Using Multiple Feature Models to Design Applications for Mobile Phones

Clément Quinton, Sébastien Mosser, Carlos Parra, Laurence Duchien

Using Multiple Feature Models to Design Applications for Mobile Phones

Clément Quinton
INRIA Lille - Nord Europe
LIFL UMR CNRS 8022
Université Lille 1, France
clement.quinton@inria.fr

Sébastien Mosser
INRIA Lille - Nord Europe
LIFL UMR CNRS 8022
Université Lille 1, France
sebastien.mosser@inria.fr

Carlos Parra
INRIA Lille - Nord Europe
LIFL UMR CNRS 8022
Université Lille 1, France
carlos.parra@inria.fr

Laurence Duchien
INRIA Lille - Nord Europe
LIFL UMR CNRS 8022
Université Lille 1, France
laurence.duchien@inria.fr

ABSTRACT

The design of a mobile phone application is a tedious task according to its intrinsic variability. Software designers must take into account in their development process the versatility of available platforms (*e.g.*, Android, iPhone). In addition to this, the variety of existing devices and their divergences (*e.g.*, frontal camera, GPS) introduce another layer of complexity in the development process. These two dimensions can be formalized as *Software Product Lines* (SPL), independently defined. In this paper, we use a dedicated meta-model to bridge the gap between an application SPL and a mobile device one. This meta-model is also the support for the product derivation process. The approach is implemented in a framework named AppLIDE, and is used to successfully derive customer relationship management software on different devices.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies, Representation*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*

General Terms

Software Product Line, Meta–Model, Feature Model, Application for Mobile Phones

Keywords

Smartphones, Meta–Model

1. INTRODUCTION

Mobile phones, and smartphones in particular, are highly versatile ubiquitous systems. To automate the software development process for this particular subset of devices, it is important to take into account the different variations at the level of software, in the configuration of a mobile application product, but also at the level of hardware, and in particular, at the features supported by different devices running on different operating systems. For example, a functionality for a given application can be implemented in several ways, and, there can also be different mobile devices available (*e.g.*, with or without frontal camera, GPS sensor). This combination of possibilities represents a major challenge for developers that want to reach a maximum of end-users.

One way to automate this development process is to use a *Software Product Line* (SPL) approach. In an SPL, multiple products are *derived* by combining a set of different core assets. One of the most important challenges of SPL engineering concerns variability management, *i.e.*, how to describe, manage and implement the commonalities and variabilities existing among the members of the same family of software products. A well-known approach to variability modeling is by means of *Feature Diagrams* (FD) introduced as part of *Feature Oriented Domain Analysis* (FODA) [9] back in 1990. A FD typically consists of *(i)* a hierarchy of *features*, which may be *mandatory* (commonality) or *optional* (variability), and *(ii)* a set of *constraints* expressing inter-feature dependencies. Every feature can be realized by one or more assets. To build a product, the assets of different features are combined to obtain a software product. In this context, features can be seen as a way of modularizing a software product. Modularization, as introduced by Parnas [14], refers to the separation and localization of concerns. Separating such concerns helps specialists to focus on small modules of their expertise.

Contrarly to frameworks (see Sec. 5) that rely on code-driven approaches, we propose to use models as a support for *(i)* product derivation and *(ii)* consistency reasoning. Our approach use Model Driven Engineering (MDE) and SPL to bridge the gap between *business variability* and *device variability*. We propose to separate each domain with two different FDs. For the business domain, we define an application dependent FD that represents the different configurations available for a given business. For the device do-

SPLC'11 August 21-26, 2011, Munich, Germany.

main, we define an application independent FD that aggregates the variability of different and heterogeneous mobile devices. Using this separation we propose a derivation process that automates the development process through model merging and code generation, and that additionally analyses automatically the constraints of each product derived, with regard to the available mobile devices.

As a result, our derivation process allows developers to identify conflicts between a given product and the underlying hardware on which products are executed. To validate our approach, we use as a running example a *Customer Relationship Management* (CRM) product family, that contains up to 360 different product configurations. Nonetheless, the approach proposed here is not specific to CRM applications and can cover different domains.

The remainder of this paper is organized as follows. In SEC. 2 we present the MOBICRM example based on CRM applications. In SEC. 3 we describe the approach and the way MDE and SPL can be combined to support mobile application derivation. SEC. 4 presents the implementation and the results obtained from applying the approach. Finally, SEC. 5 compares our approach with close-related approaches in the literature and SEC. 6 concludes the paper.

2. MOTIVATION & CHALLENGES

In this section, we present a motivating example based on a CRM system. We discuss its representation as an FD and we describe the challenges our proposition faces.

2.1 Adaptation to heterogenous systems

In this paper, we propose a framework to manage the variability of mobile applications and devices to make them available to the largest number possible. To maximize the number of reachable end-users, a smartphone application (*i.e.*, a product) has to be available on a maximum of mobile platforms. Unfortunately, a company has to develop concurrently several versions of the same piece of software, or complete software, targeting the different operating systems or devices. A piece of software, or a part of application specifications, can also be reused in another application, allowing time-saving and increasing development productivity. An alternative way is to develop it on a restricted set of platforms, accepting to lose end-users. This is not acceptable for a company that wants its application (*e.g.*, products catalog, shops location) to be available for the largest set of end-users.

2.2 Running example

To illustrate our proposition, we introduce MOBICRM, a CRM application family for mobile devices, *e.g.*, smartphones. A CRM system allows the user to have information about the company customers, (*e.g.*, name, address). The main advantage of a CRM application dedicated to smartphones is that the information can be retrieved anywhere. For end-users, the entry point of the application is a login screen. There are different ways to access the user account. In other screens, the user retrieves a local or remote customers list depending on his/her login. He/she also interacts with these customers (*e.g.*, calls them, sends them emails or SMS, takes a picture). As a result, the MOBICRM application family presents multiple variation points enabling a wide range for customization in order to address different user needs.

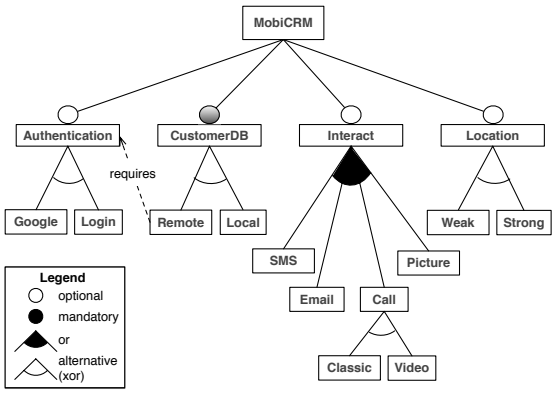


Figure 1: $FM_{MobiCRM}$

We formalize these variations through the definition of a MOBICRM feature model, $FM_{MobiCRM}$ (FIG. 1).

The second variability dimension in mobile applications development is in the mobile devices and systems. For example, a MOBICRM product $P_1 = \{Login, Remote, Video\}$ will run on iPhone 4 but won't be able to run on older devices, *e.g.*, iPhone 3GS. This kind of device does not have a front-facing camera to make a video call. Furthermore, even if the camera is available, devices running the Android operating system cannot provide this functionality on versions 2.3.3 and below. Being able to identify such inconsistencies gives strong support to designers as they are able to accurately know which device is able to run their software. We formalize these variations through the definition of a mobile device feature model, FM_{device} (FIG. 2).

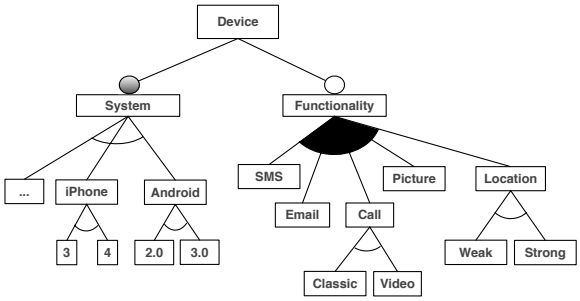


Figure 2: FM_{device} **(extract)**

2.3 Challenges

We propose in this contribution to combine the FM_{device} with a $FM_{application}$, whatever the application (in our case MOBICRM), into a single solution for product derivation that gives strong support to mobile application developers. For this, we have to face several challenges:

$\mathcal{C}_1$: *Supporting application variability.* The essential role played by the variability is to use it as a starting point for deriving products. The first challenge that has to be faced regards the definition of a process that uses as input any configuration obtained from the FD

$FM_{application}$ and that results in a fully functional product.

$\mathcal{C}_2$: *Supporting device variability.* To maximize the number of reachable end–users for the different software products, it is necessary to identify the inconsistencies between such products and the mobile devices on which they have to be executed. The second challenge refers to the automatic detection of these inconsistencies, using as input the the products of the $FM_{application}$, and a set of mobile devices like the ones represented by the FM_{device}.

