Model Driven Engineering for Crop Monitoring Applications

Haeng-Kon Kim
Dept. of Computer Information & Communication Engineering, Catholic University of Deagu, Korea

Abstract

Along with the boom of Mobile services and the thriving Model Driven Architecture (MDA), we must consider the growing significance and utility of modeling in the development of software and solutions. The main advantages of MDA are the ability to transform one PIM into several PSMs, one for each platform or technology in which the final system will be deployed, and the automatic code generation that implements the system for those platforms from the corresponding PSMs. Service-oriented architectures (SOA) are also touted as the key to business agility, especially when combined with a model-driven approach. Model-Driven Architecture (MDA) is a well-developed concept that fits well with SOA, but until now it has been a specialized technique that is beyond practical application scope of most enterprises. Model driven and Domain engineering is the foundation for emerging “product line” software development approaches and affects the maintainability, understandability, usability, and reusability characteristics of family of similar systems. However, the existing domain engineering methods do not elicit information necessary for the Component-based Software Development process, such as, selecting and configuring appropriate components.

In this paper, we suggest a method that systematically defines, analyzes and designs a crop monitoring frameworks domain to enhance reusability effectively in Model Driven Engineering and Component-based Software Development. For this, we extract information objectively that can be reused in a crop monitoring domain from the requirement analysis phase. We sustain and refine crop monitoring information, and match them to artifacts of each phase in domain engineering. Through this method, reusable domain components and malleable domain architecture can be produced. In addition, we show the practical applicability and features of our approach via a new clouding and IOT technology.

Keywords: Crop Monitoring Domain Engineering, Crop Monitoring Domain analysis, Software Components Reuse, Component-based Software Development, Model Driven Domain Model, Crop Monitoring Domain Architecture

1. Introduction

Domain engineering supports application engineering by producing artifacts necessary for efficiency of application development. Therefore, domain engineering has to be tailored to CBSD process. The existing domain engineering methods don’t elicit information necessary for Component-based Software Development process, such as, selecting and configuring appropriate components. Also, the existing domain analysis and design method do not represent objective analysis processes that extract and determine the properties of the domain, such as commonality and variability. In addition, the method that domain information are explicitly reflected to the domain component and the domain architecture is deficient.

In this paper, we present a model-driven approach to SOA modeling and designing
complex crop monitoring for convergence agriculture information systems based on MDA. MDA separates the Platform Independent Model (PIM) from the Platform Specified Model (PSM) of the system and transforming between these models is achieved via appropriate tools. The paper proposes a new approach to modeling and designing service-oriented architecture. In this approach the PIM of the system is created and then the PSM based on SOA is generated (this PSM is a PIM for next level). Then the final PSM based on a target platform is generated. These models are generated with transformation tools in MDA and an approach to the model driven development for crop monitoring applications on SOA is presented as in figure 1. The goal of the approach is to minimize the necessary human interaction required to transform a PIM into a PSM and a PSM into code for a SOA. To further separate the architectures specific components of the PSM from the business specific components of the PSM, a UML crop monitoring Profile is introduced and a separation of the PSM layer into two parts is proposed which make the automated transformations from PIM to PSM to code easier to implement and more transparent for system designers, developers, and users. The separation of concerns introduced on the PSM layer is mirrored on the code layer by the use of Java annotations, allowing the same business code to run in different domains simply by exchanging the annotations and thus decoupling application code and SOA middleware.

![Visual editing](image)

**Figure 1. Crop Monitoring Model Application**

### 2. Related Works

#### 2.1. Modeling Mobile services metadata based on MDA

Mobile services are emerging as the perfect framework for application-to-application integration or collaboration, to make these applications available as Mobile services. To standardize the use of Mobile services, the World Wide Mobile Consortium (W3C) proposed the Mobile Service Description Language (WSDL) standard, an XML-based language that describes Mobile service functionality. Essentially, a WSDL file is a language-independent XML-based version of an IDL (Interface Definition Language) file that describes the operations offered by a Mobile service, as well as the parameters that these operations accept and return. Thus, WSDL has become the standard that supports the description of Mobile services: What they do, how they should be used, and where they are localized [8, 9].


- The **WS-Policy** specification describes the syntax for expressing policy alternatives and for composing them as combinations of domain assertions. The WS-Policy
specification also describes the basic mechanisms for merging multiple policies that apply to a common subject and the intersection of policies to determine compatibility.

