Reusing Platform-specific Models in Model-Driven Architecture for Software Product Lines

Frédéric Verdier[1,2], Abdelhak-Djamel Seriai[1] and Raoul Taffo Tiam[2]
[1]*LIRMM, University of Montpellier / CNRS, 161 rue Ada, 34095, Montpellier, France*
[2]*Acelys Informatique, Business Plaza bât. 3 - 159 rue de Thor, 34000, Montpellier, France*

Keywords: Reuse, Model-Driven Architecture, Software Product Line, Variability, Platform-specific Model.

Abstract: One of the main concerns of software engineering is the automation of reuse in order to produce high quality applications in a faster and cheaper manner. Model-Driven Software Product Line Engineering is an approach providing solutions to systematically and automatically reuse generic assets in software development. More specifically, some solutions improve the product line core assets reusability by designing them according to the Model-Driven Architecture approach. However, existing approaches provide limited reuse for platform-specific assets. In fact, platform-specific variability is either ignored or only partially managed. These issues interfere with gains in productivity provided by reuse.
In this paper, we first provide a better understanding of platform-specific variability by identifying variation points in different aspects of a software based on the well-known "4+1" view model categorization. Then we propose to fully manage platform-specific variability by building the Platform-Specific-Model using two sub-models: the Cross-Cutting Model, which is obtained by transformation of the Platform-Independent Model, and the Application Structure Model, which is obtained by reuse of variable platform-specific assets. The approach has been experimented on two concrete applications. The obtained results confirm that our approach significantly improves the productivity of a product line.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Software development industry has been evolving from hand-craft to semi-automatic processes to improve productivity and quality. An approach to achieve this goal is reuse (Jacobson et al., 1997).

Model-Driven Engineering (MDE) (Schmidt, 2006) permits to design more reusable assets than source code sections by improving their genericity. MDE consists in designing an application with models which are abstractions of a system. Models are managed using model transformation operations to target another abstraction level producing another model or source code.

More specifically, Model-Driven Architecture (MDA) is an approach based on MDE. It specifies a separation of concerns for assets based on 3 abstraction levels which starts from the representation of the client's needs in the Computation-Independent Model (CIM). The CIM is transformed into the representation of the platform-independent software conception in the Platform-Independent Model (PIM). Then the PIM is refined into a Platform-Specific Model (PSM) which includes platform specificities. Finally, the PSM is transformed into the product source code.

While MDE and MDA permit to design highly reusable assets, Software Product Line Engineering (SPLE) permits to systematically identify, organize then select and integrate assets of any nature in new applications depending on the client's needs. A Software Product Line is a "set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (Clements and Northrop, 2001). SPLE defines two parallel development processes (Pohl et al., 2005). The domain engineering phase focuses on the identification and design of reusable core assets which are organized according to the commonalities and variabilities of the product line applications. In the application engineering phase, reusable assets are derived to produce a new application fitting the client's needs.

Some approaches (Deelstra et al., 2003; Kim et al., 2005) propose to combine MDA and SPLE to capitalize on advantages of each. To do so, they con-

Verdier, F., Seriai, A-D. and Tiam, R.
Reusing Platform-specific Models in Model-Driven Architecture for Software Product Lines.
DOI: 10.5220/0006582601060116
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 106-116
ISBN: 978-989-758-283-7
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

sider MDA models as the core assets of a product line.

Nevertheless, those approaches focus primarily on reuse of CIM and PIM assets, not on PSM assets which are obtained through transformation of the PIM. However, adding platform-specific implementation variants increases the complexity of transformation operations. That is why the majority of approaches either ignore or only partially manage platform-specific variability. In this case, the produced source code can contain unwanted implementation patterns caused by ignored variants and must be modified manually. Those issues interfere with gains in productivity provided by reuse (in terms of cost reduction, quality, etc.).

We propose to improve productivity by fully managing platform-specific variability. We believe that transformation operations cannot efficiently manage variability. That is why we propose to build the PSM as a combination of two sub-models: one obtained by transformation of the PIM; and the other one obtained by reuse of variable platform-specific assets defined in the domain engineering. To do so, we first identify how platform-specific variability impacts the software implementation. Then we propose to structure the PSM to distinguish assets obtained through transformation of the PIM and reused assets. Platform-specific reusable assets are organized according to platform-specific variability. Finally, we conducted experiments on two concrete applications to validate the approach.

The remaining of the paper is structured as follows. Firstly, impacts of platform-specific variability on software are described in Section 2. Then, the proposed structure of the PSM is explained in Section 3. Section 4 details how our solution has been validated. A state of the art is presented in Section 5. Lastly, we conclude in Section 6.