We propose the following framework to accurately address these challenges.

3. PROPOSAL

To face these challenges, we define a model–driven framework used to *(i)* support the derivation process and *(ii)* bridge the gap between an application FD and the mobile device one.

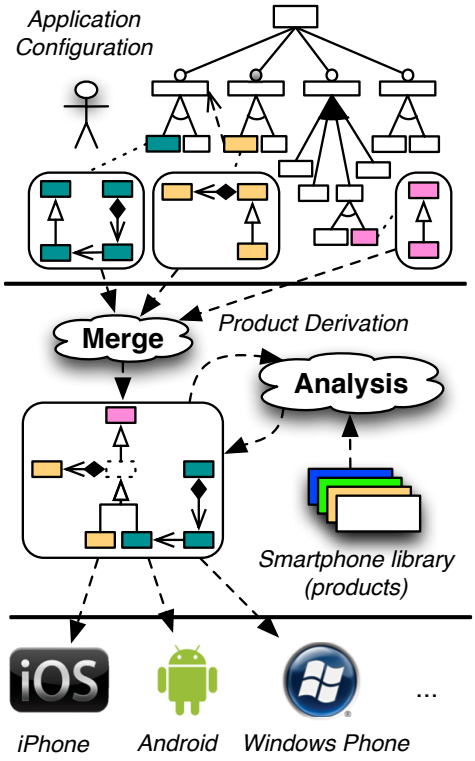


Figure 3: Approach overview

We propose in a first step to tackle challenge $\mathcal{C}_1$ by *(i)* associating each application–dependent feature to a fragment of model, *(ii)* merge these fragments into a model that represents the expected product. In a second step, we handle challenge $\mathcal{C}_2$ by checking if this model conforms to a meta–model representing a given device. At the end, the application is generated using classic model transformations.

3.1 AppliDE, a Model-Driven Framework

We define AppliDE as a model–driven bridge to support the previously described approach. It models the intrinsic variability associated to mobile phones applications. AppliDE is built upon industrial foundations, through an industrial transfer grant between INRIA Lille–Nord Europe and the UbInnov startup company. The main idea of this partnership is to reduce the time–to–market between the design of an application and its availability on multiple mobile devices.

AppliDE defines a meta–model to represent mobile applications. This meta–model allows one to model such a system in a platform independent way, which reifies smartphones application concepts. A model that conforms to this meta–model describes: *(i)* application functionalities (*e.g.*, sending an email, retrieving device's location), *(ii)* data sources used to feed the application (*e.g.*, local file, Web service invocation), and finally *(iii)* user–interface elements (*e.g.*, buttons, inputs). A coarse–grained description of the AppliDE meta–model, called AppliDE MM, is depicted in Fig. 4 using the class–diagram formalism. In AppliDE, an application is the root element of the meta–model and is defined as a set of datasources and views. `DataSources` can be local (*e.g.*, `File`) or remote (*e.g.*, `WebService`). `Views` define several graphical objects (*i.e.*, `GUIObject`) that can be both view or widget (*e.g.*, `Button`, `InputText`). The `Instruction` concept represents a functionality (*e.g.*, `SendSMS`, `TakePicture`, `Call`) that is triggered when an event occurs on a graphical element.

3.2 Supporting Product Derivation

To tackle the first challenge, we propose to bind each application–dependent feature to a fragment of an AppliDE model. Thus, a feature defined in the MobiCRM feature diagram holds as assets its corresponding model elements, that conforms to the AppliDE meta–model.

Modeling Assets.

Feature assets are defined in terms of model elements, with regard to the previously defined meta–model. Each feature is bound to fragments of model conform to the AppliDE meta–model. Such fragments can contain both structural (*e.g.*, a button and its associated event) and behavioral (*e.g.*, an action triggered by an event) elements.

The main asset is associated to the `MobiCRM` feature and with the mandatory features, in our case `CustomerDB`. It models the core of the application. It is composed by three `Views`, implementing the common user–interface provided to the user. The `Authentication` feature is associated to the `LoginView` that hosts the `Login` component, used to implement the authentication interaction. The `CustomerDB` feature is associated to a `DataSource`, which contains the customers. It also contains graphical elements used to display these customers and interact with them. `Interact` features (*e.g.*, `SendSMS`) are associated to logical elements that enrich the available interactions with the customers.

We consider here two examples of model fragments extracted from the MobiCRM running example: *(i)* supporting customer interaction through SMS and *(ii)* user authentication. The former is implemented through a button and its associated event. A click on the button triggers a SMS to be send (Fig. 5a). The latter (`Login` feature) is associated to a fragment of model (depicted in Fig. 5b using UML object–diagram syntax) that implements a *Login* component. This fragment defines graphical elements (*i.e.*,`Button` and `InputText`) used to enter login and password, embedded into a

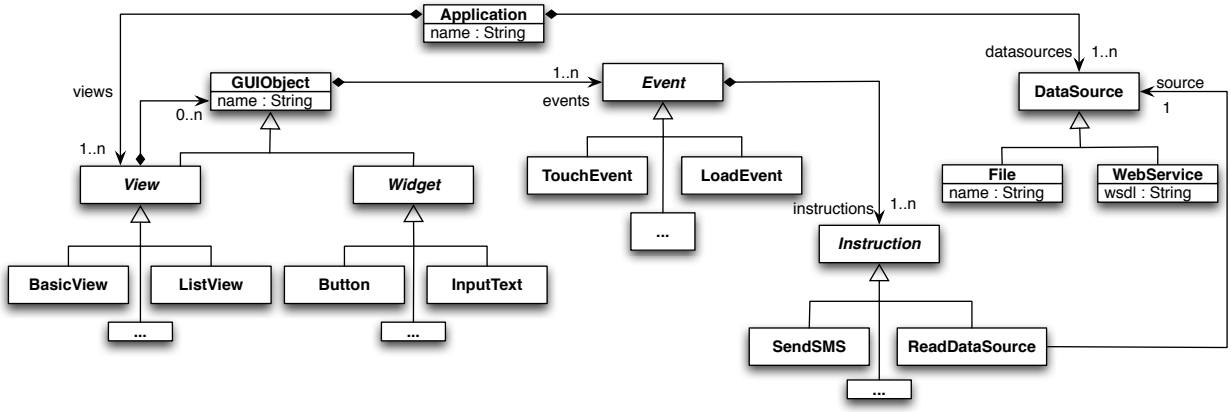
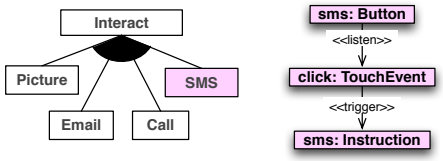
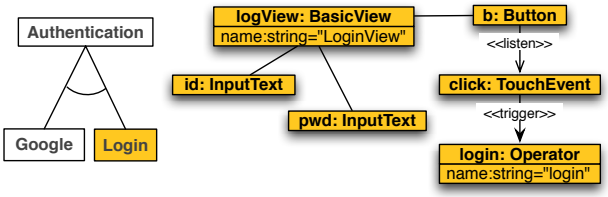


Figure 4: The AppliDE Meta–Model (extract)



(a) The SMS feature and its asset



(b) The Login feature and its asset

Figure 5: Features and associated model elements

Figure 6: The MobiCRM application (extract)

view. A click on the *login* button triggers a call to the Web Service *login* operator. We depict in Fig. 6 the running application (in the iPhone simulator) obtained after product derivation and model transformation.

Product derivation.

The derivation of a given product is thus supported by usual model composition techniques. We propose to use classical merge algorithms (*e.g.*, Kompose [5]), denoted as μ in this paper. The algorithm takes as input the set of models to be merged, and produces as output the "merged" model, *i.e.*, the expected product. In this example, we rely on a name–based matching mechanism, that is, the merge algorithm identifies elements with the same name and triggers their merge. We use a binding function β to associate a given feature F and its associated model elements $\{me_1, \ldots\}$. Considering a product as a set of selected features $p = \{F_1, \ldots, F_n\}$, we obtain its associated model m_p as the following:

$$m_p = \mu(\bigcup_{i=1}^{n} \beta(F_i))$$

Let us now consider a product that includes customer interaction with SMS, and authentication (Fig. 7). Thus, the product is composed by `SMS` and `Login` features, in addition to the application core: $p_1 = \{Login, SMS, MobiCRM\}$. The composed model (*i.e.*, the derived product) is depicted in Fig. 8.

Application generation.

AppliDE allows the user to generate applications source code for several mobile devices *e.g.*, smartphones. We rely on model–transformation techniques to support this generative process. The AppliDE model associated to a given product (*i.e.*, obtained from the derivation process) is transformed into its representation with regard to a given execution platform. Thus, a given model m can be transformed into an iPhone application a_i thanks to a dedicated model–transformation τ_i: $a_i = \tau_i(m)$. To support a new execution platform pl, one defines a new transformation τ_{pl}, and adds

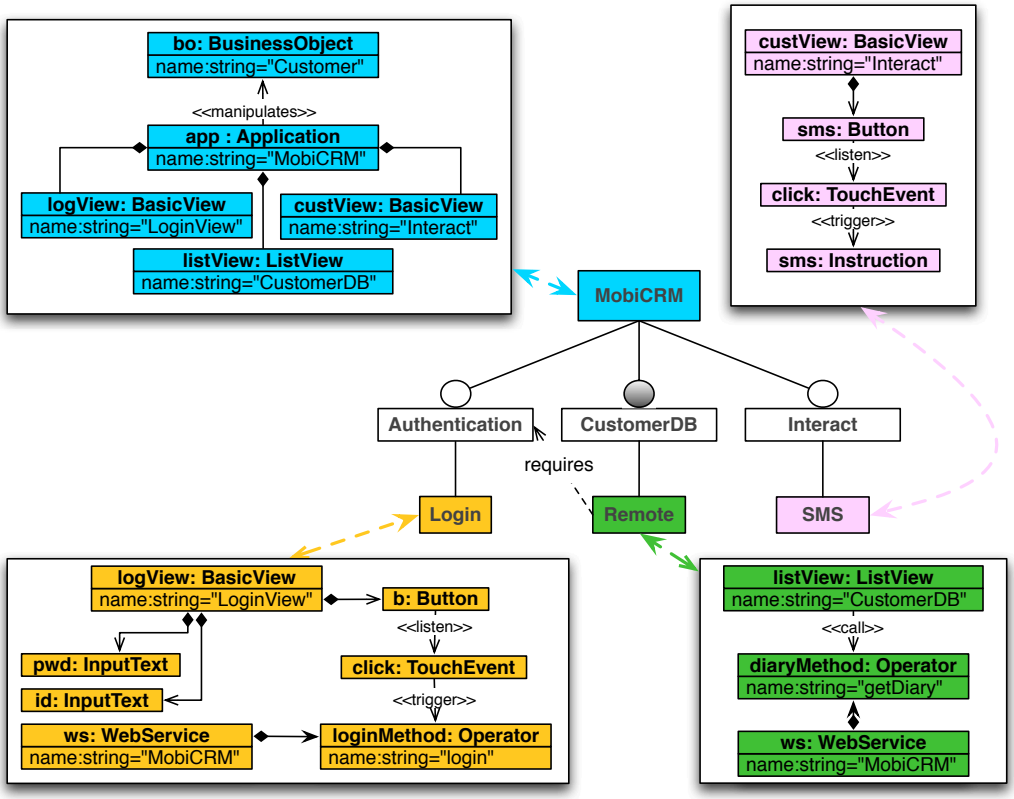


Figure 7: Merging assets (model fragments)

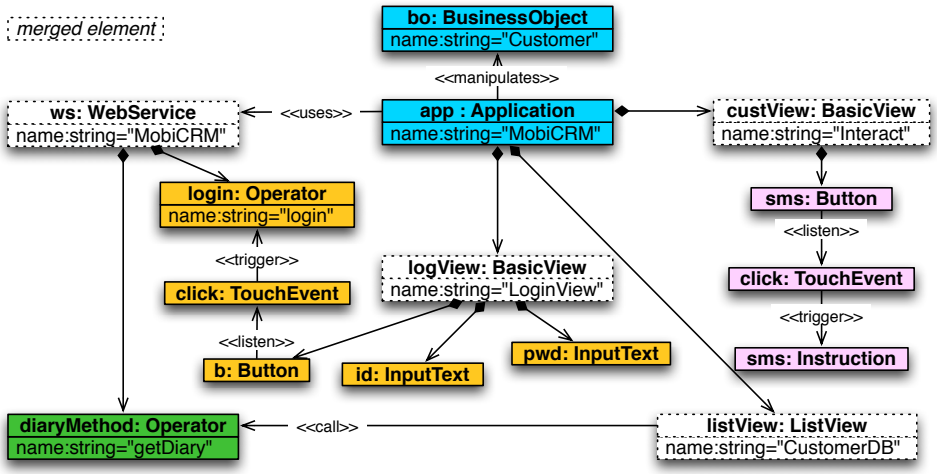


Figure 8: A MobiCRM product

it in the framework[1].

3.3 Bridging the gap

The previous subsection illustrates how features defined in the application FD are associated to fragments of models. To tackle challenge $\mathcal{C}_2$, we need to bridge the gap between this FD and the mobile device one. Following an endogenous approach, we rely on the AppliDE meta–model to implement such a bridge: application–dependent features holds fragments of model as assets, and device–dependent features are associated to meta–model concepts.

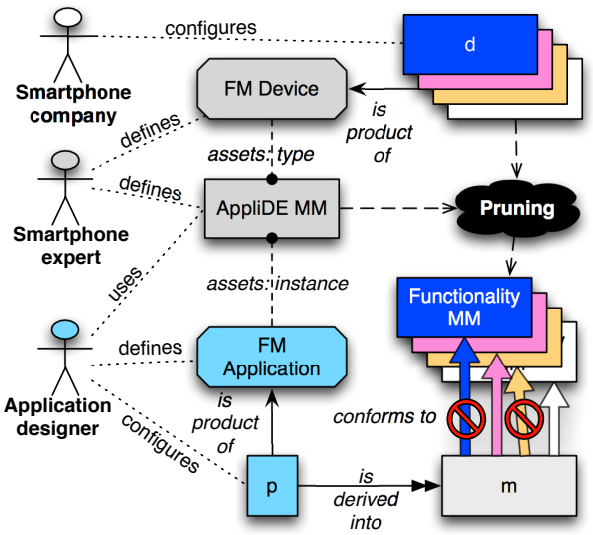


Figure 9: Consistency verification process overview

Fig. 9 depicts the consistency verification process used to confront a given application to a given device. It involves three actors, who interact with the defined artifacts at different time. The smartphone expert defines both AppliDE MM (containing smartphone application concepts) and FM_{device} (holding the features corresponding to existing device configurations, *i.e.*, the set of d configurations) provided by smartphone companies. The application designer uses the AppliDE MM to configure an application, whatever the application (in our case MobiCRM), and define its associated features. Considering AppliDE as the bridge that connects all the different artifacts, an application designer benefits from the two previously described advantages: *(i)* automatic generation of the executable applications and *(ii)* device analysis.

The AppliDE MM can be pruned [16] into a new meta–model called Functionality MM. For a given device d, a transformation extracts all the AppliDE MM concepts associated to the *Functionality* features of the device FD to obtain the Functionality MM. If a given model m derived from a product p conforms to the Functionality MM, then one can say that p can be executed on d.

[1]We rely on usual techniques (*e.g.*, model–to–text template engines) to implement the model–transformation chain, which eases the development of new ones and their adoption by engineers in an industrial context.

Sketching formal analysis.

Thus, we can reduce the analysis of an application product to a type–conformance problem. On the one hand, let $p \in FM_{MobiCRM}$ a valid product, and m_p its associated merged model. It requires a set of functionalities (denoted as F_p), *e.g.*, SMS sending, GPS sensor. These functionalities are implemented in AppliDE through the concepts defined in the meta–model (*e.g.*, the `SendSMS` instruction). Thus, one can understand F_p as the set of concepts used by the model elements contained by m_p. On the other hand, let us consider now d a given device, *i.e.*, a product defined according to FM_{device}. According to the previously given definition, the assets associated to each feature are concepts defined in AppliDE. Thus, the assets A_d associated to d represent a set of concepts that one can use in an application to safely execute it on d (a pruned meta–model [16]). An immediate analysis to assess if a given product can be executed in a given device is to check if $F_p \subseteq A_d$. Nevertheless, as we reduce the analysis as a type conformance problem, elaborated techniques from this dedicated research field can be used to tackle this goal.

For instance, consider the following configurations:
$p_1 = \{SMS, Email\}$
$p_2 = \{SMS, Video\}$
$d_1 = \{iPhone\ 3, SMS, Email, Call\ Classic\}$
$d_2 = \{iPhone\ 4, SMS, Email, Call\ Video\}$.

Using d_1 in our transformation, the Functionality MM (called MM_{Func1}) obtained after pruning holds `SMS`, `Email` and `Call Classic` as concepts. Using d_2, it (MM_{Func2}) holds `SMS`, `Email` and `Call Video` concepts. Let m_1 the model derived from p_1 and m_2 the model derived from p_2. m_1 conforms to MM_{Func1} and MM_{Func2} while m_1 conforms only to MM_{Func2} We can deduce that the application associated to the p_1 configuration can be executed on d_1 and d_2 while the one associated to the p_2 configuration can only be executed on d_2.

This analysis allows us to determine on which device(s) a MobiCRM product is able to run. Assuming the availability of a device catalog (expressed using FM_{device}), the result of such analysis gives to the designers the set of device products a given MobiCRM application is consistent with and consequently a set of "real" devices that can execute the application. Thus, one can immediately know the market share reachable by his/her current product.

4. VALIDATION

In this section we present the results obtained from the experimentation, as well as the implementation details of AppliDE.

4.1 Experimentation

As stated before, the MobiCRM FD represents 360 different product configurations. To validate our approach, we have used these configurations as input in our derivation process. In each case, we successfully obtain a complete product, and additionally, depending on the case, we verify if the product can be executed on a given device.

Consider the products we have presented in Sec. 3.3. For the product p_1, after the merging and the generation of code, we obtain a fully functional software product of around 1500 lines of code. The analysis shows that this product is compatible with the devices d_1 and d_2. For the product p_2 we

also obtain a fully functional software product with a number of lines of code similar to p_1. However, in this case, we find an inconsistency with the specified device. The video calling feature can only be used on devices with a front-facing camera. The functional meta–model obtained after the pruning process does not contain the meta-classes for video calling. This product however, can be executed in devices that support such functionality like the iPhone 4 and several Android smartphones. In total, 120 out of the 360 configurations cannot be executed in the iPhone 3 (device d_1).

4.2 Implementation

Regarding the implementation, the AppliDE framework relies on *Eclipse Modeling Framework* (EMF) [17] meta–models, which is one of the most widely accepted meta–modeling technologies. The EMF provides code generation facilities for building tools and other applications based on a structured data model. The XMI format is used to support model persistence.

The feature diagrams for both MobiCRM and the mobile devices have been described using the *FAMILIAR* environment [1] (Fig. 10). We also used the *FAMILIAR* tools to compute the number of available configurations.

```
FM(MobiCRM:
 [Authentication] CustomerDB
   [Interact] [Location];
 Authentication: (Google | Password);
 CustomerDB: (Local | Remote);
 Interact: (SMS|Email|Call|Picture)+;
 Location: (Weak | Strong);
 Remote -> Authentication;)
```

Figure 10: MobiCRM fd defined with FAMILIAR

From a model specification described in XMI, AppliDE is able to generate source code files. The source code generators for Android and iPhone have been developed as *Acceleo* templates, each model element being associated to code templates. iPhone template sample is depicted in (Fig. 11). The entry point of the generators is the Application root element.

```
[template public genInstruction(sms : SendSMS)]
['[['/]UIApplication sharedApplication[']'/] openURL:
['['/]NSURL URLWithString:[sms.parameters.genDataAccess()/]['']']'/];
[/template]
```

Figure 11: The Android template associated to the SMS feature

This choice brings high flexibility to the AppliDE framework. Indeed, one can target a new platform by adding corresponding source code generator templates (*e.g.*, for Windows Phone).

5. RELATED WORKS

Multi–view modeling approaches are used to support the design of complex systems, based on different points of views. In Ram (Reusable Aspect Models [12]), one can use both class diagrams and sequence diagrams to design a system. The approach supports the concurrent weaving of these two kinds of artifacts, and maintains consistency. The class diagrams are merged according to [15], and sequence diagrams according to Klein's method [11]. The Theme/Uml approach [3, 6] are another interesting multi–view modeling approach [2], very similar to Ram. Our proposition is clearly complementary with these ones, as we bring to the model–composition universe the architectural knowledge that a feature diagram reifies. Moreover, these powerful mechanisms can be reused in our approach, as we are based on the same approach of name-matching mechanism for the underlying composition.

Code-driven approaches, such as Rhomobile, PhoneGap, Titanium Mobile or MoSync translate HTML and JavaScript code into iPhone or Android source code. Contrarily to these syntactically-driven approaches, we propose to use models as a reasoning support. Our meta–model bridge the gap between two software product lines, thus ensuring their consistency and co-evolution. Dhungana *et al.* [4] support evolution of complex systems by defining model fragments representing subsystem parts and checking their consistency. The complete model is then obtained by merging fragments. In our approach, we systematically bind a model fragment to a given feature, introducing feature expressiveness into these artifacts. The originality is to ensure product (merged model) consistency according to two dimensions: application and mobile device variability.

With regard to derivation safety, the Mata approach [18] supports the weaving of models aspects using a graph–based approach. This approach supports powerful conflict detection mechanisms, used to support the "safe" composition of models [13]. The underlying formal model associated to this detection is based on critical pair analysis [7]. Initially defined for term rewriting system and then generalized to graph rewriting systems, critical pairs formalize the idea of a minimal example of a potentially conflicting situation. This notion supports the development of rule–based system, identifying conflicting situations such as "the rule r will delete an element matched by the rule r'" or "the rule r generates a structure which is prohibited according to the existing preconditions". These mechanisms were demonstrated as relevant to identify composition conflicts in the software product line domain [8]. Unlike these approaches, in our case we do not focus on the derivation safety of a given product configuration. We rather aim at identifying the compatibility between a given software product and a set of available mobile devices. This is done by specifying separately the application and device variability, and pruning the application meta–model with regard to each specific device configuration.

6. CONCLUSION & FUTURE WORKS

In this paper, we have presented a model-driven approach that supports mobile application development, ensuring application products to be platform independent. We start with a product family represented through a FD. For every feature in the FD, we define an associated model fragment defining an architectural element. We use the AppliDE framework and in particular, the meta–model to bridge the gap between the software SPL and the device one. The *functionality* features of the device FD are associated to meta–model concepts. If the merge of fragments model conforms

to the meta–model we can deduce which device is able to run the application.

Our approach faces the challenges identified in Section 2. For the first challenge regarding the variability of the application, we have presented the merging of fragments as well as the generation of code for different mobile platforms. For the second challenge, regarding the device variability, we have defined a pruning process that creates a reduced version of our application meta–model. This meta–model is used to check if the product being derived can be executed in a given hardware. To validate the approach, we have used it to develop the MobiCRM product family. The results of our experimentation show that our approach helps to deal with the variability of a MobiCRM product (360 different configurations) and the intrinsic variability of mobile phone devices. Furthermore, using the model–driven process provided by the AppliDE framework, we are able to generate fully functional software products on two different mobile platforms.

For future work, we think that our approach can be extended to face the challenge of inconsistency resolving, helping the developer to reach the maximum of end-users. Some decision rules could be added to the framework and proposed to the developer, such as remove a feature that is only available on a few systems in order to reach a bigger number of end-users (*e.g.*, if one removes the `Video` feature from an iPhone 4 application, one will reach all the iPhone 3 owners).

Acknowledgments

This work is supported by Ministry of Higher Education and Research, Nord–Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region Campus Intelligence Ambiante (CPER CIA) 2007-2013.

7. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. A domain-specific language for managing feature models. In *Symposium on Applied Computing (SAC)*. Programming Languages Track, March 2011.

[2] O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke. Composing Multi-View Aspect Models. In *ICCBSS*, pages 43–52. IEEE Computer Society, 2008.

[3] A. Carton, C. Driver, A. Jackson, and S. Clarke. Model-Driven Theme/UML. In *T. Aspect-Oriented Software Development VI* [10], pages 238–266.

[4] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122, 2010.

[5] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A Generic Approach For Automatic Model Composition. In *Aspect Oriented Modelling workshop at MoDELS*, 2007.

[6] H. Gomaa and M. E. Shin. A multiple-view meta-modeling approach for variability management in software product lines. In *ICSR*, pages 274–285, 2004.

[7] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 161–176, London, UK, 2002. Springer-Verlag.

[8] P. K. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, The Software Engineering Institute, 1990.

[10] S. Katz, H. Ossher, R. France, and J.-M. Jézéquel, editors. *Transactions on Aspect-Oriented Software Development VI, Special Issue on Aspects and Model-Driven Engineering*, volume 5560 of *Lecture Notes in Computer Science*. Springer, 2009.

[11] J. Kienzle, W. Al Abed, and J. Klein. Aspect-Oriented Multi-View Modeling. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2009. ACM.

[12] J. Klein and J. Kienzle. Reusable Aspect Models. In *11th Workshop on Aspect-Oriented Modeling, AOM at Models'07,*, 2007.

[13] G. Mussbacher, J. Whittle, and D. Amyot. Semantic-Based Interaction Detection in Aspect-Oriented Scenarios. In *RE*, pages 203–212. IEEE Computer Society, 2009.

[14] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[15] Y. Reddy, S. Ghosh, R. France, G. Straw, J. Bieman, N. McEachen, E. Song, and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. In *Transactions on Aspect-Oriented Software Development I*, pages 75–105, 2006.

[16] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model pruning. In A. Schürr and B. Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2009.

[17] D. Steinberg, et al. *EMF: Eclipse Modeling Framework (2nd Edition)*. 2nd revised edition (rev). edition, 2009.

[18] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, and J. Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In *T. Aspect-Oriented Software Development VI* [10], pages 191–237.