- The **WS-Policy Attachment** specification describes how to associate policies with a particular subject. It gives normative descriptions of how this applies in the context of WSDL and UDDI, (Universal Description, Discovery, and Integration), and it provides an extensible mechanism for associating policies with arbitrary subjects through the expression of scopes.

Along with the boom of Mobile services and the thriving Model Driven Architecture (MDA), we must consider the growing significance and utility of modeling in the development of software and solutions. MDA, which was proposed by the Object Management Group (OMG), is a model-driven framework for software development that proposes to model the business logic with Platform-Independent Models (PIMs) to later transform them on Platform-Specific Models (PSMs) by using transformation guides between the different models. The main advantages of MDA are the ability to transform one PIM into several PSMs, one for each platform or technology in which the final system will be deployed, and the automatic code generation that implements the system for those platforms from the corresponding PSMs.

Because Mobile services are software components, the development of Mobile services must exploit the advantages of MDA. To apply the MDA principles in the development of Mobile services, a modeling process must be considered. According to MDA principles, this modeling activity should result in automatic code generation. If we want to abstract from the platform in which the Mobile service will be deployed, the code that should be generated is the WSDL document that contains the Mobile service description in a standard format.

### 2.2. MDA Main Concepts

The main concepts of the MDA are beginning to be identified [6,7] A model represents a particular aspect of a system under construction, under operation or under maintenance. A model is written in the language of one specific meta-model. A meta-model is an explicit specification of abstraction, based on shared agreement. A meta-model acts as a filter to extract some relevant aspects from a system and to ignore all other details. A meta-meta-model defines a language to write meta-models. There are several possibilities to define a meta-meta-model. Usually the definition is reflexive, *i.e.*, the meta-meta-model is self defined. A meta-meta-model is based at least on three concepts (entity, association, package) and a set of primitive types. The OMG MOF contains all universal features, *i.e.*, all those that are not specific to a particular domain language. Among those features we find all that is necessary to build meta-models and to operate on them. Maintaining a specific tool for the MOF would be costly, so the MOF is aligned on the CORE part of one of its specific metamodels: UML. UML thus plays a privileged role in the MDA architecture. As a consequence, any tool intended to create UML models can easily be adapted to create MOF meta-models. MDA utilizes models and a generalized idea of architecture standards to address integration of enterprise systems in the face of heterogeneous and evolving technology and business domains.
MDA combines computer-aided verification and machine intelligence during modeling to discover and remove design bugs before code reviews and testing. MDA Meta model acts as a filter to extract some relevant aspects from a system and to ignore for all other details. A meta-meta-model defines a language to write meta-models. The application of MDA to a use case begins by focusing on the development of the models. Figure 2 shows the MDA process that includes: Computation Independent Model (CIM): describes concepts of a given domain but does not describe the software system. Platform Independent Model (PIM): describes software behavior that is independent of some platform. Platform Specific Model (PSM): describes software behavior that is specific for some platform. The first step in using MDA is to develop a CIM which describes the concepts for a specific domain. The CIM focuses on the environment and requirements of the system; the details of the structure and processing of the system are hidden or as yet undetermined. The next step involves developing the PIM. The term “platform” can have various meanings and can include one or more system aspects such as operating system, network configurations, and programming language. The meanings of PIM and PSM models are therefore relative to the definition of platform used in the use case. More important than the definition of platform is the recognition that PIMs and PSMs are supposed to separate aspects of program behavior from aspects of implementation. The third step is developing one or more PSMs which characterize a particular deployment of a software application. This could, for example, focus on the properties of a mobile application, whether the application should be generated in Java or Visual Basic, or whether the installation was for a standalone or networked machine. MDA requires development of explicit transformations that can be used by software tools to convert a more abstract model into a more concrete one. A PIM should be created, and then transformed into one or more PSMs, which then are transformed into code.” The mappings between models are meant to be expressed by a series of transformation rules expressed in a formal modeling language. “A CIM is a software independent model used to describe a business system. Certain parts of a CIM may be supported by software systems, but the CIM itself remains software independent. Automatic derivation of PIMs from a CIM is not possible, because the choices of what pieces of a CIM are to be supported by a software system are always human. For each system supporting part of a CIM, a PIM needs to be developed first.”

It is possible for concepts defined in a CIM to be automatically associated with properties defined in a PIM. For example, the concept “protein” defined in a CIM about proteomics experiments could be associated with PIM concepts such as a help feature that defined protein for users or a drop down list of protein names.

A meta-model in MDA defines a specific domain language. It may be compared to the formal grammar of a programming language. In the case of UML the need to define
variants of the base language was expressed. The UML meta-model was then equipped with extension mechanisms (stereotypes, tagged values, constraints) and this allows defining specialization of the basic meta-models as so called profiles.

The MOF contains features to serialize models and meta-models in order to provide a standard external representation. The XMI standard defines the way serialization is performed. This is a way to exchange models between geographical locations, humans, computers or tools. When a tool reads a XMI serialized model (a UML model for example), it needs to check the version of the meta-model used and also the version of the XMI applied scheme.

2.2. SOA

SOA exposes real dependencies against artificial ones [11]. A real dependency is a state of affairs in which one system depends on the functionality provided by another. Beside real dependencies there are always artificial dependencies in which the system becomes dependent to configurations and various musts other systems expose. The target of SOA is to minimize artificial dependencies (although it can never be completely removed), and maximize real ones. This is done via loosely coupling, and the concept of service. A service is a coarse grain functionality objects, with interfaces expressed via a well defined platform independent language. When using services as computational objects, systems can register, find and invoke each other based on a well defined, every one accepted, language hence no one, highly becomes dependent to another system and a high degree of loosely coupling is achieved.

3. Applying MDA to Crop Monitoring Applications

3.1. Basic Ideas

The MDA organization may be viewed as a set of artifacts, some being standard building blocks, some being user developed. We may envision, in the not too far future, an organization starting with a hierarchical library of meta-models and extending it as an adaptation to its own local context (models as assets). Model reusability will subsume code reusability, with much more efficiency. This may be seen as orthogonal to code class libraries (e.g., Java, Swing, EJB, etc.). Inside a company, the various business and service models will be developed and maintained to reflect the current situation.

Combining a service-oriented modeling architecture with MDA for crop monitoring can bring many unique benefits. Firstly the clear organization of models and information based on the stereotypes derived from the service-oriented architecture and select perspective as stereotypes for development process. Secondly the productivity, quality and impact analysis benefits of the use of MDA with its emphasis on automation, transformation and synchronization. MS2Mobile solution for MDA in our approach is uniquely positioned to take advantage of the unified modeling architecture which results from bringing these
two key architectures together. MDA combines a uniquely powerful implementation of the mobile services vision, together with the industry leading solutions for modeling service-based solutions.

Figure 3 shows our architecture for applying MDA to crop monitoring and mobile service application in this paper. First it defines the language used for describing object-oriented software artifacts. Second, its kernel is synchronized with the MOF for practical reasons as previously mentioned. There is much less meta-modelers (people building meta-models) than modelers (people building models). As a consequence it is not realistic to build specific workbenches for the first category of people. By making the MOF correspond to a subset of UML, it is possible with some care to use the same tool for both usages. As a consequence the MDA is not only populated by first class MOF meta-models, but also with UML dialects defined by UML profiles for specific purposes languages. This is mainly done for practicality (widening the market of UML tools vendors) and there is some redundancy between UML profiles and MOF meta models (It is even possible to find conversion tools). There are many examples of profiles. Some are standardized by OMG working groups and other are independently defined by user groups or even by individuals. Examples of profiles are "UML for APL", "UML for CICS", "UML for Scheduling Process and Time" (real-time applications), "UML for EJB", "UML testing", "UML for EAI", "UML for QoS and fault tolerance", "UML for Cust Sys".

One important kind of model that is being considered now is the correspondence model. A correspondence model explicitly defines various correspondences that may hold between several models. In the usual case, there are only two models: the source and the target. There may be several correspondences between a couple of elements from source and target. The correspondences are not always between couples of elements and they are strongly typed. There is not yet a global consistent view on correspondence models since this problem is appearing from different perspectives.

