Platform-Variant Applications from Platform-Independent Models via Templates

Nuno Amálio[1] Christian Glodt[2] Frederico Pinto[3]
Pierre Kelsen[4]

University of Luxembourg
6, r. Coudenhove-Kalergi, L-1359 Luxembourg

Abstract

By raising the level of abstraction from code to models, model-driven development (MDD) emphasises design rather than implementation and platform-specificity. This paper presents an experiment with a MDD approach, which takes platform-independent models and generates code for various platforms from them. The platform code is generated from templates. Our approach is based on EP, a formal executable modelling language, supplemented with OCL, and FTL, a formal language of templates. The paper's experiment generates code for the mobile platforms Android and iPhone from the same abstract functional model of a case study. The experiment shows the feasibility of MDD to tackle present day problems, highlighting many benefits of the MDD approach and opportunities for improvement.

Keywords: Software Product families, model-driven development, executable models, templates.

1 Introduction

A goal of model driven development (MDD) [21] is to enable software engineers to focus on design. This is achieved through the use of models expressing design concepts that abstract away from implementation and platform-specific details. Despite the increase in level of abstraction of programming languages and platforms in the past two decades, the diversity and complexity of current

[1] Email: nuno.amalio@uni.lu
[2] Email: christian.glodt@uni.lu
[3] Email: fgasparp@gmail.com
[4] Email: pierre.kelsen@uni.lu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

platform technologies makes manual development of code an arduous and expensive effort [21]. Modern platforms require considerable in-depth technical knowledge that is difficult to grasp by non-expert developers, a prominent example being mobile devices [15,14,1]. This cuts off an important source of creativity: talented people may be inspired to create novel applications, but only few have the time, energy and technical skill to dig into the intricacies of low-level platform programming. The problems of platform complexity and diversity suggest a move to a higher level of abstraction. However, novel features of modern devices require implementations to properly evaluate design decisions. Interaction [15,14,1], performance, and power consumption [22], common in mobile computing, are difficult to analyse from abstract models; they require experimentation with implementations.

In MDD, these issues can be tackled through a model-centric approach: all lower level code is generated from functional models of the system, also called *platform-independent models* (PIMs) in the *model-driven architecture* (MDA) [18], which is possible provided models fully describe the system's structure and behaviour. This approach tackles platform complexity and diversity, and enables the early construction of implementations from design models. Due to their level of abstraction, models can be articulated to describe families of related products, abstracting away from many intricacies of execution platforms. From such models, it is possible to build reusable transformations that enable the derivation of platform-variant products. Finally, from models and transformations to code, it is possible to obtain prototypes for experimentation of design decisions.

This paper presents an experiment with our MDD approach based on executable modelling and templates. Our approach enables the generation of code for various platforms from the same functional model. It is as follows:

- Applications are described using PIMs, describing structure and behaviour. PIMs are expressed in terms of abstract design primitives, yet concrete enough to enable generation of multi-platform code from them. PIMs' level of abstraction mitigates the need for platform expertise.
- Platform-specific artifacts are generated by instantiating templates of the platform's catalogue. The choice for the alternative execution platforms is a variation point in a product family, where variants are obtained automatically through code-generation by instantiating templates. Catalogues of templates constitute a repository of knowledge that is maintained by platform experts.

The approach presented here is based on formal languages: (a) models are expressed using the executable modelling language EP [16,17] supplemented with OCL [23], (b) catalogues of templates are expressed using the Formal Template Language (FTL) [4,2]. This approach gives generation of platform

artifacts a first-class status: generative reusable assets are described in FTL. The experiment presented here evaluates this approach using a present day problem: building mobile-applications that have the same functionality, but need to run on different execution platforms. This is illustrated with Google's Android and Apple's iPhone mobile platforms.

2 Background

We give some background on both EP, the language used to express abstract models, and FTL, the language used to express templates.

2.1 EP

EP [16,17] is a formal modelling language designed to express executable models visually. EP expresses both structure and behavior of a system. An EP model is structured around *classes*; each class comprises *properties*, *queries* and *events*, which are supplemented with OCL textual descriptions. Appendix A provides several definitions of EP classes; the different element types are distinguished with letter decorations: 'P' denotes properties, 'Q' denotes queries, and 'E' denotes events.

Properties represent structural features of EP classes. A property comprises a name, a type and an initialisation (described in OCL). Class `Book` of Fig. A.1a (p. 23) defines properties `bookId`, `title`, `isbn`, `authors` and `copies`. Initialisation of `authors`, for instance, is defined by the OCL expression:

```
Set {}
```

This initialises the set of authors to the empty set.

Queries retrieve information from some class (semantically, a query property is a mathematical function). A query comprises a name, parameters, a type, and a OCL definition. For instance, the OCL that defines the query `existsMember` of class `Library` (Fig. A.1e) is:

```
members->exists (m: Library::Member |
   m.libraryNo = libNo and m.password = pw)
```

Events describe how the state of objects of a class change. An event comprises several event edges, parameters and a guard (an OCL predicate), which specifies a condition for executing the event: if the guard is true then the event edges are executed; otherwise the event does nothing. There are three types of event edges: *impact*, *pull* and *push*. An impact edge says how an event changes the state of some class's property For instance, the event `setCopies` of class `Book` has an impacts edge with the property `copies` (impact edges are represented as filled arrows); the OCL text of this impacts edge is:

```
copies
```

This sets the property to the parameter `copies` of the event `setCopies`.

A push edge is like an event call, whereby another event is called. It includes a guard, which specifies a condition for calling (or pushing) the event. The pushed event can either be local or an event of some instance for which there is a local reference. For instance, the event `reserve` of class `Copy` (Fig. A.1c) pushes the events `reserveOk` and `reserveFailed`; each have a guard; the parameter maps for each pushed event is described with an OCL expression.

A pull edge defines event triggers; it says that the current event is triggered when some other event (as defined in the pull edge) is also triggered. Like push edges, pull edges also include a guard. For instance, the event `searchClicked` of class `SearchController` (Fig. A.5a) pulls the event `clicked` of class `Button` (Fig. A.7b).

