Vol.2009-SE-165 No.4
2009/7/2

Platform Independent Model Transformation based on Architectural Patterns

Rafael Weiss,[†1] Hiroshi Kazato,[†1,†2]
Shinpei Hayashi[†1] and Motoshi Saeki [†1]

In this paper, we present the development of a model-driven approach to transform platform independent models (PIMs) based on architectural patterns. Model transformation is a fundamental concept in nowadays software development to manipulate models during its lifecycle e.g. due to changing requirements or platform technologies. We use model transformation techniques to transform profile-enriched UML2 models into platform specific models (PSMs). These PSMs could be used later as an input for common code generation frameworks to derive platform specific implementations (PSIs). As an example of a possible architectural pattern, we define a UML profile that is based on the well-known Model-View-Controller (MVC) pattern, an architectural pattern commonly used in software engineering to isolate business logic from user interface considerations.

1. Introduction

Model-driven Engineering (MDE)[1)] is a software development methodology, which places models as primary artifacts and can be used to derive executable code from them by means of model transformation. The goals are to increase interoperability, maintainability and reusability of systems by specifying their structure and behavior in a more abstract, platform independent way.

UML is the modeling language of choice for this purpose. UML models are commonly used together with profiles that are a powerful and flexible generic extension mechanism of UML used to customize models for particular domains or platforms. Such profile-enriched models can be used by some MDE tools such as openArchitectureWare (oAW)[2)] and AndroMDA[3)] to attach platform specific information to UML models. Along with recent evolution in model transformation techniques, they have shown the possibility and effectiveness of MDE in practice to some extent.

However, there is still a lack of support for platform independence, another crucial characteristic of MDE. Since the platform of an application may change during its lifecycle, in response to requirement changes or platform evolutions, MDE has to handle such a situation properly. Nevertheless, because of the diversity of model definitions and profiles used for the code generation, it is a labor-intensive task to build, maintain and reuse such models.

To cope with these problems, we propose a model-driven approach called ACCURATE, which uses existing code generators and their profiles as building blocks and introduce another abstraction layer over them. Through this additional abstraction layer, one can support platform evolutions more sustained by using platform independent models for the specification of the system. To achieve this, we define a possible example of a UML profile to describe the functionalities of applications, independently of any platforms. By specifying mappings between platform independent model (PIM) and platform specific model (PSM) elements, developers are able to use PIMs to model an application in a more comprehensive, maintainable and platform independent manner as well as easily switch the platform specific implementation (PSI) to other platforms at any time. The main contributions of this paper are: 1) introducing our model-driven approach called ACCURATE, 2) arguing on model transformation techniques for PIM-to-PSM transformations and 3) showing a possible implementation of a model-driven toolkit that realizes the ACCURATE approach.

The rest of this paper is organized as follows. First, Sect. 2 introduces a brief overview of the proposed ACCURATE approach and argues about the usage of models and model transformation in a software development process. Afterwards, Sect. 3 presents our implementation of the PIM-to-PSM transformation using model transformation techniques. In Sect. 4, we survey some related works and finally close with a conclusion and future work in Sect. 5.

2. ACCURATE Approach

In this section we present the ACCURATE approach. The key idea is that we

†1 Tokyo Institute of Technology
†2 NTT DATA CORPORATION

consider code generators as platform specific and introduce a platform independent abstraction layer on top of them. Since architecture styles are essentially immutable and independent of any platforms, we adopt them to define platform independent models and categorize existing UML profiles according to the established concepts defined in the styles. By defining mapping from architectural concepts to those of the profiles, a PIM can be transformed to PSMs (and thus a PSI) for various kinds of platforms. Furthermore, since architecture styles are used in the earlier design stages, they help developers to elaborate requirements into models.

The name ACCURATE comes from an acronym for 'A Configurable Code generator Unifying Requirements and model Transformation tEchniques'. As it implies, requirements and model transformation techniques are used jointly to establish a continuous workflow from basic analysis artifacts until the derivation of PSIs. Even though the overall ACCURATE approach consists of many different tasks, which will be introduced shortly hereafter, the major focus of this paper is the model transformation of a PIM into a PSM.

2.1 Overview

A brief overview of our approach is illustrated in **Fig. 1**. It defines the four major activities PIM modeling, platform selection, PIM-to-PSM transformation and code generation. These activities are carried out by two kinds of actors, application designers and requirements engineers.

In the proposed workflow, models have to run through different stages during their lifecycle (e.g. a PIM is transformed into a PSM). The workflow has to start with the definition of a formal specification of the structure and the functions of the system by application designers. This PIM modeling is done using our own UML profiles as described in the following Sect. 2.2. Using profiles for the modeling of PIMs enables application designers to describe the functionality of the model elements detailed enough to support a concrete mapping in the following model transformation while still not focusing on any platform specific details in the initial design of the system.