![Figure 4. Architecture for Crop Monitoring Development in this Paper](image)

When the notion of PDM and virtual machine is clarified we may then tackle the definition of a PIM, a model containing no elements associated to a given platform. In other times this was simply called a business model, but as for platform models we need to progress now towards a less naive and a more explicit view. The first idea is that the PIM is not equivalent to a model of the problem. We propose the architectural model for many elements of the solution that may be incorporated in a PIM as long as they don't refer to a specific deployment platform as in Figure 4.

In our architecture model as in Figure 3, the PIM of the system is created using UML diagrams by the analyst of the system. The PIM of the system will be designed simply without thinking about services that is pretty simple and is accomplished as CBD(Component-Based Development). The SOA-based PSM (which is a PIM for the
next level) would be derived from the present PIM. The way which is used to identify this PSM must be quite different from the one used to identify PSM in component-based systems; because in component based systems the patterns which are used to determine the PSM of the system have a specific form. For each service in e-business applications, there is a single instance which manages a set of resources and consequently, unlike components, services are for the most part stateless that means need to view a service as a manager object that can create and manage instances of a type, or set of types. According to above discussion, in our approach after creating the PIM, this PIM is transformed -with a transformation tool- to another PIM based on SOA. In this transformation, for each class diagram in PIM for crop monitoring, a Service Manager is created that manages the Instant Services. This management involves creation, deletion, updating a service and state management of services. To complete this transformation, we need some other special patterns for dealing with associations between classes. When this PIM based on SOA is created, the PSM of the system can be created based on a target platform such as Mobile Services, crop monitoring and/or other platforms with transforming tools. Some operations apply on a single model and are called monadic by opposition to dyadic operations applying to two models. Operations applying on more than two models are more rare.

![Figure 5. PIM of Crop Monitoring Applications](image)

Obviously the most apparent components in an MDA workbench are the precise tools composing this workbench. Fortunately in this context we should be able to propose a rather precise definition of a tool: it is an operational implementation of a set of operations applicable on specific models. The meta-models supported by a tool should be exhaustively and explicitly defined.

**Generating the PIM for Crop Monitoring**

PIM for crop monitoring application is an abstract design of a computerized solution which does not include any platform specific elements. The core of the platform independent model (PIM) is a UML model – ranging from use cases through classes,
interactions, states and other UML elements to the components as in Figure 5.

**Translation from PIM to PSM**

While the PSM entities, i.e. Java classes or entity beans, bear the structure and deliver the behavior of inventory entities as described in the original inventory PIMs, end-users should not interact directly with these entities. Rather, entities should be accessed through a single interface that exposes a simple set of management methods and hides their complexity. This is a standard design guideline, which conforms to the related design pattern and influences the architectural design of components. In order to comply with the guideline, the case-study aims at implementing an application tool that allows users to manage the inventory content through a simple GUI. Example users of such a tool may be front-desk operators who respond to customer calls and access the inventory to setup a new or change the state of an existing product/service instance.

![Figure 5. Overall Architecture for PIM to PSM Translation](image)

**Figure 6. Overall Architecture for PIM to PSM Translation for Crop Monitoring**

The case-study uses MDA to automatically generate the tool and associated GUI in Java and J2EE (session bean) in order to deliver the required embedded pattern and design guideline. Again, this paper only concentrates on the Java outputs. Figure 6 shows the overall architecture for PIM to PSM translation. Transforming PSM based on SOA to the PSM based on crop monitoring Services using WSDL is a straightforward task. In our approach, each value object and each interface in PIM will be transformed to WSDL Type and Port Type in the PSM respectively and the parameters of methods will be transformed to the Messages (Input/Output) in the PSM.

A transformation $t$ transforms a model $M_a$ into another model $M_b$: $M_a \rightarrow M_b$. Model $M_a$ is supposed based on meta-model $M_Ma$ and model $M_b$ is supposed based on meta-model $M_Mb$. We note this situation as: $\text{sem}(M_a, M_Ma)$ $\text{sem}(M_b, M_Mb)$ As a matter of fact, a transformation is like any other model. So we'll talk about the transformation model $M_t$: $M_a \rightarrow M_b$. Obviously since $M_t$ is a model, we postulate the existence of a generic transformation meta-model $M_Mt$, which would be similar to any other MOF based MDA meta-model:

In some cases the transformation takes some particular form if the source and target meta-models are in the relation of refinement like a CORBA and a CCM meta-model. Figure 7 show the examples of translation interface.

### 3.2. Model Driven Engineering for Crop Monitoring Applications

The general mutual progression of Domain-driven Component-based Software development process presented in this study is as in Figure 7.

Our process model for component-based software development explicitly considers reuse-specific activities, such as componential design, component identification, and component adaptation. It is comprised of seven major activities, starting with context comprehension and requirement analysis, continuing with the combination of
componential design and component identification, component creation, component adaptation, and finally ending with component assembly. Throughout the process, explicitly stated domain artifacts- domain specifications, domain model, and domain architecture - are produced. Component-based Domain Engineering depends on the component-based software development process. In the first step of domain engineering, domain definition, the purpose of the domain is decided, and its scope is confirmed. In the domain modeling step, a domain model is obtained by analyzing the domain. Domain analysis has to identify the stable and the variable parts of the domain. Based on this domain model, the domain components are identified, and the domain architecture is created. Our process model for domain engineering has an objective analysis activity in each step, i.e., generalization process. The generalization process is tasks that classify the properties of domain requirement, domain usecase, and domain component and transform these into reusable form according to the properties. The artifacts of each step are maintained and saved with interrelationships. They are reused as useful information during component-based software development.

In this paper, we suggest some domain engineering processes- domain definition, domain modeling, and domain design- for launching a study among the Domain-driven CBSD process that is represented in Figure 1. Detailed activities are shown in Figure 7.

![Crop Monitoring System Architecture](image)

**Figure 7. Crop Monitoring System Architecture**

The purpose of the Crop monitoring Domain Definition step is to create domain specifications by bounding the domain scope and defining the domain purpose. In addition, requirements of domain are extracted from legacy and new systems in the domain and converted to generalized type reflecting properties-common, optional, variable.

1. **Decide Domain Scope**

As defined earlier, the domain is a collection of related systems, which can lead to vague interpretations, so it is imperative that ambiguities are made clear. If the scope of the domain is large, more systems can be contained in that domain, and it will be easy to contain new systems in the future. However, this leads to a reduction in commonality in the domain. Consequently, more commonality can be extracted in a domain of smaller scope.

**Distinguish Domain External Stakeholder.** Domain external stakeholder means people with interest in the functions provided by a domain. People who are interested in input or
output of a domain or people who handle an external system related to the domain can be extracted as external stakeholders.

Figure 8. Detailed Activities of Crop Monitoring Domain Engineering

Define Domain Assumption. The domain assumption means pre-conditions that are to be satisfied by using components that are provided by the domain. Domain assumption performs a basic role to decide whether or not a system can be included in the domain in the initial step. Subsequently, it has influence on the decision of the domain’s component properties that will be extracted later.

Describe Domain Environment. Domain environment is divided into domain external environment and domain internal environment. Domain external environment presents clearly the boundary of the domain by analyzing interaction between the domain and its external factors. Domain internal environment presents factors that should be distributed within the domain and its functions accordingly.

(2) Define Crop Monitoring Domain Purpose

After the scope of the domain is set, a rough outline centering on the functionality of the domain is explained. Additionally, a domain concept schematic diagram is drawn outlining the domain’s business processes related to its purpose.

Describe Domain Purpose. The important function of the domain is described. It is an essential factor that all the systems belonging to the domain should have. Furthermore, it functions as a basis to decide whether the system should be included in the domain.

Model Domain Concept. Major tasks, which need to be clearly defined within the domain, and related terminology are extracted. Furthermore, relationships among these are identified in general drawings. Through these activities concepts within a domain are expressed.

(3) Domain Modeling

The purpose of the Domain Modeling step is to analyze the domain and to develop the domain model composed of a domain requirement model and a domain type model with commonality and variability. A domain model captures the most important “things” – business objects or process and prepares variable things within the context of the domain. We use usecase analysis technique as an appropriate way to create such a model. The usecase leads to a natural mapping between the business processes and the requirements [9].

The domain requirement model expresses the requirements extracted from the domain by the usecase diagram of UML. This induces the analyzed primitive requirements to be bundle into a suitable unit.
Construct Domain Usecase Diagram. The actor is extracted from the domain stakeholder and the domain external environment. The requirements of such an actor and domain operations are extracted as a domain usecase.

![Figure 9. Domain Usecase Description](image)

Then a domain usecase diagram is drawn. The domain usecase is written with different levels of detail. The primitive requirements identified during the prior step should all be allocated to usecases. This provides an important link in terms of traceability between the artifacts. The domain usecase diagram should be modified to reflect the properties of domain usecase after the domain usecase generalization process.

Describe Domain Usecase Description. Domain usecase description is described in the items of Figure 9 on each usecase. At this time, a reference appointment of variable requirement is presented with template marking - ◁.

(4) Domain Usecase Generalization Process

A task to classify the properties of domain usecase and reconstruct the domain usecase according to these properties is defined as ‘domain usecase generalization process’. Properties of domain usecase are influenced by PR’s properties.

Construct PR-Usecase Matrix. Create a PR-Usecase matrix to recognize the property of each usecase by referring to the domain usecase diagram and description and the PR-Context matrix. The usecase name, primitive requirement, and the property of primitive requirement are displayed in the matrix. Moreover, the primitive requirements that are contained in each usecase are analyzed. Figure 10 presents the PR-Usecase matrix.

Generalize Domain Usecase. When analyzing usecase, we can consider usecase conditions as the following; at this time, we can divide and rearrange usecases on their necessity. This is presented in Figure 10. First in considering the usecase condition, a usecase contains primitive requirements, which does not overlapped with that of other usecases. (1 of Figure 10). In this case, no re-arrangement is necessary. Second, the primitive requirement is spread over to many usecases (2 of Figure 10). In this case, separate commonly overlapped primitive requirements, make it an independent usecase, and connect it to include-relationships. Third, a usecase includes variable primitive requirements. (3 of Figure 10). In this case, a confirmation on the possibility of whether variable primitive requirements can be separated and created as independent usecase is addressed.
If possible, they are separated. If not, they are maintained as involved in a usecase and a variable point is stored. Finally, a usecase includes optional primitive requirements (④ of Figure 10). In this case, the optional primitive requirements are separated and connected to the extend-relationship.

The usecases that were reorganized by this process are classified by the properties as follows:

- **Common Usecase** – When the usecase has primitive requirements, which must exist within the domain, it is classified as a common usecase and represents an important process in the system.
- **Variable Usecase** – When the usecase is composed of variable primitive requirements, this is classified as a variable usecase. It means a usecase with requirements that exist in each specific application of a domain but can be variable. Mainly, it tends to appear overlapping in many usecases. In case it was divided into an independent usecase through the PR-Usecase matrix analysis, it belongs to this class.
- **Optional Usecase** – It represents the usecase that doesn’t always need to exist when handling a process in the system; this corresponds to a usecase composed of optional primitive requirements.
- **Usecase with variables** - When primitive requirements with variation are difficult to be separated independently, this is involved in the usecase. Even though this cannot be divided separately, it can be used when identifying a domain component of the next step and draw the component interaction diagram at the domain design step by classifying this status.

**Figure 11. Domain Component Extraction Standard**

(5) **Develop Crop Monitoring Domain Architecture**

Domain architecture presents the structure of domain components, interaction between
domain components in multiple views, and specifications of the domain component. Domain architecture should be independent of any specific technology or set of developmental tools. Domain architecture should reflect properties such as commonality, variability and optional that were initialized from the requirement analysis step, refined, and maintained. Such features allow part of the architecture to be divided and replaced according to the component’s property when creating the component-based software development. So, malleable architecture can be created.

**Domain Component Interaction View.** Domain Component Interaction View represents interactions between domain components that perform specific requirements. In addition, a domain component interface is extracted by analyzing operations between components. Domain Component Interaction View is presented by using an Interaction diagram, and component interface is described by using class notation. Figure 8 presents Domain Component Interaction View.

One interface is a set of externally visible operations provided by a component. The property of the operation is defined based on the property of the domain component that was extracted. That is, all operations that variable and optional domain component provide are defined as variable and optional operations respectively. But, variable operation may be additionally identified from common domain component by refining the function of the domain component. Such operation is presented as «v.p>> (variant point) when describing interface. The interface of common domain component has not only provided operations but also required operations information. At this time, the properties of a required operation can be identified by analyzing the interaction of the domain components that have a different property. For example, a required operation of the ComponentB, operationO1 is defined as optional property based on ComponentO1 with optional property as shown in Figure 8. Details on interface will be described in domain component specifications, and expected cases on variable operations will also be described together.

In this view, variable domain components can be templated and optional components can be pruned in this view. Through these processes, the domain component interaction view becomes the analysis model that has common elements only, which is defined as Commonality Analysis Model in this paper. The Commonality Analysis Model presents an execution view of most basic steps of component based software development.

**Domain Development View.** All computer applications have three general areas of functionality; User services, Business services and Data services. Domain structure can be divided into common, variable and optional parts by the features of the domain.

In the domain development view, the domain structure is divided and presented in namely 2\textsuperscript{nd} dimension layers through logical partitioning of functionality as horizontal and property division of the domain as vertical. Not only does the systemized view allow for independent performance and quick change at each step, but also is becomes the foundation for various physical partitioning (deployment alternatives) such as 2 tier or 3 tier, n tier, and Web-enabled applications.
Domain Component Specifications. Domain Component Specifications describes the purpose and interfaces of a component and furnishes information about what the component does and how it can be used. A deployable component, which is developed using a Domain Component Specifications, can differ in granularity according to applications. Hence, we will describe the related functions as interface and supplement required interfaces to effectively support variable granularity of the component. In this way when interfaces are developed independently, required interfaces can be recognized easily. Also the Domain Component property is explicitly represented using a ‘type’ tag in the interface. The ‘type’ tag can have common, variable, or optional values. If the ‘type’ tag has a variable or optional value, it can be described as a predictable case by a ‘rule’ tag in the interface. Fig. 10 presents Domain Component Specifications.

4. Conclusion and Future Work

The problems of modeling solutions based on SOA have largely been resolved through the recognition of the importance of loose coupling and the consequent separation of concerns. Reinforced by the Supply-Manage-Consume concept, the separate modeling of solutions and services is a well established practice incorporated into advanced development processes that support SOA, including Select Perspective. Service
Interfaces are shared amongst models showing the implementation and re-use of the services. Whilst the use of modeling within SOA is well established, it has suffered from the same issues as modeling in other architectures. The abstraction gap between the level of detail expressed in the model and the level of detail expressed in the code is a key issue.

Yet it is the abstraction gap which is one of the key targets for the Model Driven Architecture. Combining a service-oriented modeling architecture with MDA can bring many unique benefits. Firstly the clear organization of models and information based on the stereotypes derived from the service-oriented architecture and Select Perspective as development process. Secondly the productivity, quality and impact analysis benefits of the use of MDA with its emphasis on automation, transformation and synchronization. Select Solution for MDA is uniquely positioned to take advantage of the unified modeling architecture which results from bringing these two key architectures together.

In this paper we introduced an approach to modeling and design of complex Crop Monitoring systems using SOA and MDA. In fact, to exploit the benefits of SOA effectively and duly, we propose an approach that involves MDA into the context. In this approach the PIM of the system is created and then the PSM based on SOA is generated. Then the final PSM based on a target platform is generated. These models are generated with transformation tools in MDA.

In this study, which is different from the existing methods, processes were suggested for domain analysis and design method that have mutual action suited to component-based software development.

Namely, in the existing study, it could not obtain information on procedures to assemble components in consideration of their relationship using architecture and recognition of domain components. Therefore, this study recognized and selected components that were required during the development of component base software. Furthermore, a connected relationship between the components and the interface information through domain engineering processes were calculated which supported the component base software development process.

Also, this study deducted a method to find commonality and variability, which is necessary to extract information with objectivity through the generalization processes while existing studies depended solely on experience and intuition by a domain specialist. In addition, this information was relocated into matrix form to be maintained, refined, and used in each step to find common usecase and common domain component.

This study reflected such features into the shape of the domain architecture, created a malleable architecture that can be partially separated architecture, and replaced them by the property of the components during component based software development.

Future studies will progress in two directions. We will review the possibility to implement a domain component by using domain architecture and study ideas and technologies that can be applied. Also we will study a process that can be implemented for the development of component based software by using a proposed domain analysis and design method. Hereby, the general study process that was proposed from the beginning will be completed.

**Acknowledgment**

This research was Supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the C-ITRC (Convergence Information Technology Research Center) support program (IITP-2015-H8601-15-1007) supervised by the IITP (Institute for Information & communication Technology Promotion).
This research was also supported by the International Research & Development Program of the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (Grant number: K 2013079410).

References