Global system behaviour is a chain of events. To tackle complexity, EP modes are divided into *domains*, representing different subject matters. Domains are self-contained; they contain a collection of EP-classes that do not have references external to this domain. Realistic systems, however, are made of various domains that have to interact. In EP, this implies behaviour propagation from one domain into another. EP links domains using *bridges*. Unlike domains, bridges can include EP classes with external links to the domains being linked. Through these external links, events can be propagated across domains. A EP model divided into bridges and domains is given in Fig. 6.

Democles [5] [12] is the tool supporting the EP language. EP-models can be used to describe a platform-independent model of a system [12,11]. This paper shows how EP can describe a family of related systems, and presents a code-generation strategy for EP based on templates.

2.2 FTL

The Formal Template Language (FTL) [4,2] is a formal language for expressing templates of any target textual language. It has a general mathematical semantics, not being bound to any platform or execution environment. FTL is generative; it describes sentences of some target language (here Java and Objective-C) and generates sentences when provided with an instantiation.

FTL's original definition [4,2] included constructs for placeholders, lists and choice. The version of FTL used here adds naming of templates and modularity. An FTL module comprises a set of templates; modules can import other modules. Despite these changes, the semantics of the language is essentially the one defined in [4,2]. We have built a Java implementation of FTL, which has been integrated in Democles [12].

[5] http://democles.lassy.uni.lu/

```
module JavaClassCat
Class ==
   public class ⩽ClName⩾{
      ⟦private ⩽propName⩾ : ⩽propTy⩾;⟧
      ⟦↑ClassMethod⟧
   }■
```

```
public class Book {
   private title : String;
   private isbn : String;
}
```

Fig. 1. FTL template of a Java class (left) and a sample instantiation of the template (right).

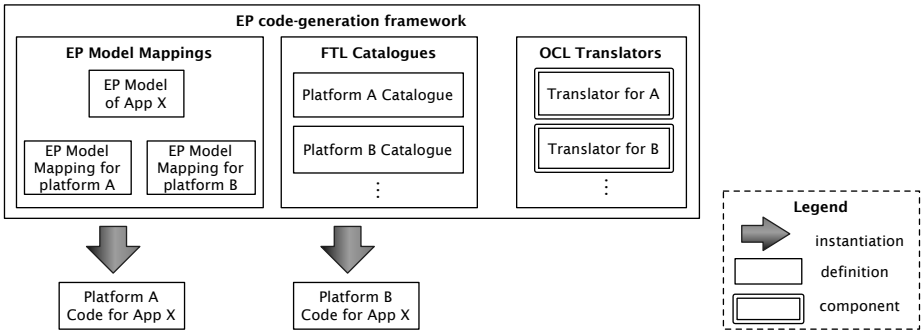


Fig. 2. Platform-specific code is generated from EP Models by instantiating FTL templates of a platform catalogue.

To illustrate FTL, consider the FTL catalogue of Fig. 1 (left). This defines template module `JavaClassCat`, containing templates to describe a Java public class, comprising a number of java private attributes and methods. The catalogue is made of the templates `Class` and `ClassMethod` (not shown). `Class` includes placeholder `ClName` (placeholders are placed between ⩽ ⩾), representing the name of the class to generate, and two FTL lists for attributes and class methods (lists are placed between ⟦ ⟧). The first list includes two placeholders representing the attribute's name and type. The second list includes a template reference (symbol ↑) to another template (not shown here). The template can be instantiated to give the Java class `Book` of Fig. 1 (right).

3 Overview of the Approach

The MDD approach explored in this paper is sketched in Fig. 2. The EP code generation framework comprises three main components: *EP Model Mappings*, *FTL Catalogues* and *OCL Translators*. They are as follows:

- The backbone of the generation infrastructure is formed by the FTL template catalogues targeting different execution platforms (Fig. 2, inner middle box). Platform specific-code is generated from these template catalogues.
- EP PIMs need to be mapped to platform templates before generation of platform code. This involves building a platform mapping for each EP PIM (Boxes `EP Model Mappings` in Fig. 2), which is partially specified by the user. Generation is based on instantiating the templates with information

```
// Catalogue of FTL templates
// for the Android platform
module AndroidCat

import CoreJava
import AndroidGUI
import AndroidPersistence
import AndroidDateSupport
import JavaStringUtils
import AndroidSupport
```

```
// Catalogue of FTL templates
// for the iPhone platform
module iPhoneCat

import CoreObjC
import iPhoneGUI
import iPhonePersistence
import iPhoneDateSupport
import ObjCStringUtils
```

Fig. 3. FTL main catalogues of Android (left) and iPhone (right) platforms

coming from the EP model, according to what is defined in the mapping.

- EP-based PIMs contain OCL code snippets. These are translated into the platform language by resorting to OCL translators (see box `OCL translators` Fig. 2). These translators yield platform language text that is used to instantiate the templates of the catalogue.

The process of building applications using this approach is as follows:

(i) Application developers build EP application models for a family of applications using platform-independent concepts.

(ii) Platform technical experts develop platform catalogues of FTL templates. The process of developing templates involves consulting application designers to know what platform constructs they need.

(iii) Application developers and platform experts then work together to build a platform mapping for the EP model.

(iv) From the EP model and its platform mapping, platform-specific applications can be generated by instantiating templates of the platform's catalogue.

4 FTL Template Catalogues for iPhone and Android

To support the experiment presented here, we have built catalogues of FTL templates for Android and iPhone platforms (available at `http://democles.lassy.uni.lu/`). These catalogues have been structured using FTL's modularity constructs. Figure 3 presents the main FTL catalogues of Android and iPhone, defining modules `AndroidCat` and `iPhoneCat` that import several subcatalogues (modules). These template modules support code-generation: the *core* modules define core constructs for Java (common to all Java-based platforms) and Objective-C; *GUI* modules support the construction of GUIs in Android and iPhone; *persistence* supports handling of files; *string utils* supports string handling. The catalogue `AndroidSupport` provides some extra templates that are required in Android (such as xml files required for the configuration of Android applications).

```
InitAndGetPropMethods ==
[public ⩽localPropType⩾ initial⩽localPropName⩾() {
   ⩽localPropCode⩾
   return ⩽localPropCodeResultVariableName⩾;
}
public ⩽localPropType⩾ get⩽localPropName⩾(){
   if (this.⩽localPropName⩾_isInitialized)
      return ⩽localPropName⩾;
   else
      this.set⩽localPropName⩾(this.initial⩽localPropName⩾());
   this.⩽localPropName⩾_isInitialized = true;
   return this.⩽localPropName⩾;
}
]■
ClassImplementation ==
public class ⩽classType⩾ implements OCLAny {
   ↑Attributes
   ↑ConstructorMethod
   ↑ConstructorMethodWithValues
   ↑InitAndGetPropMethods
   ↑SetPropMethods
   ↑RemoteEventMethods
   ↑QueryPropMethods
   ↑OCLAnyImplementation
}■
```

Fig. 4. FTL template definitions from the `CoreJava` FTL catalogue.

