A feature-oriented model-driven engineering approach for the early validation of feature-based applications

Glenn Cavarlé, Alain Plantec, Steven Costiou, Vincent Ribaud

A feature-oriented model-driven engineering approach for the early validation of feature-based applications

Glenn Cavarlé [a,b,*], Alain Plantec [a,*], Steven Costiou [a], Vincent Ribaud [a]

[a] Univ. Bretagne-Occidentale, UMR CNRS 6285, Lab-STICC, F-29200 Brest, France
[b] Libre Informatique, 29000 Quimper, France

ARTICLE INFO

Article history:
Received 15 January 2017
Received in revised form 23 December 2017
Accepted 1 January 2018
Available online xxxx

Keywords:
Feature-oriented development
Early validation
Model driven engineering
Smalltalk

ABSTRACT

The software industry has to offer increasingly individualized software for a large number of platforms. In a constantly evolving technical context, the appropriateness and the profitableness of a software has to be ensured earlier, before most of the costs have been incurred and before most of the risks have been taken. Feature-Oriented Model-Driven Development (FOMDD) is a promising paradigm to tackle the issue of developing software variants when multiple platforms are targeted. However, because of its model-driven fundament, FOMDD suffers from limited capabilities regarding model execution and early validation. In this paper, we present CrossFabrik, an approach for the design and the early functional validation of feature-based applications. This approach allows the live debugging and editing of the underlying models during a simulation without being forced to stop and restart a validation process. Such an approach relies on the reflective capability of the development environment. An implementation of our approach within Pharo is also presented.

1. Introduction

In recent years, the software industry has faced many challenges as a result of the growing influence of mobile platforms. While the cost of purchasing applications decreases, the cost and the complexity of development increases. Software industry has to deal with emerging user's needs, with new business models and with the fragmentation of platforms. Since software platforms are moving toward fragmentation rather than unification and with the growing differences between platform features, dealing with multiple platforms is one of the most costly aspects in software development [1]. Moreover, the software industry has to offer increasingly individualized features to react to changing markets in order to keep a competitive advantage. This raises the question of software reuse across individualized software and across platforms. Thus, in a constantly evolving technological context and market, it becomes riskier to invest in software development. The appropriateness and the profitableness of a proposed software has to be ensured earlier, before most of the costs have been incurred and before most of the risks have been taken.

A common practice in software industry is to treat separately multiple variants of the same software to reach several market segments as well as to cope with the heterogeneity of platforms. As an example, a software could be distributed in two different versions: a first version with a restricted set of features which is made available for free and a full-featured paid version. To maximize the number of reachable end-users, each version is proposed on several devices (*i.e.*, on mobile

* Corresponding authors.
E-mail addresses: glenn.cavarle@univ-brest.fr (G. Cavarlé), alain.plantec@univ-brest.fr (A. Plantec).

https://doi.org/10.1016/j.scico.2018.01.001

and on desktop), taking into account the main operating systems for each device. Following this common practice, each version is treated for each platform separately to benefit from native look and feel, features and performances. Finally, each software is manually deployed and validated to ensure that the functionality is preserved across the multiple platforms and devices. In this example, making available a free and a paid version of a software on multiple platforms involves the actual development of at least twelve software variants to reach most of the desktop users (*i.e.*, MS Windows, macOS and GNU Linux users) and most of the mobile users (*i.e.*, WindowsPhone, iOS and Android users). This common practice have been criticized for a lack of software reuse and early feedback. Even if all software variants share a common conceptual architecture and a set of common features, treating them separately makes it tedious the reuse of software artifacts across variants and increases the development and maintenance effort. Each targeted platform can impose its specific operating system, a particular development environment, the use of a dedicated programming language and the integration of specialized libraries. The validation process for each software variant arises close to the end of the development effort. This increases the risk of having developed the wrong product. Finally, it can increase the cost of a change. During this late validation phase, many bugs might be discovered. Moreover, some bugs might lead to a deep and costly redesign that might impact the other software variants already developed.

The Feature-Oriented Software Development (FOSD) [2] paradigm is an alternative to developing multiple software variants separately. FOSD aims at explicitly representing common parts and differences of a family of software systems with the goal of reusing software artifacts among the family members. From a set of features, several software variants can be produced by merging corresponding software artifacts. However, FOSD relies on a single platform-specific technology to realize features. Software artifacts have to be implemented using the same programming language to be subsequently merged together. This constraint makes it difficult to tackle the development of software variants for different platforms.

The association between FOSD and Model-Driven Development (MDD) [3], also named Feature-Oriented Model-Driven Development (FOMDD) [4,5], allows the management of the variability even when multiple platforms are targeted. MDD is considered well suited to manage the fragmentation of platforms. Indeed, with MDD and especially with the Model-Driven Architecture (MDA) [6], a software can be designed in a platform-independent way and can be automatically refined to obtain platform-specific models from which source code can be generated. FOMDD approach benefits from both approaches. FOSD helps to define variants and MDD helps to implement software artifacts as platform-independent models. A well known approach for users and developers to assess what is really needed is to use a “working” system [7] but, even with FOSD or MDD, such a “working” system is available only close to the end of the development process, after most of the costs have been incurred and after most of the risks have already been taken.

In this paper, we propose a FOMDD approach associated with a dynamic modelling environment for the early design and the functional validation of software variants for multiple platforms. We propose an alternative to reduce the development risk by mean of software prototyping using FOMDD and software simulation within the development environment. With our approach, an even incomplete system can be run and tested early in the development cycle. A running system can be inspected and related models can be debugged live without being forced to stop or restart the execution.

We believe that model-driven prototyping can significantly help in defining the application scope, in assessing feasibility and in estimating efforts during early development activities [8]. By executing early and visualizing the software system to be built, developers and users can identify the true requirements and can detect problems in the early stages [9].

This paper is an extended version of a previous work [10] which was more focused on the synchronization process between models and the runnable artifacts. We extend our previous work by providing a larger overview of our approach in which the synchronization process takes place. In this paper, we describe with much more details the background concepts and how we extend the FOMDD approach to support the prototyping, the simulation and the debugging activities. Moreover, the technical aspects and the CrossFabrik implementation is also more precisely presented.

This paper makes three contributions. First, we propose a FOMDD approach to cope with the issues that arise when prototyping and validating software variants for multiple platforms. Second, we provide insights on how early validation can be achieved by means of simulation and live debugging of models. Finally, we show how the development environment can be adapted to bring together modelling and early validation activities. The remaining of this paper is organized as follows: Section 2 illustrates the approach with an example. Section 3 highlights and discusses the background concepts. Section 4 presents the overview of the proposed approach while Section 5 provides details about the design of CrossFabrik. Section 6 describes the implementation details of CrossFabrik using Pharo while Section 7 overviews the related work. Section 8 concludes the paper.

2. Illustrative example: the ContactApp software family

This section describes a simple application for managing contacts. This example is used throughout the rest of this paper to illustrate the background concepts and our approach.

This application named *ContactApp* includes several features and can be summarized as follows. The entry point of the *ContactApp* is the login screen. The end-user has to be authenticated using a dedicated login service or an external provider like Twitter or Facebook. After the end-user has been authenticated, the *ContactApp* allows users to interact with contacts which are stored in a remote database. The end-user can list and edit his contacts and can quickly have access to his favourite contacts. Moreover, the end-user can send an SMS or an email, can make an audio or a video call and can also retrieve contacts around his current position.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

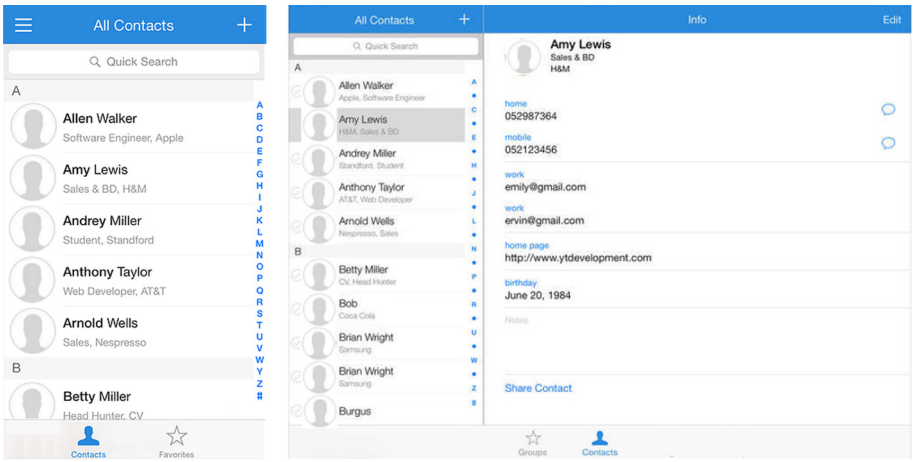

Fig. 1. Example of mock-ups for the *ContactApp* mobile and desktop versions (from left to right).

The hypothetical company developing this application wants to provide two commercial versions: a free and a paid version. The free version has limited features, it does not allow to login with external providers, to manage favourite contacts, to send SMS, to make video calls and to retrieve contacts around the current position.

According to the targeted market segment, this company wants to provide this application for mobile and desktop devices. The definition of the two previous versions is not sufficient to take into account the variability between devices (*i.e.*, between the mobile and desktop variant of each version). As illustrated by the two mock-ups in Fig. 1, whatever the specific platform targeted, a desktop software diverges from a mobile software in the provided features and in the composition of the user-interface.

For example, the geolocation of the end-user in order to retrieve contacts around him is specifically useful for the mobile application but not for the desktop one. However, this feature has impacts on data model because it have to provide the mean to store the geolocation information, in this case the user-interface has to provide some specific menus and buttons, etc.

In the end, from the *ContactApp* specifications, four versions or variants of the same application have to be produced.

3. Background concepts

This section covers important concepts to understand the early validation in the context of feature-based applications. *Model-Driven Engineering* and *Feature-Oriented Software Development* are two key approaches discussed in this section. Then, the concept of Feature-Oriented Model-Driven Development is introduced and explained.

3.1. Early validation

The validation process consists in evaluating software to ensure its conformity with respect to its expected use and the user requirements. Testing software and fixing bugs or design issues in the later stages of the development process or after the first deployment is time consuming. This often leads to an additional cost in terms of budget and planning. More than 50% of the total cost of a software come from the testing and maintenance activities [11,12]. This is why it is important that validation activities be initiated very early in the development process. The more complex a system becomes, the more problematic the validation of the software is.

Thus, reducing the time allocated to correcting defects and validation is an important challenge answered by early validation. According to Boehm et al. [13], one of the main methods for the reduction of software defects is the ability to detect problems in the design phases. This article speaks of a cost 100 times higher for a correction made after the first delivery compared to a correction made upstream. This observation encourages to invest and focus on ways to improve the validation process in the early design stages, upstream of the development. In addition, this observation encourages the practices of prototyping and simulation in order to avoid or to limit the additional and unexpected downstream cost.

Applying early validation to our illustrative example would imply to be able to prototype, simulate and assess the four *ContactApp* variants before developing them in platform-specific technologies.

3.2. Model-driven engineering

With the Model-Driven Engineering (MDE) [3,14] paradigm, a program is represented using models. A model captures particular details of a program's design. Several models can be used together to specify multiple aspects of a system with different levels of details. The Model-Driven Architecture (MDA) [6] is a particular approach that introduces a clear separation

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

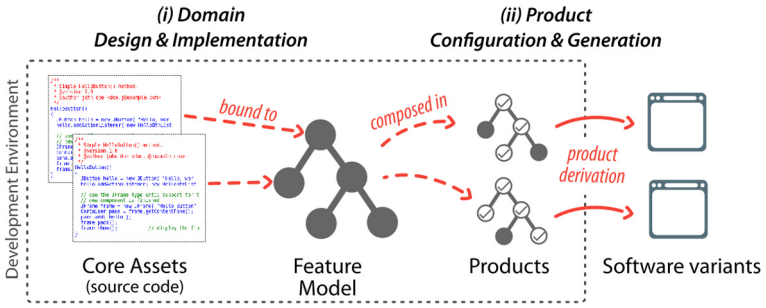

Fig. 2. The SPL approach.

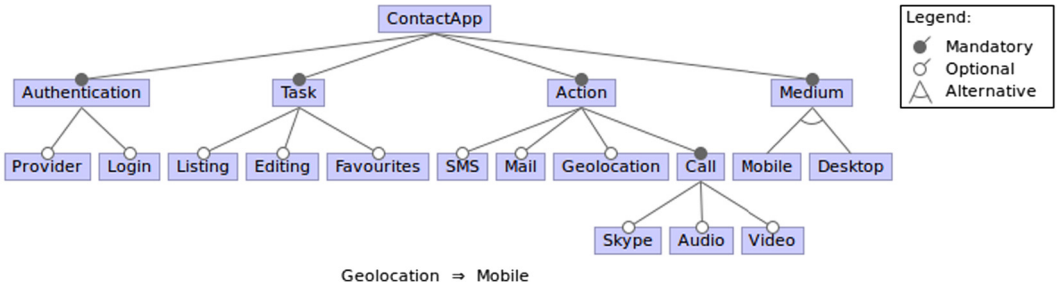

Fig. 3. Extract of the ContactApp feature model (FM_{app}).

between the business logic and the implementation details. It aims at developing a set of models, linked by transformations. These transformations allow mainly to start from a Platform Independent-Model (PIM) which is refined into several Platform-Specific Models (PSM), according to the platform-specific transformation rules. A PSM represents the concrete design and the implementation details of a system regarding a given platform. Hence, a model is an abstraction of a system from which it is possible to reason, communicate and automate part of the development process using model transformation and code generation.

Based on our illustrative example, MDE can solve the issue in managing desktop and mobile software variants by fostering common models from which platform-specific source code will be generated. However, MDE does not provide the mean to describe which features have to be assembled together in order to produce a customized product (*e.g.* product with or without the geolocation feature).

3.3. Feature-oriented software development

The Feature-Oriented Software Development (FOSD) [2,15] paradigm comes from the Software Product Line Engineering (SPLE) [16] practices and promotes the application of the feature concept to analyze, design and implement software systems. As depicted in Fig. 2, the FOSD process is built around a Feature Model (FM) [17,18] and can be summarized in two phases: (i) the domain design and implementation and (ii) the product configuration and generation.

A FM is a logical presentation of the common and variable features for a set of software variants, also named products. A FM is represented as a tree of named nodes organized using a *generalization/specialization* relationship. In a FM, some constraints can be specified on a feature individually and between several features.

The domain design and implementation phase consists in defining the FM and specifying how each feature is implemented. Implementing features involves developing software artifacts, also named *core assets*. Such a software artifact is generally developed using a general-purpose programming language. The whole set of core assets is called the *core asset base*.

During the product configuration and generation phase, features are selected by the designer to form the desired products. Such a features selection conforms to the constraints and relationships defined in the FM. This set of products is named a product family. Finally, the *product derivation* takes place. The product derivation is the process of constructing a particular software from a set of core assets. The complete source code of a software is generated for each product by composing and merging core assets which are bound to the product features.

Back to our illustrative example, Fig. 3 shows an excerpt of a feature model used to present the variations in the *ContactApp* software family. This feature model is called FM_{app}. The FM_{app} comprises *product features* and *process features* [5]. By product features we mean those that characterize the product as such, whereas process features refer to features which drive the feature composition during the product derivation. Features related to the targeted device are such process features.

Constraints can be specified between features. For example in Fig. 3, some features are marked as *Optional* or *Mandatory*. This means that each *ContactApp* variant will have the *Authentication*, the *Task*, the *Action*, the *Call* and the *Medium* features.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

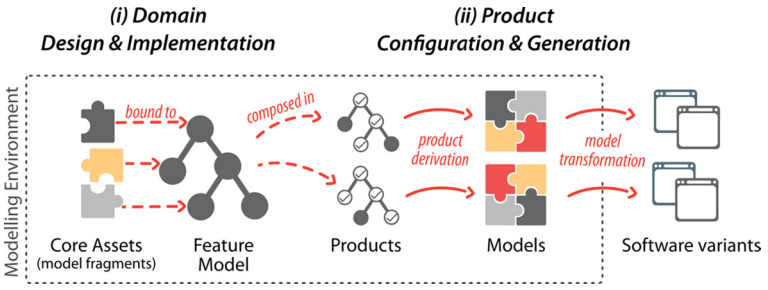

Fig. 4. The FOMDD approach.

To facilitate the definition of optional and mandatory features, three kind of parent-child relationships can be specified: *AND*, *OR* and *ALT*. The *AND* relationship does not restrict the selection, zero or more child features can be present in a product according to the optional and mandatory constraints. The *OR* relationship ensures that at least one child feature is present in all products. The *ALT* relationship ensures that only one child feature is present in all software variants. In addition, other restrictions can be specified separately. They can be used between unrelated features to define conditional inclusions or exclusions. In Fig. 3, the *ALT* relationship is used for the *Medium* feature to ensure that *Mobile* and *Desktop* features cannot be selected at the same time in the same product. Fig. 3 shows the definition of such a constraint between the *Mobile* and the *Geolocation* features. This constraint means that the *Geolocation* features implies the *Mobile* feature and therefore cannot be part of a product which includes the *Desktop* feature.

3.4. Feature-oriented model-driven development

The Feature-Oriented Model-Driven Development (FOMDD) [4,5] paradigm combines FOSD with MDE to produce models from the product derivation. This approach is depicted in Fig. 4.

In FOMDD, core assets are not platform-specific source code artifacts as in FOSD but model fragments which are weaved together during the product derivation. According to an EMOF-compliant [19] meta-model, a model fragment can be any element involved in the description of the structural and behavioural aspects of a software system (*e.g.*, Package, Class, Property, Operation, etc.). After the product derivation, the resulting product-specific model can be used as an input for model checking, model execution, model transformations and code generation. Hence, taking into account non-functional features and actual platform-specific constraints (*e.g.* performances, memory consumption, programming language specificities, etc.) can be postponed at the end of the process, after the product derivation, when the platform-specific source code have to be generated.

Applied to our illustrative example, FOMDD tackles both issues mentioned with MDE and FOSD. Customized products are defined for multiple devices and features are implemented not by platform-specific source code but with a combination of model fragments. Taking into account non-functional aspects and actual platform-specific implementation details is achieved late, during the code generation process.

3.5. Discussion

Section 3.2 showed how MDE helps in automating developments in the context of cross-platform development: stemming from a cohesive set of models, different platform-specific systems can be fully or partially generated. Section 3.3 showed how FOSD improves the software reuse through the definition of several products sharing common features. Reusing a feature in several products involves reusing software artifacts in several actual software variants. But the definition of core assets using a single programming language implies difficulties to target multiple platforms. The FOMDD approach depicted in Section 3.4 combines the advantages of both FOSD and MDE to cope with the issue of the software reuse across individualized software and across platforms. However, FOMDD has also the same drawbacks as MDE and FOSD: platform-specific source code, most frequently stored in files, has to be produced before executing and validating the software under development. Indeed, based on MDE, FOMDD is geared towards code generation. Platform-specific source code have to be generated, compiled and deployed on a given platform. The executable system is the final result of this process. An important issue is that, from the high level models to the final system to be executed, the production chain can be time and resource consuming. Moreover, the causal connection between the models and the generated artefacts is lost because of the code generation step [20]. Some approaches provide a synchronization support based on meta-data included within the generated artefacts or written in an associated file. When a model is updated, the artefact and the meta-data have to be re-generated and the execution has to be restarted. Even with this kind of approach, the “live” aspect of the causal connection is lost. The running system does not always reflect the current state of the related model. The impact of model changes cannot be immediately observed and developers are faced with a traceability issue between a malfunction that is observed during the execution and the source model that should be fixed [21].

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

The prototyping	The modelling of several software variants using a common meta-model which supports the variability management and the structural as well as the behavioural aspects of a software.
The simulation	The execution of one software variant from models and within the modelling environment.
The debugging	The live inspection and correction of models through the running instance of a software variant.

Fig. 5. The main activities supported by the approach.

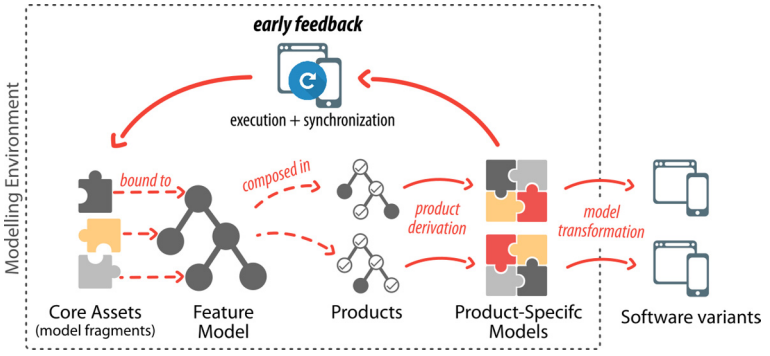

Fig. 6. Overview of the approach.

4. An approach for early validation of feature-based applications

This section presents an overview of the proposed approach to debug feature-based software variants early in the development process. As introduced in Section 3.1, the early validation should help identify and solve problems of a software product before it is actually developed in its final version. By prototyping models of a feature-based application in the early stages of the development process, we aim to collect the maximum knowledge and feedback about the application to be developed with minimal costs involved.

Our approach includes the following activities shown in Fig. 5.

As shown in Fig. 6, we extend the FOMDD approach to support these activities. During the simulation step, one can inspect and debug models while preserving the causal connection between the model fragments (*i.e.* the core assets) and the executed artifacts (*i.e.* the product-specific model).

Following the FOMDD approach, the specification of the variability is achieved through the use of a feature model together with the definition of products. The specification of core assets is achieved through the definition of model fragments which are bound to features.

According to Model-Driven Engineering, supporting the early execution of a modelled software implies the ability to define low-level specifications in the same way as high-level aspects of a software. For this purpose, the FOMDD process has to be supported by a dedicated meta-model which allows the specification of the behaviour.

At the product derivation step, the resulting models describe the distinctive software variants according to the defined products. We call these models the *product-specific models* (PrSM). A PrSM relies on the dynamic assembly of a subset of the *core asset base* and remains platform-independent. Given a PrSM, a software variant can be validated early by its simulation in the modelling environment. We call *validation process* this simulation step. Several simulation contexts can be used to validate a software variant across multiple devices and platforms. In case of an issue during this validation process, the core assets and the feature model can be fixed, this is depicted in Fig. 6 by the *early feedback*. When the PrSM is considered as mature enough, the code generation can take place to produce the final software variants. Then, non-functional aspects can be validated by running each software variant on the dedicated platforms.

Fig. 7 depicts an iteration in the development cycle including the validation process. The core assets, the product-specific models and the related executed artefacts are dynamically kept up to date. A change in a model fragment is automatically reflected in all related PrSMs. A running simulation is also dynamically impacted by such a change. This synchronization process allows the live debugging and editing of the underlying model fragments during a simulation without the necessity to stop and restart the validation process.

5. CrossFabrik: a prototyping framework and environment

In this section, we describe the key aspects of CrossFabrik, a framework and an environment that fit our approach for validating early feature-based software variants for multiple devices.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

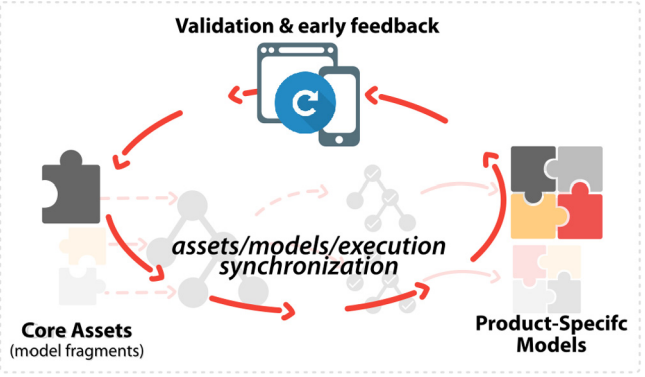

Fig. 7. The incremental and iterative development cycle within the modelling environment.

Feature modelling	The meta-model supports the creation of a Feature Model and the configuration of products.
Class modelling	The meta-model supports the creation of detailed Class model in a package hierarchy.
Feature/Element mapping	A relationship between an element and features can be defined at the meta-level and this relationship is navigable.
Elements overloading	Any element of the meta-model can be defined multiple time as long as it is distinguished by its associated features.

Fig. 8. The main concepts embodied in the CrossFabrik meta-model.

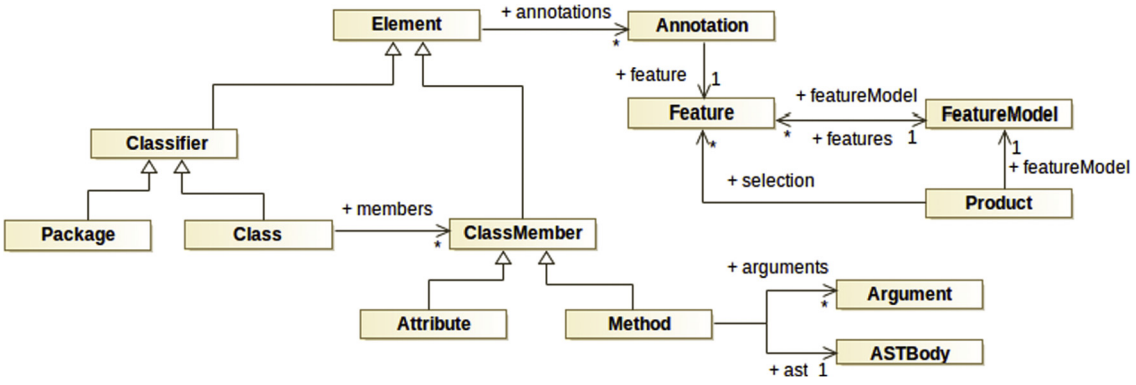

Fig. 9. The simplified CrossFabrik meta-model.

5.1. CrossFabrik meta-model

This section presents the CrossFabrik meta-model. The concepts embodied in this meta-model are summarized in Fig. 8. The goal is to define the core assets of an application and to specify the mapping between core assets and features.

Fig. 9 depicts the main elements of the CrossFabrik meta-model and the relation between core asset and feature modelling. This core meta-model is inspired by object-oriented meta-model such as EMOF [19] and ECore [22], especially for the description of the class structure. The originality relies on the four following aspects. First, any element can be associated with a set of annotations. The annotations are used to mark an element as part of the assets of one or several features. This creates a relationship between features and assets. Second, the meta-model supports overloading elements. The annotations are distinctive attributes of *Element*. For instance, unlike an EMOF Class, a *Class* element can contain several *Method* elements with exactly the same signature but with distinctive annotations in order to specify different behaviours dedicated to different devices. Third, high-level abstractions, subclasses of *Class*, are also proposed and described in Section 5.2. Fourth, the behaviour specification is managed in an abstract form and it is associated with a *Method* element through models named *ASTBody*. This abstract form typically consists in an interpretable Abstract Syntax Tree (AST) that can be produced from a given detailed action language. We observed that the notions of *Variable*, *Message*, *Message Receiver*, *Arguments* and *Return* are the minimum requirements to obtain the basis of an interpretable AST. By the use of such a kind of AST, the behaviour can be directly interpreted in the modelling environment and later partially translated to platform-specific AST using additional metadata (*e.g.* type information) in order to generate source code.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

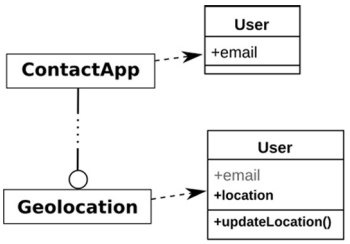

Fig. 10. The User model and associated features.

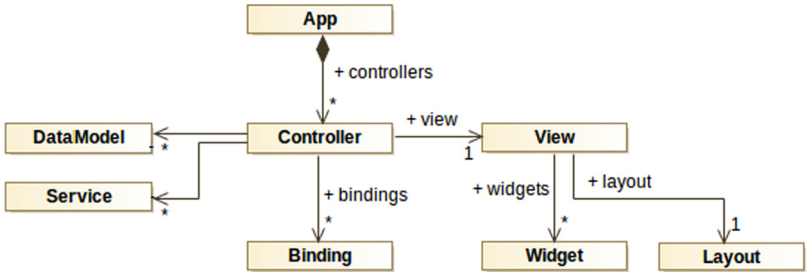

Fig. 11. Extract of the CrossFabrik meta-model for application modelling.

Thus, this meta-model provides abstractions allowing fine-grained design of applications in a platform-independent way. It includes information required for managing variability, for modelling structural and behavioural aspects of an application, and also the support for the behaviour execution and the code generation.

5.2. Modelling core assets

Once the FM_{app} is defined, the next step is to model core assets. As introduced in Section 4, core assets are defined in terms of model fragments. Such fragments are used to define the business logic and the graphical user-interface of the software variants. As introduced in Section 5.1, core assets are basically defined using *Package*, *Class*, *Attribute* and *Method* elements and all of these elements can be bound to features by means of annotations. From this basis, part of the architecture and the business models can be shared across software variants and specialized according to the business requirements and the targeted devices.

Fig. 10 depicts an example based on a *User* entity that represents the authenticated user in the *ContactApp* example. For all variants of the software family, a *User* is made of some attributes including his email. But, in our example, some specific additions have to be made into the *User* entity for the *Geolocation* feature. These additions make it possible for a user to be geolocalized. In this way, two additional properties are inserted in the *User* entity, namely the *location* field and the *updateLocation* method. These properties are annotated to be part of the *Geolocation* feature. With regard to the ContactApp example, only the desktop and mobile paid versions will benefit from these geolocation additions. To stick with this example, the *updateLocation* method might be only needed for the mobile device because the location of a desktop user should not change too much. To achieve this, the *updateLocation* method can be also annotated to be part of the *Mobile* feature. In this case only the paid version for mobile will include the *updateLocation* capability.

Based on these fine-grained modelling possibilities, CrossFabrik also provides high-level application-specific abstractions, subclasses of *Class*, that ease the definition of the software architecture and the user-interface. Applications expressed in CrossFabrik consist in the descriptions of model-view-controller (MVC) components. The MVC architecture is well suited to serve as an architecture abstraction for the major part of platform-specific frameworks and helps developers to structure applications.

As depicted in Fig. 11, the controller manages both the domain models, the services and the views. High-level abstractions, subclasses of *Binding*, are proposed to define how elements are bound together. Event listeners, data bindings, navigation, etc., are such a kind of high-level bindings. The View is the main abstraction responsible for the user interface. The view is used as a root panel which contains widgets arranged along a layout strategy. As for any other abstraction provided by the CrossFabrik meta-model, the widget layer has to stay platform-independent while providing the mean to be executed. The widgets in CrossFabrik are logical and rely on the native widgets provided by the modelling environment to be simulated.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

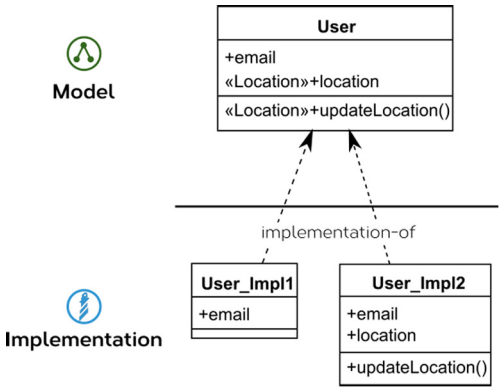

Fig. 12. The User model and its two implementations.

5.3. Validating software variants

The development cycle can be decomposed into two phases. First, the developer starts to model a software family by the means of feature, product and core asset modelling. Then, the developer can instantiate a defined product in a simulation context and can interact with it to validate and debug all or part of a software variant.

With CrossFabrik, the modelling environment is also used by the designer as the runtime environment for early validation. This lets developers explore new models or model changes with immediate and dynamic feedbacks. Moreover, this allows for a rapid prototyping style at the modelling phase. The modelling and validation phases take place in parallel. The execution of a software variant relies on the reflective capabilities provided by the modelling environment. Moreover, the execution has to take place in the same object space as the modelling tools to ensure the causal connection between meta-model instances and the executed artifacts.

In Section 5.1, we described how the structural and behavioural aspects of an application are defined using the CrossFabrik meta-model. To be executed, these abstractions must be interpreted by the modelling environment. On the one hand, we propose to generate code *in situ* and to inject it in the modelling environment to be later executed. The Class elements defined using the CrossFabrik meta-model are transformed on the fly as native classes of the modelling environment. We call these generated classes *the implementation classes*. On the other hand, we provide an adaptation layer between the CrossFabrik logical widgets and the native widgets provided by the modelling environment. During the validation process, this adaptation layer binds the logical widgets as the data model of the native ones.

In the context of FOMDD, one model can be implemented in a slightly different way depending on the product configuration. This create a one-to-many relationship between a model and its multiple implementations. Given the previous *User* entity example, Fig. 12 depicts two different implementations for the *User* class.

These two implementations have to be generated from the source model together with the specification of the product and have to stay synchronized. In currently available environments, after code generation, the implementation classes *User_Impl1* and *User_Impl2* are self-contained and are no longer linked with the related *User* model. This implies also a code duplication between the several implementations. In CrossFabrik, information duplication is avoided. An implementation class is associated with the model element that describes it. At runtime, this association allows all instances of an implementation class to have access to their related model. This means that information contained in the source model does not have to be duplicated in implementation classes. As an example, a method in an implementation class does not contain any source code. The source code of the method is only described once in the source model and it is dynamically used by instances of the implementation classes.

6. Implementation

In the previous sections, we explained the approach, the CrossFabrik main concepts and the requirements that the development environment together with the architecture of the infrastructure must meet. In this section, we present the key aspects of our first implementation of CrossFabrik using Pharo, a Smalltalk inspired language and environment.

6.1. Focus on the Pharo reflectivity

Pharo [23] is an object-oriented programming language highly influenced by Smalltalk. It is also an extensible and flexible programming environment. This environment is image-based and built around a Meta-Object Protocol (MOP) [24] that avoids the separation between the development and the runtime context. In Pharo, everything is an object. A class, an instance variable, a class variable (similar to a static attribute in Java) and a method namely *Class*, *Slot*, *ClassVariable* and *CompiledMethod*, are first-class objects which can be manipulated as any other object in the environment. A Class is therefore an instance of a MetaClass and updating its structural aspects is achieved by calling methods defined at the

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

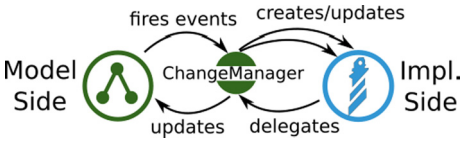

Fig. 13. The ChangeManager acts as a mediator object.

MetaClass level. Such methods can be considered as the default "meta" behaviour of classes and, as any methods, they can be overridden to customize this behaviour.

Using the reflective capabilities of Pharo, it is possible to dynamically change the structural aspects of a Class. Such a change affects immediately the running system (*i.e.*, all instances of the updated class). In the scope of a class, Slot and ClassVariable instances [25] are objects which contain the executable statements associated with getting (reading) and setting (writing) the instance and class variables. Slot and ClassVariable can be subclassed to provide a specific reading and writing behaviour. Regarding methods, any object can play the role of a CompiledMethod as long as it has the specific method named *run:with:in:*. Such an object is called a method wrapper or a method proxy [26]. The method *run:with:in:* takes three arguments: the original method name, the arguments and the object instance which own the method wrapper. In this way, the actual behaviour to be executed can be delegated to another object. To maintain the dynamic synchronization between CrossFabrik elements and generated implementation classes, the synchronization mechanism implemented in CrossFabrik relies on slots and method proxies. The implementation of this synchronization mechanism is presented in the next section.

6.2. A runnable business behaviour

The difficulty in executing models resides in the gap between models and implementation details. The main issue is the representation of a runnable business behaviour within models. As introduced in Section 5.1, we propose to extend our meta-model using AST nodes which can be presented in the form of a concrete syntax.

In our implementation, we opted for the Pharo AST. Pharo provides a minimal syntax which can be represented by about ten different node types [27]. These node types include our minimal requirements for an interpretable AST: *Variable*, *Message*, *Message Receiver*, *Arguments* and *Return*. Contrary to mainstream programming languages such as C++, Java or Python, Pharo has only 5 reserved words (nil, true, false, self and super), no control structure statements, no built-in types and no operators. This makes Pharo AST an attractive target for language transformation from an arbitrary action language. The main benefit of using the Pharo AST is that it can be directly manipulated and executed in the Pharo environment. This choice was also motivated by the possibility to reuse the Pharo concrete syntax, the runtime and its infrastructure.

6.3. The synchronization implementation

We use an internal on the fly code generation approach to produce the implementation classes introduced in Section 5.3. Implementation classes are materialized as native Pharo classes in the runtime environment. Pharo native classes are generated from core assets and loaded in the runtime environment on demand. In the rest of this paper, we call *Pharo classes* the generated implementation classes to clarify the distinction between a model and its executable representations in the runtime environment. A model may have several implementation variants according to the features set. For one model, a Round-Trip Engineering (RTE) process has to keep multiple Pharo classes synchronized. As any RTE, our implementation aims to automatically manage forward and reverse engineering:

- First, the forward engineering capability. Any change in a model has to be automatically applied to related Pharo classes.
- Second, the reverse engineering capability. Any change in a Pharo class has to be automatically reflected in the source model.

Due to the one-to-many relationship, implementing the reverse engineering can be tedious. A change in a particular Pharo class can also impact other Pharo classes generated from other products. Moreover, reverse changes have to be limited, controlled and validated before being applied to models. Any ambiguity must be resolved and improper changes must be rejected in order to preserve the model integrity. The forward and the reverse engineering are implemented by a mediator object named *ChangeManager*. The role of the ChangeManager is depicted in Fig. 13:

- Forward engineering involves generating Pharo classes from a source model. This mechanism is event-based. Events are emitted each time a source model is updated. Then, the ChangeManager reacts by propagating changes in the related Pharo classes. The ChangeManager takes care of the one-to-many relationship between the source model and the Pharo classes.
- Reverse Engineering is change-based. It relies on the redefinition of the default meta behaviour of the generated Pharo classes. Indeed, a Pharo class is an object and it provides reflective facilities, also called meta behaviour, in order to

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

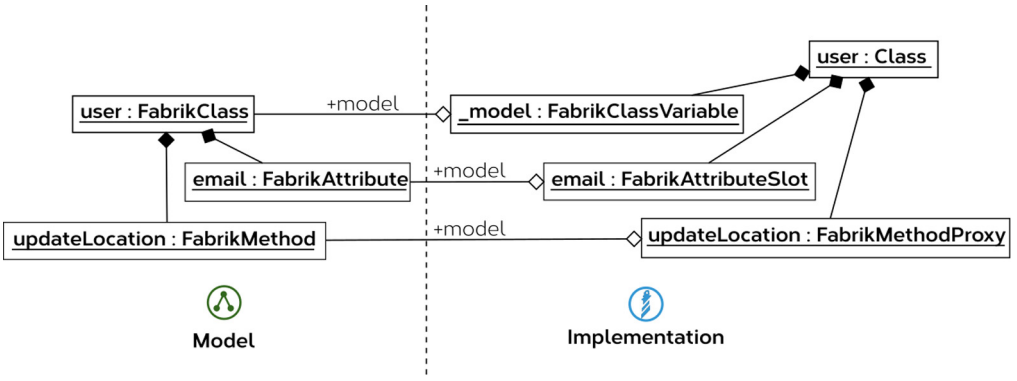

Fig. 14. User model instance (left) with related meta-entities (right).

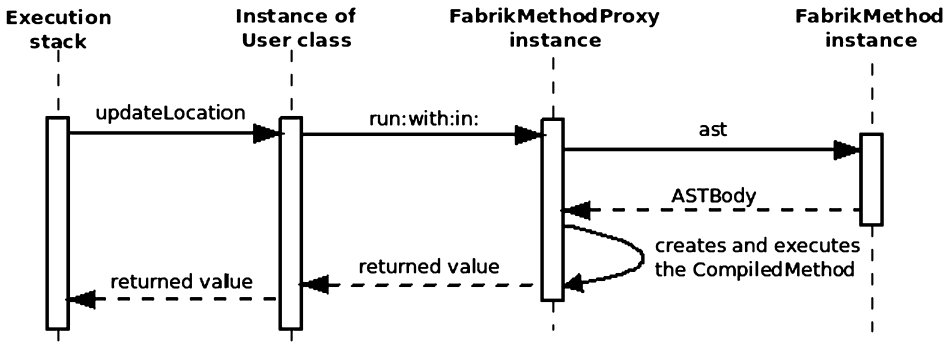

Fig. 15. Sequence diagram to execute the *updateLocation* method of a *User* instance.

update its own structure and behaviour. The reverse engineering is based on the overriding of this meta behaviour. For instance, when a generated Pharo class is asked to add a new attribute, its default meta behaviour is prevented and the class is not actually updated. Instead, the ChangeManager is asked to update the related source model.
Sometime, a change is ambiguous. The relationship between the change to propagate and the features which have to be associated with it cannot be inferred automatically. For example, when a method associated with several features is updated, we cannot predict if one or all features are targeted. Thus, in case of an ambiguous change, the developer is warned by the ChangeManager and he can choose the way the models have to be updated. Hence, instead of having read-only Pharo classes, the reverse engineering allows developers to indirectly update models through the Pharo classes. This capability allows the transparent use of the native tools of the Pharo environment (*e.g.* code browser, inspector and debugger).

6.3.1. The forward engineering

The forward engineering takes place each time a source model is updated. When a CrossFabrik element is changed, an event is fired by the element. The ChangeManager reacts to these events by generating or updating related Pharo classes. In order to implement the RTE, CrossFabrik has to manage the way a class is installed in the system and has to change the default behaviour. For that purpose, CrossFabrik implements its own meta-entities to be used in place of Slot, ClassVariable and CompiledMethod. The use of these CrossFabrik meta-entities for our User example is depicted by Fig. 14. Instead of using a Slot, the mail attribute is implemented through an instance of *FabrikAttributeSlot*. Instead of using a CompiledMethod, the updateLocation method is implemented though an instance of *FabrikMethodProxy*. Instead of using a ClassVariable, the User source model is referenced by a *FabrikClassVariable* instance. All these meta-entities hold a reference to the CrossFabrik elements.

The running of a software variant relies on a proxy mechanism from the Pharo class to the source model. For each generated Pharo classes, methods are replaced by *FabrikMethodProxy* instances. A *FabrikMethodProxy* instance does not contain any source code and executing the method body relies only on the source code specified in the source model. Fig. 15 indicates the flow of execution when the *updateLocation* method of a *User* instance is called. The role of *FabrikMethodProxy* instance is to retrieve the AST nodes on demand from the source model. From the AST nodes, a CompiledMethod is created on the fly and executed to return the expected result.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

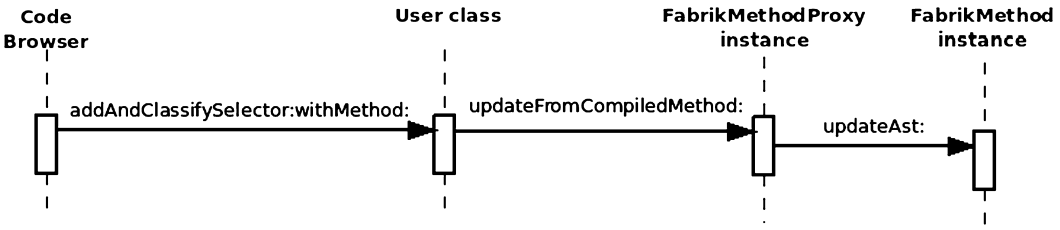

Fig. 16. Sequence diagram to update a method of the User model from a related Pharo class and using the native code editor.

6.3.2. The reverse engineering

To perform the reverse engineering, the ChangeManager has to be aware of changes applied in generated Pharo classes. In Pharo, modifying structural aspects of a class as well as modifying body of methods is managed by the relevant Class instance as part of its meta behaviour. This meta behaviour can be overridden to implement a new update strategy directly in generated Pharo classes. Two kinds of changes have to be intercepted and propagated at the model side: the structural changes denoted by the addition/deletion of instance variables or of methods of an object and the behavioural changes denoted by the modification of a method's body.

In Pharo, all structural changes are first handled by the class and finally delegated to the *PharoClassInstaller*. In CrossFabrik, the PharoClassInstaller is not directly used by generated Pharo classes. Instead, the ChangeManager acts as a mediator between the generated classes and the PharoClassInstaller.

When an instance variable is added, the ChangeManager is asked to update the related source model. It first checks the validity of the change, then it applies the change on the related source model and finally the forward engineering takes place.

In Pharo, the methods management is under the responsibility of the class itself. As depicted in Fig. 16, when a method is compiled, the class method named *addAndClassifySelector:withMethod:* is invoked to perform the actual changes. The term *selector* is the technical term to define the name of a method. The method *addAndClassifySelector:withMethod:* takes the selector and the related CompiledMethod as arguments and add the CompiledMethod in the method dictionary of the class using the selector as the key to identify it. In CrossFabrik, this method is redefined to delegate method changes to the ChangeManager. Fig. 16 depicts how the class method *addAndClassifySelector:withMethod:* is redefined. In the case of the update of an existing method, the AST of the source model is directly updated. In the case of the addition or the removal of a method, the request is forwarded to the ChangeManager in order to perform relevant changes on the source model.

6.4. Editing models

The CrossFabrik meta-model has been implemented using the Pharo language which makes the Pharo infrastructure fully reusable with CrossFabrik. This allows us to basically create and manipulate instances of this meta-model through the native Pharo tools. However, to ease the edition of source models and the debugging of generated classes, a set a tools has been specifically implemented. These tools are presented in this section.

6.4.1. Model browser and editor

Dedicated tools including editors and a specific declarative language were implemented to manipulate models. CrossFabrik provides an Integrated Development Environment (IDE) for that purpose.

Fig. 17 shows the CrossFabrik IDE for models editing. It is mainly composed on the left by a package browser which presents the hierarchy of models. From the package browser, *Package* and *Class* elements can be created, organized and opened. Tabulations are presented on the center. They show the details of a *Class* element from which *Attribute* and *Method* elements can be edited. From the package browser and related tabulations, any element can be associated with features. The pop-up window at the bottom right shows a view to select a feature from the Feature Model currently in use. The developer can also open a model in a textual mode. Instead of the default model editor made by a set of panes, a text editor is opened that contains the textual representation of the selected model. This textual representation is based on a Domain-Specific Language (DSL) embedded in the Pharo language which makes it fully compliant with native Pharo tools. This embedded DSL is based on the builder pattern. Each statement constructs a particular instance of the CrossFabrik meta-model and pushes it in the parent instance previously built. This parent-child hierarchy is materialized by a block closure (*i.e.* the square brackets in the Smalltalk syntax).

As an example, the Fig. 18 shows the DSL syntax used to describe the User model. First, a Class element is instantiated by calling the class method *named:def:* of the *CfClass* builder. The first argument is the name of the class we want to define and the second argument is a block closure which contains the declaration of the class body. The class body contains statements which construct the attributes and the methods of the Class element. The relation between the features and the model instances is declared using an annotation syntax, by calling the @ unary method of the CfFeature builder with the name of the targeted feature as an argument.

Using the Pharo language as the host language for our syntax allows us to directly evaluate our DSL within any Pharo editor. We use this capability to use the DSL even during debugging.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

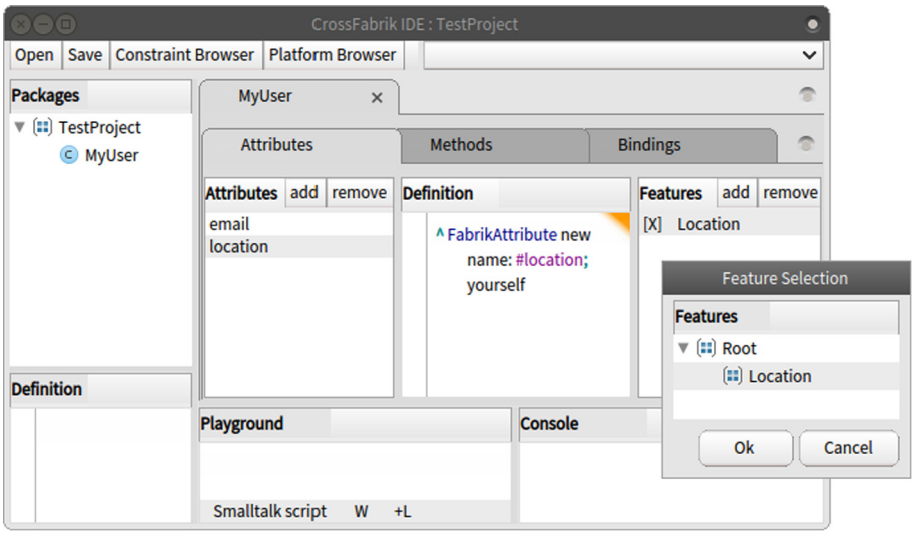

Fig. 17. The CrossFabrik IDE.

```
CfClass named: #User def:[
    CfAttr named: #email.

    CfFeature@#Geolocation.
    CfAttr named: #location.

    CfFeature@#Geolocation.
    CfMeth named: #updateLocation args: #() def: [
      "method body here"
    ].
].
```

Fig. 18. The declarative syntax used to describe the user model.

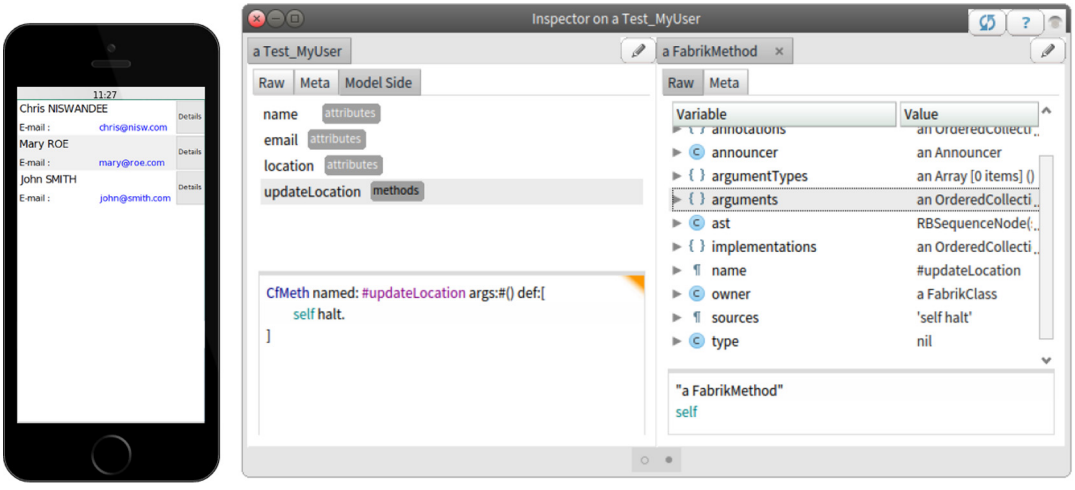

Fig. 19. The debugging of one User implementation during a mobile simulation.

6.4.2. Model inspector and debugger

Pharo provides the *GTDebugger* [28] that offers a way for developers to adapt visualization to new domains and scenarios. It is object-centric and retrieves the relevant visualizations directly from the inspected object. Thus, the debugger can be dynamically adapted when a particular inspected object contains methods that describe new visualization to use.

As an example, Fig. 19 shows a running mobile simulation to the left which displays a list of contacts. To the right, Fig. 19 shows the debugging visualization from the first item of the list.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

The CrossFabrik debugging visualization is composed of two columns. The first column, in addition to allowing inspection of the current state of the inspected object, allows access to the attributes and methods resulting from the related model. When an item is selected in the list, the related source model (*i.e.* the *Attribute* or *Method* element) is displayed below the list in a code editor using the CrossFabrik textual representation. The second column displays the structure and the current state of this source model. As any other inspected object, the source model can be manipulated and edited.

In Fig. 19, the method named *updateLocation* is selected and its body can be directly edited within the code editor below the list. Updating the textual representation of the *updateLocation* method means updating the *Method* model displayed in the second column and, consequently, updating the current running object which reference this *Method* model.

7. Related work

In this section, we discuss related work on Model-Driven approaches, on model executability and on dynamic environments. First, we focus on FOD and FOMDD approaches applied to cross-platform development and we highlight the contribution of our work with regard to the early functional validation. Then, we discuss executable models and we address the causal relationship and the reuse of native tools. Finally, we describe different Round-Trip Engineering processes and we expose how the CrossFabrik RTE applied to FOMDD differs from them.

7.1. Model-driven approaches

Albassam and Gomaa [29] present an application of variability modelling in the context of the video games domain in order to take advantage of the unique features of each platform and/or to address limitations. Based on their PLUS method that combine SPL and UML, they exploit feature modelling to express the different platform-specific capabilities and features. They use this variability as the basis for designing a component-based SPL architecture for multi-platform video games. They focus mainly in generating hardware-specific variants of video games using code generation.

Quinton et al. [30] present ApplIDE, an FOMDD approach to produce mobile software variants for multiple platforms. They propose to use two separated feature models, one for describing the application variability and the other for describing the device variability. To be able to determine on which devices an application product is able to run, a mapping is made between some functional features from the application and device features. Core assets are implemented using a meta-model specific to the mobile application domain that allows to model the business logic and the user-interface in a platform independent way. Restricted parts of the software behaviour can be described using the notion of action which are triggered by events. The proposed approach relies on code generation to produce platform-specific executable software.

Usman et al. [31] presents another FOMDD approach applied also to the mobile specific area. Instead of using two separated feature models as in [30], they propose a generic feature model that presents the variations for the mobile application domain specific concepts. From this generic feature model, device products can be defined and used as the basis for an extended application feature model. Core assets are implemented through UML modelling profiles and notations. Some parts of the business logic can be described using use-case diagrams and the definition of the user-interface relies on external platform-specific UI builders. As the previous approaches, a working software can be produced only after a platform-specific code generation process.

Our approach is close to the one proposed in [30]. We rely on a specific application feature model associated with a meta-model for implementing core assets which include the business-logic and the user-interface. Our approach aims to be more generic to take into account the business-specific as well as the device-specific functional variations in a software family. We argue that taking into account the non-functional aspects can be postponed to the end of the prototyping and development cycle when most of the software requirements are validated. Furthermore, none of the discussed approaches allows the fine-grained specification of the business logic and the early validation of the software variants without being forced to generate platform-specific software.

7.2. Model executability

The CrossFabrik approach is close to EMF [22]. Both approaches implement a MOF-oriented kernel in a general purpose language. The main difference is that EMF is geared towards code generation. To be run, the code of an application model must be generated and executed as a standalone program. In our approach, an application model is instrumented to be fully executed and debugged within the development environment.

The UML virtual machine and the runtime model for fUML [32,33], make it possible the interpretation of an application model within the modelling environment. However, as far as we know, the causal connection is not maintained at runtime. With CrossFabrik, structural as well as behavioural changes of the application model have immediate effects during the execution of a model.

Kermeta [34] use an executable meta-language close to EMOF. A dedicated language is provided to specify operational semantics of meta-models [35]. In addition, following an aspect oriented approach, the business behaviour is included as part of the runnable application model thanks to a weaving process. In this approach, the tools from the underlying development environment are not suitable and the development and the debugging tools have to be specifically implemented.

In our approach, application model can be early executed within the development environment and the native tools are adapted to be used during modelling and debugging phases.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

7.3. Round-trip engineering approaches

As far as we know, CrossFabrik is the only environment providing dynamic debugging and RTE support in the context of FOMDD.

In [36], a Change-Oriented Advanced Round-Trip Engineering (COARTE) is envisioned. In COARTE, changes are considered as first-class objects and the entire set of "changes" objects represents the complete history of a software system. As in COARTE, CrossFabrik uses a change-oriented RTE. Our implementation relies on an event model and on the change mechanism provided by Pharo. But we do not maintain the history of changes made. Changes cannot be stored and replayed.

COARTE also provides multiple views on a software system with the distinction between static and runtime views and between design and implementation views. In CrossFabrik this distinction is also made. The CrossFabrik IDE depicts how a software is designed whereas the Pharo code browser and the debugger depict respectively how a software is statically implemented and how implementation instances live in the system.

But COARTE does not address the issue of the synchronization of multiple implementations from the same model. In CrossFabrik, the first requirement regarding the RTE is to maintain this one-to-many relationship. As far as we know, COARTE does not allow such a kind of synchronization.

SelfSync [37] is an approach and a set of tools for providing dynamic support for RTE in the context of Entity-Relationship diagrams. SelfSync is implemented in Self, a prototype-based dynamic object-oriented programming language, environment, and virtual machine. In SelfSync, the modelling and the execution take place within a unique environment. A source model and the corresponding implementation object are *one and the same*. Both share the same structure and the same behaviour namely its *prototype* and its *traits*. In CrossFabrik the modelling and the execution also take place within a unique environment at design and running time. As SelfSync, CrossFabrik consists in a dynamic support for RTE, synchronizing a model and its implementation parts even at runtime. In CrossFabrik, the behaviour is also shared between source models and implementation classes. It is made of AST nodes that can be produced at design time and executed during the assessment of implementation classes.

Our solution differs from SelfSync in the management of the relationship between model and implementation parts. In CrossFabrik, because a source model may have several implementations, the same object cannot be used, implementation classes have to be generated and a one-to-many relationship has to be maintained. Such a synchronization cannot be achieved with SelfSync. Moreover, SelfSync does not support a runtime RTE in the reverse direction [38]. In CrossFabrik, we provide such a reverse support especially during the debugging process. From an instance of an implementation class, the source model can be inspected and updated. Such updates automatically affect the source model, its implementation classes and their instances currently in use.

8. Conclusion

This paper addresses the issue of early validation when several software variants have to be envisaged. Such variants can basically differ in the proposed features and in the targeted device. This brings new challenges in the early validation of software and the reuse of software artefacts.

In this paper, we presented CrossFabrik, a Feature-Oriented Model-Driven framework and environment which allows debugging and early validation of feature-based applications. Typically, validating a software in the context of FOMDD involves generating the final software from models and executing it separately from models. The CrossFabrik synchronization process manages a one-to-many relationship between the source models and the implementation parts, even at runtime.

We implemented our approach using the Pharo environment that provides us with an image-based virtual machine where our modelling tools and the simulation process take place. Regarding our synchronization implementation, changes propagation is supported in the forward direction as well as the reverse direction and includes structural and behavioural changes. Such changes can be achieved statically or dynamically during the validation process. Regarding the reuse of the Pharo infrastructure, we presented how an existing development environment can be adapted and extended to allow custom visualization and dynamic debugging of source models during execution of implementation parts.

References

[1] M.E. Joorabchi, A. Mesbah, P. Kruchten, Real challenges in mobile app development, in: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 15–24.

[2] D. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, Portland, Oregon, 2003, pp. 187–197.

[3] S. Kent, Model driven engineering, in: M. Butler, L. Petre, K. Sere (Eds.), Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15–18, 2002 Proceedings, in: LNCS, vol. 2335, Springer, Berlin, Heidelberg, 2002, pp. 286–298.

[4] M. Anastasopoulos, T. Forster, D. Muthig, Optimizing model-driven development by deriving code generation patterns from product line architectures, NetObject Days.

[5] S.T. Gonzalez, Feature Oriented Model Driven Product Lines, Ph.D. thesis, University of the Basque Country, 2007.

[6] J. Miller, J. Mukerji, MDA guide version 1.0.1, 2003.

[7] A.M. Davis, Software prototyping, in: Advances in Computers, vol. 40, 1995, pp. 39–63.

[8] A. Forward, O. Badreddin, T.C. Lethbridge, UMPLE: towards combining model driven with prototype driven system development, in: Proceedings of the International Workshop on Rapid System Prototyping, IEEE, 2010, pp. 1–7.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001

[9] J. Fu, F.B. Bastani, I.-L. Yen, Model-driven prototyping based requirements elicitation, in: LNCS, vol. 5320, 2007, pp. 43–61.
[10] G. Cavarlé, A. Plantec, S. Costiou, V. Ribaud, Dynamic round-trip engineering in the context of fomdd, in: Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies, IWST'16, ACM Press, New York, New York, USA, 2016, pp. 1–7.
[11] M.J. Harrold, Testing: a roadmap, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, 2000, pp. 61–72.
[12] P. Berander, L.-O. Damm, J. Eriksson, T. Gorschek, K. Henningsson, P. Jönsson, S. Kågström, D. Milicic, F. Mårtensson, K. Rönkkö, P. Tomaszewski, L. Lundberg, M. Mattsson, C. Wohlin, in: C.W. Lars Lundberg, Michael Mattsson (Eds.), Software Quality Attributes and Trade-offs, Blekinge Institute of Technology, 2005, pp. 1–100.
[13] B. Boehm, V.R. Basili, Software defect reduction top 10 list, Computer 34 (1) (2001) 135–137, https://doi.org/10.1109/2.962984.
[14] A. Rodrigues Da Silva, Model-driven engineering: a survey supported by the unified conceptual model, Computer Languages, Systems & Structures 43 (2015) 139–155, https://doi.org/10.1016/j.cl.2015.06.001.
[15] S. Apel, C. Kastner, An overview of feature-oriented software development, J. Object Technol. 8 (5) (2009) 49–84, https://doi.org/10.5381/jot.2009.8.5.c5.
[16] P.C. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison–Wesley, 2001.
[17] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Distribution 17 (November 1990) 161, https://doi.org/10.1080/10629360701306050.
[18] K. Lee, K.C. Kang, J. Lee, Concepts and guidelines of feature modeling for product line software engineering, in: Software Reuse Methods Techniques and Tools, vol. 2319, 2002, pp. 62–77.
[19] Object Management Group, OMG meta object facility (MOF) core specification version 2.4.1, 2013.
[20] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe, The architecture of a UML virtual machine, in: Environment, vol. 36, 2001, pp. 327–341.
[21] M. Eisenstadt, My hairiest bug war stories, Commun. ACM 40 (4) (1997) 30–37, https://doi.org/10.1145/248448.248456.
[22] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, 2nd edition, Addison–Wesley Professional, 2008.
[23] O. Nierstrasz, S. Ducasse, D. Pollet, Pharo by Example, Square Bracket Associates, 2010.
[24] G. Kiczales, J.M. Ashley, L. Rodriguez, A. Vahdat, D.G.D. Bobrow, Metaobject protocols: why we want them and what else they can do, in: Object-Oriented Programming: The CLOS Perspective, 1993, pp. 101–118.
[25] T. Verwaest, C. Bruni, M. Lungu, O. Nierstrasz, C. Bruni, Flexible object layouts: enabling lightweight language extensions by intercepting slot access, in: OOPSLA, 2011, pp. 959–972.
[26] M.M. Peck, N. Bouraqadi, L. Fabresse, M. Denker, C. Teruel, S. Ducasse, Ghost: a uniform and general-purpose proxy implementation, Sci. Comput. Program. 98 (2015) 339–359, https://doi.org/10.1016/j.scico.2014.05.015, arXiv:1310.7774.
[27] L. Renggli, Dynamic Language Embedding With Homogeneous Tool Support, Ph.D. thesis, University of Bern, 2010.
[28] A. Chis, O. Nierstrasz, T. Girba, Towards a moldable debugger, in: Proceedings of the 7th Workshop on Dynamic Languages and Applications, DYLA '13, 2013, pp. 1–6.
[29] E. Albassam, H. Gomaa, Applying software product lines to multiplatform video games, in: Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change, IEEE Press, San Francisco, California, 2013, pp. 1–7.
[30] C. Quinton, S. Mosser, C. Parra, L. Duchien, Using multiple feature models to design applications for mobile phones, in: MAPLE/SCALE Workshop, Colocated with SPLC'11, Munich, Germany, 2011, pp. 1–8.
[31] M. Usman, M.Z. Iqbal, M.U. Khan, A product-line model-driven engineering approach for generating feature-based mobile applications, J. Syst. Softw. 123 (2016) 1–32, https://doi.org/10.1016/j.jss.2016.09.049.
[32] M.L. Crane, J.J. Dingel, Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities, in: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research Meeting of Minds, CASCON '08, 2008, p. 8.
[33] T. Mayerhofer, P. Langer, G. Kappel, A runtime model for fuml, in: Mrt@Runtime, 2012, pp. 53–58.
[34] J.-M. Jézéquel, O. Barais, F. Fleurey, Model driven language engineering with Kermeta, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), in: LNCS, vol. 6491, 2009, pp. 201–221.
[35] P.-A. Muller, F. Fleurey, J.-M. Jézéquel, Weaving executability into object-oriented meta-languages, in: Model Driven Engineering Languages and Systems, October 2005, pp. 264–278.
[36] P. Ebraert, E.V. Paesschen, Change-oriented round-trip engineering, in: ICDL '07 Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007, Lugano, Switzerland, 2007, pp. 3–24.
[37] E. Van Paesschen, M. D'Hondt, W. De Meuter, Rapid prototyping of extended entity-relationship models, in: ISIM 2005, Czech Republic, 2005, pp. 194–209.
[38] E. Van Paesschen, M. D'Hondt, Selfsync: a dynamic round-trip engineering environment, in: Satellite Events at the Models 2005 Conference, vol. 3844, 2006, pp. 347–352.

Please cite this article in press as: G. Cavarlé et al., A feature-oriented model-driven engineering approach for the early validation of feature-based applications, Sci. Comput. Program. (2018), https://doi.org/10.1016/j.scico.2018.01.001