BOM–Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques

Abel Gómez
Dep. of Information Systems and Computation
Universidad Politécnica de Valencia
Valencia, Spain
agomez@dsic.upv.es

Mª Eugenia Cabello
Faculty of Telematics
Universidad de Colima
Colima, México
ecabello@ucol.mx

Isidro Ramos
Dep. of Information Systems and Computation
Universidad Politécnica de Valencia
Valencia, Spain
iramos@dsic.upv.es

***Abstract*—This paper presents Baseline Oriented Modeling–Lazy (BOM–Lazy): an approach to develop applications in a domain, Expert Systems, by means of Software Product Lines and model transformations techniques. A domain analysis has been done on the variability of Expert Systems that perform diagnostic tasks in order to determine the general and individual features, (i.e. common and variants features) of these systems. The variability of our Software Product Line is managed by means of models and model transformations; and the production plan is automatically generated and driven by the variability model and the core assets (which take part in the reference architecture) of the domain, in order to produce the base architecture of the Software Product Line.**

***Keywords*-Software Product Lines; Variability Management; Model Transformations; Software Architectures; Diagnostic Expert Systems**

I. Introduction

Expert Systems (ES) are a family of software products that are gaining great importance in recent years [12]. They are applications that can be used to simulate the behaviour of experts when they solve a problem in a specific domain. Such systems try to capture the knowledge and decisions of experts, improving the speed and the quality of the answers that are obtained, and increasing the productivity of the domain experts. ES have been widely used in several fields of our lives: medicine, education, military intelligence, aeronautics, archeology, agriculture, law, geology, electronics, computer science, telecommunications, etc. That is why they are becoming an important reference point in decision-making processes, especially in systems in charge of diagnostic tasks. However, the development of this kind of systems is complex because the basic elements that conform their architecture vary not only in their behaviour, but also in their architecture. That is why there is an increasing need in supporting them properly.

The changing nature of technology leads us to need multiple versions of the same or similar application in short time periods. Because of that, Software Engineering must provide the tools and methods which allow us to develop a family of products with different capabilities and adaptable to changeable situations, in place of developing only a single product. Under these circumstances, the Software Product Line (SPL) [8] concept arises with the aim of controlling and minimizing the high costs of the software development process. This approach is based on the creation of a design that can be shared among all the members of a family of programs within an application domain. This way, a design that has been done explicitly for a product can get benefit from the common assets (models, components, code…) that can be reused in different products, reducing costs and development time.

In order to develop an ES, the SPL approach is more adequate than the traditional one: this family of systems involve a wide range of application domains, with different features, that vary from a case study to another. Moreover, SPL make the development of software products for several target platforms and technologies easier.

This work integrates different technical spaces [15]: Expert Systems (ES), Software Product Lines (SPL) and the Model–Driven Architecture (MDA) [16], as the technical framework. MDA promotes to separate the description and the functionality of a system (Platform Independent Model, PIM) from its implementation (Platform Specific Model, PSM). MDA suggests to define and use models at different abstraction levels as the main assets of the software development process. These models can be manipulated and refined (by means of model transformations) in successive steps to obtain the final implementation of the system. The standard that the OMG proposes to define these model transformations is the Query/View/Transformation (QVT) [17] standard. This way, in our approach we use the tools that are supported on the MDA framework to represent and manage the variability of the SPL by means of domain models and their transformation to architectural models. In the end, this infrastructure allows us to produce a SPL of ES at design level using the architectural specification language PRISMA [5] as the target domain. The PRISMA framework [20] automatically compiles the architectural configurations and executes them by using a middleware built on top of the .NET platform.

This document is structured as follows: section II de-

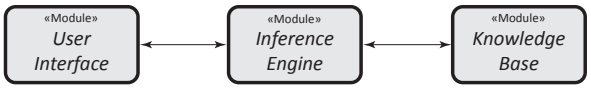


Figure 1. Reference Architecture of Expert Systems

scribes the variability found in the Expert Systems technical space; section III explains how this variability is managed in our approach. Section IV describes how our ideas have been captured in a prototype. Finally, section V describes some related works and section VI presents our conclusions.

II. Variability of the Diagnostic Expert Systems Domain

A subset of the Expert Systems domain has been chosen to describe our approach: the ES that are used in diagnostic tasks, the so-called Diagnostic Expert Systems (DES). The diagnosis of an entity lies on the evaluation of its state by interpreting its properties. A field study about diagnostic systems has been done [7] to identify the key variability points. This study allows us to know the DES behaviour and structure in several specific domains. The remainder of this paper will refer to two paradigmatic cases: systems for medical diagnosis and systems for educational diagnosis.

In the medical diagnosis example, the entity to be diagnosed is the patient and the result of the process is the disease he/she suffers. First, a clinical diagnosis is performed, which must be validated then by a laboratory-based diagnosis. In the end, both diagnosis are merged in a final diagnosis where the previous ones are taken into account. Thus, we can identify three basic functionalities: *get laboratory diagnosis*, *get clinical diagnosis*, and *get diagnosis*. The first one is used by the *laboratory assistant* and the last ones are used by the *doctor*. In this case, the properties of the entities considered in the process vary during the whole process, which implies the existence of several hypotheses that must be evaluated to determine the valid one using differential reasoning.

In the educations reasoning example the entity to be diagnosed is a post-graduate educational program where several quality criteria are evaluated, and the result of the diagnosis is the advance of the given program. The properties of the entities remain the same throughout the diagnostic process, therefore only one hypothesis is created applying deductive reasoning. In this case the DES only performs one task: *get program advance*, which is invoked by the user of the tool.

A. Diagnostic Expert Systems Reference Architecture

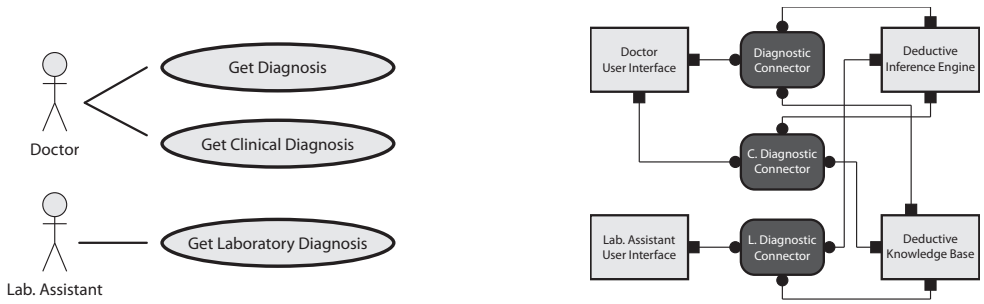
In SPL, there are parts that are shared among all the products, but some other parts vary from one product to another. The common parts are represented by the *reference architecture*, which captures the shared functionality. The variable part shows additional features that are specific for some products, and such parts are represented by the *base architecture*. The *reference architecture* of DES is expressed in our approach by a modular model made up of three basic modules (see Fig. 1):

- *Inference Engine* module. This module contains the inference process that solves a problem in a specific domain.
- *Knowledge Base* module. The *Knowledge Base* module contains the knowledge about the domain
- *User Interface* module. This module allows the communication between the user(s) and the system.

The reference architecture is used as the shared structure of an application that is member of the product line, but, there also exist additional features that are particular to a specific application. This implies the creation of a specific base architecture when a product of the SPL is obtained from the reference architecture. However, the base architecture that is generated from the reference architecture is not unique, because systems vary not only in their structure but also in their behaviour as explained in the following section.

B. Diagnostic Expert Systems Structural Variability

To illustrate how the architectural elements of a DES vary in their structure, we have modeled the functional requirements that the final product must satisfy using UML Case Diagrams. These diagrams show the different functionalities that the user expects from the system and how the system



(a) (b)

Figure 2. Medical diagnosis use case diagram (a) and its corresponding base architecture (b)

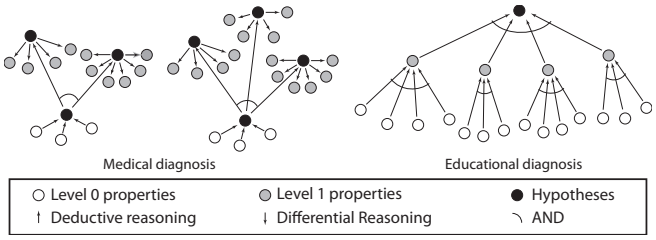


Figure 3. Graph describing the inference processes for medical diagnosis and educational diagnosis

interacts with its environment. In particular, the structure of the architectural elements vary according to the number of use cases, the number of actors and the number of use cases that are accessed by each actor.

Fig. 2 shows the use case diagram for the medical diagnosis domain together with its corresponding base architecture. As Fig. 2a shows, the ES for medical diagnosis of our case study has two actors: *doctor* and *lab. assistant*. The first one uses the system to get clinical diagnosis and the final diagnosis; and the second one uses the system to get the laboratory diagnosis. These use cases affect the final base architecture of the system. Fig. 2b shows, for example, that a user interface module is used for each one of the actors shown in the diagram.

C. Diagnostic Expert Systems Behavioral Variability

Behavioral variability of the architectural elements of an ES depends on the type of diagnosis, and therefore the reasoning to be used. As presented in the previous section, the inference process to apply is defined according to the reasoning (it can be deductive or differential). Moreover, we say that the inference process is *static* if there is only one hypothesis to evaluate and the entities involved keep the same properties throughout the whole diagnostic process. However, if the properties of the entities change during the process and there is more than one hypothesis, we say that it is a *dynamic* process. This way, medical diagnosis is a dynamic process that requires differential reasoning (Fig. 3-left); but educational diagnosis is an static process which requires to apply deductive reasoning (Fig. 3-right) [6]. Thus, the base architecture for ES in the medical diagnosis domain will have a *Differential Inference Engine* component. The behaviour of this component will differ from the behaviour of the inference engine of the ES in the educative diagnosis domain, which will have a *Deductive Inference Engine* component. Fig. 3 represents the medical and educative diagnosis processes as inference graphs.

III. Variability Management

Variability management is the key point to develop our family of DES. Variability among the products of a SPL can be expressed in terms of features [14]. In BOM–Lazy (Baseline Oriented Modeling–Lazy) [7] the observed features in the diagnostic processes and the user requirements are considered as the first group of variability points. Additionally, a second group of variability points arise: the variability that emerges from the application domain's properties. Thus, variability is managed in two stages. The first stage manages variability by using a feature model, which is used to obtain a specific base architecture. The second stage manages the variability of the properties and decorates this specific base architecture with the application domain features, conforming the final product.

A. BOM–Lazy Production Plan overview

The production plan of our SPL taking the BOM–Lazy approach is shown in Fig. 4 by using SPEM notation [18]. The production plan starts (task 1) when the features of the first variability (the domain variability) are obtained from the application engineer expressed as instances of the Domain Variability Model (DVM). Next (task 2), a skeleton architecture (the base architecture) is built by executing a QVT–Relations transformation using the DVM instances and

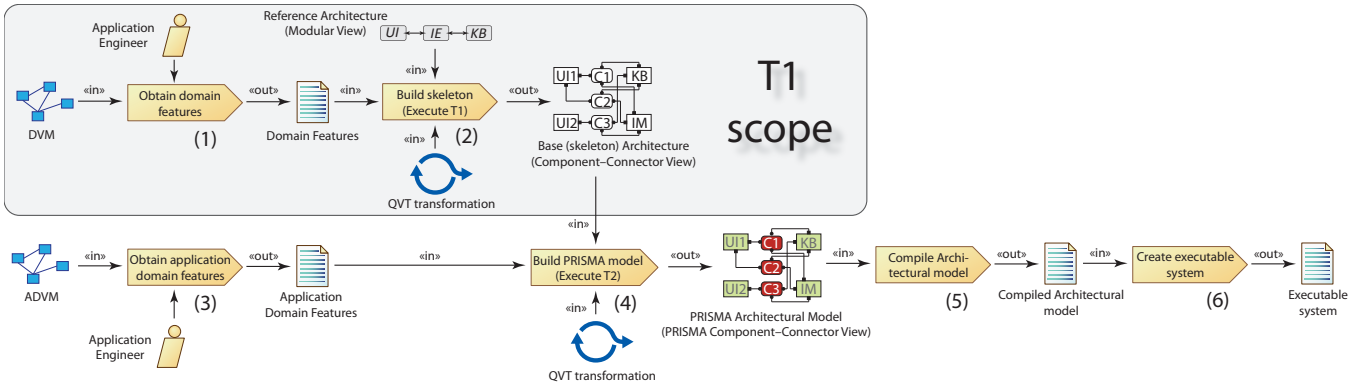


Figure 4. BOM–Lazy production plan

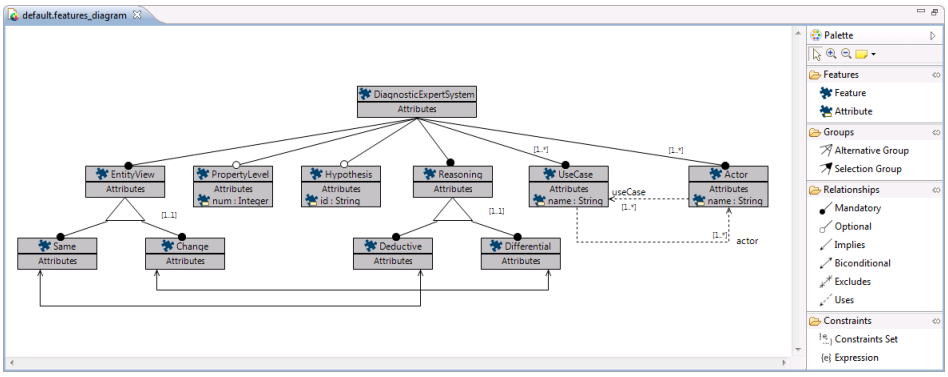


Figure 5. Feature model of the SPL (Domain Variability Model)

the reference architecture model. This first transformation (called *T1*) creates a base architecture that corresponds to the specific product of our SPL family. Additionally, the Application Domain Variability Model (ADVM) is used to define the features of the application domain (task 3). The instances of the ADVM are used to build the final model (task 4) by executing the QVT–Relations transformation (the second transformation, *T2*). This way, the base architectural model is automatically transformed to a PRISMA architectural model [5]. Finally, the PRISMA architectural model is compiled (task 5) by using the PRISMA-MODEL-COMPILER [20]. This tool automatically generates final product of our SPL: a fully functional C# .NET program that can be directly executed by using the PRISMA-Middleware.

The remainder of this paper will focus only on the first variability (highlighted in Fig. 4 as *T1 scope*), which is related with the diagnostic process and the user requirements, as they define the behaviour and the structure of the ES as described in section II. The management of the second variability will remain unexplained in this paper as it is out of its scope and it is dealt with similar process.

B. Domain Variability Management

The *Feature Model*, our *Domain Variability Model (DVM)*, represents the features of the first variability of the DES (our SPL). The selected variant is represented as an instance of the Domain Conceptual Model (DCM). The DCM is an intermediate model used to capture the selected features of the DVM. These features are used to configure the base architecture. Fig. 5 shows and screenshot of our feature model editor. In the image the feature model of the SPL for DES is described. Our prototype uses a variant of the Czarnecki-Eisenecker notation [9]. This notation for feature modeling allows feature cloning and typed feature attributes. Feature attributes can be used to identify cloned features.

IV. BOM IN PRACTICE

This section describes how to deal with the variability in the development of DES by using SPL and MDA techniques. In BOM–Lazy the transformation in charge of building the base architectures is called “T1”. This transformation is driven by the first variability that was described in section III. The transformation generates a specific base architecture (a component–connector architectural model) from a modular model (the reference architecture, which describes the common parts of the family of DES) and a domain variability model which captures the variability of the specific conceptual domain (DVM) as shown in the highlighted part of Fig. 4.

Fig. 6 depicts a simplified version of the modular view metamodel, which is used to describe the reference architectures. It shows that a model has a set of modules (which can also have different functions) that are linked to other modules by several kinds of relationships (only the Use relationship is represented for simplicity purposes).

To define a specific variant of the SPL for DES, the reference architecture must be configured with the particular features of the conceptual domain. This is done by means of the feature model (DVM) presented in section III, which is finally represented as a UML class diagram (the DCM). This representation allows us to easily define model instances in nowadays modeling tools. The translation of the Feature Model to a class diagram is automatically performed by our prototype and is described in detail in [13]. Fig. 7 shows what this model looks like by using the class diagram representation. Thus, an instance of the DVM is a set of

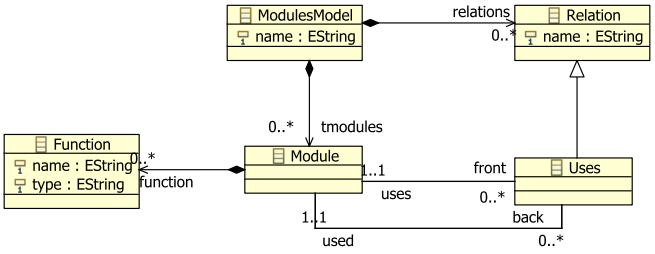


Figure 6. Simplified metamodel of the modular view.

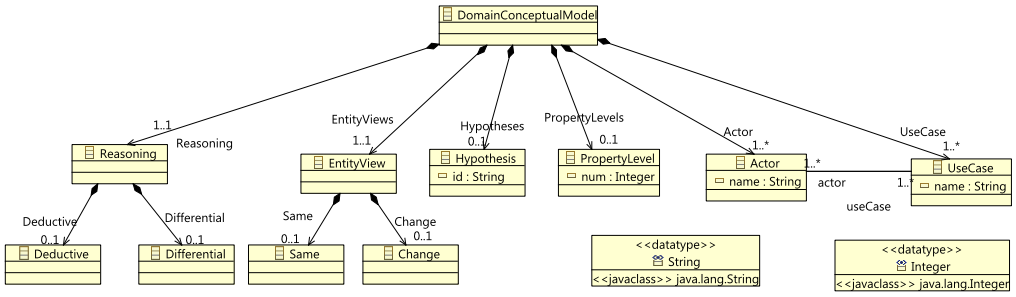


Figure 7. Domain Variability Model expressed as a class diagram (Domain Conceptual Model)

objects that are instances of such model.

Finally, Fig. 8 shows the simplified metamodel to build architectural models. An architectural model has a set of components and a set of connectors. Components provide services through a set a ports. Connectors are in charge of linking the different ports of the components by means of their roles.

A. Transformation patterns

In order to define the relationships among the different models and metamodels we have used the Query/View/-Transformation language (QVT) [17] as proposed by the OMG in their MDA proposal. The transformation establishes the correspondences among the elements of the source and the target models. Table I describes in a simplified way the relationships that have been identified, together with the elements that each one involves.

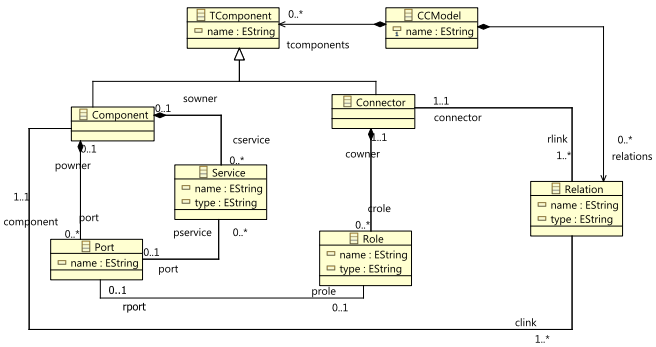


Figure 8. Simplified version of the Component–Connector metamodel

To define the correspondences among the different domain we have taken into account good practices for software development [19] in order to increase the quality of the design of the generated software architecture. Some of this practices and design decisions that the rules describe are the following:

- ***ModulesModel2ComponentsModel***. *A design must define a hierarchical organization that controls software components in a smart way*. In order to satisfy this criterion, this rule transforms the root element of the modular metamodel (ModulesModel), which contains the rest of the elements of the metamodel, to the root element of the component-connector metamodel (CCModel). The rule assigns the name of the source element to the target element.
- ***UseCase2Connector***. *A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and subfunctions.* Use cases constitute a partition of the system based on functionality. Thus, to achieve this requirement the rule transforms each use case to a connector in charge of coordinating the different components of the use case. This way, there is a one-to-one relationship between use cases and connectors.
  Fig. 9 shows the `UseCase2Connector` rule using the graphical notation of the QVT language. This rule creates, for each one of the use cases of the source variability model one connector of the component–connector model. The name of such connector stands for the concatenation of the use case name and the suffix "Connector". The modular domain (`mdomain`) and

Table I
TRANSFORMATION RULES AND INVOLVED ELEMENTS

	Involved elements		
Relation name	**Variability Model**	**Modular Metamodel**	**Component-Connector Metamodel**
ModulesModel2ComponentsModel	-	ModulesModel	CCModel
UseCase2Connector	UseCase	-	Connector
Module2Component	Actor, UseCase, EntityViews / Reasoning	Module	Component
Module2RolePort	UseCase	Module	Role, Port
Function2Service	-	Function	Service
Function2Relation	-	Function	Relation

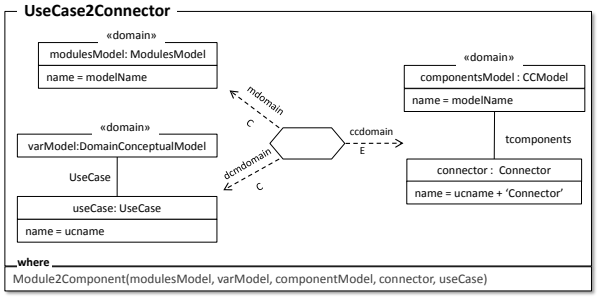


Figure 9. Graphical representation of the UseCase2Connector QVT relation

the variability model domain (`dcmdomain`) are defined as `checkonly` (as the "`C`" attached to the arrow of the domain indicates). This keyword is used to assure that at least an object validating the pattern exists. If no object validating the pattern exists the rule will not be applied (as the transformation is executed in the direction of `ccdomain`). In turn, the component–connector domain (`ccdomain`) is defined as `enforced` (as the "`E`" next to the arrow describes). The `enforce` modifier states that the pattern must be always validated. If there are not objects that make possible to validate the pattern, the rule must create them until the application of the rule is possible. Finally, the `where` clause states that the `Module2Component` relation must be considered as the post-condition of the current rule (`UseCase2Connector`). Thus, this rule should be properly applied after the `UseCase2Connector` relation is checked.

- ***Module2Component***. The reference architecture of ES is usually conformed by three modules. This rule transforms the *Knowledge Base* and *Inference Engine* modules to the *Knowledge Base* and *Inference Engine* components respectively. It also establishes their type (deductive or differential) according to the *Reasoning* and *EntityViews* elements of the variability model. Any other additional module of the reference architecture leads to a new component that represents it using the same name in the final base architecture.
  With the aim of satisfying the following design requirement: *A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment*, the *User Interface* module is transformed into as many components as actors appear in the domain variability model. This way, a relationship one-to-one is defined between the actors of the source model and the user interfaces in the target model.
- ***Module2RolePort***. The criterion *Uniqueness of the knowledge base* indicates that knowledge must be unique for the entire system. In order to meet this requirement, there is only one *Knowledge Base* component in our base architecture. Given that for each use case exists a different view of the *Knowledge Base*, we must merge each one of these views. This is represented in the *Knowledge Base* component by adding a port that allows the communication of such views with the rest of the components. To allow this communication it is also necessary to add the corresponding roles to the adequate connectors. Therefore a one-to-one relationship is defined between use cases and the *Knowledge Base* ports, which also implies a one-to-one relationship between use cases and the roles of the different connectors. This rule also creates the corresponding ports in the *Inference Engine* and *User Interfaces* components, and their corresponding roles in the involved connectors.
- ***Function2Service***. This rule creates a new service on the corresponding component of the Component–Connector model. This service is generated from a given function of a module of the modular model. The new service is created with the same name and type of the source function.
- ***Function2Relation***. This rule states that a function of a module of the source model (ModuleModel) will generate a *Relation* element in the target model (CCModel). This *Relation* will link a connector with its corresponding component in the Component–Connector model.

B. Transformation execution

Our prototype is built on top of the Eclipse platform, and makes use of the *Eclipse Modeling Framework* [11] and the *Graphical Modeling Framework* [10] tools to represent the different models and to build the different editors. Furthermore, it integrates a QVT transformations engine to easily execute the model transformations.

Different graphical editors have been developed to visually represent the models that are involved in the *T1* transformation. Fig. 10 shows a tree editor with the DCM instance of the medical diagnosis example. Fig. 11 shows the second software artifact that the T1 transformation takes: the modular view model (the reference architecture). Figure shows an screenshot of the *Modular View editor* of the BOM–Lazy prototype. Finally, Fig. 12 shows an screen-

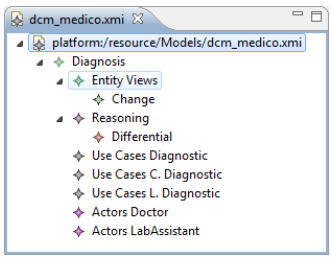


Figure 10. Domain Conceptual Model instance

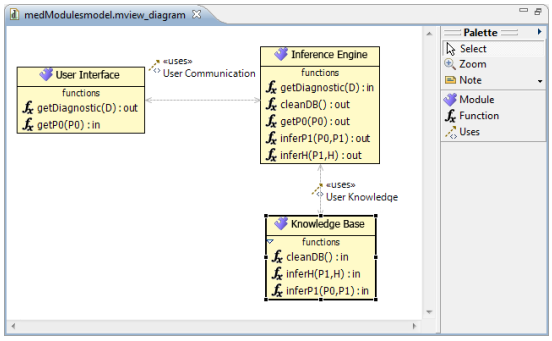


Figure 11. Expert Systems modular miew model

shot of the result model represented on the *Component–Connector View editor*.

V. Related Works

Software Product Lines have been an important discussion topic in the last decade. There are many studies on this subject. Our research is related to the following SPL ones:

- In [4] Batory et al. capture the domain features in a Feature Model. In BOM we capture features in two kinds of Feature Models. In our research, we observed that the variability problem is not solved by means of a unique feature model and the monotonic gluing of these features. We have taken a new approach, which manage the variability in two phases (one by building a base architecture using the domain features, and another one by decorating these base architectures with the application domain features) in order to obtain the final product.
- In [21] Trujillo uses Feature Oriented Programming (FOP) as a technique for inserting features into XML documents by means of XSLT templates. In BOM-LAZY we use this technique but at the model level (i.e. we use Feature Oriented Modeling) by means of QVT–Relations Transformations. The features are inserted on the skeleton model in order to obtain the PRISMA architecture model.
- Clements and Northrop use in [8] the SPL development approach, considering a clear division between domain engineering and application engineering phases for the reuse and the automation of the software process. In BOM, we have used this approach to develop our SPL.
- Ávila-García et al. [2] use process modeling in SPEM to pack reusable assets. In our approach we use this OMG standard too when we model the production plan.
- In [3] Bachmann et al. propose to separate the variability declaration of the affected assets in separate artifacts. In BOM, the specification of the variability and the functionality are captured in different feature models. The use of instances of such feature models allows the user to input the information of the domain features. Those features allow to build the associated assets, or to define the application domain features in order to configure the final application.
- Czarnecki et. al. propose in [9] a notation for cardinality-based feature modeling. In this sense, our work shares most of this notation as it is widely known and used, but we have included some variants. Although there is tool support for the Czarnecki notation (provided by the *Feature Modeling Plugin* (FMP) [1]), we have built a new tool for this notation and we have integrated it in our prototype. The new tool is necessary because FMP represents configurations as specializations of the feature models, and this representation is not well-suited for MDE processes as discussed in [13].

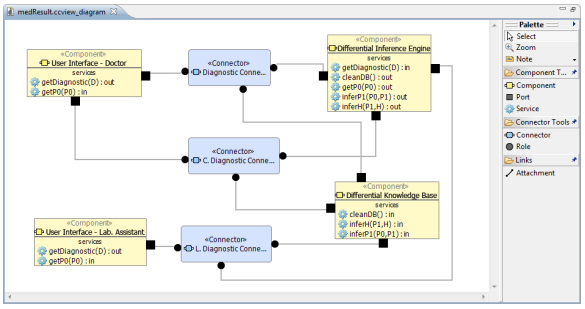


Figure 12. Resulting base architectural model

VI. Conclusions

This paper describes how to develop a Software Product Line in a specific domain (Diagnostic Expert Systems in this paper, but not limited to them) by using Model–Driven techniques. The development of such a kind of systems is a complex process because of elements that compose their architecture vary not only in their behaviour but also in their structure. This situation implies that several base architectures are obtained on the same reference architecture. Our approach uses QVT–Relations as the model transformations language to manage the variability along the whole process. Our approach also enhances the development of DES by applying SPL techniques, as they are useful when the members of a family of programs share a common design. This way, a specific design can be used in different products, reducing costs, time to market, effort and complexity. By applying MDA techniques, we are able to build systems that are platform-independent, and we can think about them from the problem perspective and not the solution perspective. This makes possible to apply such solutions to different domains. Moreover, we provide a framework with several technical spaces where modern software development techniques coexist in a coordinated way. It is noteworthy to point out that, although this paper only describe the variability management for the first stage of the development process of DES (as shown in section III),

the process continues until the base architecture is obtained. The second stage has been also implemented and the base architecture is then decorated with the application domain features. The result of the second stage produces a final and specific architecture. In our SPL the final architectural model is a PRISMA [5] model. PRISMA is a framework to describe architectural models that provides the PRISMA-MODEL-COMPILER tool [20]. This tool is able to automatically generate executable C#.NET code. This way, our proposal covers the whole development process.

Furthermore, in traditional approaches, the group of base architectures is defined and implemented at design time of the SPL (domain engineering) and it remains unchanged throughout the whole life-cycle of the SPL (application domain). In the BOM–Lazy approach, the use of the T1 transformation allows us to move the creation of the base architectures to the application domain phase. This allows us to define the base architectures by using a set of rules that encode patterns of good design practices (as well as other design decisions), in a generic way. This avoids the need to define all of them explicitly. As the LPS grows in size, BOM–Lazy becomes an adequate approach to manage variability, as it supposes a great work to build *a priori* the base architectures for all the possible products of the SPL. Thus, the main effort is done in the domain engineering stage, where the acquired knowledge is formalized and encoded in a set of declarative rules (the knowledge is stored explicitly). So, it is not necessary to develop extensively all the possible combinations of base architectures (the knowledge is stored implicitly). That will increase the efficiency on the application engineering phase, where each base architecture is obtained only when it is needed by using the explicitly stored knowledge.

VII. Acknowledgments

This work has been supported by the Spanish Government under the National Program for Research, Development and Innovation MULTIPLE TIN2009-13838 and the FPU fellowship program, ref. AP2006-00690.

References

[1] M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. *2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, 2004.

[2] O. Avila-García, A. Estévez, and J. L. Roda. Integrando modelos de procesos y activos reutilizables en una herramienta MDA. In *JISBD'06 proceedings*, pages 483–488, 2006.

[3] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In *Software Product-Family Engineering*, pages 66–80. 2004.

[4] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December, 2006.

[5] J. P. Benedí. *Prisma: aspect-oriented software architectures*. PhD thesis, Universidad Politécnica de Valencia, May 2008.

[6] M. E. Cabello and I. Ramos. Expert systems development through software product lines techniques. In *Information Systems Development*, pages 299–307. Springer US, 2009.

[7] M. E. Cabello Espinosa. *Baseline-Oriented Modeling: An MDA approach based of Software Product Lines for applications development.* PhD thesis, Dec. 2008. http://hdl.handle.net/10251/3793.

[8] P. Clements and L. Northrop. *Software product lines: practices and patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[9] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[10] Eclipse Organization. The Graphical Modeling Framework, 2006. http://www.eclipse.org/gmf/.

[11] EMF. http://download.eclipse.org/tools/emf/scripts/home.php.

[12] J. C. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1989.

[13] A. Gómez and I. Ramos. Cardinality-based feature models and model-driven engineering: fitting them together. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, volume 37, pages 61–68. ICB-research report, 2010.

[14] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[15] I. Kurtev, J. Bezivin, , and M. Aksit. Technical spaces: An initial appraisal. *In Tenth International Conference on Cooperative Information Systems (CoopIS), Federated Conferences Industrial Track, California.*, 2002.

[16] Object Management Group. MDA Guide Version 1.0.1. 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[17] Object Management Group. Meta Object Facility 2.0 QVT Specification. 2008. http://www.omg.org/spec/QVT/1.0/PDF.

[18] Object Management Group. Software Process Engineering Meta-Model, version 2.0. 2008. http://www.omg.org/cgi-bin/doc?formal/2008-04-01.

[19] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.

[20] J. Pérez, N. Ali, J. A. Carsí, I. Ramos, B. Álvarez, P. Sanchez, and J. A. Pastor. Integrating aspects in software architectures: Prisma applied to robotic tele-operated systems. *Information and Software Technology*, 50(9-10):969 – 990, 2008.

[21] S. Trujillo. *Feature Oriented Model Driven Product Lines*. PhD thesis, School of Computer Sciences, University of the Basque Country, March 2007.