Figure 4 presents templates from the `CoreJava` catalogue. Template `ClassImplementation`, generates Java classes implementing EP classes. It comprises a collection of templates references; the template `InitAndGetPropMethods` generates Java implementing properties of EP classes.

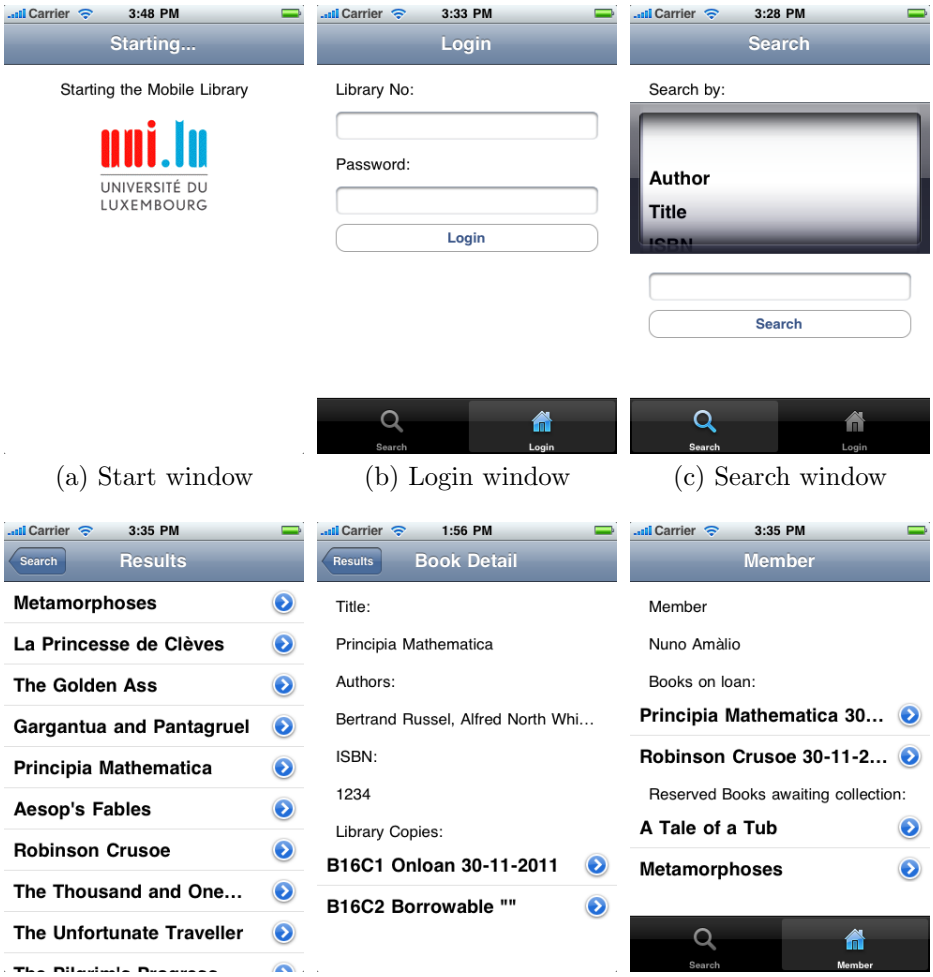
5 The Experiment and its Case Study

The experiment presented here uses a case study: the *simple mobile library* application. This application provides the following functionality:

- Users can search for books based on several search criteria, providing detailed information about a particular book from the search results; users can reserve books to be collected from the library.
- Users can see borrowing information and renew their borrowings.

Figure 5 presents this application's user interface on the iPhone. The application starts with a loading (or starting) screen (Fig. 5a). Users can then login into the system (Fig. 5b) or search for books in the library's catalogue (Fig. 5c). After submitting a search request, users can then see the results of a search in a list (Fig. 5d) and see the details of a particular book from the list (Fig. 5e). Once users are logged-in, they have access to the member window (Fig. 5f), allowing them to see the books they borrow and to request borrowings to be renewed.

The following presents the EP model and illustrate the generation process for this case study. The actual model and generated code, together with instructions on how to run the experiment on the platforms iPhone and Android,



(a) Start window (b) Login window (c) Search window

(d) Search results window (e) Book details window (f) Member window

Fig. 5. UI of simple library browser mobile application on the iPhone

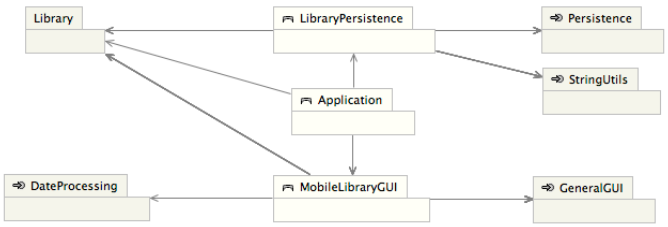
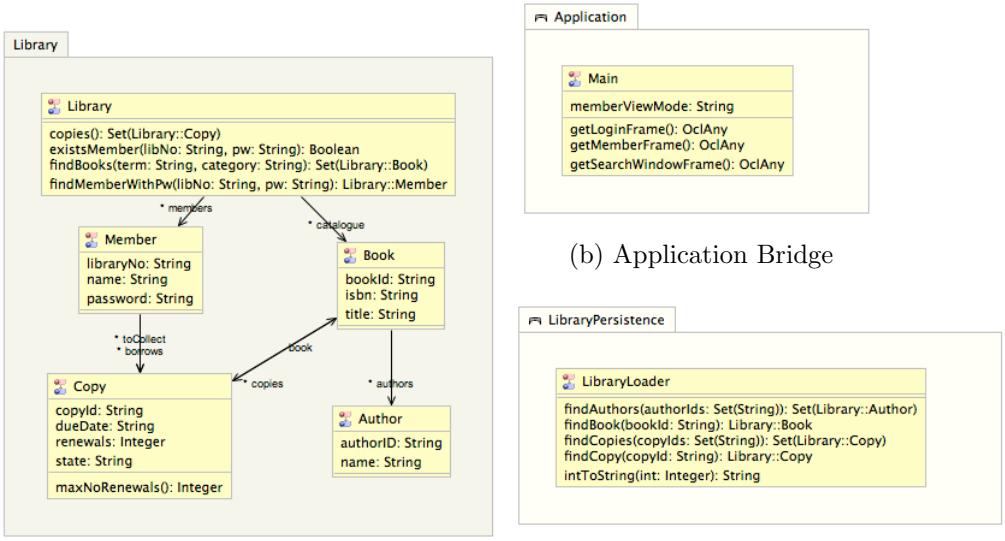


Fig. 6. Domain view of EP model of Mobile Library depicting the structure of domains and bridges

is available at http://democles.lassy.uni.lu/.



(a) Library Domain

(b) Application Bridge

(c) Persistence Domain

Fig. 7. Domains and bridges of the EP model of Mobile Library

6 EP model of *Mobile Library*

EP models are structured around bridges and domains. Figure 6 presents the bridges and domains that make the mobile library's EP model together with their dependency links. Bridges are distinguished from domains through the symbol ⊓. Unlike domains, bridges can have outgoing edges. The symbol ⇒ is used to identify binding domains, which require an explicit platform mapping (see section 7, non-binding domains do not require an explicit mapping). The model is divided into five EP domains and three bridges. The next sections describe the structure and behaviour of the mobile library's EP model.

6.1 Structure

6.1.1 The `Library` Domain

This domain encapsulates the problem domain of the mobile library application. Its structural model (Fig. 7a) is as follows:

- `Library` comprises the EP classes `Library` (representing the data of overall library), `Book`, `Copy` (represents the copy of a book), `Author` and `Member`. The EP definitions of these classes is given in Fig. A.1.
- A `Book` can have many `authors` and many `copies`. Its properties are `bookId` (a book identifier), `isbn` and `title`.
- A `Copy` has a single `Book`. Its properties record a copy identifier (`copyId`), the `dueDate` of a loaned copy, the number of `renewals` that have been made, where each renewal sets a later due date, and the copy's `state` (either

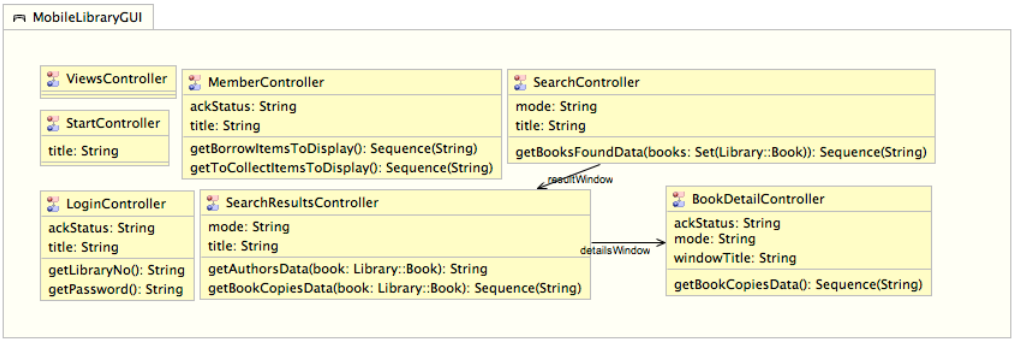


Fig. 8. The `MobileLibraryGUI` bridge

`'Borrowable'`, `'OnLoan'` or `'ToCollect'`).

- Library `Members` borrow books from the library (`borrows`), and can reserve books, which become available for collection (`toCollect`). Its properties record the member's library number (`libraryNo`), `name` and `password`.
- The properties of `Author` record the author's identifier in the library (`authorId`) and `name`.

6.1.2 The `Application` bridge

This bridge (Fig. 7b) acts as the centre of control of the overall mobile application. It interacts with bridges `LibraryPersistenceHandler` and `MobileLibraryGUI` and the `Library` domain. `Application` contains class `Main`, which is instantiated when the system is started. The EP definition of `Main` is given in Fig. A.2.

6.1.3 The `LibraryPersistence` bridge

This bridge (Fig. 7c) is responsible for loading the library data into memory. It contains class `LibraryLoader`, which reads the library data file and creates instances of the classes that make the `Library` domain. The EP definition of class `LibraryLoader` is given in Fig. A.3.

6.1.4 The `MobileLibraryGUI` bridge

This bridge (Fig. 8) encapsulates the application's platform-independent GUI design. It acts as an intermediate between the `Library` problem domain and the actual GUI elements (domain `GeneralGUI`), using the domain `DateProcessing` to process dates. Each class of this domain represents a *controller* (or *façade* [10]) of one of the windows of the GUI:

- `StartController` (see Fig. A.4a for EP definition) controls the start window (see Fig. 5a).

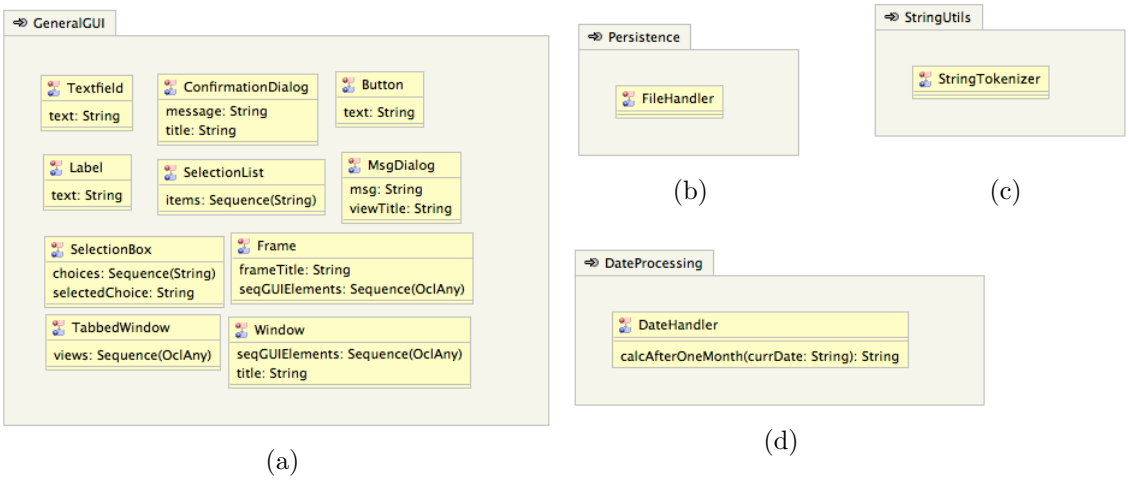


Fig. 9. Domains (a) `GeneralGUI`, (b) `Persistence`, (c) `StringUtils` and (d) `DateProcessing`

- `ViewsController` (see Fig. A.4b for EP definition) controls the the two views or tabs (see Figs. 5b, 5c, and 5f).
- `LoginController` (see Fig. A.4c for EP definition) controls the login window that may be inside the views window (see Fig. 5b).
- `SearchController` (see Fig. A.5a for EP definition) controls the search window view inside the views window (see Fig. 5c); it holds a reference to `SearchResultsController` (property `resultWindow`) to display the results of a search.
- `SearchResultsController` (see Fig. A.5b for EP definition) controls the window with the results of a search (see Fig. 5d); it holds a reference to `BookDetailController` (property `resultWindow`) to show the details of the selected book.
- `BookDetailController` (see Fig. A.6a for EP definition) controls the window that displays the details of a particular book (see Fig. 5e).
- `MemberController` (see Fig. A.6b for EP definition) controls the member window that may be inside the views window (see Fig. 5f).

6.1.5 The `GeneralGUI` Binding Domain

The classes of this binding domain (Fig. 9a) represent GUI constructs. Their EP definitions are given in Fig. A.7; they are as follows: (a) `TextField`, (b) `ConfirmationDialog` (a dialog that is presented to the user with a 'Yes', 'No' query), (c) `Button`, (d) `Label`, (e) `SelectionList`, (f) `MsgDialog` (a message dialog for the user to acknowledge), (g) `SelectionBox`, (h) `Frame`, (i) `TabbedWindow` and (j) `Window`.

6.1.6 *The* `Persistence` *Binding Domain*

This binding domain (Fig. 9b) encapsulates persistence based on files. It contains class `FileHandler`, which loads and saves data from files in the file system. The EP definition of `FileHandler` is given in Fig. A.8a; the class contains definitions of event signatures, which are bound to actual definitions of behaviour when the code is generated from the templates.

6.1.7 *The* `StringUtils` *Binding Domain*

This binding domain (Fig. 9c) encapsulates string-related functionality. It comprises class `StringTokenizer` (see Fig. A.8b for its EP definition), which breaks strings into tokens. The definitions of this domain comprise signature definitions, which are bound to actual definitions of behaviour upon code generation.

6.1.8 *The* `DateProcessor` *Binding Domain*

This binding domain (Fig. 9d) contains the class `DateHandler` (see Fig. A.8c for its EP definition), which processes dates (it calculates the new due date when a borrowing is renewed). Like the `StringUtils` domain, this domain defines signatures only, which are bound to actual behavioural definitions (or bodies) upon code generation.

6.2 *Behaviour*

In EP, behaviour is structured around events, which run on objects. Global system behaviour is a chain of events. The mobile library application comprises the following units of functionality:

(i) The application is initialised (the library data is loaded into memory).

(ii) The user is authenticated.

(iii) The user searches for books in the library.

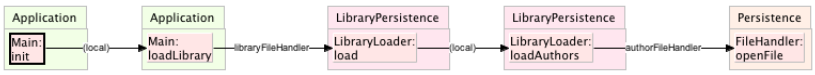
(iv) The user renews borrowings.

(v) The user reserves books for collection.

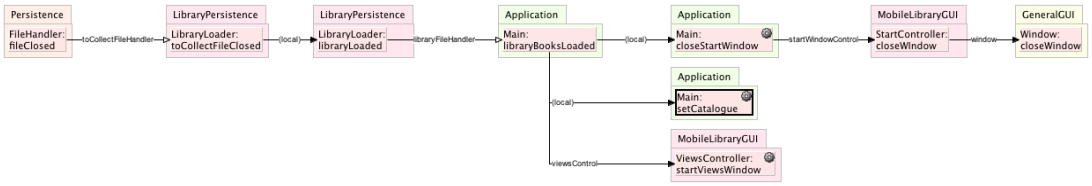
The following gives an overview over the behavioural EP model for these units of functionality by resorting to the event trees that are produced by Democles (EP's tool).

6.2.1 *Initialisation*

In the mobile library application, initialisation loads the library's data into memory. Figure 10 presents EP event trees associated with the initialisation of the system. Initialisation involves three steps: (a) the file is opened



(a) Event `init` and file opening



(b) File closing and setting of library data into memory

Fig. 10. Event trees of system initialisation

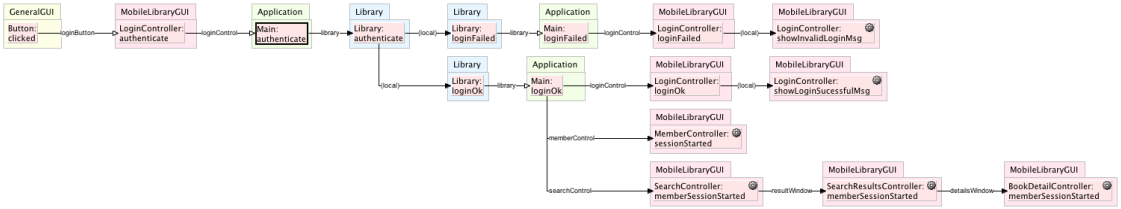


Fig. 11. Event tree of the authentication global event

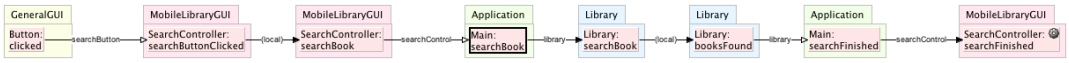


Fig. 12. Event tree of the search book global event

(Fig. 10a), (b) the data is read from the file and (c) the data is set in memory (Fig. 10b). Initialisation is triggered through the event `init`, which runs on the sole instance of `Main` [6].

6.2.2 Authentication

Figure 11 presents the event tree associated with authentication. This is triggered when the user presses the 'Login' button of the `Login` window (Fig. 5b), which results in a call to the event `authenticate` in `Main`, which is then dispatched to the `Library` instance. The event tree then branches depending on the result of the authentication: events `loginOk` and `loginFailed`.

6.2.3 Search Book

Figure 12 presents the event tree associated with search for library books using the mobile application. The event is triggered in the search book window when the user presses the 'Search' button (Fig. 5c). The event is then dispatched to the `Library` instance where the actual search is performed (event

[6] This is the first event to run in an EP system.

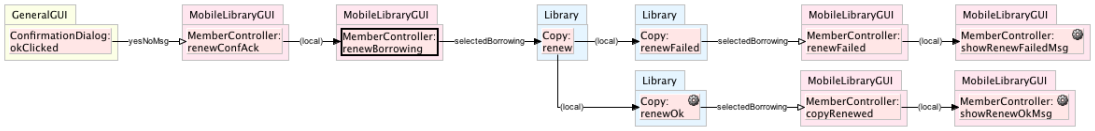


Fig. 13. Event tree associated with the renewal of books by members

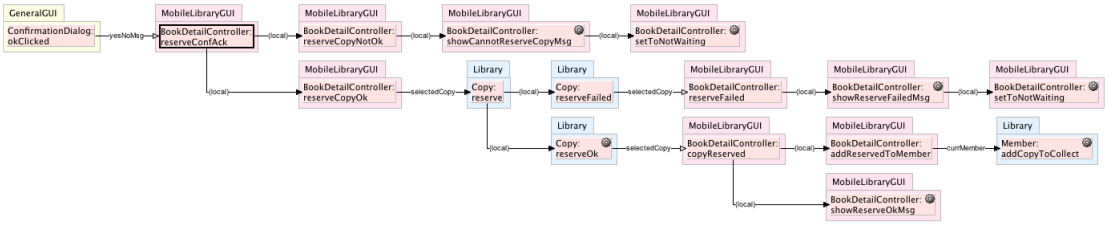


Fig. 14. Event tree associated with the reservation of books by members

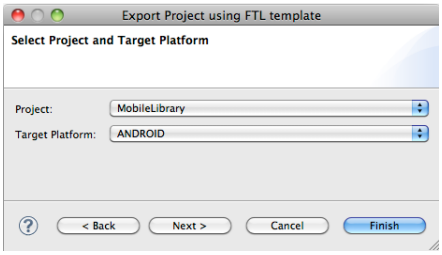
`searchBook`). The search results then come back to the `SearchController` instance through the event `searchFinished`, which results in the creation of an instance of `SearchResultsController`.

6.2.4 Renewal of Books

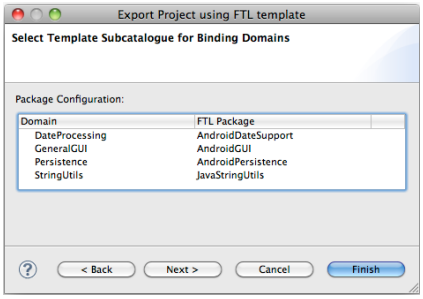
Figure 13 presents the event tree associated with the renewal of books by members, which is done via the member window (Fig. 5f). The event is triggered through a confirmation dialog when the user presses the 'Ok' button. The event is then dispatched to the `MemberController` instance, which calls the event `renew` on the `Copy` instance corresponding to the selected copy. The renew operation can either be successful (event `renewOk`) or not (event `renewFailed`); the latter occurs when the limit of renewals is reached.

6.2.5 Reservation of Books

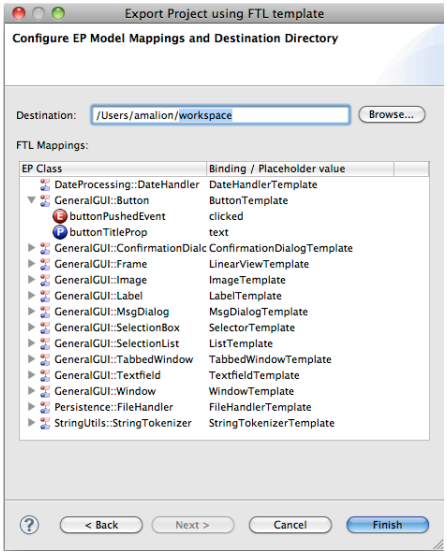
Figure 14 presents the event tree associated with the reservation of books by members, which is done via the book details window (Fig. A.6a). The event is triggered through a confirmation dialog when the user presses the 'Ok' button. The event is then dispatched to the `BookDetailController` instance. Two cases are then considered: (a) there is member that is logged-in (event `reserveCopyOk`) or (b) no member is logged-in (event `reserveCopyNotOk`). If there is a member that is logged-in, then the reservation operation can proceed and the event `reserve` is triggered on the selected `Copy` instance. The reservation can then be successful (event `reserveOk`) or not (event `reserveFailed`); the latter occurs when the copy can not be reserved.



(a) Platform Selection



(b) FTL catalogues for binding domains



(c) Templates for EP classes

Fig. 15. Configuration of a mapping for an EP model in Democles

7 Code Generation and Platform Mappings

Platform code is generated by instantiating templates. To perform this, the user needs to configure the generation settings, namely: the target platform, template catalogues of the selected platform to use, and a mapping from template parameters to model elements. The configuration process is as follows [7]:

(i) The first step is to choose the target platform. Each platform is associated with a FTL catalogue (in Fig. 15a, user selects the android platform), all platform-specific artifacts are generated from templates of the selected platform's catalogue. As illustrated in Fig. 3, a platform catalogue is made of sub-catalogues; one of these catalogues is the core, it says how to generate EP classes of non-binding EP domains.

(ii) The second step is to assign FTL template sub-catalogues to each binding domain (the core sub-catalogue cannot be assigned). This is done to reduce the number of templates that can be mapped to the classes of a binding domain. In Fig. 15b, the user assigns the sub-catalogue `AndroidGUI` to the `GeneralGUI` binding domain.

(iii) The third step is to assign EP classes of each binding domain to templates. For each class, the user needs to describe how properties and events of the EP class are mapped to template variables. In Fig. 15c, the user assigns the `ButtonTemplate` to the EP class `Button`, and maps event `clicked`

[7] The configuration step is started when the user invokes the option 'Export' on Eclipse.

Bindings and Bridges	8
Classes	28
Events and Properties	280

(a) Number of modelling units

	iPhone	Android
Template catalogues	2,432	2,266
Total application code	14,709	11,265
Infrastructure code	1,601	1,123

(b) Total lines of code of template catalogues, generated applications and platform infrastructure

Table 1
Figures of the experiment presented in this paper

and property `text` to variables of the selected template.

From the mapping, Democles builds FTL instantiation structures that are used to instantiate FTL templates to generate the platform application.

8 Evaluation

Our goals for the experiment presented here were: (a) generate code for various platforms from EP PIMs, (b) enable users that are not experts in the target platform to build platform-specific applications, and (c) provide an environment for validating requirements and experimenting design decisions. We now evaluate these goals using the experiment presented here.

8.1 Generation of Platform Code from EP PIMs

We have realised the experiment presented here with the modelling language EP supplemented with OCL and FTL, a language of templates. This approach has been incorporated into Democles, EP's tool. We have illustrated the $EP+FTL$ combination with mobile platforms Android and iPhone [19]. From the EP PIM presented in section 6, we generated code for both these mobile platforms. Both applications implement the same functionality as described in the EP PIM. The applications were generated by instantiating FTL templates of the catalogues of each platform and taking into account some infrastructure code (that supports the translation from OCL). EP model of section 6 is free of any domain-specific details; it was used to generate code for Android and iPhone, and could be used to generate code for other platforms (this requires platform FTL catalogues). EP PIM and platform templates are linked through separate mappings (section 7).

Table 1 presents several figures regarding our experiment. The case study's EP model has a total of 8 packaging units (bridges or domains), 28 EP classes and a total of 280 events and properties (Table 1a). Table 1b presents several figures based on lines of platform code. The iPhone's FTL catalogue has a total of 2,432 lines; Android's has a total of 2,266 lines. iPhone application comprises a total of 14,709 lines of platform code; Android's comprises

11,265 [8]. Although our case study can be considered to be relatively small, it is big enough to do something useful and practical. It is the biggest case study of our library of EP models at the time of writing.

8.2 Platform Applications Built by non-Platform Experts

The EP model of section 6 is platform independent. It talks about problem domain, graphical user interface and persistence concepts without committing to any platform-specific details. The team that developed the EP model presented here consists of three people; only two of them are experts in Android and iPhone. However, all developers could understand and process the EP PIM and generate iPhone and Android applications from it; EP constituted a platform-independent medium over which the different users could communicate.

In such a setting, initially the progress in developing a model is slow. This is because initially, we need more templates, and it takes time to build them and get them right. Once the templates are there and they become mature, then the development is relatively fast and straightforward, and the model developer becomes independent from platform developers.

8.3 Validation of Requirements and Experimentation of Design Decisions

The visualisation features of Democles (EP's tool) help in building the model and in evaluating design decisions. The domain view shows the dependencies between the different classes that make a model giving an overall view (e.g. Figs. 7 and 6); in our experience, its investigation helps to improve the design. The event trees that show paths of execution associated with events (e.g figures 10 and 11) help the developer in finding errors and in improving the overall model.

Our approach enables early generation of applications from the models. This allows developers to validate requirements and experiment their designs. The EP model presented here was developed incrementally; from each new version of the model, a new application would be generated for experimentation. Once the templates and the mapping is configured, the process of application generation runs smoothly. The only problem is when one is developing the model, needs feedback, but the required templates are not yet there.

[8] The iPhone platform tends to result in a higher number of lines of code; iPhone's programming language, Objective-C, is less abstract than Java and, unlike Java, it uses header files.

9 Discussion

This paper has two goals: (a) provide empirical evidence on the feasibility and applicability of MDD ideas, such as templates, executable modelling and code generation; (b) gain insight on things to improve both in our languages, EP and FTL, and MDD in general.

Our experiment suggests that our MDD approach is effective at helping developers that are not experts in some target platform in developing platform applications from models. However, our approach does have several limitations and opportunities for improvement. We now discuss these in detail.

9.1 Variability and Platform Independence

EP models are platform independent, making them portable to various platforms, but not everything is expressible in a platform independent way at the model level using EP and OCL. For instance, OCL lacks constructs to describe expressions involving strings [19] and there is not way of defining such constructions. We surrounded such limitations by defining binding EP domains and classes that provide the required functionality by defining only the signatures, which need to be mapped to some platform template when code is to be generated. This has been done in the binding domains `Persistence`, `StringUtils` and `GeneralGUI` of the EP model of section 6. Although ideally everything would be expressed at the model level, the fact that we couldn't achieve this was not a serious problem. The behaviour that was not expressed in the model is behaviour that should be part of a library (such as the behaviour involving strings and dates), or is so low level that is not relevant for the design of the application (file handling).

A benefit of the approach presented here is that the different application variants are obtained automatically. This means that application developers do not directly have to deal with variability, which is hidden in the templates infrastructure. However, sometimes one wants to deal with variability at the modelling level, but our EP based approach does not currently support this.

9.2 Usability

This experiment shows several usability benefits of developing applications at the modelling level. The abstraction from platform code enables non platform-experts to participate in the development; this encourages the formation of teams with a healthy mixture of skills, enabling those that are not platform experts to participate in the development. Furthermore, this provides a clear separation of concerns between the work that platform-experts and designers need to do and a medium that enhances collaboration. The downside in terms of usability lies in the level of tool support. Despite many advances in

recent years, modern platform-specific development toolkits are more sophisticated than MDD toolkits. We are not aware of platform-independent GUI editors that support various platforms. Through the experiment presented here, we gained insight on things to improve on our approach: (a) a platform-independent GUI editor that could generate code factored as templates; (b) EP's tool support for refactoring needs to be improved (doing simple refactorings was often tedious and time-consuming) and the EP language needs to be improved to facilitate modelling in the large; (c) is difficult to build instantiation structures for complex FTL template expressions involving lists [19] and we are looking into improving this, in particular to simplify the process of instantiation.

9.3 Validation and Experimentation

Once the required templates are defined, it is straightforward to generate a platform application from the model for validation of the requirements and experimentation of design decisions. Our approach supports the creation of templates variants of the same functionality; the choice of the appropriate variant can be selected based on experimentation with the generated prototype. The problem is that sometimes there is the need to debug the generated platform code when the templates are not yet mature, which requires programming expertise, and so the aid of the platform experts may be required.

Another possibility that we have not explored is to use the formal semantics of EP and OCL to formally validate and verify the models at the modelling before any code is generated. The advantage of this is that we can prove properties at the modelling level that are satisfied at the level of the implementation. We are currently working on an approach to formally validate and verify EP models based EP's formal semantics. However, this will require expertise in formal analysis (especially theorem proving), as the proof of certain properties cannot be automated in general. Again, we advocate the emergence of teams with a healthy mixture of skills; and in this case, a formal methods expert would be required, especially if one wants to address critical systems where verification is paramount.

9.4 Scalability

Although we have successfully built a model from which a practical mobile application could be generated, our case study is still relatively small. Things get more complicated when the EP models become large; the models tend to become cluttered. We are looking into ways of improving this to facilitate modelling in the large.

10 Related Work

The approach presented here is a variant of the FTL-based modelling framework of [2,3]. This proposes to define semantics of UML notations based on catalogues of FTL templates; each catalogue encapsulates a particular semantic interpretation of the notations. Here, we adapt the same idea to the context of code generation, where each catalogue targets an alternative execution platform. Unlike the FTL templates presented here, templates of [2,3] were instantiated manually (no FTL tool support was available at the time).

Several model-driven approaches approaches [20,7] are only partially model-centric: they produce partial skeleton code that needs to be completed by the user. Our approach is totally model-centric; all running platform-code is generated from PIMs expressed in EP and OCL.

Several MDD approaches that derive product families by resolving variability through model transformations have been proposed [9,13,5]. Unlike the work presented here, those approaches have not been realised. Our approach has been integrated in EP's tool, Democles, and illustrated here with an experiment involving the iPhone and Android platforms.

Several works propose MDD frameworks based on templates [20,7,8]. However, unlike our FTL-based approach, they use template languages that are not formally defined; [20] and [8] uses very simple template languages made of placeholder constructs only. FTL is a formally defined language with a rich set of constructs including placeholders, lists, choice, template naming, and modules.

Several works take an MDD approach to the production of mobile-phone applications or prototypes [1,22,5]. The approach presented here can also be used for this purpose; none of these approaches proposes templates as intermediate representations of platform variants.

Balasubramanian et al [6] propose an MDE approach based on domain-specific modelling languages (DSMLs), where domain modelling concepts encapsulate transformations to platform-specific artifacts. This approach covers more system aspects than the EP-based approach presented here, such as architectures and deployment. Our approach does not use DSMLs, but provided the concepts of our models are abstract they can be mapped to various platforms. Unlike [6], the approach presented here uses templates as intermediate representations of code. Our approach is based on formal languages, both EP and FTL are formal; [6] uses statecharts and data-flow diagrams to describe behaviour; our behaviour is described using EP and OCL.

In [24], Weigert and Weil report on the industrial experience of applying MDD approaches, similar to the one presented here, at Motorola. In [24], code is generated from high-level languages, such as UML, based on code-generators that are defined for the modelling language. The approach presented here uses

templates, which constitutes an intermediate medium between the high-level modelling notation and the code that is to be generated. The user needs to define mappings from concepts to templates; actual generation process is fixed and based on the rules of template instantiation defined for FTL. To support some new platform in the approach presented here, all that is required is a new catalogue of templates; in [24] a new code generator would be required.

11 Conclusions

This paper presents an experiment with our MDD approach based on the executable modelling language EP and the Formal Template Language (FTL). The approach presented here gives generation a first-class status by proposing separate description of template catalogues described in FTL; each platform catalogue is generative and encapsulates all the templates that are required for the generation of platform artifacts by instantiating them with information coming from the model. The experiment presented here illustrates this approach with a case study of a present day problem: building mobile-applications that have the same functionality, but need to run on different execution platforms. From the same EP PIM we generated Android and iPhone applications by instantiating FTL templates of these platforms' catalogues. The work presented here can be used to develop actual applications or prototypes. The Democles tool supporting the approach presented here, together with the model and generated code of the experiment can be obtained from http://democles.lassy.uni.lu/.

References

[1] Adelmann, R. and M. Langheinrich, *SPARK rapid prototyping environment — mobile phone development made easy*, in: *IMC 2009*, LNCS (2009).

[2] Amálio, N., "Generative frameworks for rigorous model-driven development," Ph.D. thesis, Dept. Computer Science, Univ. of York (2007).

[3] Amálio, N., F. Polack and S. Stepney, *Frameworks based on templates for rigorous model-driven development*, ENTCS **191** (2007), pp. 3–23.

[4] Amálio, N., S. Stepney and F. Polack, *A formal template language enabling meta-proof*, in: *FM 2006*, LNCS **4085** (2006), pp. 252–267.

[5] Balagtas-Fernandez, F., M. Tafelmayer and H. Hussman, *Mobia modeler: Easing the creation process of mobile applications for non-technical users*, in: *IUI'10*, 2010.

[6] Balasubramanian, K., A. Gokhale, G. Karsai, J. Sztipanovits and S. Neema, *Developing applications using model-driven design environments*, IEEE Computer **39** (2006), pp. 25–31.

[7] Behrens, H., *MDSD for the iPhone: developing a domain-specific language and ide tooling to produce real world applications for mobile devices*, in: *SPLASH*, 2010.

[8] Czarnecki, K. and M. Antkiewicz, *Mapping features to models: A template approach based on superimposed variants*, in: *GPCE*, LNCS **3676** (2005).

[9] Deelstra, S., J. v. G. Marco Sinnema and J. Bosch, *Model driven architecture as approach to manage variability in software product families*, in: *Workshop on model driven architecture: foundations and applications*, 2003.

[10] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns," