Número de hipótesis**: Cuando las propiedades de una entidad no cambian, se genera una sola hipótesis. Sin embargo cuando las propiedades de una entidad si cambian, se generan varias hipótesis, que deberán ser validadas para llegar a un resultado del diagnóstico.

g) **Propiedades**: Las propiedades de una entidad son la caracterización mediante datos de las causas que provocan una anomalía o disfunción de la misma. Una entidad puede ser caracterizada siempre por las mismas propiedades o bien pueden variar esas propiedades, i.e. la entidad está caracterizada por un conjunto específico de propiedades que varían según la hipótesis a validar. Por ello las propiedades se clasifican en dos tipos: propiedades iguales y propiedades distintas.

h) **Nivel de las propiedades**: El proceso de diagnóstico contempla varios niveles de abstracción de las propiedades de una entidad, y la relación entre las propiedades a través de reglas, de forma que:

- Las propiedades del nivel n y las propiedades del nivel n+1, se relacionan a través de las reglas de nivel n+1,
- Las propiedades del nivel 0 obtienen su valor del usuario,
- Las propiedades del nivel n+1 se infieren aplicando las reglas del nivel n+1,
- El valor de las propiedades superiores al nivel 0, se puede obtener del usuario o bien inferirse a través de las reglas.

i) **Vistas de una entidad**: Una entidad puede ser caracterizada siempre por las mismas propiedades (misma vista) o bien pueden variar esas propiedades (diferentes vistas), i.e. la entidad está caracterizada por un conjunto específico de propiedades para cada hipótesis. Por ello las propiedades se clasifican en dos vistas: la vista constante donde siempre se tienen las mismas propiedades, y la vista variable donde las propiedades cambian.

j) **Reglas**: Las propiedades de las entidades al relacionarse entre sí pueden cumplir reglas. Una relación entre las propiedades es una regla del tipo IF <premisa> THEN <conclusión>, i.e. una cláusula de Horn con cabeza, en donde la premisa y la conclusión de dichas reglas toman valores de las propiedades.
La Base de Conocimientos contiene este tipo de reglas que representan la información del dominio de aplicación del diagnóstico y se representan mediante una estructura de árbol.
El Motor de Inferencia al aplicar las estrategias de razonamiento, recorre el árbol, i.e. aplica las reglas.
Una estructura de árbol, como la generada por el encadenamiento de reglas, puede ser recorrida en dos sentidos: en anchura (niveles), donde se contemplan todas las posibilidades de un nudo antes de pasar al siguiente nodo o bifurcación del árbol; y en en profundidad (ramas) en el que no se toma otro nodo hasta que no se ha desarrollado completamente una rama.
En la LPS de esta tesis, el Motor de Inferencia aplica el recorrido en profundidad.

k) **Usuario final o Actor**: Un usuario (final) del sistema experto se define como una persona que solicita realizar un diagnóstico.

l) **Casos de Uso**: Un caso de uso se define en UML como "un conjunto de acciones que realiza el sistema y que tiene un resultado observable y de interés para un actor particular del mismo". Los casos de uso constituyen una especificación del sistema basada en la funcionalidad. El diagrama de casos de uso muestra las distintas operaciones que se esperan del sistema y cómo se relaciona con su entorno (usuario).

m) **Casos de uso por actor**: Un usuario (final) del sistema, puede acceder a uno o más casos de uso.

n) **Razonamientos**: Los razonamientos o estrategias de razonamiento son la forma en que el mecanismo de inferencia realiza el diagnóstico. El mecanismo de

inferencia aplica estrategias de razonamiento, utilizando la información del dominio para obtener las propiedades. Las estrategias de razonamiento que forman parte de la variabilidad en la LPS son simulaciones de los tipos de razonamientos más comunes que usa una persona que realiza un diagnóstico. Los razonamientos utilizados en esta tesisi son el deductivo, el inductivo y el diferencial, los cuales son algunos de los más utilizados para realizar un diagnóstico.

El *razonamiento deductivo* o guiado por los datos (también llamado razonamiento con encadenamiento hacia delante o razonamiento "forward") consiste en enlazar los conocimientos a partir del uso de datos (propiedades de la entidad a diagnosticar) con el fin de obtener una solución de un problema. Dado que en este razonamiento se generan nuevas propiedades o hechos, existen dos formas de tratarlos, que son: profundidad cuando un hecho en cuanto se genera se introduce a la Base de Conocimientos, o en anchura cuando no se incorporan los hechos a la Base de Conocimientos hasta que se ha terminado. En general, las soluciones pueden ser más de una. De un hecho así deducido se puede asegurar su verdad. La ventaja desde el punto de vista técnico de utilizar este modo de encadenar el conocimiento es su sencillez y que la entrada de datos es única y se da al principio del programa. Este encadenamiento está basado en el *modus ponens* de la lógica formal que dice que: Si conocemos la regla: "si A entonces B", y A es cierto, entonces podemos deducir que B es también cierto.

El encadenamiento hacia adelante se desarrolla según el siguiente procedimiento:

- Hasta que ninguna regla produzca una nueva afirmación o la hipótesis sea identificada:
  - Para cada regla:
    - Trátese de corroborar cada antecedente de la regla mediante su apareamiento con hechos conocidos;
    - Si se corroboran todos los antecedentes de la regla, hágase valer el consecuente, a menos que ya exista una afirmación idéntica;
    - Repítase el procedimiento para todos los apareamientos y alternativas de sustitución.

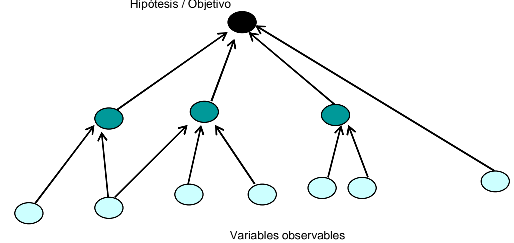


Figura 128 Grafo del razonamiento deductivo

El *razonamiento inductivo* o guiado por los objetivos (también llamado razonamiento con encadenamiento de reglas hacia atrás o razonamiento "backward") consiste en comprobar que un objetivo es cierto en base a unos hechos que forman el universo del sistema. Este encadenamiento tiene su fundamento en el *modus tollens* de la lógica formal que dice que si conocemos la regla: "Si A entonces B" y también conocemos que A es falso, entonces podemos inducir que B también lo es.

El encadenamiento hacia atrás se desarrolla según el siguiente procedimiento:

- Hasta que todas las hipótesis se hayan identificado y ninguna se pueda comprobar o hasta que las hipótesis sean verificadas:
    - Para cada hipótesis:
        - Para cada regla cuyo consecuente coincida con la hipótesis en cuestión,
            - Inténtese corroborar cada uno de los antecedentes de la regla mediante su apareamiento con afirmaciones de la memoria en funcionamiento o mediante encadenamiento regresivo a través de otra regla, creando nuevas hipótesis. Asegúrese de verificar todas las alternativas de unificación y sustitución.
            - Si todos los antecedentes de la regla logran ser corroborados, notifíquese el éxito y concluya que la hipótesis es verdadera.

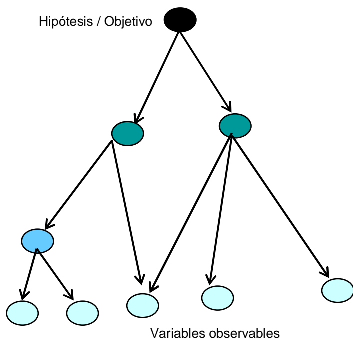


Figura 129 Grafo del razonamiento inductivo

El *razonamiento diferencial*.- El diagnóstico diferencial se presenta cuando hay que comparar entre dos o más posibilidades diagnósticas o hipótesis. En el razonamiento diferencial se lleva a cabo el siguiente proceso: primero se realiza el razonamiento deductivo con el fin de llegar a un diagnóstico; si se llegase a dos o más posibilidades diagnósticas, se realizará el razonamiento diferencial, invocando al razonamiento inductivo para reformular preguntas al usuario o buscar datos que

permitan determinar cuál de las posibilidades diagnósticas es la más certera al problema analizado. i.e. verificar las soluciones o hipótesis.

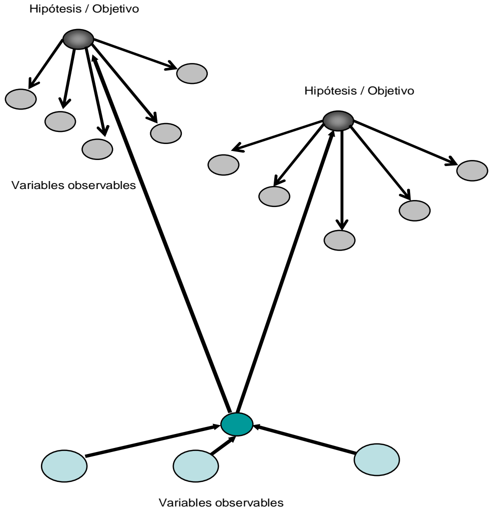


Figura 130 Grafo del razonamiento diferencial

APÉNDICE B

CONCEPTOS UTILIZADOS EN EL MODELADO DE LOS PROCESOS DE LA INGENIERÍA EN LA LÍNEA DE PRODUCTOS SOFTWARE

En este apéndice se presentan brevemente los conceptos que son utilizados para el modelado de la Baseline y el Plan de Producción de la LPS.

Conceptos utilizados en los procesos de la ingeniería de la LPS

Con el fin de clarificar la terminología utilizada en esta tesis para la descripción y el modelado de las tareas realizas en los procesos de la ingeniería de la LPS, a continuación se definen y/o describen algunos conceptos manejados tanto en la primera fase de la ingeniería del dominio, como en la segunda fase de la ingeniería de la aplicación.

a) **Asset**: El ingeniero del dominio debe ser suficientemente experto para determinar la plataforma común que comparte toda la familia de productos. Una plataforma de una LPS es "el conjunto de subsistemas software e interfaces que forman una estructura común desde la cual un conjunto de productos pueden ser producidos y desarrollados eficientemente" [Meyer et al., 1997]. Las partes que componen la plataforma son denominados componentes básicos reutilizables o *assets* [Clements et al., 2001]. Estas pueden incluir la arquitectura, los componentes software, los modelos diseñados, etc. En general, puede ser utilizado cualquier artefacto [Pohl et al., 2005]. La plataforma es la base sobre la cual los productos pueden ser creados adicionando características (variabilidad).

Nuestra plataforma consiste en un conjunto de *assets* que incluye elementos arquitectónicos en el marco del metamodelo PRISMA, esqueletos o plantillas de aspectos PRISMA, así como procesos y modelos conceptuales.

b) **Artefacto**: Un artefacto (software) es cualquier producto de trabajo creado durante un proceso de desarrollo del software, tal como modelos, documentos de requisitos, ficheros de código fuente, ficheros de configuración XML, DOC, etc., que resuelven problemas recurrentes en el desarrollo de software. Estos artefactos aceptarán cierta variabilidad, por lo que contarán con puntos de variabilidad que podrán ser configurados para adaptarlos a las características del problema concreto donde aplicarlos.

c) **Activo**: Un activo es definido como una colección cohesiva de artefactos que resuelven un problema específico o un conjunto de problemas, así como la metainformación para facilitar su reutilización. Un activo se almacena como un fichero comprimido que empaqueta un conjunto de ficheros, cada uno representando uno de estos artefactos.

d) **Modelo RAS del activo**: El modelo RAS de un activo es un modelo que conforma con el metamodelo RAS. Este modelo hace referencia a los artefactos contenidos en el activo, así como la metainformación para facilitar su reutilización. Estas actividades permitirán configurar la variabilidad de los artefactos contenidos en el activo, para obtener del mismo soluciones concretas. Este modelo se empaqueta junto al resto de artefactos dentro del fichero comprimido que representa al activo, conteniendo la identificación y la clasificación del activo, la descripción de sus artefactos, sus puntos de variabilidad y cómo configurarlos.

e) **Metainformación**: La metainformación que describe un activo, es definida por un fichero configurado en XML, que contendrá el modelo RAS del activo.

f) **Baseline**: La Baseline como repositorio de la LPS, es una base de datos especializada que almacena activos de software y facilita la recuperación y el mantenimiento de los activos de software. Su objetivo es asegurar la disponibilidad de activos para apoyar el desarrollo de productos de la LPS.

g) **Asset empaquetado**: Un asset empaquetado es un paquete software conformado por varios *assets* Este asset se almacena como un fichero comprimido que

empaqueta un conjunto de ficheros, cada uno representando uno de estos artefactos.

h) **Esqueleto**: Los esqueletos son plantillas especificadas en el LDA de PRISMA que contienen "huecos", que posteriormente serán rellenados con las características del dominio de aplicación del diagnóstico. Los esqueletos representan plantillas de los aspectos. En la **tabla 13** se representa metafóricamente un esqueleto. Nótese que el icono está vacío, por representar una plantilla con huecos que al ser rellenado conformará un aspecto tipo PRISMA.

Los aspectos necesarios para la definición de los los componentes Motor de Inferencia, Base de Conocimientos y Módulo de Comunicación o Interfaz del Usuario son sus aspectos funcionales, así como el aspecto de coordinación del conector Diagnóstico. Dichos aspectos utilizan el conjunto de servicios de las interfaces.

i) **Artefactos tipo PRISMA**.- Se puede considerar que los aspectos tipo PRISMA son los aspectos esqueleto rellenos con las caraterísticas o "*features*" del dominio de aplicación del diagnóstico, i.e. el dominio específico del caso de estudio.

Por ello la metáfora visual de un aspecto tipo PRISMA se representa con los mismos iconos pero con color, i.e. rellenados con las características que fueron insertadas, como lo muestra la **tabla 13**. Las interfaces, los elementos arquitectónicos (componentes y conectores) y el modelo arquitectónico, son considerados como artefactos software tipo PRISMA, debido a que no involucran en su especificación a las caracterísiticas del dominio de aplicación. La **tabla 13** contiene una metáfora visula de estos artefactos.

j) **Arquitectura de la LPS**: La arquitectura de la línea de productos es la clave para la reutilización sistemática, ya que describe la estructura de los productos del dominio, mostrando sus elementos arquitectónicos y las relaciones entre los mismos. Esta arquitectura debe ser instanciada cada vez que se desarrolla un producto de la línea. Por lo tanto, los productos de la LPS serán los sistemas de diagnóstico de cada uno de los dominios específicos.

Iconos utilizados en los procesos de la ingeniería de la LPS

La **tabla 13** presenta los iconos estándar de SPEM utilizados en esta tesis, así como su significado.

ICONO	SIGNIFICADO
	Actividad
	Rol del ingeniero
	Guía
	Proceso
	Producto de trabajo
	Modelo en UML (especialización de producto de trabajo)
	Documento (especialización de producto de trabajo)

Tabla 13 Iconos estándar de SPEM utilizados en el modelado de la LPS

Asimismo, en la **tabla 14** se presentan los iconos SPEM utilizados exprofeso en esta tesis:

ICONO	SIGNIFICADO
	esqueleto (aspecto)
	aspecto tipo PRISMA
	interfaz tipo PRISMA
	componente tipo PRISMA
	conector tipo PRISMA
	esqueleto aspecto-proceso inserción de características
	híbrido empaquetado
MI 3 U 3 C 3 BC 3	modelo arquitectóinico tipo PRISMA
	configuración del modelo arquitectónico
	caja (kit-box)
	baseline

Tabla 14 Iconos de SPEM creados *exprofeso* para el modelado de la LPS

APÉNDICE C

CASO DE ESTUDIO:

DIAGNÓSTICO DE PROGRAMAS EDUCATIVOS

Con el fin de dar un panorama sobre las características de un producto específico de la LPS cuyo proceso de inferencia es estático, se presenta en este apéndice el caso de estudio del diagnóstico de programas educativos.

Para ello se trata lo siguiente:

- el escenario del diagnóstico de programas educativos,
- las características de la primera variabilidad utilizada en BOM, i.e. los puntos de variabilidad,
- las características de la segunda variabilidad utilizada en BOM, i.e. las características del dominio de aplicación;
- la descripción del proceso de selección de la Kit-Box de este caso de estudio,
- la descripción del proceso de inserción de las características de este dominio aplicación ,
- la especificación en el LDA de PRISMA de cada uno de los esqueletos y tipos correspondientes a este caso de estudio.

En el diagnóstico de programas educativos la entidad a diagnosticar es un programa educativo de postgrado, y el resultado del diagnóstico es la etapa de desarrollo de dicho programa.

El diagnóstico de programas educativos es entendido como el proceso encaminado a la identificación o reconocimiento del estado de desarrollo del programa, sobre la base de propiedades del programa que son evaluadas. Dichas propiedades pueden ser clasificadas en los rubros (p.e. profesorado) y subrubros (p.e. productividad científica del profesorado) a evaluar.

El escenario para el proceso del diagnóstico educativo es el siguiente: Con valores de los subrubros evaluados del programa educativo, es inferido por deducción el valor de los rubros y posteriormente es inferida la hipótesis, obteniendo la etapa de desarrollo del programa educativo.

Para lograr una mayor comprensión del dominio de aplicación es necesario introducir los siguientes conceptos y términos académicos relacionados: profesorado, alumnado, plan de estudios, e infraestructura y servicios.

Profesorado.- El profesorado de un programa de posgrado contempla a todos los que prestan sus servicios como docentes e investigadores de tiempo completo, y que están adscritos a la dependencia de educación superior o centro de investigación que ofrece dicho programa educativo. La evaluación del profesorado de un programa, se realiza a través de la evaluación de su masa crítica, su formación académica, su productividad científica, y sus grupos y /o líneas de investigación.

Alumnado.- El alumnado son todos los estudiantes matriculados en el programa educativo. La evaluación del alumnado de un programa, se realiza a través de la evaluación de la población estudiantil, la tasa de graduación, el tiempo de graduación, la calidad de egresados, y el control y seguimiento escolar.

Plan de estudios.- El plan de estudios de un programa educativo es evaluado por medio del diseño general del plan de estudios, el control y calidad de los alumnos, la experiencia de investigación de los alumnos, el número de cursos y la duración programada del plan de estudios.

Infraestructura y servicios.- La infraestructura y los servicios del plantel que atiende al profesorado y al alumnado de un programa educativo, son evaluados a través de la biblioteca y hemeroteca, el equipo de cómputo, los laboratorios, los edificios e instalaciones, y los servicios generales.

Características de la primera variabilidad:

Dado que las características de la primera variabilidad gestionada en BOM están representadas en el Modelo de Características, a continuación se presentan en la **figura 131** (con color resaltado) dichas características para el caso de estudio del diagnóstico de programas educativos.

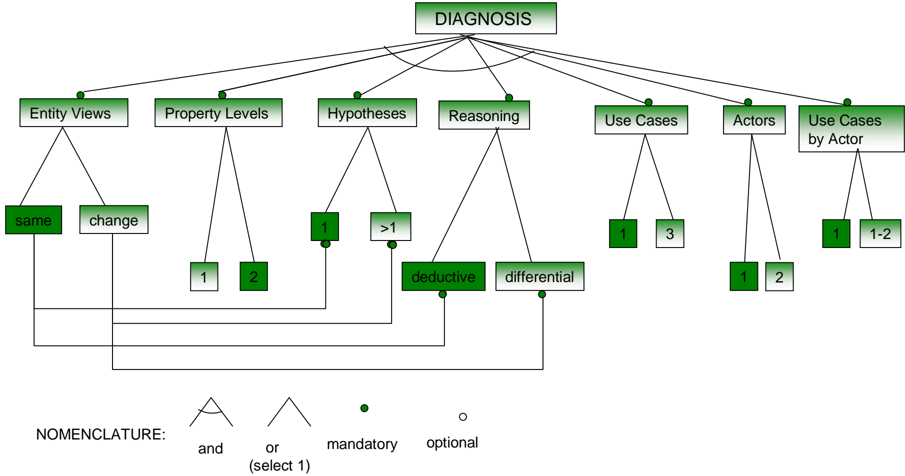


Figura 131 Características del diagnóstico de programas educativos indicadas en el Modelo de Características

Las características del Modelo de Características, representadas en el Árbol de Decisión, son las indicadas en la **figura 132** con color resaltado.

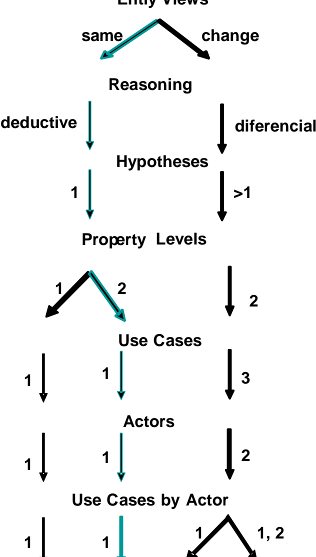


Figura 132 Características del diagnóstico de programas educativos indicadas en el Árbol de Decisión

Los puntos de variabilidad del árbol de decisión y las variantes correspondientes a este caso de estudio, son mostrados a través de las **figuras 133 y 134.**

Los cuatro primeros puntos de variabilidad del árbol de decisión se muestran en el grafo de la **figura 133**.

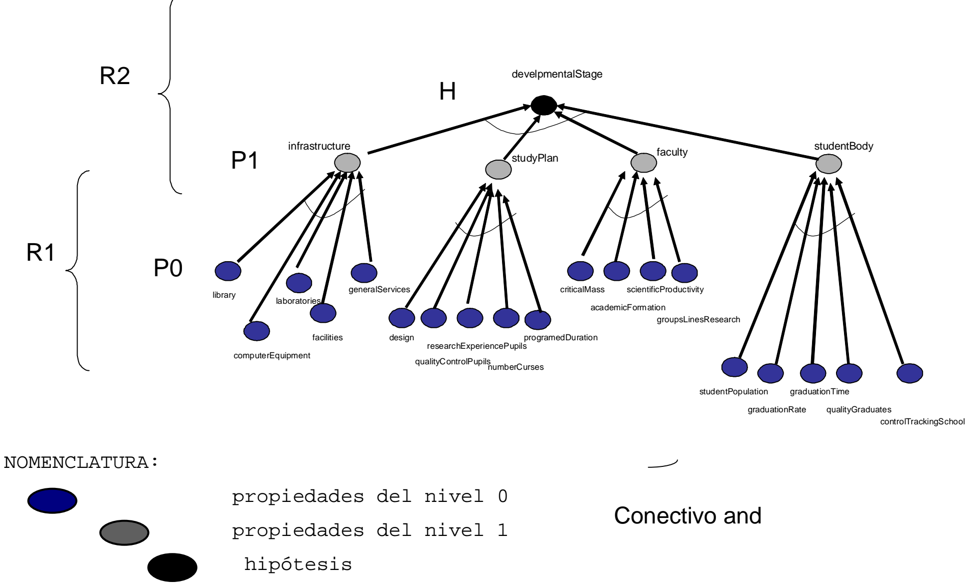


Figura 133 Grafo que muestra un diagnóstico de programas educativos

En dicho grafo se puede observar que:

- Las propiedades son los rubros y subrubros evaluados del programa educativo.
- Las propiedades del nivel 0 son los rubros (p.e. alumnado).
- Las propiedades del nivel 1 son los subrubros (p.e. tasa de graduación del alumnado).
- La hipótesis es la calificación que se le otorga al plan de estudio por medio de su etapa de desarrollo (p.e. consolidado), i.e. el resultado del diagnóstico inferido a través de las reglas del nivel 2.
- Las propiedades de las entidades son las mismas durante el proceso del diagnóstico.
- El tipo de razonamiento aplicado es el deductivo.
- Las reglas de inferencia son:

$$P0, \quad P1 \leftarrow P0, \quad P2 \leftarrow P1, \quad H \leftarrow P2.$$

El diagrama de casos de uso de la **figura 134** muestra los 3 últimos puntos de variabilidad del árbol de decisión.

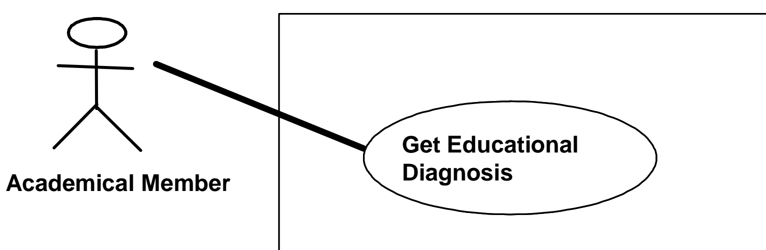


Figura 134 Diagrama de casos de uso correspondiente al diagnóstico de programas educativos

En dicho diagrama se puede observar que:

- El caso de uso es: realizar diagnóstico del programa
- Un solo actor i.e. el usuario final (p.e. el evaluador académico) puede acceder al único caso de uso.

La **figura 135** muestra la trayectoria del árbol de decisión computada por el proceso de selección de la caja (Kit-Box) para el caso de estudio del diagnóstico de programas educativos. Dicha trayectoria viene representada por:

KIT-BOX 1 ← [VP11,VP21,VP31,VP41,VP51,VP61,VP71]

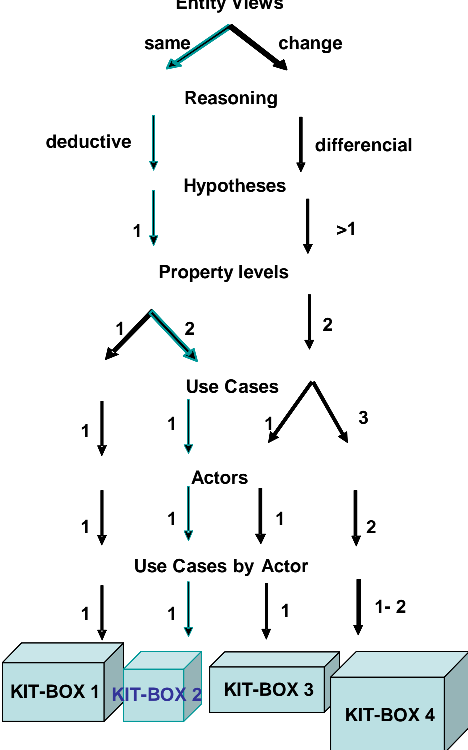


Figura 135 Trayectoria del árbol de decisión computada por el proceso de selección de *assets* correspondiente al diagnóstico de programas educativos

Características de la segunda variabilidad:

Las características de la segunda variabilidad manejada en BOM, i.e. las características del dominio de aplicación, del caso de estudio del diagnóstico de programas educativos son las siguientes:

a) propiedades de nivel 0:

población estudiantil, tasa de graduación, tiempo de graduación, calidad de egresados, control y seguimiento escolar, masa critica, formación académica, productividad científica, grupos y líneas de investigación, diseño general, control general de alumnos, experiencia de investigación de alumnos, numero de cursos, duración programada, biblioteca y hemeroteca, equipo de cómputo, laboratorios, edificio e instalaciones, servicios generales.

b) propiedades de nivel 1:

alumnado, profesorado, plan de estudios, y infraestructura y servicios

c) hipótesis:

etapa de desarrollo

d) reglas de nivel 1: (sólo se muestran algunas)

{poblaciónEstudiantil="buena" and tasaGraduación="buena" and tiempoGraduación="bueno" and calidadEgresados="bueno", and control&seIGUmientoEscolar="bueno" } alumnado="bueno"

{masaCritica="buena", formaciónAcadémica="buena", productividadCientífica="buena",gruposLíneasIinvestigación="buenos"} profesorado= "bueno"

d) reglas de nivel 2: (sólo se muestran algunas)

{alumnado="bueno" and profesorado="bueno" and plan de estudios="bueno" and infraestructura y servicios="bueno"} etapa de desarrollo= "consolidado"

{**alumnado**="malo" and profesorado="malo" and planEstudios="malo" and infraestructura& servicios="malo"} etapa de desarrollo= "incierta"

Estas caracterísiticas son las que "decoran" a los aspectos esqueletos para crear los aspectos tipos PRISMA, mediante un proceso de inserción. El algoritmo computado por dicho proceso en el caso de estudio del diagnóstico médico, es el siguiente:

$$\text{E-BC-DPE}_1 = \text{FP.0} \bullet \text{E-BC-DPE}_0$$

$$\text{E-BC-DPE}_2 = \text{FP.1} \bullet \text{E-BC-DPE}_1$$

$$\text{E-BC-DPE}_3 = \text{FH} \bullet \text{E-BC-DPE}_2$$

$$\text{E-BC-DPE}_4 = \text{FR.1} \bullet \text{E-BC-DPE}_3$$

$$\text{E-BC-DPE}_5 = \text{FR.2} \bullet \text{E-BC-DPE}_4$$

donde: Fx.i •¨E-BC-DPEj significa: "agregar la característica Fx.i al modelo E-BC-PE", E-BC-DPE$_0$ es el Esqueleto de la Base de Conocimientos del Diagnóstico de Programas Educativos, j= 0,..,n y "•" denota la aplicación de la función.

El modelo arquitectónico de este caso de estudio presenta la metáfora visual de la **figura 136**. Dicho modelo está conformado por los siguientes elementos arquitectónicos:

- Componente Motor de Inferencia (con 1 puerto)
- Componente Base de Conocimientos (con 1 puerto)
- Componente Interfaz del Usuario (con 1 puerto)
- Conector Coordinador (con tres puertos)

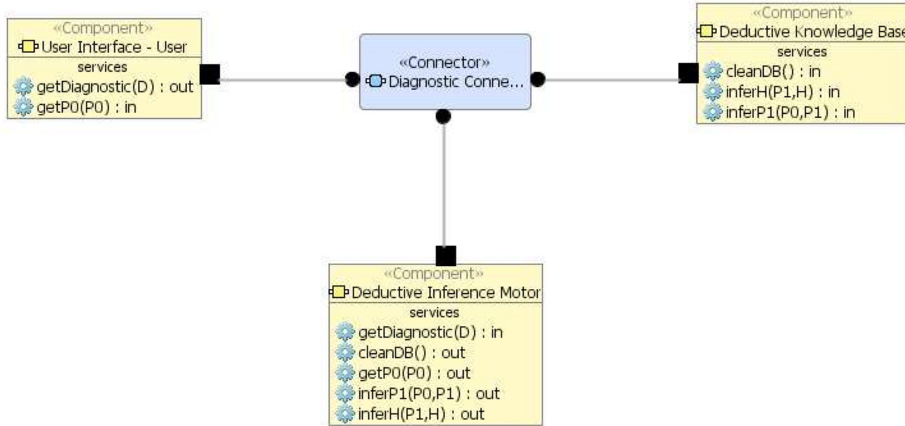


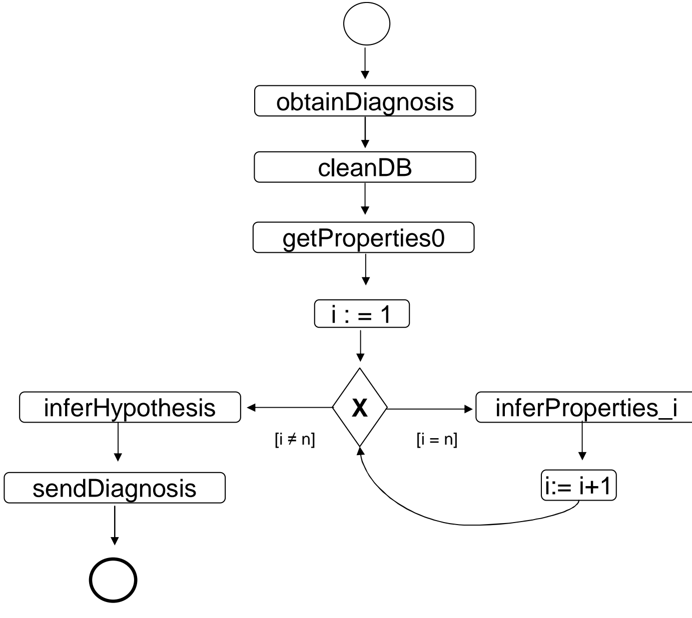
Figura 136 Modelo arquitectónico del caso de estudio: diagnóstico de programas educativos

El código *C#* generado (a través de BOM) de los elementos arquitectónicos de dicho modelo, se presenta a continuación:

```
using System;
using System.Reflection;

using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace
{

[Serializable]
public class InferenceMotor : ComponentBase
{
        public InferenceMotor(string name ) : base(name)
        {
        AddAspect(new FMotor());
        InPorts.Add("InferencePort", "IInference", "INFERENCE");
        OutPorts.Add("InferencePort", "IInference", "INFERENCE");
        }
}

[Serializable]
public class KnowledgeBase : ComponentBase
{
        public KnowledgeBase(string name ) : base(name)
        {
        AddAspect(new FBase());
        InPorts.Add("KnowledgePort", "IDomain", "KNOWLEDGE");
        OutPorts.Add("KnowledgePort", "IDomain", "KNOWLEDGE");
        }
}

[Serializable]
public class User : ComponentBase
{
        public User(string name ) : base(name)
        {
        AddAspect(new FUser());
        InPorts.Add("OperatorPort", "IOperator", "OPERATOR");
        OutPorts.Add("OperatorPort", "IOperator", "OPERATOR");
        }
}

[Serializable]
public class Coordinator : ComponentBase , IConnector
{
        public Coordinator(string name ) : base(name)
        {
        AddAspect(new CDiag());
        InPorts.Add("InferencePortCnct", "IInference","INFERENCE");
        OutPorts.Add("InferencePortCnct","IInference","INFERENCE");
        InPorts.Add("OperatorPortCnct", "IOperator", "OPERATOR");
        OutPorts.Add("OperatorPortCnct", "IOperator", "OPERATOR");

        InPorts.Add("KnowledgePortCnct", "IDomain", "KNOWLEDGE");
```

```
        InPorts.Add("OperatorPortCnct", "IOperator", "OPERATOR");
        OutPorts.Add("OperatorPortCnct", "IOperator", "OPERATOR");

        InPorts.Add("KnowledgePortCnct", "IDomain", "KNOWLEDGE");
        OutPorts.Add("KnowledgePortCnct", "IDomain", "KNOWLEDGE");
        }
}

[Serializable]
public class SysDiagEduc : SystemBase
{
        public SysDiagEduc(string name) : base(name)
        {
//AddBindingInfo("CommunicationPortCnct" ,"Coordinator",
"CommunicationPort");
        }
}
}
```

El componente Motor de Inferencia del modelo arquitectónico de este caso de estudio realiza un proceso denominado (en esta tesis) proceso de inferencia estático. El modelado de dicho proceso se presenta en la **figura 137** haciendo uso de la notación BPM.



i= 0,….,n

Figura 137 Proceso de inferencia estático del caso de estudio: diagnóstico de programas educativos

En la ejecución del sistema, el evaluador académico (usuario final) ingresa los valores de los rubros evaluados del programa educativo, i.e las propiedades del nivel 0, para obtener el resultado del diagnóstico, que en este caso es la etapa de desarrollo del programa educativo ya evaluado, el cual puede ser consolidado, incipiente o incierta. Las **figuras 138 y 139** esquematizan este contexto. En la **figura 138** se presenta una metáfora visual de la información de entrada y de salida del sistema final, ejecutado por el usuario. Mientras que en la **figura 139** se expone el proceso de las vistas de la IGU que proporciona ProtoBOM al usuario, cuando interactúa con dicho sistema.

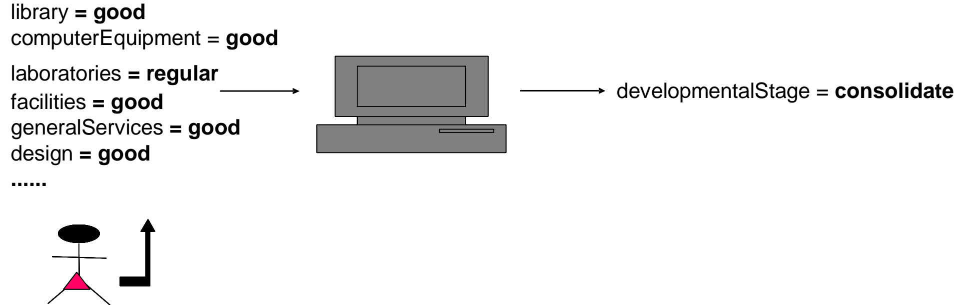


Figura 138 Entrada y salida del sistema de diagnóstico de programas educativos

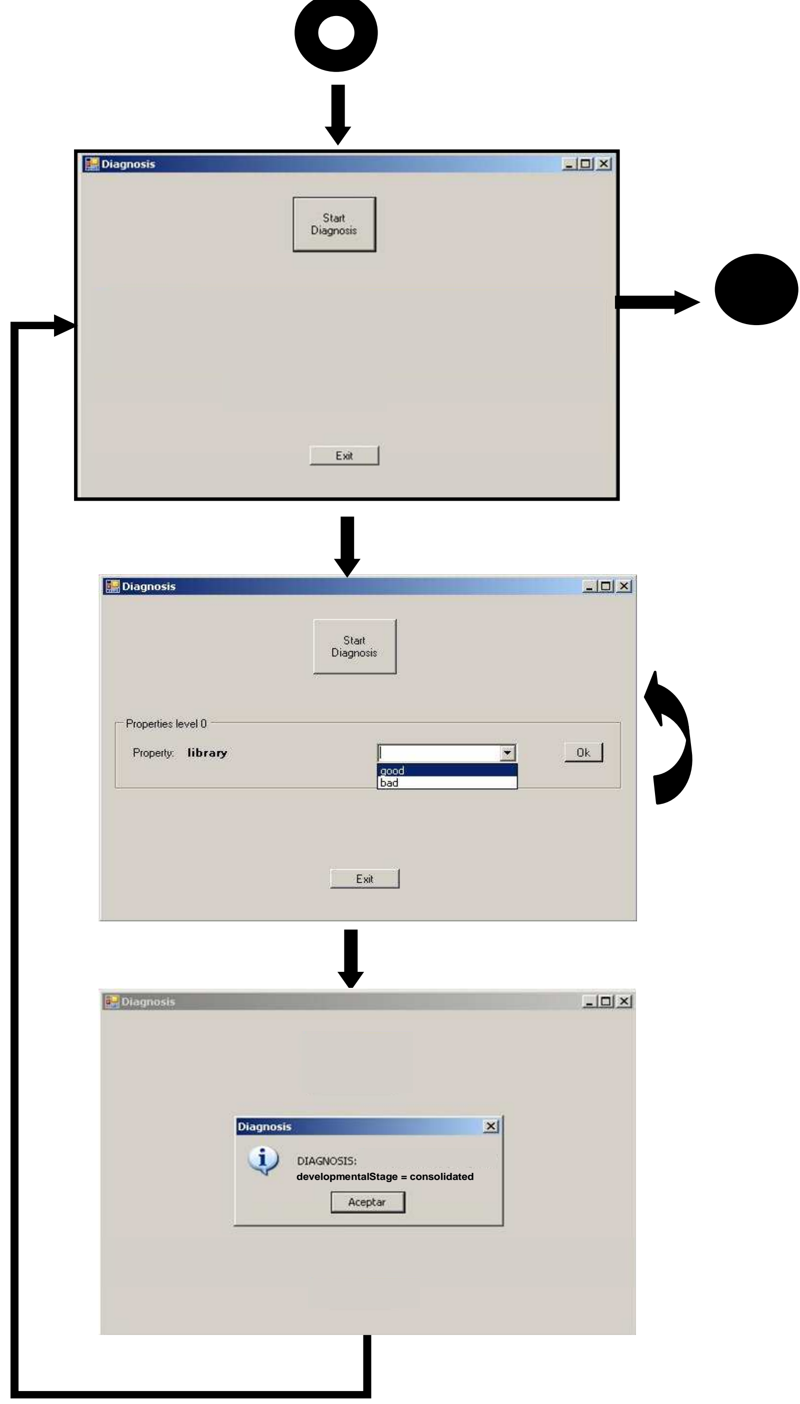


Figura 139 IGUs del sistema ejecutable (caso de estudio: diagnóstico de programas educativos)

Especificación en el LDA de PRISMA del sistema *DiagEduc*

1.- Esqueletos:

NOMENCLATURA:
Features:

FP.0	properties level 0
FP.1	properties level 1
FR.1	rules level 1
FR2	rules level 2
FH	hypotheses

(Sólo son considerados el aspecto funcional de los componentes y el aspecto de coordinación del conector)

1.1 Esqueleto del Aspecto Funcional del componente Base de Conocimientos

```
Functional Aspect  FBaseEP-TV  using IDomainEP-TV
  Attributes
     Variables
     <FP.0>,
     Deriveds
     <FP.1>,
     <FH>;
     Derivations
     <FR.1>
     <FR.2>
  Services
    begin  ( )

    in  cleanDB ( );
    Valuations
    [in  cleanDB ( )]
    <FP.0>:= nil,
    <FP.1>:= nil,
    <FH>:= nil;

    in inferProperties1 (input PROPERTIES0:list, output
      PROPERTIES1:list)
    Valuations
    [in inferProperties1 ( )]
    <FP.0>:=PROPERTIES0,
    PROPERTIESN1:=<FP.1>;

    in  inferHypothesis (input PROPERTIES1:=list, output
      HYPOTHESIS:string)
    Valuations
    [in inferHypothesis ( ) ]
    <FP.1>:=PROPERTIES1,
    HYPOTHESIS:=<FH> ;

    end;
```

```
  Played_Roles
    KNOWLEDGE for IDomainEP-TV ::=
    cleanDB ? ( )
    →
    inferProperties1 ? (PROPERTIES1)
    →
    inferProperties1 ! (PROPERTIES1)
    →
    inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
    →
    inferHypothesis ! (PROPERTIES1, HYPOTHESIS ) ;

  Protocols
    FBASE::= begin ( ):1 → P0

    P0::=KNOWLEDGE_cleanDB ? ( ):1 → P1

    P1::= ( KNOWLEDGE_inferProperties1 ? (PROPERTIES0,PROPERTIES1)
    →
    KNOWLEDGE_inferProperties1 ! (PROPERTIES0, PROPERTIES1)  ) → P2

    P2::= (KNOWLEDGE_inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
    →
    KNOWLEDGE_inferHypothesis ! (PROPERTIES1, HYPOTHESIS) ) → P3

    P3::= end ( ):1;
End_Functional Aspect FBaseEP-TV;
```

1.2 Esqueleto del Aspecto Funcional del componente Motor de Inferencia

```
Functional Aspect  FInferenceEP-TV  using IInferenceEP-TV
  Attributes
      Variables
      <FP.0>,
      <FP.1>,
      <FH>;

  Services
    begin  ( )

    out  cleanDB ( );

    in  getDiagnosis ( output  DIAGNOSIS: string)
     Valuations
        [in  getDiagnosis (DIAGNOSIS) ]
        DIAGNOSIS:= <FH>;

    out getProperties0  (output  PROPERTIES0: list )
     Valuations
        [out  getProperties0  (PROPERTIES0) ]
        <FP.0>:= PROPERTIES0;

    out inferProperties1 (input PROPERTIES0: list, output PROPERTIES1:
        list )
```

```
     Valuations
        [out  inferProperties1  (<FP.0>, PROPERTIES0) ]
        <FP.1>:= PROPERTIES1;

    out  inferHypothesis (input PROPERTIES1: list,  output HYPOTHESIS:
        string)
     Valuations
        [out inferHypothesis (<FP.1>, PROPERTIES1)]
        <FH>:= HYPOTHESIS;

    end;

   Played_Roles
    PROCESS for IInferenceEP-TV ::=
      getDiagnosis ? (DIAGNOSIS)
      →
      cleanDB ! ( )
      →
      getProperties0 ! (PROPERTIES0)
      →
      getProperties0 ? (PROPERTIES0)
      →
      inferProperties1 ! (PROPERTIES0, PROPERTIES1)
      →
      inferProperties1 ? (PROPERTIES0, PROPERTIES1)
      →
      inferHypothesis ! (PROPERTIES1, HYPOTHESIS)
      →
      inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
      →
      getDiagnosis ! (DIAGNOSIS);

   Protocols
      FINFERENCE::= begin ( ):1 → P0

      P0::= PROCESS_getDiagnosis ? (DIAGNOSIS) → P1

      P1::= PROCESS_cleanDB ! ( ):1 → P2

      P2::= ( PROCESS_getProperties0 ! (PROPERTIES0)
             →
             PROCESS_getProperties0 ? (PROPERTIES0) )  → P3

      P3::= ( PROCESS_inferProperties1 ! (PROPERTIES0, PROPERTIES1 )
            →
              PROCESS_inferProperties1 ? (PROPERTIES0, PROPERTIES1) )
      → P4

      P4::= (PROCESS_inferHypothesis ! (PROPERTIES1, HYPOTHESIS)
             →
            PROCESS_inferHypothesis ? (PROPERTIES1, HYPOTHESIS) ) → P5

      P5::= PROCESS_getDiagnosis ! (DIAGNOSIS) → P6

      P6::= end ( ):1;
End_Functional Aspect FInferenceEP-TV;
```

1.3 Esqueleto del Aspecto Funcional del componente Usuario

```
Functional Aspect  FUserEP -TV  using IUserEP -TV
  Attributes
     Variables
     <FP.0>,
     <FH>;

  Services
    begin  ( )

    out  getDiagnosis ( output  DIAGNOSIS: string)
     Valuations
     [out  getDiagnosis (DIAGNOSIS) ]
     <FH>:= DIAGNOSIS;

    in getProperties0  (output  PROPERTIES0: list )
     Valuations
     [out  getProperties0  (PROPERTIES0) ]
     PROPERTIES0:= <FP.0>;

    writeProperties0  (output "PROPERTIES0":list, input
    PROPERTIES0:list,)

    writeDiagnosis (output "DIAGNOSIS":string, output
    DIAGNOSIS:string)

    end;

  Played_Roles
     USER for IUserEP-TV ::=
     getDiagnosis ! (DIAGNOSIS)
     →
     getProperties0 ? (PROPERTIES0)
     →
     getProperties0 ! (PROPERTIES0)
     →
     getDiagnosis ? (DIAGNOSIS);

  Protocols
     FUSER::= begin ( ):1 → P0

     P0::= USER_getDiagnosis ! (DIAGNOSIS) → P1

     P1::= ( PROCESS_getProperties0 ? (PROPERTIES0)
               →
                PROCESS_getProperties0 ! (PROPERTIES0) )  → P2

     P2::= writeProperties0 ( "PROPERTIES0", PROPERTIES0) → P3

     P3::= PROCESS_getDiagnosis ? (DIAGNOSIS) → P4

     P4::= writeDiagnosis ("DIAGNOSIS", DIAGNOSIS) → P5

     P5::= end ( ):1;
End_Functional Aspect FUserEP-TV;
```

2.- Tipos PRISMA

2.1 Tipo PRISMA de la Interface IDomain

```
Interface  IDomainEP-TV
 cleanDB ( );
 inferProperties1 (input PROPERTIES0:list, output PROPERTIES1:list);
 inferHypothesis (input PROPERTIES1:list, output HYPOTHESIS: string);
End_Interface IDomainEP-TV;
```

2.2 Tipo PRISMA de la Interface IInference

```
Interface IInferenceEP-TV
 cleanDB ( );
 getDiagnosis (output DIAGNOSIS: string);
 getProperties0 (output PROPERTIES0:list);
 inferProperties1 (input PROPERTIES0:list, output PROPERTIES1:list);
 inferHypothesis (input PROPERTIES1:list, output HYPOTHESIS: string);
End_Interface IInferenceEP-TV;
```

2.3 Tipo PRISMA de la Interface IUser

```
Interface IUserEP-TV
 getDiagnosis ( output  DIAGNOSIS: string);
 getProperties0  (output PROPERTIES0:list);
End_Interface IUserEP-TV;
```

2.4 Tipo PRISMA del Aspecto Funcional del componente Base de Conocimientos

```
Functional Aspect FBaseEP-TV using IDomainEP-TV
    Attributes
      Variables
      library: string,
      computerEquipment: string,
      laboratories: string,
      facilities: string,
      generalServices: string,
      design: string,
      qualityControlPupils: string,
      researchExperiencePupils: string,
      numberCourses: string,
      programedDuration: string,
      criticalMass: string,
      academicFormation: string,
      scientificProductivity: string,
      groupsLinesResearch: string,
      studentPopulation: string,
      graduationRate: string,
      graduationTime: string,
      graduateQuality: string,
      controlTrackingSchool: string,
```

```
Deriveds
    infrastructure:string,
    studyPlan:string,
    faculty:string,
    studentBody:string,
    developmentalStage: string ;

Derivations
  {library="good"and computerEquipment="good" and
    laboratories="good" and facilities="good" and
    generalServices="good"} infrastructure:="good" ,
  {library="bad"and computerEquipment="bad" and  laboratories="bad"
    and facilities="bad" and generalServices="bad"}
    infrastructure:="bad",
   {design="good" and qualityControlPupils="good" and
    researchExperiencePupils="good" and numberCourses="good" and
    programedDuration="good"}  studyPlan="good",
  {design="bad" and qualityControlPupils="bad" and
    researchExperiencePupils="bad" and numberCourses="bad" and
    programedDuration="bad"}  studyPlan="bad",
  {critical_mass="good" and academicFormation="good" and
    scientificProductivity="good" and groupsLinesResearch="good"}
    faculty:= "good",
  {critical_mass="bad" and academicFormation="bad" and
    scientificProductivity="bad" and groupsLinesResearch="bad"}
    faculty:= "bad",
   {studentPopulation="good" and graduationRate="good" and
   graduationTime="good" and graduateQuality="good" and
   controlTrackingSchool="good"}  studentBody="good",
   {studentPopulation="bad" and graduationRate="bad" and
   graduationTime="bad" and graduateQuality="bad" and
   controlTrackingSchool="bad"}  studentBody="bad",
   ..........
  {infrastructure=" good" and  studyPlan="good" and faculty=" good"
    and studentBody="good"}   developmentalStage:= "consolidate";
  {infrastructure=" bad" and  studyPlan="good" and faculty=" bad"
    and studentBody="bad"}   developmentalStage:= "uncertain";
  ............

Services
  begin  ( )

  in  cleanDB ( );
    Valuations
    [in  cleanDB ( )]
    library, computerEquipment, laboratories, facilities,
    generalServices, design, qualityControlPupils,
    researchExperiencePupils, numberCourses, programedDuration,
    criticalMass. academicFormation, scientificProductivity,
    groupsLinesResearch, studentPopulation, graduationRate,
    graduationTime, graduateQuality, controlTrackingSchool:= nil,
    infrastructure, studyPlan, faculty, studentBody:= nil,
     developmental_stage := nil;

 in inferProperties1 (input PROPERTIES0:list, output
    PROPERTIES1:list)
    Valuations
    [in inferProperties1 ( )]
```

```
      library, computerEquipment, laboratories, facilities,
      generalServices, design, qualityControlPupils,
      researchExperiencePupils, numberCourses, programedDuration,
      criticalMass, academicFormation, scientificProductivity,
      groupsLinesResearch, studentPopulation, graduationRate,
      graduationTime, graduateQuality,
      controlTrackingSchool:=PROPERTIES0,
      PROPERTIES1:= infrastructure, studyPlan, faculty, studentBody;

  in inferHypothesis (input PROPERTIES1:=list, output
      HYPOTHESIS:string)
      Valuations
      [in inferirHypothesis ( ) ]
      infrastructure, studyPlan, faculty, studentBody:= PROPERTIES1,
      HYPOTHESIS:= developmentalStage ;

  end;

Played_Roles
 KNOWLEDGE for IDomainEP-TV ::=
      cleanDB ? ( )
      →
      inferProperties1 ? (PROPERTIES0, PROPERTIES1)
      →
      inferProperties1 ! (PROPERTIES0, PROPERTIES1)
      →
      inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
      →
      inferHypothesis ! (PROPERTIES1, HYPOTHESIS ) ;

Protocols
 FBASE::= begin ( ):1 → P0

 P0::=KNOWLEDGE_cleanDB ? ( ):1 → P1

 P1::= ( KNOWLEDGE_inferProperties1 ? (PROPERTIES0, PROPERTIES1)
 →
 KNOWLEDGE_inferProperties1 ! (PROPERTIES0, PROPERTIES1) ) → P2

 P2::= (KNOWLEDGE_inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
 →
 KNOWLEDGE_inferHypothesis ! (PROPERTIES1, HYPOTHESIS)  ) →  P3

 P3::= end ( ):1;
End_Functional Aspect FBaseEP-TV;
```

2.5 Tipo PRISMA del Aspecto Funcional del componente Motor de Inferencia

```
Functional Aspect FInferenceEP-TV using IInferenceEP-TV
  Attributes
      Variables
        library: string,
        computerEquipment: string,
        laboratories: string,
        facilities: string,
```

```
      generalServices: string,
      design: string,
      qualityControlPupils: string,
      researchExperiencePupils: string,
      numberCourses: string,
      programedDuration: string,
      criticalMass: string,
      academicFormation: string,
      scientificProductivity: string,
      groupsLinesResearch: string,
      studentPopulation: string,
      graduationRate: string,
      graduationTime: string,
      graduateQuality: string,
      controlTrackingSchool: string,

      infrastructure:string,
      studyPlan: string,
      faculty:string,
      studentBody:string,

      developmentalStage: string ;

  Services
     begin  ( )

     out  cleanDB ( );

     in  getDiagnosis ( output  DIAGNOSIS: string)
      Valuations
      [in  getDiagnosis (DIAGNOSIS) ]
      DIAGNOSIS:= developmentalStage;

     out getProperties0  (output  PROPERTIES0: list )
      Valuations
      [out  getProperties0  (PROPERTIES0)]
      library, computerEquipment, laboratories, facilities,
      generalServices, design, qualityControlPupils,
      researchExperiencePupils, numberCourses, programedDuration,
      criticalMass. academicFormation, scientificProductivity,
      groupsLinesResearch, studentPopulation, graduationRate,
      graduationTime, graduateQuality, controlTrackingSchool
      :=PROPERTIES0;

     out inferProperties1 (input PROPERTIES0: list, output
       PROPERTIES1: list )
       Valuations
        [out  inferProperties1  (library, computerEquipment,
         laboratories, facilities, generalServices, design,
         qualityControlPupils, researchExperiencePupils,
         numberCourses, programedDuration, criticalMass.
         academicFormation, scientificProductivity,
         groupsLinesResearch, studentPopulation, graduationRate,
         graduationTime, graduateQuality, controlTrackingSchool,
         PROPERTIES1) ]
        infrastructure, studyPlan, faculty, studentBody:= PROPERTIES1;

     out  inferHypothesis (input PROPERTIES1: list,  output
       HYPOTHESIS: string)
```

```
       Valuations
        [out inferHypothesis (infrastructure, studyPlan, faculty,
        studentBody , HYPOTHESIS)]
        developmentalStage:= HYPOTHESIS;

     end;

 Played_Roles
   PROCESS for IInferenceEP-TV ::=
     getDiagnosis ? (DIAGNOSIS)
     →
     cleanDB ! ( )
     →
     getProperties0 ! (PROPERTIES0)
     →
     getProperties0 ? (PROPERTIES0)
     →
     inferProperties1 ! (PROPERTIES0, PROPERTIES1)
     →
     inferProperties1 ? (PROPERTIES0, PROPERTIES1)
     →
     inferHypothesis ! (PROPERTIES1, HYPOTHESIS)
     →
     inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
     →
     getDiagnosis ! (DIAGNOSIS);

 Protocols
     FINFERENCE::= begin ( ):1 → P0

     P0::= PROCESS_getDiagnosis ? (DIAGNOSIS) → P1

     P1::= PROCESS_cleanDB ! ( ):1 → P2

     P2::= (PROCESS_getProperties0 ! (PROPERTIES0)
            →
           PROCESS_getProperties0 ? (PROPERTIES0) )→ P3

     P3::= (PROCESS_inferProperties1 ! (PROPERTIES0, PROPERTIES1)
            →
           PROCESS_inferProperties1 ? (PROPERTIES0, PROPERTIES1) )
            → P4

     P4::= (PROCESS_inferHypothesis ! (PROPERTIES1, HYPOTHESIS)
           →
           PROCESS_inferHypothesis ? (PROPERTIES1, HYPOTHESIS) )
           → P5

     P5::= PROCESS_getDiagnosis ! (DIAGNOSIS) → P6

     P6::= end ( ):1;
End_Functional Aspect FInferenceEP-TV;
```

2.6 Tipo PRISMA del Aspecto Funcional del Componente Usuario

```
Functional Aspect FUserEP-TV using IUserEP-TV
 Attributes
     Variables
     library: string,
     computerEquipment: string,
     laboratories: string,
     facilities: string,
     generalServices: string,
     design: string,
     qualityControlPupils: string,
     researchExperiencePupils: string,
     numberCourses: string,
     programedDuration: string,
     criticalMass: string,
     academicFormation: string,
     scientificProductivity: string,
     groupsLinesResearch: string,
     studentPopulation: string,
     graduationRate: string,
     graduationTime: string,
     graduateQuality: string,
     controlTrackingSchool: string,
     developmentalStage: string ;

 Services
     begin  ( )

     out  getDiagnosis ( output  DIAGNOSIS: string)
       Valuations
       [out  getDiagnosis (DIAGNOSIS) ]
       developmentalStage:= DIAGNOSIS;

     out getProperties0  (output  PROPERITES0: list )
      Valuations
      [out  getProperties0  (PROPERTIES0)]
      PROPERTIES0:= library, computerEquipment, laboratories,
      facilities, generalServices, design, qualityControlPupils,
      researchExperiencePupils, numberCourses, programedDuration,
      criticalMass. academicFormation, scientificProductivity,
      groupsLinesResearch, studentPopulation, graduationRate,
      graduationTime, graduateQuality, controlTrackingSchool;

     writeProperties0  (output “PROPERTIES0”:list, input
     PROPERTIES0:list,)

     writeDiagnosis (output “DIAGNOSIS”:string, output
     DIAGNOSIS:string)

     end;

 Played_Roles
   USER for IUserEP-TV ::=
     getDiagnosis ! (DIAGNOSIS)
     →
     getProperties0 ? (PROPERTIES0)
     →
     getProperties0 ! (PROPERTIES0)
```

```
     →
     getDiagnosis ? (DIAGNOSIS);

 Protocols
     FUSER::= begin ( ):1 → P0

     P0::= USER_getDiagnosis ! (DIAGNOSIS) → P1

     P1::= writeProperties0 ( "PROPERTIES0", PROPERTIES0) → P2

     P2::= ( PROCESS_getProperties0 ? (PROPERTIES0)
             →
           PROCESS_getProperties0 ! (PROPERTIES0) ) → P3

     P3::= PROCESS_getDiagnosis ? (DIAGNOSIS) → P4

     P4::= writeDiagnosis ("DIAGNOSIS", DIAGNOSIS) → P5

     P5::= end ( ):1;
End_Functional Aspect FUserEP-TV;
```

2.7 Tipo PRISMA del Aspecto de Coordinación del conector Coordinador

```
Coodination Aspect  CDiagnosisEP-TV  using IInferenceEP-TV, IDomainEP-
TV, IUserEP-TV
    Services
     begin ( )

     in/out cleanDB ( );

     in/out getDiagnosis ( output DIAGNOSIS: string)

     in/out getProperties0  (output PROPERTIES0: list)

     in/out inferProperties1 (input PROPERTIES0: list_string, output
         PROPERTIES1: list )

     in/out inferHypothesis (input PROPERTIES1:list, output
         HYPOTHESIS: string)

     end;

    Played_Roles
     PROCESSCNCT for IInferenceEP-TV ::=
       getDiagnosis ? (DIAGNOSIS)
       →
       cleanDB ? ( )
       →
       getProperties0 ! (PROPERTIES0)
       →
       getProperties0 ? (PROPERTIES0)
       →
       inferProperties1 ! (PROPERTIES0, PROPERTIES1)
       →
       inferProperties1 ? (PROPERTIES0, PROPERTIES1)
```

```
    →
    inferHypothesis ! (PROPERTIES1, HYPOTHESIS)
    →
    inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
    →
    getDiagnosis ! (DIAGNOSIS);

  KNOWLEDGECNCT for IDomainEP-TV ::=
    cleanDB ! ( )
    →
    inferProperties1 ! (PROPERTIES0, PROPERTIES1)
    →
    inferProperties1 ? (PROPERTIES0, PROPERTIES1)
    →
    inferHypothesis ! (PROPERTIES1, HYPOTHESIS)
    →
    inferHypothesis ? (PROPERTIES1, HYPOTHESIS)

  USERCNCT for IUserEP-TV ::=
    getDiagnosis ? (DIAGNOSIS)
    →
    getProperties0 ! (PROPERTIES0)
    →
    getProperties0 ? (PROPERTIES0)
    →
    getDiagnosis ! (DIAGNOSIS);

Protocols
  CDIAGNOSIS::= begin ( ):1 → P0

  P0::= ( USERCNCT_getDiagnosis ? (DIAGNOSIS)
        →
        PROCESSCNCT_getDiagnosis ! (DIAGNOSIS) ) → P1

  P1::= ( PROCESSCNCT_cleanDB ? ( )
        →
        KNOWLEDGECNCT_cleanDB ! ( ) ) → P2

  P2::= ( PROCESSCNCT_getProperties0 ? (PROPERTIES0)
        →
        USERCNCT_getProperties0 ! (PROPERTIES0)
        →
        USERCNCT_getProperties0 ? (PROPERTIES0)
        →
        PROCESSCNCT_getProperties0 ! (PROPERTIES0) )   → P3

  P3::= ( PROCESSCNCT_inferProperties1 ? (PROPERTIES0,PROPERTIES1)
        →
        KNOWLEDGECNCT_inferProperties1 ! (PROPERTIES0, PROPERTIES1)
        →
        KNOWLEDGECNCT_inferProperties1 ? (PROPERTIES0, PROPERTIES1)
        →
        PROCESSCNCT_inferProperties1 ! (PROPERTIES0, PROPERTIES1)
        ) → P4

  P4::= (PROCESSCNCT_inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
        →
```

```
          KNOWLEDGECNCT_inferHypothesis ! (PROPERTIES1, HYPOTHESIS)
          →
          KNOWLEDGECNCT_inferHypothesis ? (PROPERTIES1, HYPOTHESIS)
          →
          PROCESSCNCT_inferHypothesis ! (PROPERTIES1, HYPOTHESIS)  )
          → P5

     P5::= ( PROCESSCNCT_getDiagnosis ? (DIAGNOSIS)
          →
          USERCNCT_getDiagnosis ! (DIAGNOSIS)  ) →  P6

     P6::= end ( ):1;
End_Coordination Aspect CDiagnosisEP-TV;
```

2.8 Tipo PRISMA del componente Base de Conocimientos

```
   Component_Type KnowledgeBaseEP-TV
     Ports
       KnowledgePort: IDomainEP-TV,
       Played_Role  FBaseEP-TV.KNOWLEDGE;
     End_Ports;

     Function Aspect import  FBaseEP-TV;

     Initialize
         new ( )
         {  FBaseEP-TV.begin ();
         }
     End_Initialize;

     Destruction
         destroy ()
         {  FBaseEP-TV.end();
          }
     End_Destruction;
   End_Connector_Type  KnowledgeBaseEP-TV;
```

2.9 Tipo PRISMA del componente Motor de Inferencia

```
 Component_Type InferenceMotorEP-TV
     Ports
       ProcessPort: IInferenceEP-TV,
       Played_Role  FInferenceEP-TV.PROCESS;
     End_Ports;

     Function Aspect import  FInferenceEP-TV;

     Initialize
         new ( )
         {  FInferenceEP-TV.begin ();
          }
     End_Initialize;

     Destruction
```

```
        destroy ()
        {  FInferenceEP-TV.end ();
         }
    End_Destruction;
End_Connector_Type  InferenceMotorEP-TV;
```

2.10 Tipo PRISMA del componente Usuario

```
Component_Type UserInterfaceEP-TV
    Ports
      OperatorPort: IUserDEDT
      Played_Role  FUserEP-TV.USER
    End_Ports;

    Functional Aspect import  FUserEP-TV;

    Initialize
       new ( )
       {  FUserEP-TV.begin ();
        }
    End_Initialize;

    Destruction
       destroy ()
       {  FUserEP-TV.end ();
        }
    End_Destruction;
End_Connector_Type UserInterfaceEP-TV;
```

2.11 Tipo PRISMA del conector Coordinador

```
Connector_Type CoordinatorEP-TV
    Ports
      ProcessPortCnct: IInferenceEP-TV,
      Played_Role  CDiagnosisEP-TV.PROCESSCNCT;
      KnowledgePortCnct: IDomainEP-TV,
      Played_Role  CDiagnosisEP-TV.KNOWLEDGECNCT;
      UserPortCnct: IUserEP-TV,
      Played_Role  CDiagnosisEP-TV.USERCNCT;
    End_Ports;

    Coordination Aspect import  CDiagEP-TV;

    Initialize
       new ( )
       {  CDiagEP-TV.begin ();
       }
       End_Initialize;

    Destruction
       destroy ()
       {  CDiagEP-TV.end ();
       }
  End_Destruction;
End_Connector_Type  CoordinatorEP-TV;
```

3- Modelo Arquitectónico

```
Architectural_Model DiagnosticModelEP-TV
  Import Architectural Elements  InferenceMotorEP-TV,
    KnowledgeBaseEP-TV, UserInterfaceEP-TV, CoordinatorEP-TV;

  Attachments
    AttchCnctInf: Coordinator.PortCnctInf (1,1) ↔
        InferenceMotor.PortInf (1,1);
    AttchCnctKnow: Coordinator.PortCnctKnow (1,1) ↔
        KnowledgeBase.PortKnow (1,1);
    AttchCnctOp: Coordinator.PortCnctOp (1,1) ↔ UserInterface.PortOp
        (1,1);
  End_Attachments;

  new ( )
    {
     new InferenceMotorEP-TV;
     new KnowledgeBaseEP-TV;
     new UserInterfaceEP-TV;
     new CoordinatorEP-TV;
    }

  destroy ()
     {
    .destroy ( ) InferenceMotorEP-TV;
    .destroy ( ) KnowledgeBaseEP-TV;
    .destroy ( ) UserInterfaceEP-TV;
     .destroy ( ) CoordinatorEP-TV;
     }
End_Architectural_Model  DiagnosticModelEP-TV;
```

4- Configuración del Modelo Arquitectónico

```
Architectural_Model_ConfigurationEP-TV
  DIAGNOSIS_EP-TV = new  DiagnosticModelEP-TV
    {
    MOTOR_EP-TV =  new InferenceMotorEP-TV;
    BASE_EP-TV =  new KnowledgeBaseEP-TV;
    USER_EP-TV =  new UserInterfaceEP-TV;
     JOINT_EP-TV= new CoordinatorEP-TV;
    Motor_EP-TVAttchCnctInf = new AttchCnctInf (Joint_EP-TV,
      PortCnctInf, Motor_EP-TV, PortInf);
    Base_EP-TVAttchCnctKnow = new AttchCnctKnow (Joint_EP-TV,
      PortCnctKnow, Base_EP-TV, PortKnow);
    User_EP-TVAttchCnctOp = new AttchCnctOp (Joint_EP-TV, PortCnctOp,
      User_EP-TV, PortOp);
    }
End_Architectural_Model_Configuration;
```

APÉNDICE D

CASO DE ESTUDIO:

DIAGNÓSTICO MÉDICO

Con el fin de dar un panorama sobre las características de un producto específico de la LPS cuyo proceso de inferencia es dinámico, se presenta en este apéndice el caso de estudio del diagnóstico médico.

Para ello se trata lo siguiente:

- el escenario del diagnóstico médico,
- las características de la primera variabilidad utilizada en BOM, i.e. los puntos de variabilidad,
- las características de la segunda variabilidad utilizada en BOM, i.e. las características del dominio de aplicación;
- la descripción del proceso de selección de la Kit-Box de este caso de estudio,
- la descripción del proceso de inserción de las características de este dominio aplicación ,
- la especificación en el LDA de PRISMA de cada uno de los esqueletos y tipos correspondientes a este caso de estudio.

El diagnóstico médico

En el diagnóstico médico la entidad a diagnosticar es el paciente y el resultado del diagnóstico es la enfermedad que padece el paciente

El diagnóstico médico es entendido como el proceso encaminado a la identificación o reconocimiento de una enfermedad sobre la base de los signos y síntomas presentes, con el apoyo de los estudios de laboratorio y gabinete. Los signos y síntomas del paciente pueden ser clasificados en dos niveles de abstracción: de grano grueso y de grano fino.

El diagnóstico médico indica el proceso de investigación del paciente, a partir de las observaciones y razonamientos del médico para determinar una enfermedad (o bien si está sano o padece una enfermedad de otra índole a las consideradas).

El escenario del proceso del diagnóstico médico es el siguiente: Con valores de los signos y síntomas de grano grueso, es inferido el síndrome. Esta parte del proceso se realiza con razonamiento deductivo. Dicho síndrome da lugar (por deducción) a dos o más posibles enfermedades (posibles hipótesis). Estas hipótesis deben ser validadas con los signos y síntomas de grano fino para así inferir la enfermedad (hipótesis validada) que padece el paciente. Este última parte del proceso se realiza con razonamiento inductivo.

Para lograr una mayor comprensión del dominio de aplicación es necesario introducir los siguientes conceptos y términos médicos relacionados: signos, síntomas, síndrome o entidad etiofisiopatológica, enfermedad o entidad nosológica y terapia o entidad terapéutica. Para ello, se hará referencia a una de las más comunes o frecuentes de las enfermedades infecciosas infantiles, la bronquiolitis infecciosa aguda. Cabe señalar que entre otras enfermedades, dentro de este dominio, se encuentran: la rubéola, el sarampión, la roséola, la escarlatina, la varicela, el dengue, la tifoidea, la parotiditis viral o paperas, la parotiditis bacteriana, el crup espasmódico o laringitis estridulosa, la neumonía, la otitis viral y la otitis bacteriana.

Signo.- Un signo es la evidencia objetiva de una enfermedad, en especial cuando ésta es observada e interpretada por el médico y no por el paciente. Un signo físico es una indicación de la condición corporal que puede ser percibida directamente (por ejemplo, a través de la auscultación) cuando el médico examina al paciente. El signo es parte principal del proceso de diagnóstico médico, ya que su principal virtud es el ser un dato objetivo. Los signos se obtienen durante el proceso de exploración física y por lo tanto sólo son detectados por el médico que realiza la exploración. Como ejemplo de signos se pueden citar los siguientes: fiebre mayor a 39, fiebre continua, fiebre de 2 a 3 días, tos seca, tos por accesos, dificultad

respiratoria grave, odinofagia garganta roja, odinofagia de 2 a 3 días, apnea por periodos largos.

Las pruebas de laboratorio y gabinete forman parte de los signos que presenta el paciente. Estas, junto con el resto de los signos y síntomas, son una parte muy importante dentro de las reglas que se aplican en los encadenamientos, dado que el resultado del laboratorio ayuda al sistema a tomar una decisión sobre el diagnóstico diferencial que se aplica. Como ejemplo de prueba de laboratorio, se puede citar la biometría hemática leucocitosis.

Síntoma.- El síntoma es cualquier prueba subjetiva de enfermedad o del estado de un paciente. Se refiere a los datos proporcionados por el paciente, relacionados con alguna enfermedad. Son subjetivos, matizados por el padecimiento y las características psicológicas, sociológicas y biológicas del paciente. Es decir, el síntoma "dolor de ganglios" referido por un paciente pudiera estar matizado por su capacidad para resistir el dolor, su historia sobre antecedentes de dolores semejantes y su fortaleza para soportarlo. Como ejemplo de síntoma se puede citar el ardor de garganta intenso.

Síndrome.- La entidad etiofisiopatológica, comúnmente llamada síndrome, es el grupo de signos y síntomas que se manifiestan simultáneamente y que, considerados como un todo, son característicos de cierta enfermedad. Existen procesos mórbidos, o grupo de enfermedades que se caracterizan siempre por la presencia de ciertos signos y síntomas, y cuando éstos no están presentes en su totalidad, no es posible llegar a ese diagnóstico sindromático. En el proceso diagnóstico, el síndrome puede ocupar el diagnóstico final o el paso previo a ese diagnóstico cuando no se puede encontrar la causa definitiva y única de ese síndrome. Como ejemplo de síndrome podemos citar al ira.

Enfermedad.- Relacionado a la nosología (del griego nosos: enfermedad y logos: palabra, razón, tratado), la enfermedad es un dato físico y funcional, manifestado por síndromes y la presencia de signos y síntomas de grano más fino, y comprobado por análisis de laboratorio y estudios de gabinete. Como ejemplo de las enfermedades en este dominio podemos citar a la bronquiolitis infecciosa aguda.

Terapia.- La entidad terapéutica o terapia es el tratamiento dado al paciente. Como ejemplo de la terapia aplicada a la bronquiolitis infecciosa aguda, se puede citar la siguiente: Sintomático y de soporte (antitérmicos, broncodilatadores, oxigenoterapia y nebulizaciones).

Características de la primera variabilidad:

Dado que las características de la primera variabilidad gestionada en BOM están representadas en el Modelo de Características, a continuación se presentan en la **figura 140** (con color resaltado) dichas características para el caso de estudio del diagnóstico médico.

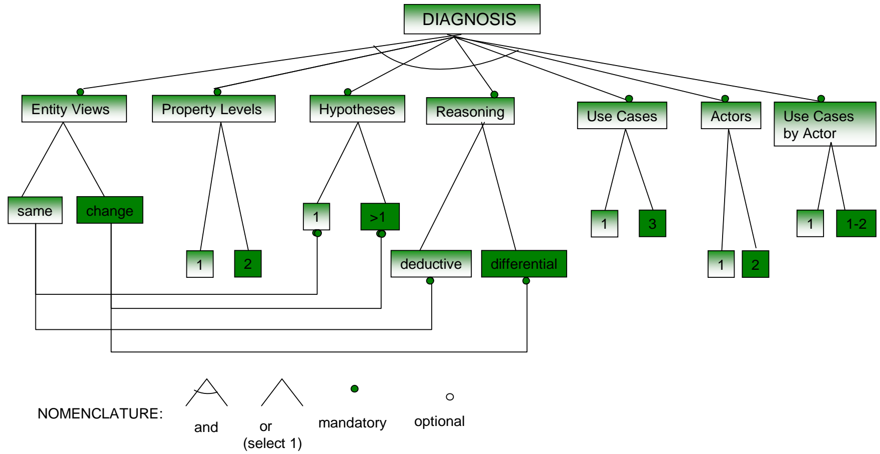


Figura 140 Características del diagnóstico médico indicadas en el Modelo de Características

Las características del Modelo de Características, representadas en el Árbol de Decisión, son las indicadas en la **figura 141** con color resaltado.

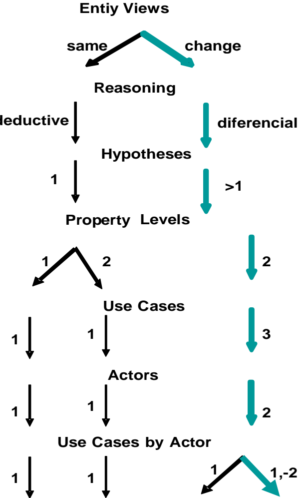


Figura 141 Características del diagnóstico médico indicadas en el Árbol de Decisión

Los puntos de variabilidad del árbol de decisión y las variantes correspondientes a este caso de estudio, se muestran en las **figuras 142 y 143**.

Los cuatro primeros puntos de variabilidad del árbol de decisión se muestran en el grafo de la **figura 142**. Por simplicidad, sólo se han incluido cinco enfermedades.

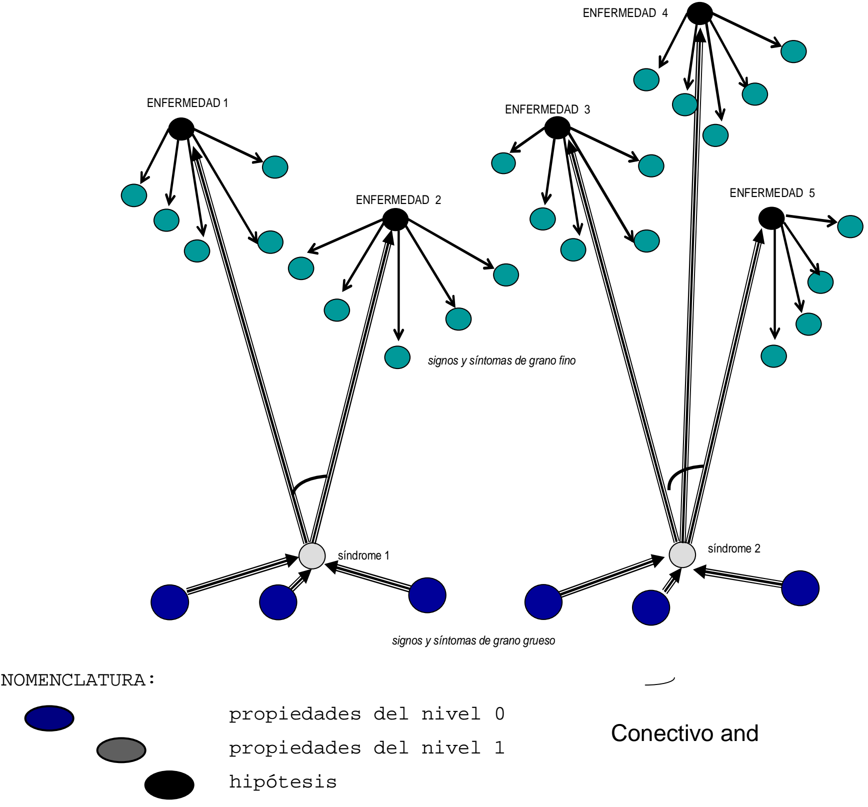


Figura 142 Grafo que muestra un diagnóstico médico

En dicho grafo se puede observar que:

- Las propiedades son los signos y síntomas del paciente; los cuales están clasificados en dos niveles de abstracción: los de grano grueso y los de grano fino.
- Las propiedades del nivel 0 son los signos y síntomas de grano grueso (p.e. tos y fiebre).
- Las hipótesis del nivel 1 son los síndromes (p.e. ira), que son inferidos a través de las reglas del nivel 1
- Las propiedades del nivel 1 son los signos y síntomas de grano fino (p.e. tos seca y fiebre continua).

- Las hipótesis del nivel 2 son las enfermedades (p.e neumonía), i.e. el resultado del diagnóstico inferido a través de las reglas del último nivel.
- Las propiedades de las entidades son diferentes en cada hipótesis que puede resultar del proceso del diagnóstico.
- El tipo de razonamiento aplicado es el diferencial. Al inicio se realiza un razonamiento deductivo el cual infiere varias hipótesis (i.e. posibles enfermedades), por lo que se invoca al razonamiento inductivo para poder diferenciar entre esas hipótesis, la hipótesis validada que es el resultado del diagnóstico (i.e. la enfermedad).
- Las reglas de inferencia son:

$$H1 \leftarrow P0, \quad H2 \leftarrow P1 \wedge H1, \quad H3 \leftarrow P2 \wedge H2.$$

El diagrama de casos de uso de la **figura 143** muestra los tres últimos puntos de variabilidad del árbol de decisión.

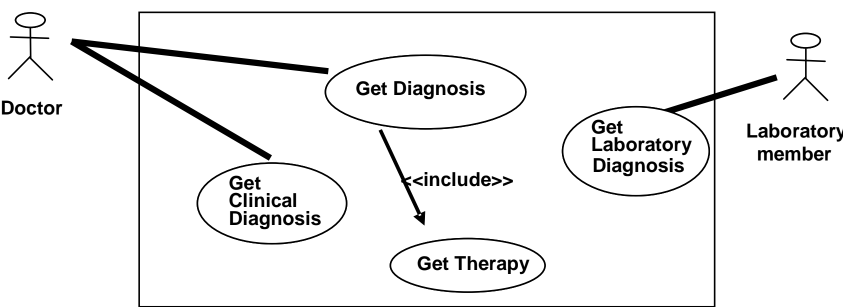


Figura 143 Diagrama de casos de uso correspondiente al diagnóstico médico

En dicho diagrama se puede observar que:

- Los casos de uso son: realizar diagnóstico clínico (caso de uso 1), realizar diagnóstico de laboratorio (caso de uso 2), obtener resultados del diagnóstico (caso de uso 3)
- Dos actores i.e. los usuarios finales (p.e. el médico y el encargado del laboratorio) pueden acceder a los tres casos de uso. Dos casos de uso son utilizados por el médico; sólo un caso de uso por el encargado del laboratorio.

La **figura 144** muestra la trayectoria del árbol de decisión computada por el proceso de selección de la Kit-Box para el caso de estudio del diagnóstico médico. Dicha trayectoria está representada por:

KIT-BOX 4 ← [VP12,VP22,VP33,VP42,VP52,VP62,VP72]

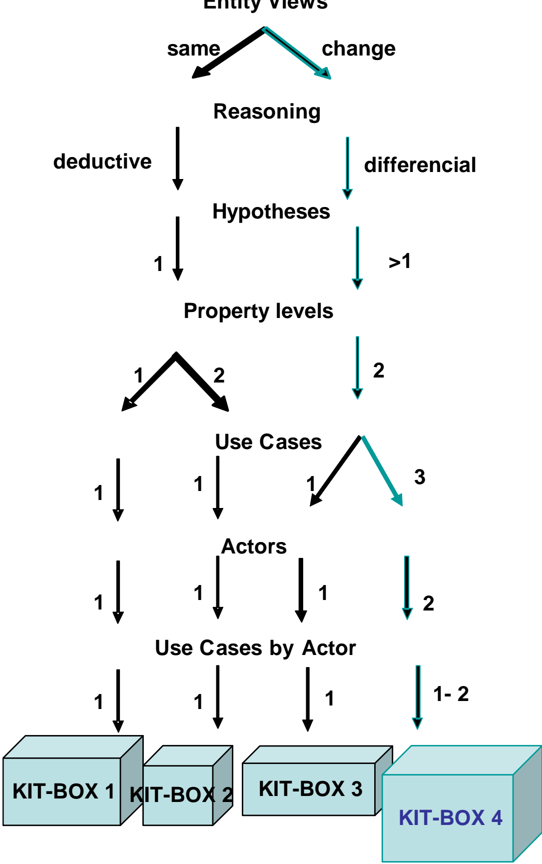


Figura 144 Trayectoria del árbol de decisión computada por el proceso de selección de *assets* correspondiente al diagnóstico médico

Características de la segunda variabilidad: (sólo se muestran algunas)

Las características de la segunda variabilidad manejada en BOM, i.e. las características del dominio de aplicación, del caso de estudio del diagnóstico médico son las siguientes:

a) propiedades de nivel 0:

tos, fiebre, dificultad respiratoria, ...

b) propiedades de nivel 1:

tos seca, tos con flema, fiebre continua, dificultad respiratoria grave, ...

c) hipótesis de nivel 1:
ira, parotiditis,...

d) hipótesis de nivel 2:
bronquiolitis, neumonía, crup espasmódico, paperas, parotiditis bacteriana, ...

e) reglas de nivel 1:
{fiebre = true and tos = true and dificultad respiratoria=true} síndrome="ira"
{fiebre = true and dolor a masticación = true and parótidas anormales=true} síndrome="parotiditis"

f) reglas de nivel 2:
{síndrome="ira" } enfermedad= "bronquiolitis"
{síndrome="ira" } enfermedad= "neumonía"
{síndrome="ira" } enfermedad= "crup espasmódico"
{síndrome=" parotiditis" } enfermedad= "paperas"
{síndrome=" parotiditis" } enfermedad= "parotiditis bacteriana"

g) reglas de nivel 3:
{síndrome="parotiditis" and fiebre continua=true and fiebre mayor a 38 =true and dolor y crecimiento de parótidas =true and dolor a masticación espontáneo agudo=true } enfermedad= "paperas"

Estas caracterísiticas son las que "decoran" a los aspectos esqueletos para crear los aspectos tipos PRISMA, mediante un proceso de inserción. El algoritmo computado por dicho proceso en el caso de estudio del diagnóstico médico, es el siguiente:

$E\text{-}BC\text{-}DM_1 = FP.0 \bullet E\text{-}BC\text{-}DM_0$

$E\text{-}BC\text{-}DM_2 = FP.1 \bullet E\text{-}BC\text{-}DM_1$

$E\text{-}BC\text{-}DM_3 = FH.1 \bullet E\text{-}BC\text{-}DM_2$

$E\text{-}BC\text{-}DM_4 = FH.2 \bullet E\text{-}BC\text{-}DM_3$

$E\text{-}BC\text{-}DM_5 = FR.1 \bullet E\text{-}BC\text{-}DM_4$

$E\text{-}BC\text{-}DM_6 = FR.2 \bullet E\text{-}BC\text{-}DM_5$

$E\text{-}BC\text{-}DM_7 = FR.3 \bullet E\text{-}BC\text{-}DM_6$

donde: Fx.i •·E-BC-DMj significa: "agregar la característica Fx.i al modelo E-BC-DM", E-BC-DM_0 es el Esqueleto de la Base de Conocimientos del Diagnóstico Médico, j= 0,..,n y "•" denota la aplicación de la función.

El modelo arquitectónico de este caso de estudio presenta la metáfora visual de la **figura 145**. Dicho modelo está conformado por los siguientes elementos arquitectónicos:

- Componente Motor de Inferencia (con tres puertos)
- Componente Base de Conocimientos (con tres puertos)
- Componente Interfaz del Usuario1 (con dos puertos)
- Componente Interfaz del Ususario2 (con un puertos)
- Conector Coordinador1 (con tres puertos)
- Conector Coordinador2 (con tres puertos)
- Conector Coordinador3 (con tres puertos)

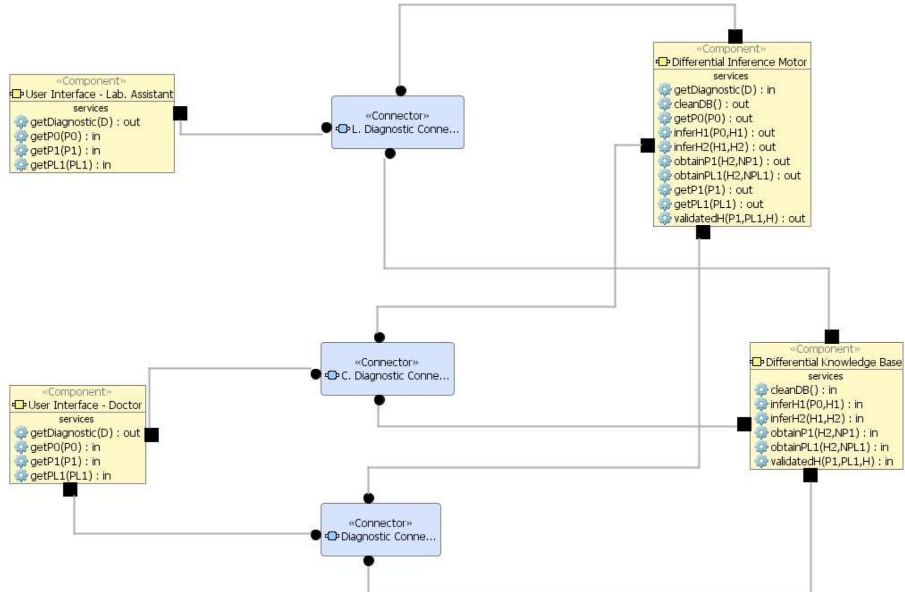


Figura 145 Modelo arquitectónico del caso de estudio: diagnóstico médico

De este modelo arquitectónico se ha elegido el componente Base de Conocimientos para presentar un ejemplo del código C# generado a través de BOM.

```
[Serializable]
  public class KnowledgeBaseMD: ComponentBase
  {
     public KnowledgeBaseMD (string name ) : base(name)
     {
     AddAspect(new FBase());

     InPorts.Add("KnowledgeClinicalPort", "IDomainMD",
  "KNOWLEDGE_CLIN");
     OutPorts.Add("KnowledgeClinicalPort", "IDomainMD",
  "KNOWLEDGE_CLIN");
     InPorts.Add("KnowledgeLaboratoryPort", "IDomainMD",
  "KNOWLEDGE_LAB");
     OutPorts.Add("KnowledgeLaboratoryPort", "IDomainMD",
  "KNOWLEDGE_LAB");
     InPorts.Add("KnowledgeResultsPort", "IDomainMD",
  "KNOWLEDGE_RES");
     OutPorts.Add("KnowledgeResultsPort", "IDomainMD",
  "KNOWLEDGE_RES");
     }
  }
```

El componente Motor de Inferencia del modelo arquitectónico de este caso de estudio realiza un proceso denominado (en esta tesis) proceso de inferencia dinámico. El modelado de dicho proceso se presenta en la **figura 146** haciendo uso de la notación BPM.

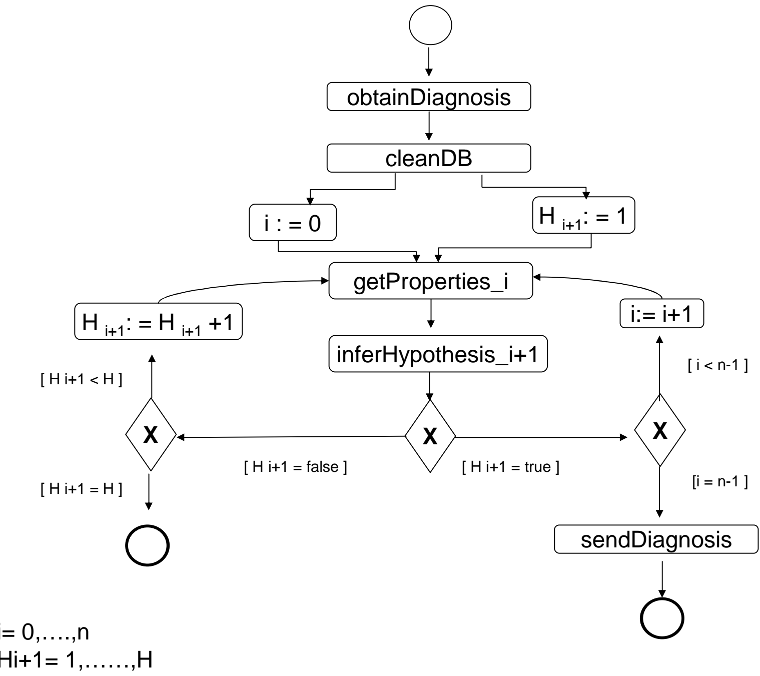


Figura 146 Proceso de inferencia dinámico del caso de estudio: diagnóstico médico

En la ejecución del sistema, el médico y el encargado del laboratorio (usuarios finales) ingresan los valores de los signos y síntomas del paciente, para obtener el resultado del diagnóstico, que en este caso es la enfermedad que padece el paciente, como se muestra en las **figuras 147 y 148.** En la **figura 147** se presenta una metáfora visual de la información de entrada y de salida del sistema final, ejecutado por el médico. Mientras que en la **figura 148** se expone el proceso de las vistas de la IGU que proporciona ProtoBOM al usuario, cuando interactúa con dicho sistema.

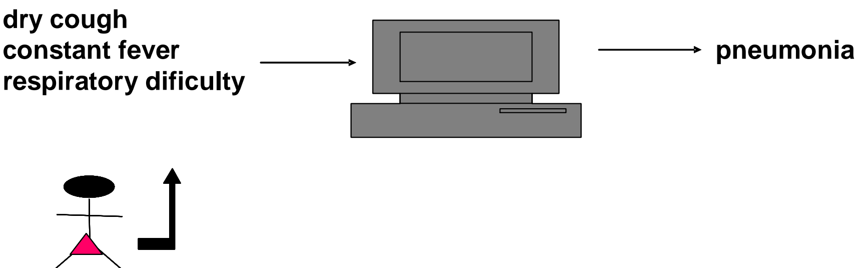


Figura 147 Entrada y salida del sistema de diagnóstico médico

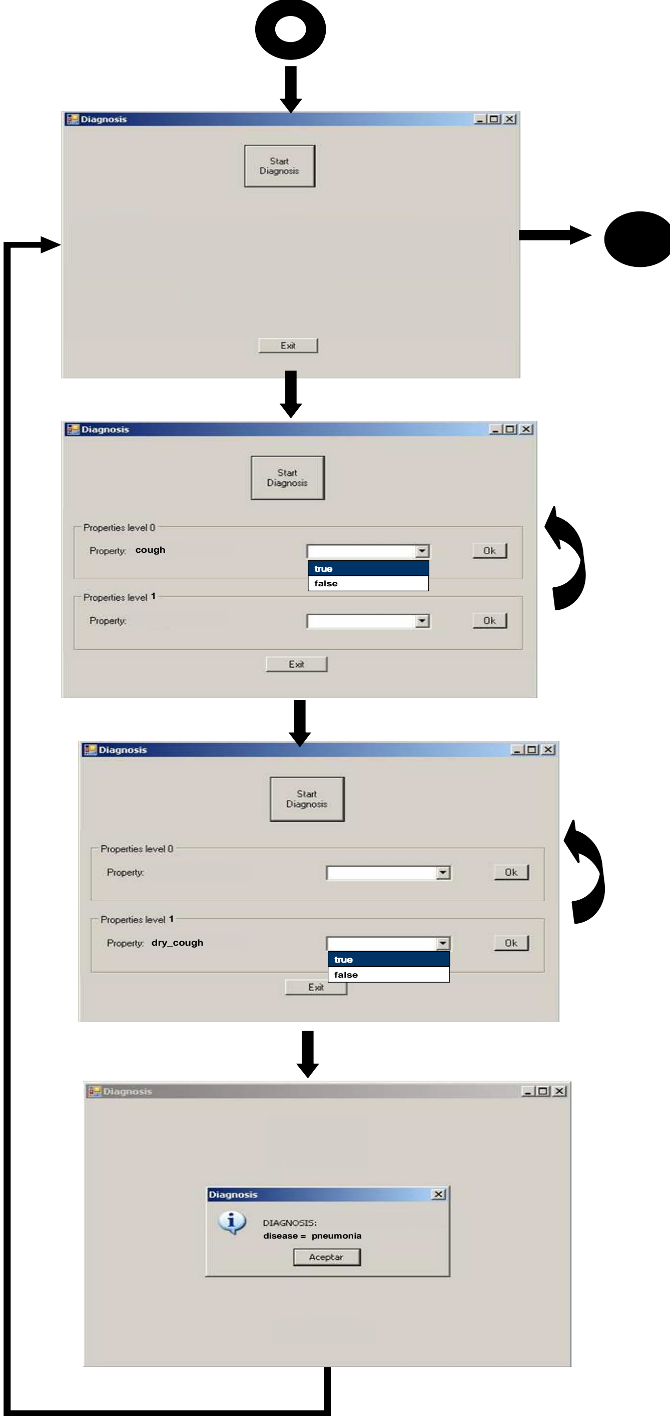


Figura 148 IGUs del sistema ejecutable   (caso de estudio: diagnóstico médico)

Especificación en el LDA de PRISMA del sistema *DiagMed*

1.- Esqueletos:

NOMENCLATURA:
Features:

FP.0	properties level 0
FP.1	properties level 1
FR.1	rules level 1
FR2	rules level 2
FR3	rules level 3
FH.1	hypotheses level 1
FH.2	hypotheses level 2

(Sólo son considerados el aspecto funcional de los componentes y el aspecto de coordinación del conector)

1.1 Esqueleto del Aspecto Funcional del componente Base de Conocimientos

```
Functional Aspect FBaseMD using IDomainMD
 Attributes
    Variable      < FP.0 >;
      < FP.1 >;
      < FPL.1 >;
    Derived           < FH.1 >,
         derivation < FR.1 >
      < FH.2 >,
         derivation < FR.2 >, < FR.3 >;

Services  Begin ( ) ;

  in cleanDB ( )
   Valuations    [in cleanDB ()]
    < FP.0 > := nil;
    < FP.1 > := nil;
    < FPL.1 > := nil;
    < FH.1 > := nil;
    < FH.2 > := nil;

  in inferHypothesis1 (input PROPERTY_0:list_bool, output
    HYPOTHESIS_1:string)
   Valuations
    [in inferHypothesis1 ( )]
    < FP.0 >:= PROPERTY 0,
    HYPOTHESIS1:= < FH.1 >;

  in inferHypotheses2 (input HYPOTHESIS1:string, output
    HYPOTHESES2:list_string)
   Valuations
    [in  inferHypotheses2 ( )]
    < FH.1 >:= HYPOTHESIS1,
    HYPOTHESES2:= < FH.2 >;
  in obtainProperties1 (input HYPOTHESES2:list_string, output
    NPROPERTY1:list_string)
```

```
    Valuations
    {posible_disease= FH.2[0]} [in  obtainProperties1 ()]
    < FH.2[0] >:= HYPOTHESES2,
    NPROPERTY1:= < "FP.1" > ;    {posible_disease= FH.2[1]} [in
    obtainProperties1 ()]
    < FH.2[1] >:= HYPOTHESES2,
    NPROPERTY1:= < "FP.1" > ;    {posible_disease= FH.2[2]} [in
    obtainProperties1 ()]
    < FH.2[2] >:= HYPOTHESES2,
    NPROPERTY1:= < "FP.1" > ;
  in obtainPropertiesLab1 (input HYPOTHESES2:list_string, output
    NPROPERTYLAB1:list_string)
    Valuations
    {posible_disease= FH.2[0]} [in obtainPropertiesLab1()]
    < FH.2[0] >:= HYPOTHESES2,
    NPROPERTYLAB1:= < "FPL.1" > ;    {posible_disease= FH.2[1]} [in
    obtainPropertiesLab1()]
    < FH.2[1] >:= HYPOTHESES2,
    NPROPERTYLAB1:= < "FPL.1" > ;    {posible_disease= FH.2[2]} [in
    obtainPropertiesLab1()]
    < FH.2[2] >:= HYPOTHESES2,
    NPROPERTYLAB1:= < "FPL.1" > ;
  in validateHypothesis (input PROPERTY1:list_bool, input
    PROPERTYLAB1:list_bool, output HYPOTHESIS:string)
    Valuations
    [in  validateHypothesis ()]
    < FP.1 >:= PROPERTY1,
    < FPL.1 >:= PROPERTYLAB1,
    HYPOTHESIS:= < FH.2 > ;
  end;

Played_Roles
  KNOWLEDGE1 for IDomainMD ::=
  cleanDB ? ( )
  →
  inferHypothesis1 ? (PROPERTY0, HYPOTHESIS1)
  →
  inferHypothesis1 ! (PROPERTY0, HYPOTHESIS1)
  →
  inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2)
  →
  inferHypotheses2 ! (HYPOTHESIS1, HYPOTHESES2)
  →
  obtainProperties1 ? (HYPOTHESES2, NPROPERTY1)
  →
  obtainProperties1 ! (HYPOTHESES2, NPROPERTY1)


  KNOWLEDGE2 for IDomainMD ::=
  obtainPropertiesLab1 ? (HYPOTHESES2, NPROPERTYLAB1)
  →
  obtainPropertiesLab1 ! (HYPOTHESES2, NPROPERTYLAB1)


  KNOWLEDGE3 for IDomainMD ::=
  validateHypothesis ? (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
  →
  validateHypothesis ! (PROPERTY1, PROPERTYLAB1, HYPOTHESIS);
```

```
Protocols
  FBASE::= begin ( ):1 → P0

  P0::=KNOWLEDGE1_cleanDB ? ( ):1 → P1

  P1::= ( KNOWLEDGE1_inferHypothesis1 ? (PROPERTY0, HYPOTHESIS1)
       →
       KNOWLEDGE1_inferHypothesis1 ! (PROPERTY0, HYPOTHESIS1)  ) →
       P2

  P2::= (KNOWLEDGE1_inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2)
       →
       KNOWLEDGE1_inferHypotheses2 ! (HYPOTHESIS1,HYPOTHESES2)) →
       P3

  P3::= (KNOWLEDGE1_obtainProperties1 ? (HYPOTHESES2, NPROPERTY1)
       →
       KNOWLEDGE1_obtainProperties1 ! (HYPOTHESES2, NPROPERTY1) ) →
       P4

  P4::= (KNOWLEDGE2_obtainPropertiesLab1 ? (HYPOTHESES2,
       NPROPERTYLAB1)
       →
       KNOWLEDGE1_obtainPropertiesLab1 ! (HYPOTHESES2,
       NPROPERTYLAB1) ) →  P5

  P5::= (KNOWLEDGE3_validateHypothesis ? (PROPERTY1, PROPERTYLAB1
       HYPOTHESIS)
    →
    KNOWLEDGE3_validateHypothesis ! (PROPERTY1, PROPERTYLAB1,
    HYPOTHESIS) ) →  P4

  P4::= end ( ):1;
 End_Functional Aspect FBaseMD
```

1.2 Esqueleto del Aspecto Funcional del componente Motor de Inferencia

```
Functional Aspect  FInferenceMD  using IInferenceMD
  Attributes
     Variables
     <FP.0>,
     <FP.1>,
     <FPL.1>,
     <FH.1>;
     <FH.2>;

  Services
    begin  ( );

    out  cleanDB ( );

    in  getDiagnosis ( output  DIAGNOSIS: string)
     Valuations
       [in  getDiagnosis (DIAGNOSIS) ]
       DIAGNOSIS:= <FH.2>;
```

```
out getProperties0  (output  PROPERTY0: list_bool )
 Valuations
    [out  getProperties0 (PROPERTY0) ]
    <FP.0>:= PROPERTY0;

out inferHypothesis1 (input PROPERTY0: list_bool, output
    HYPOTHESIS1: list )
 Valuations
    [out  inferProperties1 (<FP.0>, PROPERTY0) ]
    <FH.1>:= HYPOTHESIS1;

out inferHypotheses2 (input HYPOTHESIS1:string,  output
    HYPOTHESES2: list_string)
 Valuations
    [out inferHypothesis (<FH.1>, HYPOTHESIS1)]
    <FH.2>:= HYPOTHESES2;

out obtainProperties1 (input HYPOTHESES2:list_string, output
         NPROPERTY1: list_string )
 Valuations
    {posible_disease= FH.2[0]}
    [out  getProperties1 (<FH.2[0]>, NPROPERTY1) ]
    <"FP.1">:= NPROPERTY1;
    {posible_disease= FH.2[1]}
    [out  getProperties1 (<FH.2[1]>, NPROPERTY1) ]
    <"FP.1">:= NPROPERTY1;
    {posible_disease= FH.2[2]}
    [out  getProperties1 (<FH.2[2]>, NPROPERTY1) ]
    <"FP.1">:= NPROPERTY1;

out obtainPropertiesLab1 (input HYPOTHESES2: list_string, output
         NPROPERTYLAB1:list_string )
 Valuations
    {posible_disease= FH.2[0]}
    [out getProperties1 (<FH.2[0]>, NPROPERTYLAB1)]
    <"FPL.1">:= NPROPERTYLAB1;
    {posible_disease= FH.2[1]}
    [out getProperties1 (<FH.2[1]>, NPROPERTYLAB1)]
    <"FPL.1">:= NPROPERTYLAB1;
    {posible_disease= FH.2[2]}
    [out getProperties1 (<FH.2[2]>, NPROPERTYLAB1)]
    <"FPL.1">:= NPROPERTYLAB1;

out getProperties1  (output  PROPERTY1: list_bool )
 Valuations
    [out  getProperties1  (PROPERTY1) ]
    <FP.1>:= PROPERTY1;

out getPropertiesLab1  (output PROPERTYLAB1: list_bool)
 Valuations
    [out  getProperties1 (PROPERTYLAB1) ]
    <FPL.1>:= PROPERTYLAB1;

out  validateHypothesis (input PROPERTY1: list_bool, input
    PROPERTYLAB1: list_bool, output HYPOTHESIS: string)
 Valuations
    [out inferHypothesis (<FP.1>, <FPL.1>, HYPOTHESIS)]
    <FH.2>:= HYPOTHESIS;
```

```
 end;

Played_Roles
 PROCESS1 for IInferenceMD ::=
   getDiagnosis ? (DIAGNOSIS)
   →
   cleanDB ! ( )
   →
   getProperties0 ! (PROPERTY0)
   →
   getProperties0 ? (PROPERTY0)
   →
   inferHypothesis1 ! (PROPERTY0, HYPOTHESIS1)
   →
   inferHypothesis1 ? (PROPERTY0, HYPOTHESIS1)
   →
   inferHypothesies2 ! (HYPOTHESIS1, HYPOTHESES2)
   →
   inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2)
   →
   obtainProperties1 ! (HYPOTHESES2, NPROPERTY1)
   →
   obtainProperties1 ? (HYPOTHESES2, NPROPERTY1);
   →
   getProperties1 ! (PROPERTY1)
   →
   getProperties1 ? (PROPERTY1);

 PROCESS2 for IInferenceMD ::=
   obtainPropertiesLab1 ! (HYPOTHESES2, NPROPERTYLAB1)
   →
   obtainPropertiesLab1 ? (HYPOTHESES2, NPROPERTYLAB1);
   →
   getPropertiesLab1 ! (PROPERTYLAB1)
   →
   getPropertiesLab1 ? (PROPERTYLAB1);

 PROCESS3 for IInferenceMD ::=
   validateHypothesis ! (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
   →
   validateHypothesis ? (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
   →
   getDiagnosis ! (DIAGNOSIS);

Protocols
   FINFERENCE::= begin ( ):1 → P0

   P0::= PROCESS1_getDiagnosis ? (DIAGNOSIS) → P1

   P1::= PROCESS1_cleanDB ! ( ):1 → P2

   P2::= ( PROCESS1_getProperties0 ! (PROPERTY0)
```

```
            →
            PROCESS1_getProperties0 ? (PROPERTY0) )
            → P3

    P3::=( PROCESS1_inferHypothesis1 ! (PROPERTY0,HYPOTHESIS1)
           →
          PROCESS1_inferHypothesis1 ? (PROPERTY0,HYPOTHESIS1))  → P4

    P4::= ( PROCESS1_inferHypotheses2 ! (HYPOTHESIS1,  HYPOTHESES2)
          →
          PROCESS1_inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2) ) →
         P5

    P5::= (
           ( PROCESS1_obtainPropierties1 ! (HYPOTHESES2, NPROPERTY1)
           →
           PROCESS1_obtainPropierties1 ? (HYPOTHESES2, NPROPERTY1) )
    ^
           ( PROCESS2_obtainPropiertiesLab1 ! (HYPOTHESES2,
          NPROPERTYLAB1)
           →
           PROCESS2_obtainPropiertiesLab1 ? (HYPOTHESES2,
          NPROPERTYLAB1) )
           )   → P6

    P6::=
        ( ( PROCESS1_getPropierties1 ! (PROPERTY1)
           →
           PROCESS1_getPropierties1 ? (PROPERTY1) )
    ^

         ( PROCESS2_getPropiertiesLab1 ! (PROPERTYLAB1)
           →
           PROCESS2_getPropiertiesLab1 ? (PROPERTYLAB1) )
         )   → P7

    P7::= (PROCESS3_validateHypothesis ! (PROPERTY1, PROPERTYLAB1,
         HYPOTHESIS)
           →
          PROCESS3_validateHypothesis ? (PROPERTY1, PROPERTYLAB1,
         HYPOTHESIS))  → P8

    P8::= PROCESS3_getDiagnosis ! (DIAGNOSIS) → P9

    P9::= end ( ):1;
  End_Functional Aspect FInferenceMD;
```

1.3 Esqueleto del Aspecto Funcional del componente Usuario Clínico

```
Functional Aspect  FUserClinMD  using IUserClinMD
  Attributes
     Variables
     <FP.0>,
     <FP.1>,
     <FH>;
```

```
Services
  begin  ( );

  out  getDiagnosis ( output DIAGNOSIS: string)
    Valuations
    [out  getDiagnosis (DIAGNOSIS) ]
    <FH.2>:= DIAGNOSIS;

  in getProperties0 (output PROPERTY0: list_bool )
    Valuations
    [out  getProperties0 (PROPERTY0) ]
    PROPERTY0:= <FP.0>;

  writeProperties0 (output <"FP.0">, input PROPERTY0:list_bool);

  in getProperties1 (output PROPERTY1: list_bool )
    Valuations
    [out  getProperties1 (PROPERTY1) ]
    PROPERTY1:= <FP.1>;

  writeProperties1 (output <"FP.1">, input PROPERTY1: list_bool);

  writeDiagnosis (output <"FH.2">, output DIAGNOSIS: string);

  end;

Played_Roles
  USER1 for IUserClinMD ::=
    getDiagnosis ! (DIAGNOSIS)
    →
    getProperties0 ? (PROPERTY0)
    →
    getProperties0 ! (PROPERTY0)
    →
    getProperties1 ? (PROPERTY1)
    →
    getProperties1 ! (PROPERTY1)

  USER3 for IUserClinMD ::=
    getDiagnosis ? (DIAGNOSIS);

Protocols
  FUSERCLIN::= begin ( ):1 → P0

  P0::= USER1_getDiagnosis ! (DIAGNOSIS) → P1

  P2::= writeProperties0 ( <"FP.0">, PROPERTY1) → P3

  P3::= ( USER1_getProperties0 ? (PROPERTY0)
          →
          USER1_getProperties0 ! (PROPERTY0) )  → P4

  P4::= writeProperties1 ( <"FP.1">, PROPERTY1) → P5

  P5::= (USER1_getProperties1 ? (PROPERTY1)
         →
         USER1_getProperties1 ! (PROPERTY1) ) → P6
```

```
    P6::= USER3_getDiagnosis ? (DIAGNOSIS) → P7

    P7::= writeDiagnosis (<"FH">, DIAGNOSIS) → P8

    P8::= end ( ):1;
  End_Functional Aspect FUserClinMD;
```

1.4 Esqueleto del Aspecto Funcional del componente Usuario Laboratorio

```
Functional Aspect  FUserLabMD  using IUserLabMD
  Attributes
     Variables
     <FPL.1>,
     <FH>;

  Services
    begin ( );

    in getPropertiesLab1 (output PROPERTYLAB1:list_bool )
      Valuations
      [out  getPropertiesLab1  (PROPERTYLAB1) ]
      PROPERTYLAB1:= <FPL.1>;

    writePropertiesLab1 (output <"FPL.1">, input PROPERTYLAB1:
    list_bool);

    end;

  Played_Roles
    USER2 for IUserMD ::=
      getPropertiesLab1 ? (PROPERTYLAB1)
      →
      getPropertiesLab1 ! (PROPERTYLAB1);

  Protocols
    FUSERLAB::= begin ( ):1 → P0

    P0::= writePropertiesLab1 ( <"FPL.1">, PROPERTYLAB1) → P1

    P1::= ( PROCESS_getPropertiesLab1 ? (PROPERTYLAB1)
          →
          PROCESS_getPropertiesLab1 ! (PROPERTYLAB1) )→ P2

    P2::= end ( ):1;
End_Functional Aspect FUserLabMD;
```

2.- Tipos PRISMA

2.1 Tipo PRISMA de la Interface IDomain

```
Interface IDomainMD
 cleanDB ( );
 inferHypothesis1 (input PROPERTY0:list_bool,output
     HYPOTHESIS1:string)
 inferHypotheses2 (input HYPOTHESIS1:string, output
     HYPOTHESES2:list_string)
 obtainProperties1 (input HYPOTHESES2:list_string, output
     NPROPERTY1:list_string)
 obtainPropertiesLab1 (input HYPOTHESES2:list_string, output
     NPROPERTYLAB1:list_string)
 validateHypothesis (input PROPERTYL1:list_bool, input
     PROPERTYLAB1:list_bool, output HYPOTHESIS:string)
End_Interface  IDomainMD;
```

2.2 Tipo PRISMA de la Interface IInference

```
Interface IInferenceMD
 cleanDB ( );
 getDiagnosis (output DIAGNOSIS: string);
 getProperties0 (output PROPERTY0:list_bool);
 getProperties1 (output PROPERTY1:list_bool);
 inferHypothesis1 (input PROPERTY0:list_bool, output
    HYPOTHESIS1:string);
 inferHypotheses2 (input HYPOTHESIS1:string, output HYPOTHESES2:
    list_string);
 obtainProperties1 (input HYPOTHESES2:list_string, output
     NPROPERTY1:list_string)
 obtainPropertiesLab1 (input HYPOTHESES2:list_string, output
     NPROPERTYLAB1:list_string)
 validateHypothesis (input PROPERTYL1:list_bool, input
     PROPERTYLAB1:list_bool, output HYPOTHESIS:string)
End_Interface IInferenceMD;
```

2.3 Tipo PRISMA de la Interface IUserClin

```
Interface IUserClinMD
 getDiagnosis (output DIAGNOSIS: string);
 getProperties0 (output PROPERTY0:list_bool);
 getProperties1 (output PROPERTY1:list_bool);
End_Interface IUserClinMD;
```

2.4 Tipo PRISMA de la Interface IUserLab

```
Interface IUserLabMD
 getLabProperties1 (output LABPROPERTY1:list_bool);
End_Interface IUserLabMD;
```

2.5 Tipo PRISMA del Aspecto Funcional del componente Base de Conocimientos

```
Functional Aspect FBaseMD using IDomainMD
 Attributes
    Variables
      cough: bool;
      fever: bool;
      respiratory_dificulty: bool;
      …
      dry_cough: bool;
      phlegmatic_cough: bool;
      coughing for access: bool;
      constant_fever: bool;
      continue_fever: bool;
      high_fever: bool;
      ...
      grave_respiratory_dificulty: bool;
      esquerichia_coli_pharyngeal_exurado: bool;
      eusinofilia_hematic_biometric: bool;
      leucocitosis_hematic_biometric: bool;
      ...
      warth string;
      pneumonia: string;
      crup:string;
      broncholitis:string;
      ...
      …
     Derived       syndrome: string;
      derivation
      {cough=true and fever=true and respiratory_dificulty=true}
      syndrome="warth";

      posible_disease: string;
      derivation
      {syndrome="warth"} posible_disease="pneumonia";
      {syndrome="warth"} posible_disease="crup";
      {syndrome="warth"} posible_disease="bronchilitis";

      disease: string;
      derivation
      {constant_fever=true and greater_39_fever=true and
      2_3_days_fever=true and phlegmatic_cough=true and
      frequent_cough=true and grave_ respiratory_dificulty=true and
      eusinofilia_hematic_biometric=true)} disease="pneumonia";
      …


Services  begin;

  in cleanDB ();
  Valuations    [in cleanDB ()]
    cough:= nil;
    fever:= nil;
    resp_dificulty:= nil;
    dry_cough: nil;
    leucocitosis_hematic_biometric: nil
    syndrome: nil;
    posible_disease: nil;
    disease: nil;
```

```
    ...;

  in inferHypothesis1 (input PROPERTY_0:list_bool, output
    HYPOTHESIS_1:string)
   Valuations
    [in inferHypothesis1 ( )]
    cough, fever, resp_dificulty, .....:= PROPERTY 0,
    HYPOTHESIS1:= warth;

  in inferHypotheses2 (input HYPOTHESIS1:string, output
    HYPOTHESES2:list_string)
   Valuations
   [in inferHypotheses2 ( )]
    warth:= HYPOTHESIS1,
    HYPOTHESES2:= pneumonia, crup, broncholitis;
  in  obtainProperties1 (input HYPOTHESES2:list_string, output
    NPROPERTY1:list_string)
    Valuations
   {posible_disease="pneumonia"} [in obtainProperties1 ()]
       pneumonia:= HYPOTESES2,
       NPROPERTY1:= "phlegmatic_cough", "constant_fever",...;
   {posible_disease="crup"} [in obtainProperties1 ()]
       crup:= HYPOTESES2,
       NPROPERTY1:= "coughing for access", "constant_fever",...;
   {posible_disease="broncholitis"} [in obtainProperties1 ()]
       bronchilitis:= HYPOTESES2,
       NPROPERTY1:= "dry_cough", "fever_high",...;

  in  obtainPropertiesLab1 (input HYPOTHESES2:list_string, output
    NPROPERTYLAB1:list_string)
    Valuations
  {posible_disease="pneumonia"} [in obtainPropertiesLab1 ()]
        pneumonia:= HYPOTESES2,
        NPROPERTYLAB1:= "esquerichia_coli_pharyngeal_exurado";
   {posible_disease="crup"} [in obtainPropertiesLab1 ()]
        crup:= HYPOTESES2,
        NPROPERTYLAB1:= "eusinofilia_hematic_biometric";
   {posible_disease="broncholitis"} [in obtainPropertiesLab1 ()]
        bronchilitis:= HYPOTESES2,
        NPROPERTYLAB1:= "leucocitosis_hematic_biometric";

  in  validateHypothesis (input PROPERTY1:list_bool, input
    PROPERTYLAB1:list_bool, output HYPOTHESIS:string)
    Valuations
    [in  validateHypothesis ()]
    phlegmatic_cough, constant_fever,...:= PROPERTY1,
    esquerichia_coli_pharyngeal_exurado:= PROPERTYLAB1,
    HYPOTHESIS:= pneumonia;
  end;


Played_Roles
  KNOWLEDGE1 for IDomainMD ::=
  cleanDB ? ( )
  →
  inferHypothesis1 ? (PROPERTY0, HYPOTHESIS1)
  →
  inferHypothesis1 ! (PROPERTY0, HYPOTHESIS1)
  →
```

```
  inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2)
  →
  inferHypotheses2 ! (HYPOTHESIS1, HYPOTHESES2)
  →
  obtainProperties1 ? (HYPOTHESES2, NPROPERTY1)
  →
  obtainProperties1 ! (HYPOTHESES2, NPROPERTY1)


  KNOWLEDGE2 for IDomainMD ::=
  obtainPropertiesLab1 ? (HYPOTHESES2, NPROPERTYLAB1)
  →
  obtainPropertiesLab1 ! (HYPOTHESES2, NPROPERTYLAB1)


  KNOWLEDGE3 for IDomainMD ::=
  validateHypothesis ? (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
  →
  validateHypothesis ! (PROPERTY1, PROPERTYLAB1, HYPOTHESIS);

Protocols
  FBASE::= begin ( ):1 → P0

  P0::=KNOWLEDGE1_cleanDB ? ( ):1 → P1

  P1::= ( KNOWLEDGE1_inferHypothesis1 ? (PROPERTY0, HYPOTHESIS1)
       →
       KNOWLEDGE1_inferHypothesis1 ! (PROPERTY0, HYPOTHESIS1)  ) →
       P2

  P2::= (KNOWLEDGE1_inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2)
       →
       KNOWLEDGE1_inferHypotheses2 ! (HYPOTHESIS1,HYPOTHESES2)) →
       P3

  P3::= (KNOWLEDGE1_obtainProperties1 ? (HYPOTHESES2, NPROPERTY1)
       →
       KNOWLEDGE1_obtainProperties1 ! (HYPOTHESES2, NPROPERTY1) ) →
       P4

  P4::= (KNOWLEDGE2_obtainPropertiesLab1 ? (HYPOTHESES2,
       NPROPERTYLAB1)
       →
       KNOWLEDGE1_obtainPropertiesLab1 ! (HYPOTHESES2,
       NPROPERTYLAB1) ) →  P5

  P5::= (KNOWLEDGE3_validateHypothesis ? (PROPERTY1,PROPERTYLAB1
       HYPOTHESIS)
    →
    KNOWLEDGE3_validateHypothesis ! (PROPERTY1, PROPERTYLAB1,
    HYPOTHESIS) ) →  P4

  P4::= end ( ):1;
End_Functional Aspect FBaseMD
```

2.6 Tipo PRISMA del Aspecto Funcional del componente Motor de Inferencia

```
Functional Aspect FInferenceMD using IInferenceMD
  Attributes
     Variables
       cough: bool;
       fever: bool;
       respiratory_dificulty: bool;
       …
       dry_cough: bool;
       phlegmatic_cough: bool;
       coughing for access: bool;
       constant_fever: bool;
       continue_fever: bool;
       high_fever: bool;
       grave_respiratory_dificulty: bool;
       esquerichia_coli_pharyngeal_exurado: bool;
       eusinofilia_hematic_biometric: bool;
       leucocitosis_hematic_biometric: bool;
       ...
       warth string;
       pneumonia: string;
       crup:string;
       broncholitis:string;
       ...

  Services
    begin ( )

    out cleanDB ( );

    in getDiagnosis ( output  DIAGNOSIS:string)
     Valuations
        [in  getDiagnosis (DIAGNOSIS) ]
        DIAGNOSIS:= pneumonia;

    out getProperties0  (output PROPERTY0:list_bool )
     Valuations
        [out  getProperties0  (PROPERTY0) ]
        cough, fever, resp_dificulty, ....:= PROPERTY0;

    out inferHypothesis1 (input PROPERTY0:list_bool, output
        HYPOTHESIS1:string )
     Valuations
        [out  inferProperties1  (cough,fever,resp_dificulty, ...,
        PROPERTY0) ]
        warth:= HYPOTHESIS1;

    out inferHypotheses2 (input HYPOTHESIS1:list, output
        HYPOTHESES2:list_string)
     Valuations
        [out inferHypothesis2 (warth, HYPOTHESES2)]
        pneumonia, crup, broncholitis:= HYPOTHESES2;


    out obtainProperties1 (input HYPOTHESES2:list_string, output
             NPROPERTY1: list_string )
     Valuations
```

```
    {posible_disease="pneumonia"}
    [out obtainProperties1 (pneumonia, NPROPERTY1) ]
    "phlegmatic_cough", "constant_fever",....:= NPROPERTY1;
    {posible_disease="crup"}
    [out  getProperties1 (crup, NPROPERTY1) ]
    "coughing for access", "constant_fever",...:= NPROPERTY1;
    {posible_disease="broncholitis"}
    [out  getProperties1 (broncholitis, NPROPERTY1) ]
    "dry_cough", "fever_high",...:= NPROPERTY1;

 out obtainPropertiesLab1 (input HYPOTHESES2: list_string, output
         NPROPERTYLAB1: list_string)
  Valuations
    {posible_disease="pneumonia"}
    [out getProperties1 (pneumonia,NPROPERTYLAB1)]
    "esquerichia_coli_pharyngeal_exurado":= NPROPERTYLAB1;
    {posible_disease="crup"}
    [out getProperties1 (crup,NPROPERTYLAB1)]
    "eusinofilia_hematic_biometric":= NPROPERTYLAB1;
    {posible_disease="broncholitis"}
    [out getProperties1 (bronchiolitis,NPROPERTYLAB1)]
    "leucocitosis_hematic_biometric":= NPROPERTYLAB1;

 out getProperties1  (output  PROPERTY1: list_bool )
  Valuations
    [out getProperties1 (PROPERTY1) ]
    phlegmatic_cough, constant_fever,. .:= PROPERTY1;

 out getPropertiesLab1 (output PROPERTYLAB1:list_bool )
  Valuations
    [out  getProperties1 (PROPERTYLAB1) ]
    esquerichia_coli_pharyngeal_exurado:= PROPERTYLAB1;

 out validateHypothesis (input PROPERTY1:list_bool, input
    PROPERTYLAB1:list_bool, output HYPOTHESIS: string)
  Valuations
    [out inferHypothesis ("phlegmatic_cough",
    "constant_fever",...,"esquerichia_coli_pharyngeal_exurado",
    HYPOTHESIS)]
    pneumonia:= HYPOTHESIS;

 end;


Played_Roles
 PROCESS1 for IInferenceMD ::=
   getDiagnosis ? (DIAGNOSIS)
   →
   cleanDB ! ( )
   →
   getProperties0 ! (PROPERTY0)
   →
   getProperties0 ? (PROPERTY0)
   →
   inferHypothesis1 ! (PROPERTY0, HYPOTHESIS1)
   →
   inferHypothesis1 ? (PROPERTY0, HYPOTHESIS1)
   →
```

```
   inferHypothesies2 ! (HYPOTHESIS1, HYPOTHESES2)
   →
   inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2)
   →
   obtainProperties1 ! (HYPOTHESES2, NPROPERTY1)
   →
   obtainProperties1 ? (HYPOTHESES2, NPROPERTY1);
   →
   getProperties1 ! (PROPERTY1)
   →
   getProperties1 ? (PROPERTY1);

 PROCESS2 for IInferenceMD ::=
   obtainPropertiesLab1 ! (HYPOTHESES2, NPROPERTYLAB1)
   →
   obtainPropertiesLab1 ? (HYPOTHESES2, NPROPERTYLAB1);
   →
   getPropertiesLab1 ! (PROPERTYLAB1)
   →
   getPropertiesLab1 ? (PROPERTYLAB1);

 PROCESS3 for IInferenceMD ::=
   validateHypothesis ! (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
   →
   validateHypothesis ? (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
   →
   getDiagnosis ! (DIAGNOSIS);


Protocols
   FINFERENCE::= begin ( ):1 → P0

   P0::= PROCESS1_getDiagnosis ? (DIAGNOSIS) → P1

   P1::= PROCESS1_cleanDB ! ( ):1 → P2

   P2::= ( PROCESS1_getProperties0 ! (PROPERTY0)
           →
           PROCESS1_getProperties0 ? (PROPERTY0) )
           → P3

   P3::=( PROCESS1_inferHypothesis1 ! (PROPERTY0,HYPOTHESIS1)
          →
         PROCESS1_inferHypothesis1 ? (PROPERTY0,HYPOTHESIS1))  → P4

   P4::= ( PROCESS1_inferHypotheses2 ! (HYPOTHESIS1,  HYPOTHESES2)
         →
         PROCESS1_inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2) )
   → P5

   P5::= (
          ( PROCESS1_obtainPropierties1 ! (HYPOTHESES2, NPROPERTY1)
          →
          PROCESS1_obtainPropierties1 ? (HYPOTHESES2, NPROPERTY1) )
   ^
         (PROCESS2_obtainPropiertiesLab1 ! (HYPOTHESES2,
          NPROPERTYLAB1)
```

```
            →
            PROCESS2_obtainPropiertiesLab1 ? (HYPOTHESES2,
            NPROPERTYLAB1) )
            )   → P6

     P6::=
         ( ( PROCESS1_getPropierties1 ! (PROPERTY1)
            →
            PROCESS1_getPropierties1 ? (PROPERTY1) )
     ^

          ( PROCESS2_getPropiertiesLab1 ! (PROPERTYLAB1)
            →
            PROCESS2_getPropiertiesLab1 ? (PROPERTYLAB1) )
          )   → P7

     P7::= (PROCESS3_validateHypothesis ! (PROPERTY1, PROPERTYLAB1,
          HYPOTHESIS)
            →
           PROCESS3_validateHypothesis ? (PROPERTY1,PROPERTYLAB1,
          HYPOTHESIS))  → P8

     P8::= PROCESS3_getDiagnosis ! (DIAGNOSIS) → P9

     P9::= end ( ):1;
 End_Functional Aspect FInferenceMD;
```

2.7 Tipo PRISMA del Aspecto Funcional del Componente Usuario Clínico

```
Functional Aspect  FUserClinMD  using IUserClinMD
  Attributes
     Variables
     <FP.0>,
     <FP.1>,
     <FH.2>;

  Services
    begin ( )

    out getDiagnosis ( output  DIAGNOSIS: string)
      Valuations
      [out  getDiagnosis (DIAGNOSIS) ]
      pneumonia:= DIAGNOSIS;

    in getProperties0  (output  PROPERTY0: list_bool )
      Valuations
      [out  getProperties0  (PROPERTY0) ]
      PROPERTY0:= cough,fever,resp_dificulty,... ;

    writeProperties0  (output "cough","fever", "resp_dificulty",...,
    input PROPERTY0: list_bool)

    in getProperties1  (output PROPERTY1: list_bool )
      Valuations
      [out getProperties1 (PROPERTY1) ]
```

```
      PROPERTY1:= phlegmatic_cough, constant_fever;

    writeProperties1  (output "phlegmatic_cough", "constant_fever",..,
    input PROPERTY1:list_bool)

    writeDiagnosis (output "pneumonia", output DIAGNOSIS:string)

    end;

  Played_Roles
    USER1 for IUserClinMD ::=
      getDiagnosis ! (DIAGNOSIS)
      →
      getProperties0 ? (PROPERTY0)
      →
      getProperties0 ! (PROPERTY0)
      →
      getProperties1 ? (PROPERTY1)
      →
      getProperties1 ! (PROPERTY1)

    USER3 for IUserClinMD ::=
      getDiagnosis ? (DIAGNOSIS);

  Protocols
    FUSERCLIN::= begin ( ):1 → P0

    P0::= USER1_getDiagnosis ! (DIAGNOSIS) → P1

    P2::= writeProperties0 ("cough","fever", "resp_dificulty",..,
    PROPERTY1) → P3

    P3::= ( USER1_getProperties0 ? (PROPERTY0)
            →
            USER1_getProperties0 ! (PROPERTY0) )  → P4

    P4::= writeProperties1 ("phlegmatic_cough", "constant_fever",..,
    PROPERTY1) → P5

    P5::= (USER1_getProperties1 ? (PROPERTY1)
           →
           USER1_getProperties1 ! (PROPERTY1) ) → P6

    P6::= USER3_getDiagnosis ? (DIAGNOSIS) → P7

    P7::= writeDiagnosis ("pneumonia", DIAGNOSIS) → P8

    P8::= end ( ):1;
End_Functional Aspect FUserClinMD;
```

2.8 Tipo PRISMA del Aspecto Funcional del Componente Usuario Laboratorio

```
Functional Aspect  FUserLabMD  using IUserLabMD
  Attributes
      Variables
```

```
      <FPL.1>,
      <FH>;

  Services
    begin  ( )

    in getPropertiesLab1  (output PROPERTYLAB1: list_bool )
      Valuations
      [out  getPropertiesLab1  (PROPERTYLAB1) ]
      PROPERTY1:= <FPL.1>;

    writeProperties1  (output "esquerichia_coli_pharyngeal_exurado",
    input PROPERTY1:list_bool)

    end;


  Played_Roles
    USER2 for IUserMD ::=
      getProperties1 ? (PROPERTY1)
      →
      getProperties1 ! (PROPERTY1)


  Protocols
    FUSERLAB::= begin ( ):1 → P0

    P0::= writeProperties1 ("esquerichia_coli_pharyngeal_exurado",
    PROPERTY1) → P1

    P1::= ( PROCESS_getProperties1 ? (PROPERTY1)
            →
            PROCESS_getProperties1 ! (PROPERTY1) )  → P2

    P2::= end ( ):1;
End_Functional Aspect FUserLabMD;
```

2.9 Tipo PRISMA del Aspecto de Coordinación del conector Coordinador1

```
Coordination Aspect  C1DiagnosisMD using IInferenceMD, IDomainMD,
IUserClinMD
    Services
      begin ( );

      in/out  cleanDB ( );

      in/out getDiagnosis ( output  DIAGNOSIS: string);

      in/out getProperties0 (output  PROPERTY0: list );

      in/out inferHypothesis1 (input PROPERTY0: list_bool, output
          HYPOTHESIS1: list);

      in/out inferHypotheses2 (input HYPOTHESIS1:string,  output
          HYPOTHESES2: list_string);
```

```
    in/out obtainProperties1 (input HYPOTHESES2: list_string,
        NPROPERTY1:list_string);

    in/out getProperties1 (output PROPERTY1: list_bool);

    end;

  Played_Roles
    PROCESSCNCT1 for IInferenceMD ::=
      getDiagnosis ! (DIAGNOSIS)
      →
      cleanDB ? ( )
      →
      getProperties0 ? (PROPERTY0)
      →
      getProperties0 ! (PROPERTY0)
      →
      inferHypothesis1 ? (PROPERTY0, HYPOTHESIS1)
      →
      inferHypothesis1 ! (PROPERTY0, HYPOTHESIS1)
      →
      inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2)
      →
      inferHypotheses2 ! (HYPOTHESIS1, HYPOTHESES2)
      →
      obtainProperties1 ? (HYPOTHESES2, NPROPERTY1)
      →
      obtainProperties1 ! (HYPOTHESES2, NPROPERTY1);
      →
      getProperties1 ? (PROPERTY1)
      →
      getProperties1 ! (PROPERTY1);

    KNOWLEDGECNCT1 for IDomainMD ::=
      cleanDB ! ( )
      →
      inferHypothesis1 ! (PROPERTY0, HYPOTHESIS1)
      →
      inferHypothesis1 ? (PROPERTY0, HYPOTHESIS1)
      →
      inferHypotheses2 ! (HYPOTHESIS1, HYPOTHESES2)
      →
      inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2);
      →
      obtainProperties1 ! (HYPOTHESES2, NPROPERTY1)
      →
      obtainProperties1 ? (HYPOTHESES2, NPROPERTY1);

    USERCNCT1 for IUserMD ::=
      getDiagnosis ? (DIAGNOSIS)
      →
      getProperties0 ! (PROPERTY0)
      →
      getProperties0 ? (PROPERTY0)
      →
      getProperties1 ! (PROPERTY1)
      →
      getProperties1 ? (PROPERTY1);
```

```
  Protocols
    C1DIAGNOSIS::= begin ( ):1 → P0

    P0::= ( USERCNCT1_getDiagnosis ? (DIAGNOSIS)
          →
          PROCESSCNCT1_getDiagnosis ! (DIAGNOSIS)  ) →  P1

    P1::= ( PROCESSCNCT1_cleanDB ? ( )
          →
          KNOWLEDGECNCT1_cleanDB ! ( ) ) → P2

    P2::= ( PROCESSCNCT1_getProperties0 ? (PROPERTY0)
          →
          USERCNCT1_getProperties0 ! (PROPERTY0)
          →
          USERCNCT1_getProperties0 ? (PROPERTY0)
          →
          PROCESSCNCT1_getProperties0 ! (PROPERTY0) )→ P3

    P3::= ( PROCESSCNCT1_inferHypothesis1 ? (PROPERTY0,HYPOTHESIS1)
          →
          KNOWLEDGECNCT1_ inferHypothesis1! (PROPERTY0, HYPOTHESIS1)
          →
          KNOWLEDGECNCT1_ inferHypothesis1? (PROPERTY0, HYPOTHESIS1)
          →
          PROCESSCNCT1_ inferHypothesis1! (PROPERTY0, HYPOTHESIS1)
          ) → P4

    P4::= (PROCESSCNCT1_inferHypotheses2 ? (HYPOTHESIS1, HYPOTHESES2)
          →
          KNOWLEDGECNCT1_inferHypotheses2! (HYPOTHESIS1, HYPOTHESES2)
          →
          KNOWLEDGECNCT_inferHypotheses2? (HYPOTHESIS1, HYPOTHESES2)
          →
          PROCESSCNCT_inferHypotheses2! (HYPOTHESIS1,HYPOTHESES2)
          ) → P5

    P5::= (PROCESSCNCT1_obtainProperties1 ? (HYPOTHESES2, NPROPERTY1)
          →
          KNOWLEDGECNCT1_obtainProperties1 ! (HYPOTHESES2, NPROPERTY1)
          →
          KNOWLEDGECNCT_obtainProperties1 ? (HYPOTHESES2, NPROPERTY1)
          →
          PROCESSCNCT_obtainProperties1 ! (HYPOTHESES2, NPROPERTY1)
          ) → P6

    P6::= (PROCESSCNCT1_getProperties1 ? (PROPERTY1)
          →
          KNOWLEDGECNCT1_getProperties1 ! (PROPERTY1)
          →
          KNOWLEDGECNCT_getProperties1 ? (PROPERTY1)
          →
          PROCESSCNCT_getProperties1 ! (PROPERTY1) ) → P7

    P7::= end ( ):1;
End_Coordination Aspect C1DiagnosisMD;
```

2.10 Tipo PRISMA del Aspecto de Coordinación del conector Coordinador2

```
Coordination Aspect  C2DiagnosisMD using IInferenceMD, IDomainMD,
IUserLabMD
    Services
      begin ( );

      in/out obtainPropertiesLab1 (input HYPOTHESES2: list_string,
          output NPROPERTYLAB1: list_string);

      in/out getPropertiesLab1 (output PROPERTYLAB1: list_bool);

      end;

    Played_Roles
      PROCESSCNCT2 for IInferenceMD ::=
        obtainPropertiesLab1 ? (HYPOTHESES2, NPROPERTYLAB1)
        →
        obtainPropertiesLab1 ! (HYPOTHESES2, NPROPERTYLAB1)
        →
        getProperties1 ? (PROPERTY1)
        →
        getProperties1 ! (PROPERTY1);

      KNOWLEDGECNCT2 for IDomainMD ::=
        obtainPropertiesLab1 ! (HYPOTHESES2, NPROPERTYLAB1)
        →
        obtainPropertiesLab1 ? (HYPOTHESES2, NPROPERTYLAB1)

      USERCNCT2 for IUserLabMD ::=
        getProperties1 ! (PROPERTY1)
        →
        getProperties1 ? (PROPERTY1);

    Protocols
      C2DIAGNOSIS::= begin ( ):1 → P0
      P0::= (PROCESSCNCT2_obtainPropertiesLab1 ? (HYPOTHESES2,
            NPROPERTYLAB1)
            →
            KNOWLEDGECNCT2_obtainPropertiesLab1 ! (HYPOTHESES2,
            NPROPERTYLAB1)
            →
            KNOWLEDGECNCT2_obtainPropertiesLab1 ? (HYPOTHESES2,
            NPROPERTYLAB1)
            →
            PROCESSCNCT2_obtainPropertiesLab1 ! (HYPOTHESES2,
            NPROPERTYLAB1)   ) → P1

      P1::= (PROCESSCNCT1_getPropertiesLab1 ? (PROPERTYLAB1)
            →
            USERCNCT1_getPropertiesLab1 ! (PROPERTYLAB1)
            →
            USERCNCT1_getPropertiesLab1 ? (PROPERTYLAB1)
            →
            PROCESSCNCT1_getPropertiesLab1 ! (PROPERTYLAB1))
```

```
          → P2

     P2::= end ( ):1;
End_Coordination Aspect C2DiagnosisMD;
```

2.11 Tipo PRISMA del Aspecto de Coordinación del conector Coordinador3

```
Coordination Aspect  C3DiagnosisMD using IInferenceMD, IDomainMD,
IUserClinMD
    Services
     begin ( );

     in/out getDiagnosis ( output DIAGNOSIS: string);

     in/out validateHypothesis (input PROPERTY1: list_bool, input
        PROPERTYLAB1:list_bool, output HYPOTHESIS: string);

     end;

    Played_Roles
     PROCESSCNCT3 for IInferenceMD ::=
       validateHypothesis ? (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
       →
       validateHypothesis ! (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
       →
       getDiagnosis ? (DIAGNOSIS);

     KNOWLEDGECNCT3 for IDomainMD ::=
       validateHypothesis ! (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)
       →
       validateHypothesis ? (PROPERTY1, PROPERTYLAB1, HYPOTHESIS)


     USERCNCT3 for IUserMD ::=
       getDiagnosis ! (DIAGNOSIS)

    Protocols
     C3DIAGNOSIS::= begin ( ):1 → P0

     P0::= ( PROCESSCNCT3_validateHypothesis ? (PROPERTY1, HYPOTHESIS)
          →
          KNOWLEDGECNCT3_validateHypothesis ! (PROPERTY1, HYPOTHESIS)
          →
          KNOWLEDGECNCT3_validateHypothesis ? (PROPERTY1, HYPOTHESIS)
          →
          PROCESSCNCT3_validateHypothesis ! (PROPERTY1, HYPOTHESIS) )
          → P3

     P1::= (PROCESSCNCT1_getDiagnosis ? (DIAGNOSIS)
          →
          USERCNCT1_getDiagnosis ! (DIAGNOSIS)  ) →  P2


     P2::= end ( ):1;
      End_Coordination Aspect C3DiagnosisMD;
```

2.12 Tipo PRISMA del componente Base de Conocimientos

```
Component KnowledgeBaseMD
  Ports
      PortKnow1: IDomainMD,
      Played_Role  FBaseMD.KNOWLEDGE1;
      PortKnow2: IDomainMD,
      Played_Role  FBaseMD.KNOWLEDGE2;
      PortKnow3: IDomainMD,
      Played_Role  FBaseMD.KNOWLEDGE3;
  End_Ports;

  Function Aspect import  FBaseMD;

  Initialize
      new ( )
      {  FBaseMD.begin ();
      }
  End_Initialize;

  Destruction
      destroy ()
      {  FBaseMD.end();
          }
  End_Destruction;
End_Connector  KnowledgeBaseMD;
```

2.13 Tipo PRISMA del componente Motor de Inferencia

```
Component InferenceMotorMD
  Ports
      PortMotor1: IInferenceMD,
      Played_Role  FInferenceMD.PROCESS1;
      PortMotor2: IInferenceMD,
      Played_Role  FInferenceMD.PROCESS2;
      PortMotor3: IInferenceMD,
      Played_Role  FInferenceMD.PROCESS3;
  End_Ports;

  Function Aspect import  FInferenceMD;

  Initialize
      new ( )
      {  FInferenceMD.begin ();
      }
  End_Initialize;

  Destruction
      destroy ()
      {  FInferenceMD.end();
      }
  End_Destruction;
End_Connector  KnowledgeBaseMD;
```

2.14 Tipo PRISMA del componente Usuario Clinico

```
Component UserInterfaceClinMD
  Ports
      Port1Op: IUserClinMD,
      Played_Role  FUserClinMD.USER1;
      Port3Op: IOperatorClinMD,
      Played_Role  FUserClinMD.USER3;
  End_Ports;

  Function Aspect import  FUserClinMD;

  Initialize
      new ( )
      {  FUserClinMD.begin ();
      }
  End_Initialize;

  Destruction
      destroy ()
      {  FUserClinMD.end();
      }
  End_Destruction;
End_Connector  UserInterfaceClinMD;
```

2.15 Tipo PRISMA del componente Usuario Laboratorio

```
Component UserInterfaceLabMD
  Ports
      Port2Op: IUserLabMD,
      Played_Role  FUserLabMD.USER2;
  End_Ports;

  Function Aspect import  FUserLabMD;

  Initialize
      new ( )
      {  FUserLabMD.begin ();
      }
  End_Initialize;

  Destruction
      destroy ()
      {  FUserLabMD.end();
      }
  End_Destruction;
End_Connector  UserInterfaceLabMD;
```

2.16 Tipo PRISMA del conector Coordinador1

```
Component Coordinator1MD
  Ports
      PortCnct1Inf: IInferenceMD,
      Played_Role  C1DiagnosisMD.PROCESSCNCT1;
```

```
      PortCnct1Know: IDomainMD,
      Played_Role  C1DiagnosisMD.KNOWLEDGECNCT1;
      PortCnct1Op: IUserClinMD,
      Played_Role  C1DiagnosisMD.USERCNCT1;
  End_Ports;

  Coordination Aspect import  C1DiagnosisMD;

  Initialize
      new ( )
      {  C1DiagnosisMD.begin ();
      }
  End_Initialize;

  Destruction
      destroy ()
      {  C1DiagnosisMD.end();
      }
  End_Destruction;
End_Connector  Coordinator1MD;
```

2.17 Tipo PRISMA del conector Coordinador2

```
Component Coordinator2MD
  Ports
      PortCnct2Inf: IInferenceMD,
      Played_Role  C2DiagnosisMD.PROCESSCNCT2;
      PortCnct2Know: IDomainMD,
      Played_Role  C2DiagnosisMD.KNOWLEDGECNCT2;
      PortCnct2Op: IUserLabMD,
      Played_Role  C2DiagnosisMD.USERCNCT2;
  End_Ports;

  Coordination Aspect import  C2DiagnosisMD;

  Initialize
      new ( )
      {  C2DiagnosisMD.begin ();
      }
  End_Initialize;
  Destruction
      destroy ()
      {  C2DiagnosisMD.end();
      }
  End_Destruction;
End_Connector  Coordinator2MD;
```

2.18 Tipo PRISMA del conector Coordinador3

```
Component Coordinator3MD
  Ports
      PortCnct3Inf: IInferenceMD,
      Played_Role  C3DiagnosisMD.PROCESSCNCT3;
      PortCnct3Know: IDomainMD,
```

```
      Played_Role  C3DiagnosisMD.KNOWLEDGECNCT3;
      PortCnct3Op: IUserClinMD,
      Played_Role  C3DiagnosisMD.USERCNCT3;
  End_Ports;

  Coordination Aspect import  C3DiagnosisMD;

  Initialize
      new ( )
      {  C3DiagnosisMD.begin ();
      }
  End_Initialize;

  Destruction
      destroy ()
      {  C3DiagnosisMD.end();
      }
  End_Destruction;
End_Connector  Coordinator3MD;
```

3- Modelo Arquitectónico

```
Architectural_Model DiagnosticModelMD
    Import Architectural Elements  InferenceMotorMD, KnowledgeBaseMD,
      UserInterfaceClinMD, UserInterfaceLabMD, Coordinator1MD,
      Coordinator2MD, Coordinator3MD;

    Attachments
      AttchCnct1Inf:  Coordinator1MD.PortCnct1Inf (1,1)
        ↔  InferenceMotorMD.Port1Inf (1,1);
      AttchCnct1Know:  Coordinator1MD.PortCnct1Know (1,1)  ↔
        KnowledgeBaseMD.Port1Know (1,1);
      AttchCnct1Op: Coordinator1MD.PortCnct1Op (1,1)
        ↔ UserInterfaceClinMD.Port1Op (1,1);
      AttchCnct2Inf:  Coordinator2MD.PortCnct2Inf (1,1)
        ↔  InferenceMotorMD.Port1Inf (1,1);
      AttchCnct2Know:  Coordinator2MD.PortCnct2Know (1,1)  ↔
        KnowledgeBaseMD.Port2Know (1,1);
      AttchCnct2Op: Coordinator2MD.PortCnct2Op (1,1)
        ↔ UserInterfaceLabMD.Port2Op (1,1);
      AttchCnct3Inf:  Coordinator3MD.PortCnct3Inf (1,1)
        ↔  InferenceMotorMD.Port3Inf (1,1);
      AttchCnct3Know:  Coordinator3MD.PortCnct3Know (1,1)  ↔
        KnowledgeBaseMD.Port3Know (1,1);
      AttchCnct3Op: Coordinator3MD.PortCnct3Op (1,1)
        ↔ UserInterfaceClinMD.Port3Op (1,1);
    End_Attachments;

    new ( )
      {
       new InferenceMotorMD;
       new KnowledgeBaseMD;
       new UserInterfaceClinMD;
       new UserInterfaceLabMD;
       new Coordinator1MD;
       new Coordinator2MD;
```

```
      new Coordinator3MD;
      }

    destroy ()
       {
      destroy ( ) InferenceMotorMD;
      destroy ( ) KnowledgeBaseMD;
      destroy ( ) UserInterfaceClinMD;
      destroy ( ) UserInterfaceLabMD;
      destroy ( ) Coordinator1MD;
      destroy ( ) Coordinator2MD;
      destroy ( ) Coordinator3MD;
       }
End_Architectural_Model  DiagnosticModelEP-TV;
```

4- Configuración del Modelo Arquitectónico

```
Architectural_Model_ConfigurationMD
   DIAGNOSIS_MD = new  DiagnosticModelMD
    {
       MOTOR_MD =  new InferenceMotorMD;
       BASE_MD =  new KnowledgeBaseMD;
       USERCLIN_MD =  new UserInterfaceClinMD;
       USERLAB_MD =  new UserInterfaceLabMD;
       JOINT1_MD= new Coordinator1MD;
       JOINT2_MD= new Coordinator2MD;
       JOINT3_MD= new Coordinator3MD;
       Motor_MD_AttchCnct1Inf = new AttchCnct1Inf (Joint1_MD,
          PortCnct1Inf, Motor_MD, PortInf);
       Base_MD_AttchCnct1Know = new AttchCnct1Know (Joint1_MD,
          PortCnct1Know, Base_MD, PortKnow);
       User_MD_AttchCnct1OpClin = new AttchCnct1OpClin (Joint1_MD,
          PortCnct1OpClin, UserClin_MD, PortOpClin);
       Motor_MD_AttchCnct2Inf = new AttchCnct2Inf (Joint2_MD,
          PortCnct2Inf, Motor_MD, PortInf);
       Base_MD_AttchCnct2Know = new AttchCnct2Know (Joint2_MD,
          PortCnct2Know, Base_MD, PortKnow);
       User_MD_AttchCnct2OpClin = new AttchCnct2OpClin (Joint2_MD,
          PortCnct2OpClin, UserClin_MD, PortOpClin);
       Motor_MD_AttchCnct3Inf = new AttchCnct3Inf (Joint3_MD,
          PortCnct3Inf, Motor_MD, PortInf);
       Base_MD_AttchCnct3Know = new AttchCnct3Know (Joint3_MD,
          PortCnct3Know, Base_MD, PortKnow);
       User_MD_AttchCnct3OpLab = new AttchCnct3OpLab (Joint3_MD,
          PortCnct3OpLab, UserLab_MD, PortOpLab);
    }
  End_Architectural_Model_Configuration
```

APÉNDICE E

IMPLEMENTACIÓN DE PROTOBOM

En este apéndice se presenta lo más destacado de la implementación[7] del prototipo de BOM. Se comenta sobre las caracterísiticas generales de ProtoBOM y se incorpora la codificación más representativa como aportación de esta herramienta: el servicio Web y el proceso de inserción de caracterísiticas.

ProtoBOM es una aplicación desarrollada sobre el sistema operativo Windows XP, sin embargo es compatible con otras versiones de Windows (Windows Vista y versiones superiores a Windows 3.1). Esta aplicación fue desarrollada empleando el lenguaje de programación orientada a objetos *C#* . NET.

En ProtoBOM, la Baseline ha sido implementada como una base de datos que se accede con un servicio Web, como lo muestra la **figura 149.**

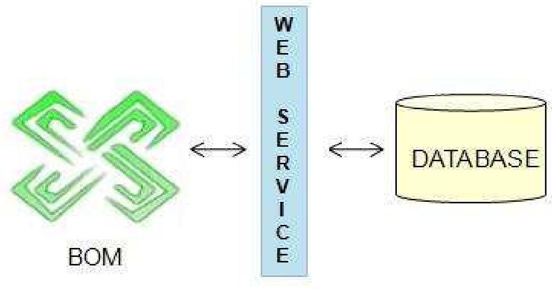


Figura 149 Arquitectura de ProtoBOM

[7] Realizada por María Gómez Lacruz

Para el acceso a la base de datos ha sido utilizado SQL Server 2005 y para el servicio Web se ha utilizado ASP.Net.

La base de datos debe ser añadida a una herramienta gestora de base de datos (en nuestro caso SQL Server Management Express). La base de datos está compuesta por las tablas de la **figura 150**.

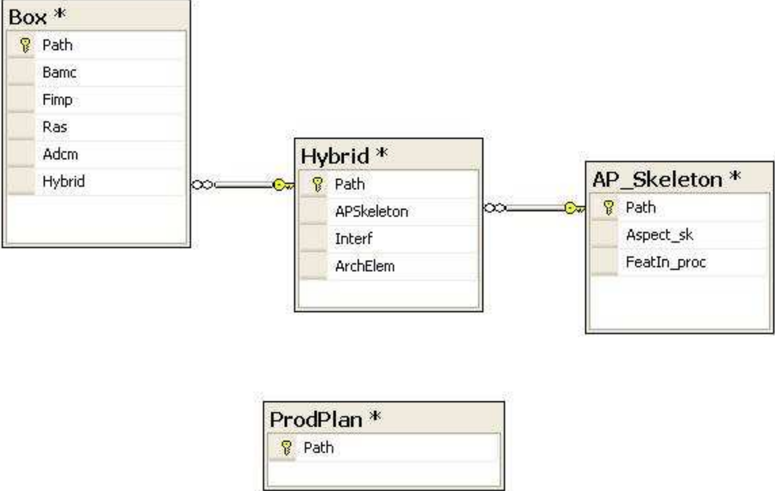


Figura 150 Tablas de la base de datos

Si no se tiene un sitio Web para que ProtoBOM trabaje con un servicio Web (como en nuestro caso), dicho servicio debe ser implementado sobre el servidor local de windows (localhost). En nuestro caso usamos Internet Information System.

ProtoBOM no requiere de algún hardware específico. Las limitaciones de hardware estarán determinadas por el ordenador en el cual trabaje la aplicación.

Implementación del servicio Web

En un sentido amplio, un servicio Web es un método o conjunto de métodos que son públicamente disponibles para aplicaciones remotas sobre el Hypertext Transfer Protocol (HTTP). La implementación de un servicio Web es completamente blindado a la aplicación que invoca al servicio Web. Dicha aplicación sólo conoce las reglas para interactuar con el servicio Web. De esta manera, la línea de productos software puede ser fácilmente compartida y distribuida.
Todas las operaciones para la creación de la base de datos son llevadas a cabo por el servicio Web. Para la inserción de las Kit-Boxes en la Baseline, así como la inserción de cada uno de los artefactos en la Baseline, ProtoBOM invoca uno de los métodos del servicio Web. El servicio Web es el encargado de llevar a cabo la

tarea. Las cabeceras de los métodos Web correspondientes para cada artefacto son las siguientes:

```
[WebMethod]
public void InsertBamc(string boxPath, string artifactName, byte[]
content)

[WebMethod]
public void InsertFimp(string boxPath,string artifactName,byte[]
content)

[WebMethod]
public void InsertRasma(string boxPath, string artifactName, byte[]
content)

[WebMethod]
public void InsertAdcm (string boxPath,string artifactName,byte[]
content)
```

ProtoBOM se encarga de invocar los servicios. El servicio Web es el encargado de servirle las tareas. ProtoBOM invocará el método Web correspondiente al artefacto que quiera insertar en la Baseline de la siguiente manera:

```
***CÓDIGO EN ProtoBOM para inserción de artefactos en la baseline***

private void buttonAdd_Click(object sender, EventArgs e)  {

string caja = "c:\\baseline\\" + labelBox.Text;
FileStream str = new FileStream(textBoxPath.Text, FileMode.Open);
BinaryReader rea = new BinaryReader(str);

BaselineServer.Service ser = new BOM.BaselineServer.Service();

switch (numArtifact)   {

case 1:
       ser.InsertBamc(caja,
       textBoxArtif.Text,rea.ReadBytes(int.Parse(rea.BaseStream.Length.T
       oString())));
      break;

case 2:
       ser.InsertFimp(caja,
       textBoxArtif.Text,rea.ReadBytes(int.Parse(rea.BaseStream.Length.T
       oString())));
      break;

case 3:
      ser.InsertRasma(caja, textBoxArtif.Text,
      rea.ReadBytes(int.Parse(rea.BaseStream.Length.ToString())));
```

```
     break;

case 4:
      ser.InsertAdcm(caja, textBoxArtif.Text,
      rea.ReadBytes(int.Parse(rea.BaseStream.Length.ToString())));
     break;

default:
     break;

}
rea.Close();


System.Windows.Forms.MessageBox.Show(this, "The artifact was
successfully inserted.", "Information", MessageBoxButtons.OK,
MessageBoxIcon.Information);
}
```

El código completo para el servicio Web se presenta a continuación:

```
***********************************************************************
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.IO;
using System.Data.SqlClient;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{
    public Service () { }


/** Métodos utilizados en Form CreateBl **/
 [WebMethod]
    public int InsertBox(string name)
    {
        if (!Directory.Exists("c:\\baseline\\" + name))
        {
            SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
            SqlCommand cmd = new SqlCommand("INSERT INTO Box (Path)
VALUES (@Path)", con);
            cmd.Parameters.Add(new SqlParameter("@Path",
"c:\\baseline\\" + name));

            con.Open();
            cmd.ExecuteNonQuery();
            con.Close();

            Directory.CreateDirectory("c:\\baseline\\" + name);
```

```
            return 0;
        }
        else return -1;
    }

 [WebMethod]
    public void InsertProdPlan(string name, byte[] content)
    {
        if (!Directory.Exists("c:\\baseline\\"))
            Directory.CreateDirectory("c:\\baseline\\");

        FileStream str = new FileStream("c:\\baseline\\" + name,
FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
        wri.Write(content);
        wri.Flush();
        wri.Close();

        SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
        SqlCommand cmd = new SqlCommand("INSERT INTO ProdPlan (Path)
VALUES (@Path)", con);
        cmd.Parameters.Add(new SqlParameter("@Path", "c:\\baseline\\" +
name));

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();
    }



/** Métodos utilizados en FormArtifacts **/

 [WebMethod]
    public string[] GetBoxes()
    {
        return Directory.GetDirectories("c:\\baseline");
    }

 [WebMethod]
    public void InsertBamc(string cajaPath, string artifactName, byte[]
content)
    {
        SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
        SqlCommand cmd = new SqlCommand("UPDATE Box SET
Box.Bamc='"+cajaPath+"' + '\\' + '"+artifactName+"' WHERE
Box.Path='"+cajaPath+"' ",con);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();

        FileStream str = new FileStream(cajaPath + "\\" + artifactName,
FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
```

```
        wri.Write(content);
        wri.Flush();
        wri.Close();
    }

 [WebMethod]
    public void InsertFimp(string cajaPath, string artifactName, byte[]
content)
    {
        SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
        SqlCommand cmd = new SqlCommand("UPDATE Box SET Box.Fimp='" +
cajaPath + "' + '\\' + '" + artifactName + "' WHERE Box.Path='" +
cajaPath + "' ", con);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();

        FileStream str = new FileStream(cajaPath + "\\" + artifactName,
FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
        wri.Write(content);
        wri.Flush();
        wri.Close();
    }

 [WebMethod]
    public void InsertRasma(string cajaPath, string artifactName, byte[]
content)
    {
        SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
        SqlCommand cmd = new SqlCommand("UPDATE Box SET Box.Ras='" +
cajaPath + "' + '\\' + '" + artifactName + "' WHERE Box.Path='" +
cajaPath + "' ", con);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();

        FileStream str = new FileStream(cajaPath + "\\" + artifactName,
FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
        wri.Write(content);
        wri.Flush();
        wri.Close();
    }

 [WebMethod]
    public void InsertAdcm(string cajaPath, string artifactName, byte[]
content)
    {
        SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
```

```
        SqlCommand cmd = new SqlCommand("UPDATE Box SET Box.Adcm='" +
cajaPath + "' + '\\' + '" + artifactName + "' WHERE Box.Path='" +
cajaPath + "' ", con);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();

        FileStream str = new FileStream(cajaPath + "\\" + artifactName,
FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
        wri.Write(content);
        wri.Flush();
        wri.Close();
    }

 [WebMethod]
    public void InsertIamt(string cajaPath, string artifactName, byte[]
content)
    {
        SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
        SqlCommand cmd = new SqlCommand("UPDATE Box SET Box.Iamt='" +
cajaPath + "' + '\\' + '" + artifactName + "' WHERE Box.Path='" +
cajaPath + "' ", con);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();

        FileStream str = new FileStream(cajaPath + "\\" + artifactName,
FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
        wri.Write(content);
        wri.Flush();
        wri.Close();
    }



 [WebMethod]
    public int InsertHybrid(string cajaPath, string artifactName)
    {
        if (!Directory.Exists(cajaPath + "\\" + artifactName))
        {
            SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
            SqlCommand cmd = new SqlCommand("INSERT INTO Hybrid (Path)
VALUES (@Path)", con);
            cmd.Parameters.Add(new SqlParameter("@Path", cajaPath + "\\"
+ artifactName));
            SqlCommand cmd1 = new SqlCommand("UPDATE Box SET
Box.Hybrid='" + cajaPath + "' + '\\' + '" + artifactName + "' WHERE
Box.Path='" + cajaPath + "' ", con);

            con.Open();
            cmd.ExecuteNonQuery();
```

```
            con.Close();

            con.Open();
            cmd1.ExecuteNonQuery();
            con.Close();

            Directory.CreateDirectory(cajaPath + "\\" + artifactName);
            return 0;
        }
        else return -1;
    }


//Añade interfaz dentro del hibrido
 [WebMethod]
    public void InsertInterf(string boxPath,string hybridName, string 
interfName, byte[] content)
    {
        SqlConnection con = new SqlConnection("Data 
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated 
Security=False;User ID=uno;Password=uno");
        SqlCommand cmd = new SqlCommand("UPDATE Hybrid SET 
Hybrid.Interf='" + boxPath + "' + '\\' + '" + hybridName + "' + '\\' + 
'" + interfName + "' WHERE Hybrid.Path='"+boxPath+"' + '\\' + '" + 
hybridName + "' ", con);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();

        FileStream str = new FileStream(boxPath + "\\" + hybridName + 
"\\" + interfName, FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
        wri.Write(content);
        wri.Flush();
        wri.Close();
    }


//Añade Arch. Elem. dentro del hibrido
 [WebMethod]
    public void InsertArchElem(string boxPath, string hybridName, string 
archName, byte[] content)
    {
        SqlConnection con = new SqlConnection("Data 
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated 
Security=False;User ID=uno;Password=uno");
        SqlCommand cmd = new SqlCommand("UPDATE Hybrid SET 
Hybrid.ArchElem='" + boxPath + "' + '\\' + '" + hybridName + "' + '\\' + 
'" + archName + "' WHERE Hybrid.Path='" + boxPath + "' + '\\' + '" + 
hybridName + "' ", con);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();

        FileStream str = new FileStream(boxPath + "\\" + hybridName + 
"\\" + archName, FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
```

```
        wri.Write(content);
        wri.Flush();
        wri.Close();
    }


/** Métodos utilizados en FormAPSkeleton **/
 [WebMethod]
    public int InsertAPSkeleton(string cajaPath, string hybridName,
string APSName)
    {
        if (!Directory.Exists(cajaPath + "\\" + hybridName + "\\" +
APSName))
        {
            Directory.CreateDirectory(cajaPath + "\\" + hybridName +
"\\" + APSName);

            SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
            SqlCommand cmd = new SqlCommand("UPDATE Hybrid SET
Hybrid.APSkeleton='" + cajaPath + "' + '\\' + '" + hybridName +
"'+'\\'+'"+APSName+"' WHERE Hybrid.Path='" + cajaPath + "' + '\\' + '"
+ hybridName+"' ", con);

            con.Open();
            cmd.ExecuteNonQuery();
            con.Close();

            return 0;
        }
        else return -1;

    }

[WebMethod]
    public void InsertAPSkelArtifacts(string cajaPath, string
hybridName, string APSName,string name, byte[] content)
    {
        FileStream str = new FileStream(cajaPath + "\\" + hybridName +
"\\" + APSName + "\\" + name, FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);

        wri.Write(content);
        wri.Flush();
        wri.Close();
    }


[WebMethod]
    public void deleteBox(string boxPath)
    {
        SqlConnection con = new SqlConnection("Data
Source=.\\SQLEXPRESS;Initial Catalog=Baseline;Integrated
Security=False;User ID=uno;Password=uno");
        SqlCommand cmd = new SqlCommand("DELETE FROM Box WHERE
Path=@Path", con);
        cmd.Parameters.Add(new SqlParameter("@Path", boxPath));
```

```
        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();
    }


}
************************************************************************
```

Implementación del proceso de inserción de características

Los aspectos esqueletos de los elementos arquitectónicos de un sistema experto son representados mediante documentos XML. Por lo tanto, el proceso de inserción consiste en rellenar esos documentos con las características del dominio de aplicación del caso específico, obteniendo un aspecto tipo PRISMA. La **figura 151** presenta (un trozo de) el resultado del proceso de inserción de características, especificado en el LDA de PRISMA.

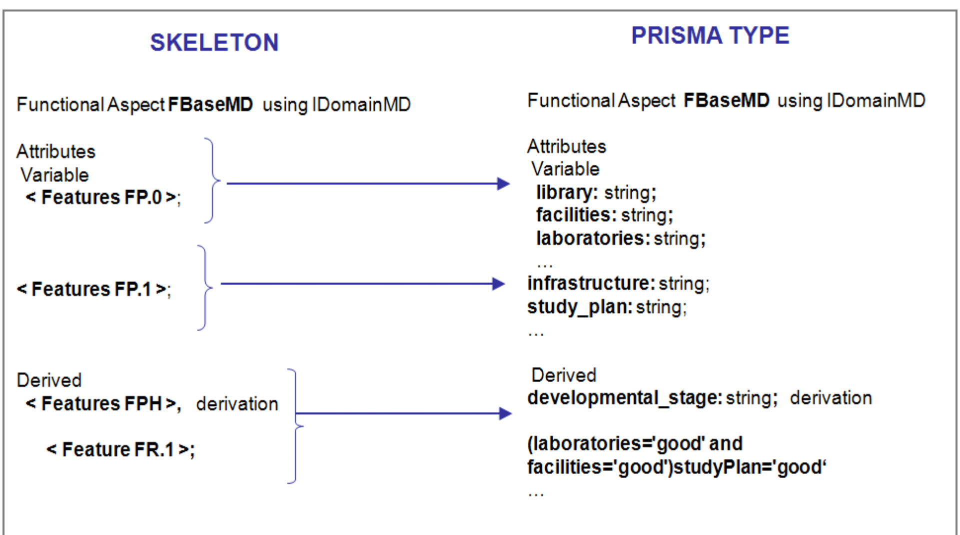


Figura 151 Metáfora visual del resultado del proceso de inserción de características

Adicionalmente, un ejemplo de una aspecto tipo PRISMA en el documento XML generado por ProtoBOM, se presenta a continuación:

```
*******************************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SKELETON>
  <ASPECT>
    <TYPE>Functional Aspect</TYPE>
    <NAME>FBaseDPE</NAME>
    <INTERFACE>using IDomainDPEDT</INTERFACE>
  </ASPECT>
```

```
  <ATTRIBUTES>
    <VARIABLES>
      <FP0>library: String</FP0>
      <FP0>computerEquipment:String</FP0>
      <FP0>laboratories: String</FP0>
      <FP0>facilities: String</FP0>
    </VARIABLES>
    <DERIVEDS>
      <FP1>infrastructure: String</FP1>
      <FP1>studyPlan:String</FP1>
      <FH>developmentalStage:String</FH>
    </DERIVEDS>
  </ATTRIBUTES>
  <DERIVATIONS>
    <FR1>(library='good'and
computerEquipment='good')infrastructure='good'</FR1>
    <FR1>(laboratories='good' and
facilities='good')studyPlan='good'</FR1>
    <FR1>(library='bad' and
computerEquipment='bad')infrastructure='bad'</FR1>
    <FR1>(laboratories='bad' and facilities='bad')studyPlan='bad'</FR1>
    <FR2>(infrastructure='good' and
studyPlan='good')developmentalStage='good'</FR2>
    <FR2>(infrastructure='bad' and
studyPlan='bad')developmentalStage='bad'</FR2>
  </DERIVATIONS>
  <SERVICES>
    <BEGIN>Begin()</BEGIN>
    <CLEAN>in cleanDB();</CLEAN>
     <VALUATIONCLEAN>Valuations</VALUATIONCLEAN>
     <INCLEAN>[in cleanDB ()]
       <NILFP0>library:=nil</NILFP0>
       <NILFP0>computerEquipment:=nil</NILFP0>
       <NILFP0>laboratories:=nil</NILFP0>
       <NILFP0>facilities:=nil</NILFP0>
       <NILFP1>infrastructure:=nil</NILFP1>
       <NILFP1>studyPlan:=nil</NILFP1>
       <NILFH>developmentalStage:=nil</NILFH>
       </INCLEAN>
```

```
    <INFERPROP1>in inferPropertiesN1 (input PROPERTIESN0:string, output
PROPERTIESN1: string)
    </INFERPROP1>
 <VALUATIONINFERPROP1>Valuations</VALUATIONINFERPROP1>
    <ININFERPROP1>[in inferPropertiesN1 ()]
       <ASIGNAFP0>library:=PROPERTY0_0</ASIGNAFP0>
       <ASIGNAFP0>computerEquipment:=PROPERTY0_1</ASIGNAFP0>
       <ASIGNAFP0>laboratories:=PROPERTY0_2</ASIGNAFP0>
       <ASIGNAFP0>facilities:=PROPERTY0_3</ASIGNAFP0>
       <ASIGNAFP1>PROPERTY1_0:=infrastructure</ASIGNAFP1>
       <ASIGNAFP1>PROPERTY1_1:=studyPlan</ASIGNAFP1>
  </ININFERPROP1>
    <INFERHYP>in inferHypotheses (input PROPERTIESN1: string, output
HYPOTHESES: string)</INFERHYP>
    <VALUATIONINFERHYP>Valuations</VALUATIONINFERHYP>
    <ININFERHYP>[in iferHypotheses () ]
<ASIGNAFP1>infrastructure:=PROPERTY1_0</ASIGNAFP1>
<ASIGNAFP1>studyPlan:=PROPERTY1_1</ASIGNAFP1>
<ASIGNAFH>HYPOTHESES:=developmentalStage</ASIGNAFH>
</ININFERHYP>
    <END>end;</END>
  </SERVICES>
  <PLAYEDROLES>Played_Roles
   <ROLE>KNOWLEDGE for IDomainEP-TV::=limpiarBD?()</ROLE>
<ROLE>inferProperties1?(PROPERTY0_0,PROPERTY0_1,PROPERTY0_2,PROPERTY0_3,
PROPERTY1_0,PROPERTY1_1)</ROLE>
<ROLE>inferProperties1!(PROPERTY0_0,PROPERTY0_1,PROPERTY0_2,PROPERTY0_3,
PROPERTY1_0,PROPERTY1_1)</ROLE>
<ROLE>inferHypotheses?(PROPERTY1_0,PROPERTY1_1HYPOTHESES)</ROLE>
<ROLE>inferHypotheses!(PROPERTY1_0,PROPERTY1_1HYPOTHESES)</ROLE>
</PLAYEDROLES>
  <PROTOCOLS>
    <FBASE>FASE::=begin():1-&gt;P0</FBASE>
    <P0>P0::=KNOWLEDGE_cleanDB?():1-&gt;P1</P0>
    <P1>PROPERTY0_3,PROPERTY1_0,PROPERTY1_1)-
&gt;KNOWLEDGE_inferProperties1!(PROPERTY0_3,PROPERTY1_0,PROPERTY1_1):1-
&gt;P2</P1>
    <P2>PROPERTY1_0,PROPERTY1_1HYPOTHESES)-
&gt;KNOWLEDGE_inferHypotheses!(PROPERTY1_0,PROPERTY1_1HYPOTHESES):1-
&gt;P3</P2>
```

```
    <P3>P3::=end():1;</P3>
  </PROTOCOLS>
  <NAMEEND>
    <ASPECTEND>
      <TYPEEND>End_Functional Aspect</TYPEEND>
      <NAME>FBaseDPE</NAME>
    </ASPECTEND>
  </NAMEEND>
</SKELETON>
************************************************************************
```

El proceso de inserción de caracterísiticas ha sido implementado específicamente para esta tesis. A continuación se presenta su código fuente:

```
************************************************************************
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Xml;
using System.Collections;
using System.IO;

namespace InsertionProcess
{
    public partial class Form1 : Form
    {

        private Specification specification;

        public Form1()
        {
            InitializeComponent();
            specification = new Specification();

//Carga la specificación
specification.loadSpecification("C:\\specif_skeleton.txt");
            rellenaFunctionalAspect(specification.getPathPrismas())
        }



//Método que rellena el SKELETON
        private void rellenaFunctionalAspect(string file)
        {
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.Load("C:\\Skeleton.xml");
```

```
            //Create an XML declaration.
            XmlDeclaration xmldecl=xmlDoc.CreateXmlDeclaration("1.0",
"UTF-8", null);
           //Add the new node (with the declaration) to the document.
            XmlElement root = xmlDoc.DocumentElement;
            xmlDoc.InsertBefore(xmldecl, root);


//ETIQUETA VARIABLES
            XmlNodeList xmlNodelistVar =
xmlDoc.GetElementsByTagName("VARIABLES");

            foreach (XmlElement xmlEl in xmlNodelistVar)
            {

//Ahora añado todas las propiedades de nivel 0
                        ArrayList featuresLevel0 =
this.specification.getPropertiesLevel(0);
                        for (int i = 0; i < featuresLevel0.Count; i++)
                        {
                            Propertie p = (Propertie)featuresLevel0[i];

                            XmlElement featuresFP0Element =
(XmlElement)xmlEl.AppendChild(xmlDoc.CreateElement("FP0"));
                            XmlText featuresFP0TextElement =
(XmlText)featuresFP0Element.AppendChild(xmlDoc.CreateTextNode("FP0"));
                            featuresFP0TextElement.Value = p.getName()
+ ":" + p.getType();
                        }
                    }



//Etiqueta DERIVEDS
            XmlNodeList xmlNodelistDer =
xmlDoc.GetElementsByTagName("DERIVEDS");

            foreach (XmlElement xmlEld in xmlNodelistDer)
            {

//Añado todas las propiedades de nivel 1
                        ArrayList featuresLevel1 =
this.specification.getPropertiesLevel(1);
                        for (int i = 0; i < featuresLevel1.Count; i++)
                        {
                            Propertie p = (Propertie)featuresLevel1[i];

                            XmlElement featuresFP1Element =
(XmlElement)xmlEld.AppendChild(xmlDoc.CreateElement("FP1"));
                            XmlText featuresFP1TextElement =
(XmlText)featuresFP1Element.AppendChild(xmlDoc.CreateTextNode("FP1"));
                            featuresFP1TextElement.Value = p.getName()
+ ":" + p.getType();
                        }

//Añado las hipótesis de nivel 1
                        ArrayList hipothesesLevel1 =
specification.getHypothesesLevel(1);
                        for (int i = 0; i < hipothesesLevel1.Count; i++)
```

```
                        {
                            Propertie p =
(Propertie)hipothesesLevel1[i];

                            XmlElement featuresFP1Element =
(XmlElement)xmlEld.AppendChild(xmlDoc.CreateElement("FH"));
                            XmlText featuresFP1TextElement =
(XmlText)featuresFP1Element.AppendChild(xmlDoc.CreateTextNode("FH"));
                            featuresFP1TextElement.Value = p.getName()
+ ":" + p.getType();
                        }
                    }




//ETIQUETA DERIVATIONS
            XmlNodeList xmlNodelistDerivations =
xmlDoc.GetElementsByTagName("DERIVATIONS");

            foreach (XmlElement xmlElDerRul in xmlNodelistDerivations)
            {

//Añado las Derivation rules de nivel 1
                ArrayList rulesL1 = this.specification.getRulesLevel(1);
                for (int i = 0; i < rulesL1.Count; i++)
                {
                    XmlElement rulesL1Element =
(XmlElement)xmlElDerRul.AppendChild(xmlDoc.CreateElement("FR1"));
                    XmlText rulesL1TextElement =
(XmlText)rulesL1Element.AppendChild(xmlDoc.CreateTextNode("FR1"));
                    rulesL1TextElement.Value = rulesL1[i].ToString();
                }

//Añado las Derivation rules de nivel 2
                ArrayList rulesL2 = this.specification.getRulesLevel(2);
                for (int i = 0; i < rulesL2.Count; i++)
                {
                    XmlElement rulesL2Element =
(XmlElement)xmlElDerRul.AppendChild(xmlDoc.CreateElement("FR2"));
                    XmlText rulesL2TextElement =
(XmlText)rulesL2Element.AppendChild(xmlDoc.CreateTextNode("FR2"));
                    rulesL2TextElement.Value = rulesL2[i].ToString();
                }
            }

// Relleno "in cleanDB()"
            XmlNodeList xmlNodelistInClean =
xmlDoc.GetElementsByTagName("INCLEAN");

            foreach (XmlElement xmlElInClean in xmlNodelistInClean)
            {

//Añado etiqueta NILFP0
//Obtengo propiedades de nivel 0
                ArrayList featuresL0 =
this.specification.getPropertiesLevel(0);

                for (int i = 0; i < featuresL0.Count; i++)
```

```
                {
                    Propertie p = (Propertie)featuresL0[i];

                    XmlElement nilFP0Element =
(XmlElement)xmlElInClean.AppendChild(xmlDoc.CreateElement("NILFP0"));
                    XmlText nilFP0TextElement =
(XmlText)nilFP0Element.AppendChild(xmlDoc.CreateTextNode("NILFP0"));
                    nilFP0TextElement.Value = p.getName() + ":=nil";
                }

//Añado etiqueta NILFP1
//Obtengo propiedades de nivel 1
ArrayList featuresL1 = this.specification.getPropertiesLevel(1);

                for (int i = 0; i < featuresL1.Count; i++)
                {
                    Propertie p = (Propertie)featuresL1[i];

                    XmlElement nilFP1Element =
(XmlElement)xmlElInClean.AppendChild(xmlDoc.CreateElement("NILFP1"));
                    XmlText nilFP1TextElement =
(XmlText)nilFP1Element.AppendChild(xmlDoc.CreateTextNode("NILFP1"));
                    nilFP1TextElement.Value = p.getName() + ":=nil";
                }


//Añado etiqueta NILFH
//Obtengo las hipotesis
ArrayList hypotheses = this.specification.getHypothesesLevel(1);

for (int i = 0; i < hypotheses.Count; i++)
                {
                    Propertie p = (Propertie)hypotheses[i];
                    XmlElement nilFHElement =
(XmlElement)xmlElInClean.AppendChild(xmlDoc.CreateElement("NILFH"));
                    XmlText nilFHTextElement =
(XmlText)nilFHElement.AppendChild(xmlDoc.CreateTextNode("NILFH"));
                    nilFHTextElement.Value = p.getName() + ":=nil";
                }
            }


// Relleno "in inferProperties1()"
            XmlNodeList xmlNodelistInInfer =
xmlDoc.GetElementsByTagName("ININFERPROP1");
            foreach (XmlElement xmlElInInfer in xmlNodelistInInfer)
            {

//Añado etiqueta ASIGNAFP0
//Obtengo propiedades de nivel 0
ArrayList featuresL0 = this.specification.getPropertiesLevel(0);

                for (int i = 0; i < featuresL0.Count; i++)
                {
                    Propertie p = (Propertie)featuresL0[i];

                    XmlElement nilFP0Element =
(XmlElement)xmlElInInfer.AppendChild(xmlDoc.CreateElement("ASIGNAFP0"));
```

```
                    XmlText nilFP0TextElement =
(XmlText)nilFP0Element.AppendChild(xmlDoc.CreateTextNode("ASIGNAFP0"));
                    nilFP0TextElement.Value = p.getName() +
":=PROPERTY0_" + i.ToString();//!!sumar1
                }

//Añado etiqueta ASIGNAFP1
//Obtengo propiedades de nivel 1
ArrayList featuresL1 = this.specification.getPropertiesLevel(1);

                for (int i = 0; i < featuresL1.Count; i++)
                {
                    Propertie p = (Propertie)featuresL1[i];

                    XmlElement nilFP1Element =
(XmlElement)xmlElInInfer.AppendChild(xmlDoc.CreateElement("ASIGNAFP1"));
                    XmlText nilFP1TextElement =
(XmlText)nilFP1Element.AppendChild(xmlDoc.CreateTextNode("ASIGNAFP1"));
                    nilFP1TextElement.Value = "PROPERTY1_" +
i.ToString() + ":=" + p.getName();
                }
            }

// Relleno "in inferHypotheses()"
            XmlNodeList xmlNodelistInHyp =
xmlDoc.GetElementsByTagName("ININFERHYP");
            foreach (XmlElement xmlElInHyp in xmlNodelistInHyp)
            {

//Obtengo propiedades de nivel 1
ArrayList featuresL1 = this.specification.getPropertiesLevel(1);

                for (int i = 0; i < featuresL1.Count; i++)
                {
                    Propertie p = (Propertie)featuresL1[i];

                    XmlElement nilFP1Element =
(XmlElement)xmlElInHyp.AppendChild(xmlDoc.CreateElement("ASIGNAFP1"));
                    XmlText nilFP1TextElement =
(XmlText)nilFP1Element.AppendChild(xmlDoc.CreateTextNode("ASIGNAFP1"));
                    nilFP1TextElement.Value = p.getName() + ":=" +
"PROPERTY1_" + i.ToString();
                }

//Añado etiqueta NILFH
//Obtengo las hipotesis
ArrayList hypotheses = this.specification.getHypothesesLevel(1);

                for (int i = 0; i < hypotheses.Count; i++)
                {
                    Propertie p = (Propertie)hypotheses[i];
                    XmlElement nilFHElement =
(XmlElement)xmlElInHyp.AppendChild(xmlDoc.CreateElement("ASIGNAFH"));
                    XmlText nilFHTextElement =
(XmlText)nilFHElement.AppendChild(xmlDoc.CreateTextNode("ASIGNAFH"));
                    nilFHTextElement.Value = "HYPOTHESES:=" +
p.getName();
                }
            }
```

```
// Relleno ROLES
            XmlNodeList xmlNodelistPR =
xmlDoc.GetElementsByTagName("PLAYEDROLES");
            foreach (XmlElement xmlElPR in xmlNodelistPR)
            {



// ROLE inferProperties1?
                XmlElement role1Element =
(XmlElement)xmlElPR.AppendChild(xmlDoc.CreateElement("ROLE"));
                XmlText role1TextElement =
(XmlText)role1Element.AppendChild(xmlDoc.CreateTextNode("ROLE"));
                role1TextElement.Value = "inferProperties1?(";

// ROLE inferProperties1!
                XmlElement role2Element =
(XmlElement)xmlElPR.AppendChild(xmlDoc.CreateElement("ROLE"));
                XmlText role2TextElement =
(XmlText)role2Element.AppendChild(xmlDoc.CreateTextNode("ROLE"));
                role2TextElement.Value = "inferProperties1!(";

// role inferHypotheses?
                XmlElement role3Element =
(XmlElement)xmlElPR.AppendChild(xmlDoc.CreateElement("ROLE"));
                XmlText role3TextElement =
(XmlText)role3Element.AppendChild(xmlDoc.CreateTextNode("ROLE"));
                role3TextElement.Value = "inferHypotheses?(";

// role inferHypotheses!
                XmlElement role4Element =
(XmlElement)xmlElPR.AppendChild(xmlDoc.CreateElement("ROLE"));
                XmlText role4TextElement =
(XmlText)role4Element.AppendChild(xmlDoc.CreateTextNode("ROLE"));
                role4TextElement.Value = "inferHypotheses!(";


//Obtengo propiedades de nivel 0
ArrayList featuresL0 = this.specification.getPropertiesLevel(0);
                for (int i = 0; i < featuresL0.Count; i++)
                {
                    role1TextElement.Value += "PROPERTY0_" +
i.ToString() + ",";
                    role2TextElement.Value += "PROPERTY0_" +
i.ToString() + ",";
                }

//Obtengo propiedades de nivel 1
ArrayList featuresL1 = this.specification.getPropertiesLevel(1);
                int j;
                for (j = 0; j < featuresL1.Count - 1; j++)
                {
                    role1TextElement.Value += "PROPERTY1_" +
j.ToString() + ",";
                    role2TextElement.Value += "PROPERTY1_" +
j.ToString() + ",";
                    role3TextElement.Value += "PROPERTY1_" +
j.ToString() + ",";
```

```
                    role4TextElement.Value += "PROPERTY1_" +
j.ToString() + ",";
                }
                role1TextElement.Value += "PROPERTY1_" + j.ToString() +
")";
                role2TextElement.Value += "PROPERTY1_" + j.ToString() +
")";
                role3TextElement.Value += "PROPERTY1_" + j.ToString() +
"HYPOTHESES)";
                role4TextElement.Value += "PROPERTY1_" + j.ToString() +
"HYPOTHESES)";

            }



// PROTOCOLOS
            string protocolo1 = "";
            string protocolo2 = "";

            XmlNodeList xmlNodelistPro =
xmlDoc.GetElementsByTagName("PROTOCOLS");
            foreach (XmlElement xmlElPR in xmlNodelistPro)
            {

/****** P0  ********/
                XmlElement P0Element =
(XmlElement)xmlElPR.AppendChild(xmlDoc.CreateElement("P0"));
                XmlText P0TextElement =
(XmlText)P0Element.AppendChild(xmlDoc.CreateTextNode("P0"));
                P0TextElement.Value = "P0::=KNOWLEDGE_cleanDB?():1->P1";

/***** P1 ********/
                XmlElement P1Element =
(XmlElement)xmlElPR.AppendChild(xmlDoc.CreateElement("P1"));
                XmlText P1TextElement =
(XmlText)P1Element.AppendChild(xmlDoc.CreateTextNode("P1"));
                P1TextElement.Value =
"P1::=KNOWLEDGE_inferProperties1?(";

/***** P2 ********/
                XmlElement P2Element =
(XmlElement)xmlElPR.AppendChild(xmlDoc.CreateElement("P2"));
                XmlText P2TextElement =
(XmlText)P2Element.AppendChild(xmlDoc.CreateTextNode("P2"));
                P2TextElement.Value = "KNOWLEDGE_inferProperties1!(";

/***** P3  *******/
                XmlElement P3Element =
(XmlElement)xmlElPR.AppendChild(xmlDoc.CreateElement("P3"));
                XmlText P3TextElement =
(XmlText)P3Element.AppendChild(xmlDoc.CreateTextNode("P3"));
                P3TextElement.Value = "P3::=end():1;";


//Obtengo propiedades de nivel 0
ArrayList featuresL0 = this.specification.getPropertiesLevel(0);
                for (int i = 0; i < featuresL0.Count; i++)
                {
```

```
                    protocolo1 = "PROPERTY0_" + i.ToString() + ",";
                }

//Obtengo propiedades de nivel 1
ArrayList featuresL1 = this.specification.getPropertiesLevel(1);
                int j;
                for (j = 0; j < featuresL1.Count - 1; j++)
                {
                  protocolo1 += "PROPERTY1_" + j.ToString() + ",";
                  protocolo2 += "PROPERTY1_" + j.ToString() + ",";

                }
                 protocolo1 += "PROPERTY1_" + j.ToString() + ")";
                 protocolo2 += "PROPERTY1_" + j.ToString() +
"HYPOTHESES)";


                P1TextElement.Value = protocolo1 + "-
>KNOWLEDGE_inferProperties1!(" + protocolo1 + ":1->P2";
                P2TextElement.Value = protocolo2 + "-
>KNOWLEDGE_inferHypotheses!(" + protocolo2 + ":1->P3";
            }


            xmlDoc.Save(file+"\\FBaseDPE PrysmaType.xml");


//Borro archivos auxiliares
            deleteAuxFiles();
        }

//Método para borrar los archivos auxiliares, que se crean para ejecutar
este InsertProcess
        private void deleteAuxFiles()
        {
            File.Delete("C:\\Skeleton.xml");
            File.Delete("C:\\specif_skeleton.txt");
        }



//Barra de progreso
        private void progress()
        {
            pBar1.Visible = true; // Display the ProgressBar control.
            pBar1.Minimum = 1; // Set Minimum to 1 to represent the
first file being copied.
            pBar1.Maximum = 2000; // Set Maximum to the total number of
files to copy.
            pBar1.Value = 1; // Set the initial value of the
ProgressBar.
            pBar1.Step = 1; // Set the Step property to a value of 1 to
represent each file being copied.

// Loop
            for (int x = 1; x <= 2000; x++)
                pBar1.PerformStep(); // Perform the increment on the
ProgressBar.
```

```
        }

        private void Form1_Shown(object sender, EventArgs e)
        {
            progress();
            System.Windows.Forms.MessageBox.Show(this, "The prisma types
have been created succesfully.", "Information", MessageBoxButtons.OK,
MessageBoxIcon.Information);
            this.Close();
        }



    }
}

************************************************************************
```

APÉNDICE F

IMPLEMENTACIÓN EN QVT-RELATIONS

En este apéndice se presenta lo más destacado de la implementación de la transformación del modelo modular a los modelos de componentes-conectores de los SE, utilizando las QVT-Relations

Definición de las QVT-Relations

El código[8] correspondiente para la transformación del modelo modular a los modelos de componentes-conectores, se muestra a continuación:

```
***************************************************************
transformation modules2components(mdomain : mview,  dcmdomain: dcm,
ccdomain : ccview) {

       key ccview::Component{name};
       key ccview::Connector{name};
       key ccview::Port{name,powner};
       key ccview::Role{name,cowner};
       key ccview::Service{name,sowner};
       key ccview::PeerToPeer{name,service};


       top relation ModulesModel2ComponentsModel {

              checkonly domain mdomain modulesModel :
mview::ModulesModel {
              };

              checkonly domain dcmdomain varModel :
dcm::DomainConceptualModel {
```

[8] Realizado por Abel Gómez LLana

```
		};

		enforce domain ccdomain componentsModel : ccview::CCModel
{
			name = modulesModel.name
		};

		where {
	UseCase2Connector(modulesModel,varModel,componentsModel);
		}
	}

	relation UseCase2Connector {

		checkonly domain mdomain modulesModel :
mview::ModulesModel {
		};

		checkonly domain dcmdomain varModel :
dcm::DomainConceptualModel {
			useCases = useCase : dcm::UseCase {}
		};

		enforce domain ccdomain componentsModel : ccview::CCModel
{
			tcomponents = connector : ccview::Connector {
				name = useCase.name + ' Connector'
			}
		};

		where {
			Module2Component(modulesModel, varModel,
componentsModel, connector, useCase);
		}
	}

	relation Module2Component {

		checkonly domain mdomain modulesModel :
mview::ModulesModel {
			tmodules = module : mview::Module{}
		};

		checkonly domain dcmdomain varModel :
dcm::DomainConceptualModel {
			actors = actor : dcm::Actor {
				uses = useCaseActor : dcm::UseCase {}
			}
		};

		enforce domain ccdomain componentsModel : ccview::CCModel
{
			tcomponents = component : ccview::Component {
				name = getComponentName(varModel, actor,
module)
			}
		};
```

```
		enforce domain ccdomain connector : ccview::Connector {
		};

		checkonly domain dcmdomain useCase : dcm::UseCase {
		};


		where {
			if module.name = 'User Interface' then
				Module2RolePort(module, useCaseActor,
connector, component)
			else
				Module2RolePort(module, useCase, connector,
component)
			endif;

			Function2Relation(module, componentsModel,
connector, component);
		}
	}

	relation Module2RolePort {

		checkonly domain mdomain module : mview::Module {
		};

		checkonly domain dcmdomain useCase : dcm::UseCase {
		};

		enforce domain ccdomain connector : ccview::Connector {
			crole = role : ccview::Role {
				name = module.name + ' Role'
			}
		};

		enforce domain ccdomain component : ccview::Component {
			port = port : ccview::Port {
				name = useCase.name + ' Port'
			}
		};
		where {
			ConnectRoleAndPort(role, port);
			Function2Service(module, port, component);
		}
	}

	relation ConnectRoleAndPort {

		checkonly domain ccdomain role : ccview::Role {
		};

		enforce domain ccdomain port : ccview::Port {
			prole = role
		};
		when {
			role.rport.oclIsUndefined() and
port.prole.oclIsUndefined();
		}
	}
```

```
        relation Function2Service {

                checkonly domain mdomain module : mview::Module {
                        function = function : mview::Function {}
                };

                checkonly domain ccdomain port : ccview::Port {
                };

                enforce domain ccdomain component: ccview::Component {
                        cservice = service : ccview::Service {
                                name = function.name,
                                type = function.type,
                                port = port
                        }
                };

        }

        relation Function2Relation {

                checkonly domain mdomain module : mview::Module {
                        function = function : mview::Function {}
                };

                enforce domain ccdomain componentsModel : ccview::CCModel
{
                        relations = relat : ccview::PeerToPeer {
                                name = module.name + ' Attachment',
                                type = 'attachment',
                                service = function.name,
                                constraints = function.type,
                                component = component,
                                connector = connector
                        }
                };

                checkonly domain ccdomain connector : ccview::Connector {
                };

                checkonly domain ccdomain component: ccview::Component {
                };

        }

        query getComponentName(varModel : dcm::DomainConceptualModel,
actor : dcm::Actor, module : mview::Module) : String {
                if module.name = 'User Interface' then
                        module.name + ' - ' + actor.name
                else
                        if module.name = 'Inference Motor' or module.name =
'Knowledge Base' then
                                if varModel.reasoning.type =
dcm::ReasoningType::deductive then
                                        'Deductive ' + module.name
                                else
                                        if varModel.reasoning.type =
dcm::ReasoningType::inductive then
```

```
                                        'Inductive ' + module.name
                                else
                                        'Differential ' + module.name
                                endif
                        endif
                else
                        module.name
                endif
        endif
    }

}
*************************************************************************
```