Furthermore, requirements engineers choose an appropriate platform combination for the system. Once the platform configuration is determined (see Sect. 2.3), the PIM of the system can be transformed automatically to a PSM according to

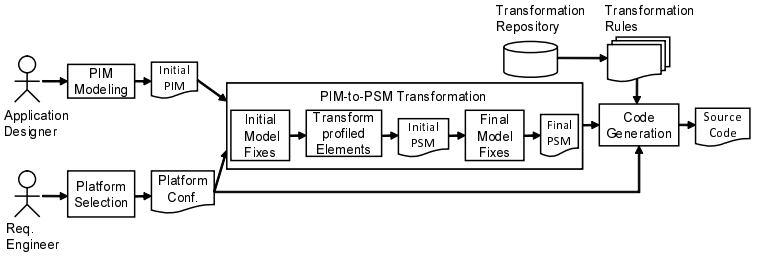


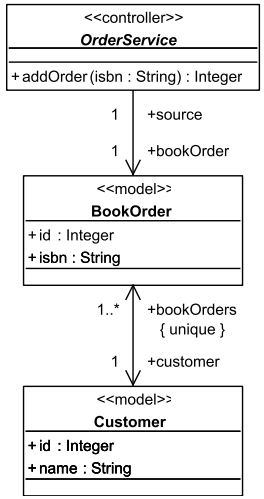
Fig. 1 ACCURATE overview.

the pre-defined mappings of the model elements. The output from this activity that we refer to as PIM-to-PSM transformation is not only a PSM that conforms to the designated platform, but also assures model integrity. Furthermore, common design flaws are corrected by applying model fixes at the beginning and end of the PIM-to-PSM transformation (see Sect. 2.4). This way, the resulting PSMs can be used directly as an input for the code generation.

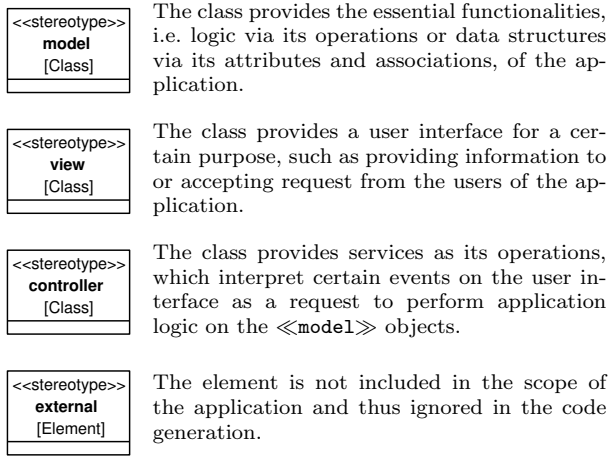
Source code for the application is generated as the final artifact of the workflow. Here we make use of existing code generator frameworks that support multiple platforms by separating transformation rules from their execution engine and storing these rules in the transformation repository. According to the platform configuration, the framework chooses transformation rules from the repository and configures a code generator specific to that platform. Since a valid PSM is provided from the previous stage, a code generation using the ACCURATE approach is as a result less error-prone (see Sect. 2.5). In the following subsections, these activities are explained in more detail.

2.2 PIM Modeling

The PIM is an artifact used in the early design stages to describe the structure and functionality of a system without focusing on any technology specific details. In this paper, we define our own profile for UML, called the ACCURATE profile, which is adopting concepts defined in the architecture style of MVC[4]. The profile has fewer stereotypes and tagged values than a common platform specific UML profile. Thus, developers are able to easily learn and use the profile, while it is still expressive enough to specify an application independently of any platform specific details.



(a) PIM example.



(b) MVC profile.

Fig. 2 Example for using ACCURATE together with MVC.

Figure 2 illustrates a possible ACCURATE profile definition for the well-known `Model-View-Controller` architecture style. According to this style, ≪`model`≫ classes provide core functionalities of an application and propagate changes to ≪`view`≫ and ≪`controller`≫ classes. One can model this mechanism using unidirectional associations from ≪`view`≫ or ≪`controller`≫ classes to ≪`model`≫ classes. One might notice that a single ≪`view`≫ class can only have one ≪`model`≫ class connected while a ≪`model`≫ class can have unlimited ≪`view`≫ classes associated. Furthermore, a ≪`view`≫ class needs to be connected to a ≪`model`≫ class either directly or through a ≪`controller`≫ class, which always has to be associated to a ≪`model`≫ class directly. Other kind of associations (e.g. between two ≪`model`≫ classes or from a ≪`view`≫ class to ≪`controller`≫ classes) are not restricted.

2.3 Platform Selection

Besides the abstract, platform independent description of a system, it is often necessary to extend models for technology specific elements in order to generate concrete PSIs for several platforms of choice. In the model-driven software development, one may achieve this e.g. by adding platform specific information to the model elements via additional UML profiles. It should be considered that this information not necessarily have to be information focusing on platforms only. Thus, a PSM might e.g. contain additional information about data storage techniques or specific middleware that the user wants to adopt for the project.

Table 1 Decision matrix of platform combinations.

≪`view`≫	JSF			Java UI		
≪`model`≫ ≪`controller`≫	Spring	POJO	EJB	Spring	POJO	EJB
Hibernate	+	o	–	+	+	–
POJO	+	o	o	+	+	o
EJB	o	o	+	o	o	+

+: recommended, o: possible, –: unusual

In terms of the platform selection, it should be taken into account that some techniques should not be combined due to some technical constraints or system design standards. One possible solution to this is to adopt architectural patterns to define the system of choice and support the user during the platform selection process. In case of using the MVC pattern within the ACCURATE approach, the only remaining task for the user should be the selection of an appropriate combination of techniques for ≪`model`≫, ≪`view`≫ and ≪`controller`≫ parts of the application conforming to **Table 1**. For example, the selection of Hibernate as ≪`model`≫ together with Java Server Faces (JSF) for the ≪`view`≫ and Spring for the ≪`controller`≫ is a recommended combination of implementation techniques, while the combination of Hibernate for ≪`model`≫, Java UI (used as a summary of native Java UI paradigms such as Swing or AWT) for the ≪`view`≫ and Enterprise Java Beans (EJB) as ≪`controller`≫ is an unusual combination due to technical restrictions of the data access of EJB and Hibernate. This decision matrix can be used as a basic guideline for the platform selection process since more detailed studies of a platform selection via non-functional requirements are out of the scope of this paper and mainly driven by personal experience and flavor of the developer.

2.4 PIM-to-PSM Transformation

To transform a PIM into a PSM, elements of a PIM need to be mapped to conforming PSM elements in a consistent way. Thus, the overall process of trans-

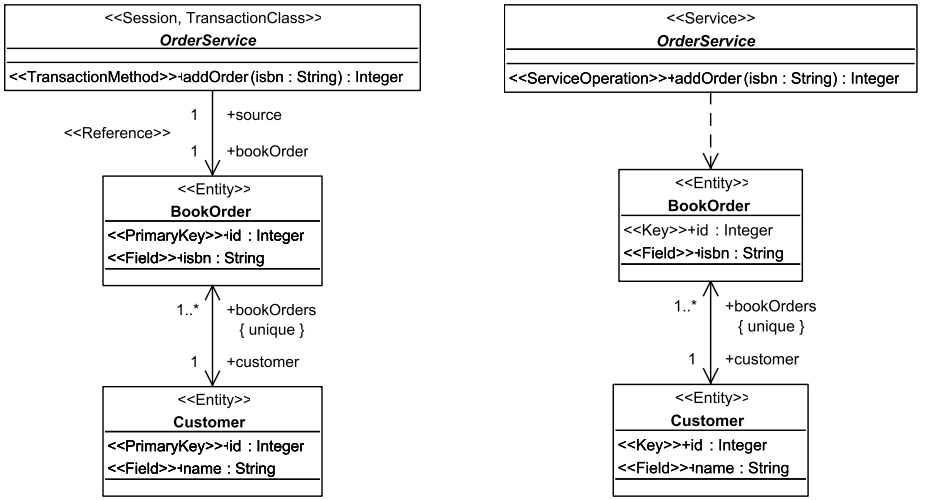
forming a PIM into a PSM consists of three general tasks:

(1) Initial model fixes
(2) Transformation of profiled elements
(3) Final model fixes

To assure the correct structure of the PIM before the transformation of the elements begins, we apply various PIM-to-PIM transformations in the first step to guarantee a consistent PIM structure (as to be seen in Sect. 3.2). We call these transformations the initial model fixes (see Fig. 1). Thus, the correct usage of namespaces, profiles and packaging inside the model is checked and corrected in the initial step of the PIM-to-PSM transformation.

After the PIM is transformed into a consistent model, the second step of the PIM-to-PSM transformation, the transformation of profiled elements, can be triggered. We regard the transformation of a PIM to a PSM as a stepwise conversion of all contained structural elements (e.g. classes, attributes, operations and references) of the PIM due to the defined mappings (as to be seen in Sect. 3.3). For example, one may consider a mapping of the PIM element `Customer` with the stereotype ≪`model`≫ (see Fig. 2) to the PSM element `Customer` with the stereotypes ≪`Entity`≫, ≪`Key`≫/≪`PrimaryKey`≫ and ≪`Field`≫ (see **Fig. 3**). One should notice that besides the transformation of stereotypes of classes and their owned attributes and operations, it is also sometimes required to change the type of model links (e.g. association to dependency) during the PIM-to-PSM transformation and add all required platform specific profiles to the resulting PSM. Furthermore, it is also required to remove unnecessary elements (such as ≪`external`≫ stereotyped elements or use-cases) that where necessary during the requirements and analysis phase of the software development process but are not needed for the PSI generation. As a result, such elements are excluded during the PIM-to-PSM transformation.

As the final step of the PIM-to-PSM transformation, the resulting PSM is checked again for its correctness during the final model fixes. In this step, common design flaws such as unnamed classes or links and missing return values of methods are corrected via PSM-to-PSM transformation. This way, we can guarantee a more accurate and flawless PSI generation based on our generated PSMs.



(a) EJB profile. (b) Spring and Hibernate profiles.

Fig. 3 PSM Examples.

The resulting PSMs for each selected platform combination may vary a lot. Because of this, every PSM has to be transformed independently, which may cause a lot of work and is more error-prone if done manually. Thus, we assume that it is easier to learn the basic modeling technique of a PIM, select a possible combination of platforms and leave the mapping and transformation of the PIM into a PSM up to model-driven transformation tools.

2.5 Code Generation

Taking into account that the PSM may vary during the software development of a system, the required code generator should also be exchangeable to support the generation of arbitrary PSIs. One possible solution for this is to apply code generation frameworks that support various generator components. Such pluggable code generator components are commonly named `cartridges` due to their replaceable character. As a side-effect, cartridge components can be linked to achieve very specific code generation that even conform to platform decisions based on the MVC approach (e.g. Spring and Hibernate can be link together in a shared PSM as to be seen in **Fig. 3(b)**). Since cartridges provide sets of model

transformation rules we refer them as transformation rules that are selected from a transformation repository due to the platform selection. Since the details of the PSM-to-PSI transformation is out of the scope of this paper, we avoid more concrete details at this point and leave it for future work.

3. Implementation of the PIM-to-PSM Transformation

We have developed our tool as a plug-in that can be used together with any Eclipse distribution. It supports the user in transforming a PIM automatically into a PSM that is conforming to the platform selection. Since the tool is following our ACCURATE approach, the resulting PSM can be used directly afterwards as an input for the PSM-to-PSI transformation to derive compilable source code from the models.

The transformation of the PIM into a PSM is a very labor-intensive and error-prone task if done manually each time. To enhance this process the ACCURATE plug-in provides an automatic transformation of an inputted PIM into a PSM. To achieve this, the ACCURATE plug-in offers the following major functionalities:

- Specification of the target platform according to the MVC approach
- Transformation of the PIM into a PSM that is according to the platform selection
- Assurance of model integrity
- Offer user guidance and help system

Besides this, the plug-in can generate configuration files based on the platform selection that can be used by common code generation frameworks such as oAW.

In the following subsections, we are going to specify the technical details of the PIM-to-PSM transformation, introduce our strategy for assuring the model integrity and describe in details how to transform all required elements of the PIM into a platform specific representation.

3.1 Technical Details

One may recognize that both the PIM-to-PSM and the PSM-to-PSI transformation have to be considered as model transformations. The most common used techniques for specifying model transformations are XMI[5] manipulation (such as JDOM[6]), graph transformation and Domain Specific Languages (DSLs) that can be used for transformations (such as xText[7]). In the case of the PIM-to-PSM transformation, we use XMI manipulation techniques for transforming the PIM. Since most of the common UML CASE tools (such as MagicDraw[8]), that can be used for modeling the PIM, have support for exporting the model as a XMI file, we can use XMI manipulations techniques such as JDOM directly to modify the PIM. JDOM offers a lightweight but matured API that can be used to specify detailed transformations of the UML model. It has to be mentioned that JDOM is a programming language specific transformation technique based on Java. Since our plug-in is also developed in Java, JDOM can be adopted without any technical problem. It has to be noticed that any of the above transformation techniques can be used to specify model transformations. Thus, it depends on the developer, which technique of choice is adopted for a specific model transformation.

As it can be seen in Fig. 1, the platform configuration is needed as an input for the PIM-to-PSM transformation and thus need to be selected before the PIM-to-PSM transformation can be triggered. Especially for the mapping of the stereotyped elements to PSM elements it is necessary to know which technique should be used for each MVC component in order to achieve a PSM that is conforming to the platform selection. In our plug-in, the platform selection for a MVC application is done by selecting appropriate techniques for the ≪`model`≫, ≪`view`≫ and ≪`controller`≫ parts of the target system from a drop-down menu as shown in **Fig. 4**. In addition, if the user already has a platform decision model available that holds the information about the selected techniques, this model can be used as an input to derive the platform selection and preselect the corresponding techniques in the drop-down menu. Thus, the user just has to check, if these values are according to the requirements.

Once the platform configuration is determined, the PIM-to-PSM transformation can be triggered, which is described in the following sections.

3.2 Assuring Model Integrity

As a side-effect of the automatic transformation of a PIM into a PSM the ACCURATE plug-in also offers some functionality to assure the model integrity by fixing common modeling flaws. These flaws would result in errors during the code generation process and thus cause time-consuming bug-fixing of the PIM, which can be avoided partially through this automation. The concrete model

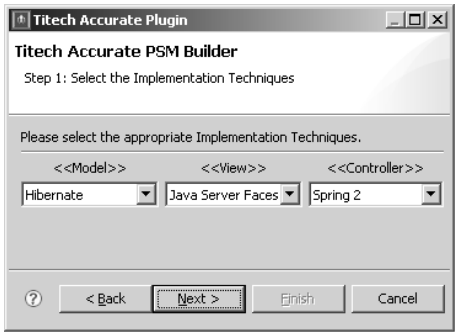


Fig. 4 MVC selection dialog.

fixes that are done during the PIM-to-PSM transformation are:

- Assuring the correct structure of the PIM
- Adding missing ACCURATE profile
- Removing whitespaces in element names
- Adding missing return types of operations
- Adding names for unnamed links
- Removing use-case elements

It has to be mentioned that some of these model fixes are applied before and some after the PIM-to-PSM transformation.

It is mandatory to guarantee a consistent PIM structure before the PIM-to-PSM transformation starts. Thus, the structure of the PIM is checked initially and corrected if necessary. A consistent PIM structure consist at least of a `XMI` root element, a `Model` element as its child and a `Package` element as a child of the `Model` element. Since there is no existing specification on how a PIM should be structured the usage of different UML modeling tools may result in different structures for the same software system. We use the above structure as a default PIM structure since it is well supported by common code generation frameworks. Besides this, modeling errors may also result in a divergent PIM structure. To approach this problem, the plug-in checks the actual structure of the PIM and adds the missing structural elements with default names to the PIM. Furthermore, a PIM using the ACCURATE approach needs to have the ACCURATE profile attached to the namespace of the model XMI file. Because a missing profile would result in errors during the PIM-to-PSM transformation, the plug-in adds the ACCURATE profile to the model file if necessary. Since these transformations don't add any platform specific information to the model, we regard these modifications as PIM-to-PIM transformations.

After the PIM has been successfully transformed into a PSM, it is again checked for its consistency to assure a more accurate and flawless PSI generation based on our generated PSMs. At this point, common design flaws such as whitespaces in model element names, unnamed links and missing return types of operations are corrected via PSM-to-PSM transformations. As before, the plug-in checks using JDOM if one of the above mentioned design flaws exists in the PSM and corrects the flaw if necessary. In detail, whitespaces in names are removed directly from the elements name. In addition, unnamed links get a default and unique name assigned and if an operation has no return type specified, the return type is set to void. As a last step, all remaining elements that are not necessary for the PSI generation are removed from the PSM. This affects all fragments from the early requirements and analysis phase of the software development process such as use-case elements. In this example, any use-case, actor or reference to a use-case is excluded from the PSM before the resulting PSM is finally saved.

3.3 Transformation of stereotyped elements

As stated out before, the mappings of the PIM and PSM elements vary a lot and thus the PIM-to-PSM transformation needs to be regarded as nontrivial. The complete PIM-to-PSM transformation has to treat all ≪`model`≫, ≪`view`≫, ≪`controller`≫ and ≪`external`≫ elements of the PIM and transform them into platform specific elements. For the ≪`model`≫, ≪`view`≫ and ≪`controller`≫ elements this means to:

(1) Find all according elements
(2) Check the platform selection
(3) Transform these elements and their attributes, operations and references according to the platform selection

The overall workflow of the transformation of ≪`model`≫ elements is illustrated in **Fig. 5**. First of all, the model file is analyzed and searched for any elements that are associated with the stereotype ≪`model`≫. To apply the correct mapping to these elements to a platform specific technique, furthermore the platform

selection has to be looked up in order to trigger the correct transformation of the elements. As to be seen in the figure, the ACCURATE plug-in supports three different techniques for ≪`model`≫ element. These are Hibernate, Plain Old Java Objects (POJOs) and Enterprise Java Beans. Depending on the platform selection, the corresponding UML profile is applied to the PSM. Afterwards, each ≪`model`≫ elements' stereotype is transformed into the proper platform specific one. For example, any ≪`model`≫ element is transformed to the stereotype ≪`Entity`≫ if the selected platform is Hibernate. In addition, the attributes, operations and references of these elements sometimes also need to be attached to platform specific stereotypes. Since the PIM notation is intentionally kept simple, it is at some point difficult to obtain a clear and unambiguous mapping of these PIM elements. Especially, if PSM stereotypes are bound to some functional and behavioral characteristics of the stereotyped elements that cannot be retrieved from the static structure of a system only, mappings to such stereotypes are hard to handle. For example, from an abstract view of a class in a PIM one may have problems to distinguish if some attributes of a database class should be mapped to the property stereotypes ≪`Index`≫ or ≪`Key`≫, which are commonly used to define the structure of database tables more precisely in a PSM. We address this problem by simplifying the mapping of such attribute stereotypes. Since the most essential stereotypes can be derived unambiguous from the PIM, the remaining stereotypes are either considered as optional or derived by the naming of the attributes. For example, if a class attribute is named `key` we map the element to the stereotype ≪`Key`≫ or if an attribute is named `index` it will be mapped to the stereotype ≪`Index`≫.

We would like to mention at this point that since the workflow of the complete ACCURATE process not necessarily has to executed consecutively, one may want to stop the process after the PIM-to-PSM transformation and manually refine the PSM. This way the user has total freedom of design, while still sparing a lot of time and work in the PSM design process, since most of the PIM-to-PSM transformations are already achieve automatically. Even so, the integrity of the modified PSMs can be assured through model checks before the PSM-to-PSI transformation takes place.

For the Hibernate example, the last step during the model transformation

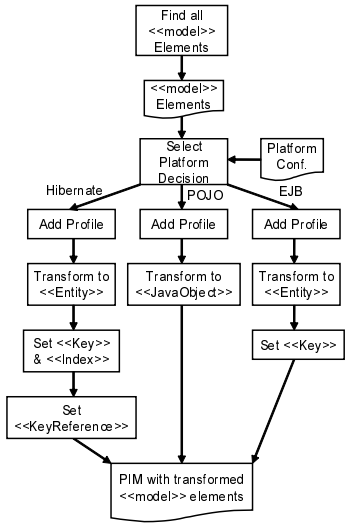


Fig. 5 Transformation of ≪`model`≫ elements.

is to set the stereotype ≪`KeyReference`≫ for all links between two ≪`model`≫ elements. Since UML offers different options of how to model links between model elements, this task can be complicated. For example, **Fig. 6** shows a possible example of the two ≪`model`≫ elements `Entry` and `AddressBook`, which are each associated to a stereotype ≪`Entity`≫ through the `xmi:id` of the two classes. In this example, both classes own an attribute holding the information about the association between them. The association is again stored by the association's `xmi:id` within these owned attributes. Besides this, it is also a possible way to store the reference to the classes directly in the association as owned ends (as shown in **Fig. 7**). One should notice that from the viewpoint of a class the linked class and its stereotype cannot be received directly. To address this problem, one has to start the traversal of the elements at the association itself. Using the JDOM API, it is possible to find UML associations quickly and unambiguously using the build-in filter functionality. Furthermore, JDOM offers support for traversing the parent and child elements of a given element. This way, it is in both examples possible to reach the two referenced elements of an association easily

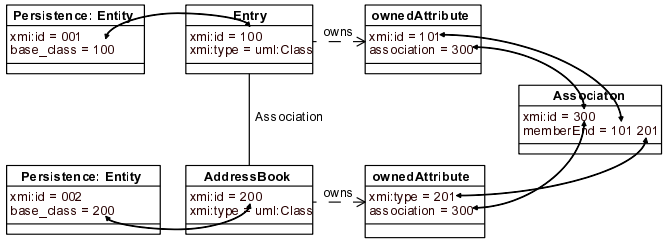


Fig. 6 Structure of a nested Association.

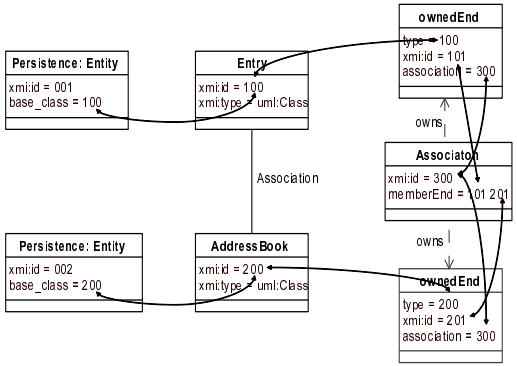


Fig. 7 Structure of a separated Association.

and analyze their stereotypes. As a result, one can find ≪model≫-to-≪model≫ links and attach the desired stereotype ≪KeyReference≫ to the association.

The transformation for POJO or Enterprise Java Beans as technique for the ≪model≫ classes is straightforward. Since these platforms use different stereotypes for their elements the corresponding ≪model≫ elements need to be either set to ≪JavaObject≫ in the case of POJO or to ≪Entity≫ for EJB. In the case of Enterprise Java Beans, attributes of ≪Entity≫ classes that are named key also get the stereotype ≪Key≫ assigned.

The result after these transformations is a PSM that contains all mapped ≪model≫ elements but still owns PIM elements for the ≪view≫, ≪controller≫ and ≪external≫ classes that need to be transformed in the following steps.

In the next step of the PIM-to-PSM transformation all ≪view≫ classes need to be transformed to the according platform specific stereotypes (see **Fig. 8**). As for the ≪view≫ elements, first all ≪view≫ elements need to be found within the model. Corresponding to the platform selection the UML profile for Java Server Faces or a standard Java UI is attached to the PSM. In the case of JSF, all ≪view≫ classes need to be transformed into ≪ManagedBean≫ classes. Furthermore, the JSF profile we make use of is using UML dependencies instead of associations to connect the ≪view≫ elements to ≪model≫ and ≪controller≫ elements. Thus, any association needs to be changed to a dependency. If the platform selection is Java UI then it is necessary to additionally check the technique used for the ≪controller≫ components. Depending on the result, the ≪view≫ elements are either transformed to ≪SpringService≫ elements if the platform selection is Spring or otherwise transformed to ≪JavaObject≫. Again, if Spring is used as the ≪controller≫ technique all links connecting the ≪view≫ class to other classes need to be set to dependency. After all of the ≪view≫ elements have been transformed the resulting PSM contains all mapped ≪model≫ and ≪view≫ elements.

The last MVC elements that need to be transformed during the PIM-to-PSM transformation are the ≪controller≫ elements. As to be seen in **Fig. 9**, the workflow for transforming ≪controller≫ elements is straightforward. As before, all ≪controller≫ elements are initially selected. Again, depending on the platform selection, the corresponding UML profile is attached to the PSM and the ≪controller≫ elements transformed into platform specific elements. These are ≪Service≫ for Spring, ≪JavaObject≫ for POJO and ≪SessionBean≫ for Enterprise Java Beans. In the case of Spring as the ≪controller≫ technique, two additional transformations need to be triggered. First, all operations of ≪controller≫ classes need to be mapped to ≪ServiceOperation≫. Finally, all links that connect ≪controller≫ classes to other elements have again to be changed a UML dependency. As the result, all MVC elements are transformed into a platform specific representation that conforms to the platform selection.

Since the resulting PSM still contains all ≪external≫ classes, which are not necessary in a PSM, the final task during the PIM-to-PSM transformation has

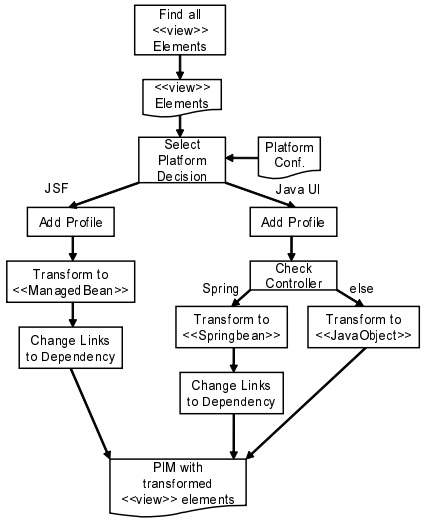


Fig. 8 Transformation of ≪`view`≫ elements.

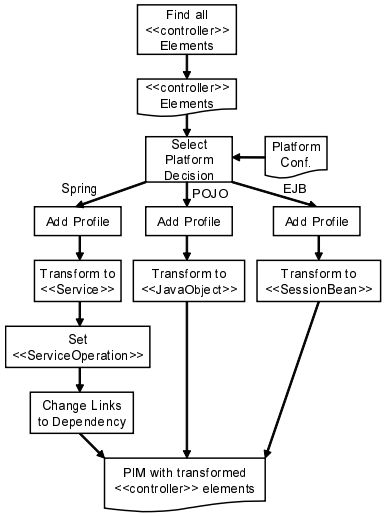


Fig. 9 Transformation of ≪`controller`≫ elements.

to be the removal of all ≪`external`≫ elements from the PSM. Since in this case no mapping to a platform specific technique is necessary, first all ≪`external`≫ elements need to be selected from the PSM, then all references to these elements are removed and finally the ≪`external`≫ elements themselves are excluded from the PSM.

After all steps of the PIM-to-PSM transformation process the result is a PSM that conforms to the platform selection and furthermore only contains elements that are relevant for a possible PSI generation on the basis of the PSM. It should be noticed that through adopting JDOM together with a plug-in environment it is possible to structure the overall PIM-to-PSM transformation in a modular and sequential way. Besides this, the transformation can be extended easily by adding branches for additional techniques to any of the transformations of the stereotyped elements or add further model fixes at the beginning or end of the PIM-to-PSM transformation. Thus, using XMI manipulation techniques for the model transformation offers both an easy to use way of designing the transformations through a matured and well-design API and flexible and extendable mechanism to structure the complete PIM-to-PSM transformation process.

4. Related Work

There is already some existing work focusing on platform independent modeling and model transformation in a different problem domain. Bezivin et al. propose to use ATL transformation[10] to transform PIMs defined by Enterprise Distributed Object Computing into PSMs for different web service platforms[9]. Even though this work is focusing on web platforms, the technique for defining the model transformation is different from our approach. Furthermore, the proposed workflow is not focusing on a jointly combination of requirements techniques for the platform selection as in our proposed approach. Billig et al.[11] define PIM-to-PSM transformations in the context of EJB by using QVT[12] and thus also use a different technique for defining model transformations. Since this approach only focuses on Enterprise Java Beans it is also technology specific and would cause some work to adapt to a new combination of platform alternatives.

Besides this, some other related work also defines PIMs via UML profiles. Link et al. propose to use the GUIProfile to model PIMs and transform them into PSMs[13]. Richly et al. focus on a UML profile to define PIMs for databases[14]. He et al. use template role model together with PIM profiles for templates to design PIMs, which are specific for web applications[15]. Ayed et al. propose a UML profile for modeling platform independent context-aware applications[16]. Lopez-Sanz et al. define a UML profile for service-oriented architectures[17]. Finally Fink et al. combine UML and MOF profiles for access control specifications[18]. Thus, there are a lot of approaches, which describe a PIM on a more abstract level than a PSM. Even so, these approaches are still tailored to a specific technology or architecture and thus need some detailed knowledge of the concrete problem domain. Furthermore, the adoption to a different problem domain or architecture such as MVC is hindered due to the specific notations of these PIMs.

5. Conclusion and Future Work

In this paper, we have discussed the actual situation of software development and stated out some clear problems when trying to adopt MDE techniques to projects that have to handle a great variety of target platforms. To address

this problem, we have introduced an approach called ACCURATE to handle the lack of support for changing platforms, in respond to requirement changes or platform evolutions. Our approach shows how to specify systems easily without any PSM modeling skills. Furthermore, the approach offers much automation of the development process and thus reducing costs due to a shorter time-to-market.

Furthermore, we described detailed a possible implementation of a PIM-to-PSM transformation according to the ACCURATE approach. The prototype tool we provided, both assures the integrity during the model transformation and offer guidance through the software development process to the user. The current implementation of the tool provides a workable and extendable solution to address the stated problems. However, there are still some enhancements that we would like to adopt to our approach in the near future. These possible extensions can be summarized as follows:

(1) **Evaluation:** This paper focused on applying the ACCURATE approach to the MVC architecture style. Since this is just one possible example for an architecture style, we would like to evaluate our approach through a case study and validate the usability of our approach more sustained adopting different architecture styles.

(2) **Platform decision models:** As stated out before, the platform selection could be derived automatically from a suitable input model. In the future, we are going to introduce decision models such as Bayesian networks[19] more precisely and use them as possible standard inputs for our plug-in.

(3) **PSM-to-PSI generation:** In future work, we would like to focus more detailed on how to implement existing code generation frameworks into the ACCURATE approach and toolkit. This way, we are also going to argue more on the technical differences between existing model transformation techniques.

References

1) Schmidt, D. C.: Guest Editor's Introduction: Model-Driven Engineering, *Computer*, Vol.39, No.2, pp.25–31 (2006).
2) openArchitectureWare.org: Official openArchitectureWare Homepage, `http://www.openarchitectureware.org/`.
3) AndroMDA.org: AndroMDA.org - Home, `http://www.andromda.org/`.
4) Reenskaug, T.: The original MVC reports, `http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf` (2007).
5) OMG: MOF 2.0/XMI Mapping, Version 2.1.1, `http://www.omg.org/docs/formal/07-12-01.pdf` (2007).
6) jdom.org: JDOM, `http://www.jdom.org/`.
7) Efftinge, S., Voelter, M.: oAW xText: A framework for textual DSLs, `http://www.voelter.de/data/workshops/EfftingeVoelterEclipseSummit.pdf` (2006).
8) No Magic: UML 2 diagramming, OO software modeling, Source code engineering Tool MagicDraw UML from No Magic, `http://www.magicdraw.com/`.
9) Bezivin, J., Hammoudi, S., Lopes, D., Jouault, F.: Applying MDA approach for Web service platform, *EDOC '04: Proceedings of the 8th International Conference on Enterprise Distributed Object Computing*, pp.58–70 (2004).
10) eclipse.org: ATL Project, `http://www.eclipse.org/m2m/atl/`.
11) Billig, A., Busse, S., Leicher, A. and Süss, J. G.: Platform Independent Model Transformation Based on TRIPLE, *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pp.493–511 (2004).
12) OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0, `http://www.omg.org/docs/formal/08-04-03.pdf` (2008).
13) Link, S., Schuster, T., Hoyer, P., Abeck, S.: Focusing Graphical User Interfaces in Model-Driven Software Development, *ACHI '08: Proceedings of the 1st International Conference on Advances in Computer-Human Interaction*, pp.3–8 (2008).
14) Richly, S., Habich, D., Lehner, W.: GignoMDA - Generation of Complex Database Applications, `http://dbs.informatik.uni-halle.de/GvD2006/gvd06_habich.pdf` (2006).
15) He, C., He, F., He, K., Tu, W.: Constructing Platform Independent Models of Web Application, *SOSE '05: Proceedings of the IEEE International Workshop on Service-Oriented System Engineering*, pp.85–92 (2005).
16) Ayed, D., Berbers, Y.: UML Profile for the Design of a Platform-Independent Context-Aware Applications, *MODDM '06: Proceedings of the 1st Workshop on MOdel Driven Development for Middleware*, pp.1–5 (2006).
17) López-Sanz, M., Acuña, C., Cuesta, C., Marcos, E.: UML Profile for the Platform Independent Modelling of Service-Oriented Architectures, *Software Architecture*, Vol.4758, pp.304–307 (2007).
18) Fink, T., Koch, M., Pauls, K.: An MDA approach to Access Control Specifications Using MOF and UML Profiles, *Electronic Notes in Theoretical Computer Science*, Vol.142, pp.161–179 (2006).
19) Pearl, J.: Bayesian Networks: a Model of Self-Activated Memory for Evidential Reasoning, *Proceedings of the 7th Conference of the Cognitive Science Society*, pp. 329–334 (1985).