2 PLATFORM-SPECIFIC VARIABILITY

Platform-specific variability consists of variation points that appear only when a specific platform or technology is selected. Each variation point represents a decision impacting the source code related to the use of the selected platforms.

Platform-specific variability consists of a large amount of different variation points in numerous parts of a software.

2.1 Identifying Platform-specific Variability

To describe the impacts of platform-specific variability on an application implementation, we identified platform-specific variation points in different parts of a system. Those parts are based on the ”4+1” view model (Kruchten, 1995). This model is composed of 5 views defining the different parts of a system: the logical view, the process view, the physical view, the development view and the scenario view.

In the following, we focus on views in which we identified platform-specific variation points:

Platform-specific Variability in the Process View. The process view describes how the main blocks of system functionalities interact with each other. This view captures concurrency, inter-process communication, distribution of blocks, etc.

One of the elements the process view can describe is the messaging system between the software and some distant system.

A messaging system design can vary entirely from a framework to another. For example, a system implementation using RabbitMq describes the map of the message exchanges while a system implementation using Kafka describes the structure of messages without managing how those messages are sent. For each platform, the different possible configurations can be reused and organized with a variation point specific to this platform.

Platform-specific Variability in the Physical View. The physical view describes the infrastructure of the system and how it is deployed.

Then, the physical view can describe the hardware peripherals the software interacts with. For example, mobile applications often interact with different captors which can be specific to the device they are deployed on and the operating system. The heterogeneity of hardware devices in mobile applications is identified in (Usman et al., 2017). But this variety of devices is different in each operating system. Consequently, this variation point cannot be included in the PIM without adding platform specificities.

Platform-specific Variability in the Development View. The development view describes how functional blocks are implemented (using layer, component or class diagrams for example).

One of the elements the development view can describe is the persisted data structure in the system using a database management system.

Some database management systems like MySQL in Figure 1 can encode generic primitive types (like *String*) with different concrete types (*VARCHAR* and *TEXT*). Those types differ in their length to fit different use cases. The available types differ from a technology to another one. In fact, MongoDB provides only one concrete type to encode string fields. Consequently, the PIM cannot be responsible to solve those variation points without including platform specificities in the model. Then, by selecting MySQL, a new platform-specific variation point is included for each field corresponding to the variety of available types.

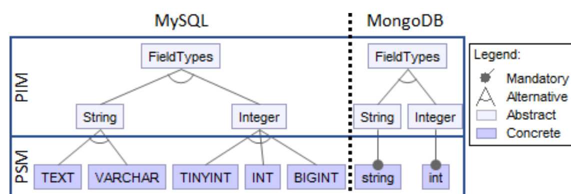

Figure 1: Variability of types for a field depending on the database management system used.

We identified platform-specific variability in three aspects of the software design. It is now possible to understand what are the impacts of platform-specific variability on reuse.

2.2 Impacts of Platform-specific Variability on Reuse

In MDA and SPLE combinations, platform-specific variability influence on reuse is not as visible as impacts of variability in higher abstraction levels.

In fact, if platform-specific variability is ignored, only one platform-specific asset variant is always selected. In our example, ignoring the *String* variation point for MySQL implies that only one concrete type is always selected such as *TEXT*. Then, the code must be manually modified to use the right concrete type for each field.

However, according to (Brambilla et al., 2017), it is preferable to produce source code from only one source of information to be able to identify how a source code section is produced (through code generation or manual development). In this way, it is possible to produce the application source code incrementally (e.g. after multiple iterations of code generation). Moreover, modifying generated source code can be expensive. In fact, developers must manually identify where corrections are required. Some modifications can have large impacts on the application implementation. In our example, changing the concrete type of a field modifies the database structure implementation as well as functional data validation rules implementation. Impacts can also vary depending on how MySQL is combined with other technologies. For example, if an Object Relational Mapping is used like the Hibernate framework (O'Neil, 2008), the mapping logic is also impacted.

Similarly, if platform-specific variability is partially managed, some platform-specific asset variants are ignored. Then, it leads to the same consequences previously mentioned.

Existing approaches rely on PIM to PSM and PSM to text transformation operations to realize platform-specific variation points. They do not manage platform-specific variability efficiently (refer to Section 5 for a description of related works) and thus, productivity gains provided by reuse are reduced. We propose to cater the platform-specific variability management problem by reusing platform-specific assets designed at the PSM abstraction level.

3 REUSE AT THE PLATFORM-SPECIFIC MODEL (PSM) ABSTRACTION LEVEL

In order to efficiently reuse platform-specific implementation patterns, we propose a way to capitalize on SPLE techniques to manage platform-specific variability. Assets defined at the PSM abstraction level are considered as variable core assets similarly to assets defined in higher abstraction levels.

To do so, the PSM is first structured to distinguish its elements obtained by transformation of the PIM from those obtained by derivation of platform-specific core assets. Then, two different mechanisms are used to represent platform-specific variability. Finally, assets are composed to obtain the application engineering PSM.

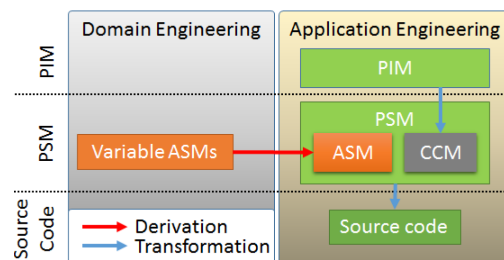

Figure 2: Content of the PSM and how it is obtained.

As depicted in Figure 2, the content of an application engineering PSM is composed of more fine-grained models: the Cross-Cutting Models (CCM), which are resulting of the PIM transformation, and the Application Structure Models (ASM), which are

obtained by deriving reusable models of the same nature.

3.1 Cross-Cutting Model (CCM)

The CCM is obtained by transformation of the PIM. Consequently, CCMs do not contain reusable assets and so, are not produced in the domain engineering.

A CCM describes the application realization including the business domain, functionalities, graphical user interfaces as well as cross-domain features such as security management protocols.

Section 2 showed that PIM to PSM transformation operations could not manage platform-specific variability efficiently. That is why they must not define any platform-specific reusable asset. This implies that those operations are responsible to translate PIM information to the PSM formalism but must not add information that is not described in the PIM. In this way, generic concepts in the PIM are translated to platform-specific concepts.

For example, a string field in a PIM class diagram is transformed into a class diagram field in PSM typed as *String* if Java is used or *string* if C# is selected. However, PIM to PSM transformation operations cannot add new information like the database architecture.

Therefore, a CCM is similar to the PIM it comes from. Its formalism can change in different development contexts. In fact, different companies can use different diagrams to model their softwares. For example, the CCM can be composed of different UML diagrams such as component, class and sequence diagrams. An excerpt of CCM using a class diagram is depicted in Figure 3.

Similarly to the PIM, the CCM cross-cuts the source code. Thus, it is necessary to bridge the gap between source code and Cross-Cutting Models. This bridge is realized by ASMs.

3.2 Application Structure Model (ASM)

ASMs are platform-specific reusable models designed in the domain engineering as variable core assets. They are reused to design the application PSM.

An ASM represents an application physical structure to generate. Variable ASMs are reusable physical structure patterns. A pattern describes the use of a platform, for defined concerns, as a tree of implementation structure elements (such as files, folders, libraries and distant systems) and sub-patterns.

Therefore, ASMs are specific to platforms. However, they are independent from business domains. Consequently, an ASM can be reused in several product lines to produce applications using the described set of technologies.

ASM elements define the concrete implementation by referencing the PSM to text transformation operations to use for each file to generate. In this way, those operations can have well defined and separated responsibilities:

- An operation produces the implementation of using a specific platform for a defined concern.
- An operation produces local implementation sections such as the content of a method, a single file or a set of similar files.
- An operation can use intermediate PSM to PSM refinement operations if required. Those operations refine the PSM with the addition of new information (for example, the description of the database architecture).

Then, ASMs promote the use of template-based approaches (Czarnecki and Helsen, 2003) to realize PSM to text transformations. In fact, a template is responsible to describe the code to produce for a source code section such as a method or a file.

Domain engineering ASMs can describe how any platform is used with its generic formalism. In fact, an application implementation is always structured using elements such as files and folders.

An application engineering ASM represents the software physical structure to generate for an application. It is built as a composition of derived domain engineering ASMs. It reuses and modify the physical structure of the product line applications and, therefore, its architectural components that are related to specific platforms.

For example, an excerpt of an absence demand scheduler software PSM is depicted in Figure 3. Elements with the stereotype *Pattern* are ASMs. The ASM named *MySQLDatabase* describes how a MySQL database is implemented. It defines a subtree composed of two sub-patterns (*DomainDefinition* and *MySQLDatabase*). In this way, if the content of a sub-pattern evolves, the content of *MySQLDatabase* is also impacted. Elements with the stereotype *File* represent a file to generate. The element named *DomainClass* references a generation template. The latter produces a file implementing a C# class for each class defined in the CCMs.

Domain engineering Application Structure Models are variable assets. The following section describes how the ASM variability is represented.

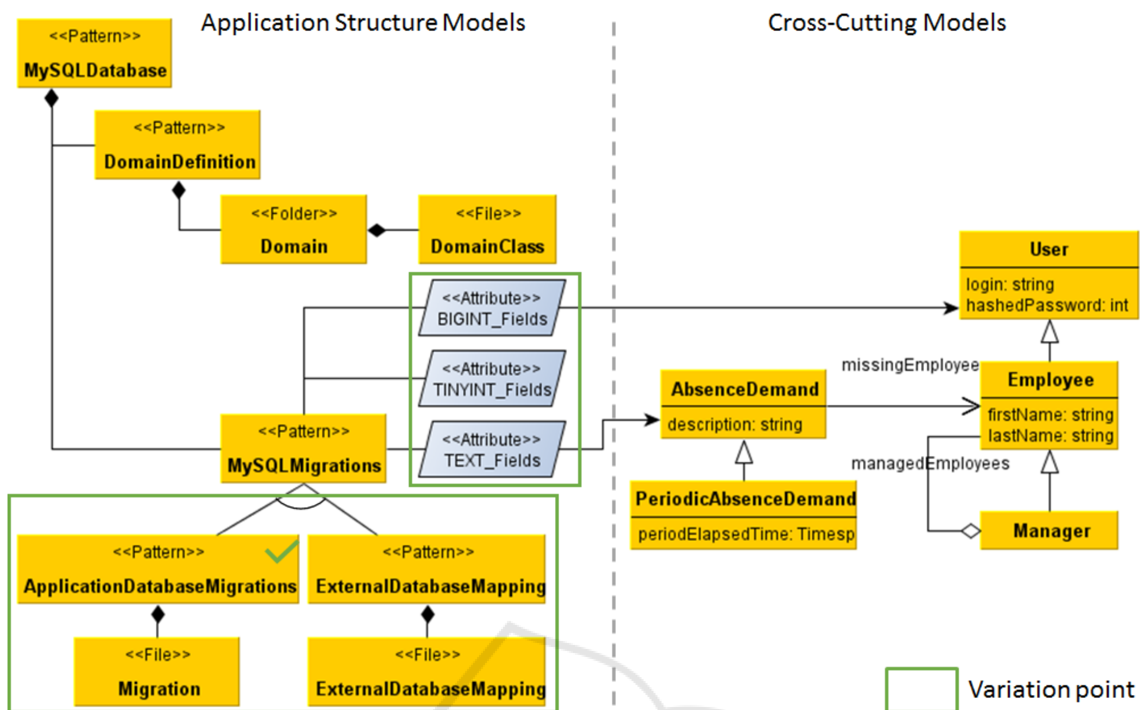

Figure 3: PSM of the absence demand scheduler.

3.3 Realizing Variability in the PSM Abstraction Level

Representing the variability in the PSM abstraction level lets developers choose which platform-specific reusable solution to integrate among several variants. In this way, it is possible to reuse more efficiently a wider range of platform-specific assets by using variability management to organize them.

In the PSM abstraction level, only ASMs are reusable variable assets. Then, variability can be represented only on them.

ASMs variation points are platform-specific decisions performed by developers. Platform-specific variation points allow to reuse alternative file structures and configure the PSM to text transformation operations they reference.

We distinguish two mechanisms to realize variability in models:

Asset Variants. Those variation points provide choices between different variants which we commonly see in SPLE approaches.

In Figure 3, applications can use a specific MySQL database or use an already existing database. Both cases are handled with specific variants (*ApplicationDatabaseMigrations* and *ExternalDatabaseMapping* assets) of the ASM *MySQLMigration*.

Model Attributes. Models are configurable as advised in (Deelstra et al., 2003). In this way, their genericity is improved. Those variation points can have an uncountable number of solutions unlike asset variants.

Figure 3 provides an example of model attribute related to the variability of MySQL types described in Section 2.1. A default variant is chosen for each type. In this way, string fields are by default typed by *VARCHAR* and integer fields are typed by *INT*. However, *VARCHAR* have a limited length that does not fit to long string like descriptions. That is why the *description* field of *AbsenceDemand* is typed as *TEXT* which has an unlimited length.

An application engineering PSM is a composition of CCMs obtained by transformation of the PIM with derived domain engineering ASMs. The following section describes how this composition is realized.

3.4 Composing CCMs and ASMs

The application PSM is obtained by composing the CCM (obtained by transformation of the PIM) with reused ASMs (obtained by derivation of domain engineering ASMs). CCMs and ASMs are loosely coupled to simplify the PSM construction. This is possible for two reasons.

On the one hand, CCMs cross-cut the source code.

They do not rely on specific source code areas. Thus, their definition is independent from ASMs which define the source code structure.

On the other hand, when an ASM depends on the application engineering CCM content, the domain engineering ASM declares model attributes to define those dependencies. Then, in application engineering, solving the ASM dependencies consists in solving its model attributes. Although this resolution is manual, dependencies are organized and identified with specific variation points.

An example of CCM and ASM composition is depicted in Figure 3. Only the resolution of model attributes is creating relations between ASM and CCM elements. This resolution is realized by the application engineering development team.

We defined the proposed loosely coupled separation of concerns of the application engineering PSM and how each sub-model is obtained. The following describes how the approach has been validated.

4 VALIDATION OF THE PROPOSAL

In order to measure the efficiency of the proposal, we conducted experiments on two concrete applications to answer the following questions:

Q1 Does the proposal permit to reuse variable platform-specific assets?

- **Q1.1:** Can the proposal model variable platform-specific reusable assets?
- **Q1.2:** Can the proposal integrate platform-specific reusable assets in new applications?

Q2 Does the proposal improve productivity?

- **Q2.1:** Does the proposal improve the management of platform-specific variability?
- **Q2.2:** Does the proposal reduce development efforts for new applications?

4.1 Experiment Protocol

How to Answer Q1. A domain engineering phase has been performed to identify reusable platform-specific assets in the server part of several client-server applications from diverse product lines using a similar set of technologies. In this way, we ensure that identified assets are platform-specific and independent from business domains.

Then, the development team of two small client-server applications (described in table 1) modeled their application engineering PSM with our help. Those applications are located in different product lines provided by our industrial partner that use a similar set of technologies.

The first application is a client-server absence demand management application already used in production. This software is partially described in the previous sections to provide simplified examples.

The second application is a client-server mobility advisor application. This application provides tools to manage users' travels with routing optimization advices. The experiment was realized alongside its manual development.

How to Answer Q2. We developed a tool which helps the development team to design an application engineering PSM that reuses domain engineering ASMs and creates CCMs. In this way, we implemented the composition of reused ASMs with CCMs and we also observed how the tool was used to create the PSM of both applications by the development team. Moreover, we developed PSM to text transformation operations corresponding to the identified platform-specific assets. Those operations were realized as templates for a template engine used by our industrial partner.

The PSM of both applications have been transformed into source code using the developed PSM to text transformation operations. The obtained source code has been compared to their hand-crafted version. With this comparison, we could analyze the ratio of generated source code that we could obtain and evaluate the ability of the proposal to handle platform-specific variation points. In fact, when a variant is ignored or inefficiently managed, the produced implementation is different from the expected one. To compare our solution results with related works, we looked for generic approaches that could use similar PSM to text transformation operations. As far as our knowledge, only (Lahiani and Bennouar, 2014) could fit to our experimentation context. We estimated the amount of required manual modifications in the source code.

We estimated the amount of gained time on the applications realization obtained by using the proposal compared to the time needed to implement them manually. We also estimated the amount of gained time obtained by using (Lahiani and Bennouar, 2014). These estimations were compared to the real cost of the application overall project including all development phases (**Project cost** in Table 1). Knowing that our estimations concerned the gained time obtained in the realization phase (thanks to code generation), we also compared our results to the real server realization phase costs (**Server realization cost** in Table 1).

Table 1: Statistics of 2 case studies applications.

Application	Lines of code	Nb. PSM variation points		Nb. functionalities	Project cost[1]	Server realization cost[1]
		Variants	Model attributes			
Absence Manager	9741	5	14	81	130 M/D	40 M/D
Mobility Advisor	2820	3	14	31	98 M/D	32.5 M/D

4.2 Obtained Results

Obtained results are described in Table 2. We use the following metrics to evaluate our results:

- Correctly generated code: ratio of the application source code successfully generated for the server part.
- Generated code requiring corrections: ratio of the application generated source code that required additional manual modifications for the server part.
- Gained time (realization phase): estimation of time that could be gained with the code generation.

Using these results, it is now possible to answer to the previously mentioned questions.

Answering Q1. During the domain engineering phase, we identified ASMs from applications related to different product lines with similar technologies. They could be organized following their commonalities and variabilities. We identified variation points related to each aspect of the software mentioned in Section 3.3. The obtained product line realized the platform-specific variability using the two mechanisms described in Section 3.3. In Table 1, **Nb. PSM variation points** represents the count of variation points solved to produce each application (*Q1.1*).

Results show that with a single domain engineering phase, we were able to generate a significant amount of source code correctly by reusing numerous variable ASMs identified in the domain engineering phase. Experimented cases are related to different business domains. Therefore, those assets are reusable in different product lines (*Q1.2*).

Thus, the proposal permits to reuse variable cross-domain platform-specific assets.

Answering Q2. The application engineering PSM of each experiment case has been successfully realized by the development team.

All the produced source code was effectively used thanks to the ability, provided by ASMs, to select which fine-grained PSM to text transformation operations must be used.

[1]Estimated costs measured in Man/Day (M/D).

Moreover, less additional manual corrections were needed in the source code produced by our proposal comparatively to source code produced using (Lahiani and Bennouar, 2014). In fact, the latter could not generate platform-specific implementation variants which had large impacts on the source code. Therefore, the proposal manages more efficiently platform-specific variability than (Lahiani and Bennouar, 2014) does (*Q2.1*).

The estimated gained time difference between existing approaches and the proposed one is explained by the required manual correction ratio difference between the experimented solutions. In fact, considering the transformation operations used, the generated source code proportion is the same. But additional manual efforts are required in existing solutions for code sections that have a large impact on the application implementation. Thus, using (Lahiani and Bennouar, 2014), it is sometimes preferable to not generate source code sections that can vary depending on a platform-specific variation point instead of generating them.

The estimated gained time over real realization cost ratio was lower than the ratio of successfully generated source code. In fact, for a given source code section, it is slower to manually modify generated source code sections than to produce the same entire code section manually. Some modifications involved different parts of the software and were difficult to perform. However, most of the required modifications on the source code produced using our approach could be handled with automatic tools provided by any IDE.

In the second experiment, the estimated gained time was lower than in the first experiment. This was mainly caused by the use of new technologies that were not used in Absence Manager. Consequently, developers spent more time to realize a task due to their lack of experience in the new involved technologies. And the source code could not be generated because no domain engineering ASM was available to describe the use of those technologies.

Although the estimated gained time involves only the realization phase, we believe that the proposed so-

Table 2: Results of using the studied solutions on the case studies.

Application	Approach	Generated code		Gained time (realization)[1]
		Correctly	Requiring corrections	
Absence Manager	(Lahiani and Bennouar, 2014)	71%	14%	16 M/D (40%)
	Proposition	80%	5%	25 M/D (62.5%)
Mobility Advisor	(Lahiani and Bennouar, 2014)	70%	11%	4 M/D (12.3%)
	Proposition	78%	3%	10 M/D (30.7%)

lution can reduce the amount of time required in other development phases. For example, costs in project management are reduced because we lower human resource costs in the realization phase. Moreover, after the realization phase, the produced application must be tested before being released to the customer. The cost of this qualification phase is variable. It depends on several factors such as the requirements about the product quality, the application complexity or even the testing process reliability. By reusing assets tested in previous projects, we reduce the application complexity by providing standardization. We can also improve the testing process reliability because a large amount of the source code is obtained by reusing assets that are linked to features. Therefore, it is possible to provide test scenarios, designed in previous applications containing the same features, to help testers in the qualification phase.

Therefore, the proposal reduces the development effort in application engineering (*Q2.2*).

Experiments showed that the proposal improves the reuse of platform-specific assets and therefore productivity in MD-SPLE. However, we have also identified some threats to our solution validity which are discussed in the next section.

4.3 Threats to Validity

External Threats. Firstly, the comparison between our proposition and the existing approaches is limited. In fact, we could compare our results to only one solution. Furthermore, estimations involving the existing approach are theoretical and not obtained by measuring results using a concrete tool. These estimations rely on the estimations of the development team regarding specific implementation and correction tasks.

Moreover, the tested applications have a similar architecture (client-server) to experiment variable ASMs reuse. The size of tested applications is small because their architecture improved reuse of components "on the shelf". Therefore, the application source code focuses only on domain-specific behaviors and connections between components implementations. Consequently, the proposal applicability in a more generic context is not guaranteed.

Finally, adoption in an industrial context is a main concern for both model-driven engineering and software product line approaches. In our case, we believe that this threat is reduced by the maturity of available MDE tools and by the fact that the approach ensures that code can be generated by template engines.

Internal Threat. A potential limitation of the proposal is that the PIM and PSM meta-models must evolve alongside the evolution of the product line. But those meta-models' evolutions can be difficult because they impact other reusable assets like model transformation operations meaning those assets might have to evolve too. This threat is reduced by the large amount of existing MDE solutions addressing the problem of the meta-model evolution (Paige et al., 2016).

5 RELATED WORKS

Previous works addressed the problem of handling platform-specific variability. We evaluate their capabilities to handle, partially handle or not handle each case of platform-specific variation points identified in the "4+1" view model listed in Section 2.1. Our results are summarized in Table 3.

Reusing MDA models in a software product line is an idea introduced by (Deelstra et al., 2003). However, only platform variability is addressed in this paper. But the variability of platforms is not platform-specific variability. In fact, platform variability consists of the set of available platforms which can address a specific problem. For example, to implement a software, the different programming languages are platform variants, all addressing the problem of implementing the system.

Alternatively, (Czarnecki et al., 2004) proposed a staged configuration process in which variability

[1] Estimated costs measured in Man/Day (M/D).

Table 3: Capabilities of existing approaches to handle platform-specific variability aligned on the "4+1" view model.

Approaches	Platform variability	Platform-specific variability		
		Process view	Physical view	Development view
(Deelstra et al., 2003)	✓	✗	✗	✗
(Czarnecki et al., 2004)	✓	!	!	!
(Hamed and Colomb, 2014)	✓	!	!	!
(Dageförde et al., 2016)	✓	✓	✗	✗
(Usman et al., 2017)	✓	✗	✓	✗

Legend: ✓ Handled   ! Inefficiently handled   ✗ Not handled

is represented in several feature models. Each feature model represents the system features or a subsystem features to correspond to the abstraction level of a development team. Then, an application is designed by selecting features from the highest abstraction level model. Selected features are refined into specific feature models in lower abstraction levels. Once the staged configuration is complete, core assets are automatically selected and integrated in the system to produce. The staged configuration process can be adapted to our problem using the MDA abstraction levels. Thus, platform-specific variability can be managed. However, it is necessary to design core assets for each platform or technology possible because they are realizing the lowest abstraction level features which are specific to platforms. Then, the staged configurations approach using MDA abstraction levels does not scale up with the addition of new platforms.

(Hamed and Colomb, 2014) identifies the problem of platform-specific variability lack of management in existing approaches. This approach addresses the problem of handling Non-Functional Requirements (NFR). NFRs are client's requirements that are not related to the application functionalities or business domain such as the quality requirements (performance, maintainability). Depending on the chosen NFRs, platform-specific design variants are selected by integrating variation points in PIM to PSM transformation operations. Consequently, those operations complexity increases quickly with the addition of new design variants because each alternative has to be described in its dedicated transformation rule. Moreover, NFRs are global features often related to quality criteria. Therefore, some selected implementation variants are applied without considering the application specific use cases.

(Dageförde et al., 2016) proposed to use a MDA and SPLE combination to realize a cross-platform mobile product line. It extends the model-driven cross-platform framework MD2 to adapt it to a SPLE context. MD2 describes an application using a textual Domain Specific Modeling Language. The application is designed by deriving a variable workflow of tasks similar to the Business Process Model and Notation (Group, 2011). The approach permits to produce applications that can collaborate with each others thanks to the expressiveness of MD2. Managing the variability with modification of the workflow implies that only high abstraction variability is handled. In fact, workflow models are coarse grained models using tasks as basic elements. Consequently, the workflow describes the application behavior but not its architecture. Therefore it is not possible to select an alternative platform-specific implementation pattern such as a different architecture (physical and development views). Modifying the application design to satisfy a non-functional requirement would require to be able to choose between different model transformation operations automatically.

Then, (Usman et al., 2017) proposed a MD-SPLE approach which combines MDA and SPLE for mobile development context. This approach addresses the problem of using product lines in the mobile development context with its extensive use of different platforms and hardware devices. It uses UML2 models. The approach integrates platform variability by using UML profiles. Those profiles specialize PIM elements to add platform specificities. They can be seen as a Platform Description Models (PDM) which describe platform specificities. However, variation points are related to features or hardware choices (physical view) and not to implementation choices (development view). Similarly to (Dageförde et al., 2016), modifying the application design is hard because the approach relies on a common architecture implicitly described in model to text transformation operations. Consequently, variability related to the process view is not managed. Moreover, PDMs modify uniformly every elements of a targeted kind. Therefore, it is not possible to select different variants for different elements in the same model.

6 CONCLUSIONS

We propose a generic solution which goes further in the combination of MDA and SPLE that consider MDA models as configurable core assets of a product line. The proposal improves productivity regarding existing MDA and SPLE combinations by enhancing reuse of platform-specific assets and fully managing platform-specific variability.

Firstly, we provide a definition of platform-specific variability by identifying platform-specific variation points in different aspects of the software design. These aspects are defined accordingly to the "4+1" view model which is a well-known categorization of a system concerns. We also show that platform-specific variability has a not negligible impact on the reuse capabilities of the product line.

Then we propose a new PSM structure based on a composition of two sub-models. On the one hand, the Cross-Cutting Model (CCM) is obtained by transformation of the PIM which defines the application conception. On the other hand, the Application Structure Model (ASM) is obtained by reuse of variable models of the same nature defined in domain engineering.

Platform-specific variability is represented on ASMs with two mechanisms: asset variants which permit to replace an ASM with one of its variants and model attributes which permit to configure the assets to reuse. In this way, domain engineering ASMs are generic configurable assets organized by their commonalities and variabilities.

We experimented our proposal to produce two concrete applications. The obtained results confirmed that fully handling platform-specific variability significantly increases the productivity of a product line. In fact, the generated code could vary according to cross-domain, platform-specific variation points.

Finally, the capabilities of existing approaches addressing the problem of managing platform-specific variability are analyzed. Results showed that platform-specific variability is either ignored, only partially managed or fully managed but implying shortcomings in terms of maintainability of the product line.

The presented work involved only the PSM abstraction level. We expect that the proposed PSM definition could impact the CIM and PIM contents. Our future works will address impacts on higher abstraction levels to integrate the proposal in a full approach involving all MDA abstraction levels. We plan to work on two main axis.

Firstly, further works will be required to understand how the selection of CIM and PIM abstraction level assets can impact the selection of PSM abstraction level ones. We expect that the selection of ASMs will be motivated by two criteria: the non-functional requirements expressed by the client represented in the CIM and the software architecture design of the PIM.

Then, the proposal core assets organization is managed by a feature model. This model purpose is to represent coarse-grained features. However, PSM reusable assets can be fine-grained models. Integrating Common Variability Language (Haugen et al., 2013) is a promising solution that might help us organize more fine-grained assets.

ACKNOWLEDGEMENTS

We would like to thank the National Association of Research and Technology (ANRT in French) for its contribution to this research.

REFERENCES

Brambilla, M., Cabot, J., and Wimmer, M. (2017). Model-driven software engineering in practice, second edition. *Synthesis Lectures on Software Engineering*.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional.

Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, pages 1–17.

Czarnecki, K., Helsen, S., and Eisenecker, U. (2004). *Staged Configuration Using Feature Models*, pages 266–283. Springer Berlin Heidelberg.

Dageförde, J. C., Reischmann, T., Majchrzak, T. A., and Ernsting, J. (2016). Generating app product lines in a model-driven cross-platform development approach. In *49th Hawaii International Conference on System Sciences (HICSS)*.

Deelstra, S., Sinnema, M., van Gurp, J., and Bosch, J. (2003). Model driven architecture as approach to manage variability in software product families. *ResearchGate*.

Group, O. M. (2011). *Business Process Model and Notation. Version 2.0*.

Hamed, A. and Colomb, R. M. (2014). End to end development engineering. *Journal of Software Engineering and Applications*, pages 195–216.

Haugen, O., Wasowski, A., and Czarnecki, K. (2013). Cvl: Common variability language. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 277–277. ACM.

Jacobson, I., Griss, M., and Jonsson, P. (1997). *Software Reuse: Architecture Process and Organization for Business Success*. ACM Press Books. ACM Press.

Kim, S. D., Min, H. G., Her, J. S., and Chang, S. H. (2005). Dream : A practical product line engineering using model driven architecture. *Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05)*.

Kruchten, P. B. (1995). The 4+1 view model of architecture. *IEEE Software*, pages 42–50.

Lahiani, N. and Bennouar, D. (2014). An mda based derivation process for software product lines. In *The International Arab Conference on Information Technology (ACIT2014)*.

O'Neil, E. J. (2008). Object/relational mapping 2008: Hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1351–1356. ACM.

Paige, R. F., Matragkas, N., and Rose, L. M. (2016). Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc.

Schmidt, D. C. (2006). Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*.

Usman, M., Iqbal, M. Z., and Khan, M. U. (2017). A product-line model-driven engineering approach for generating feature-based mobile applications. *Journal of Systems and Software*, 123:1 – 32.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS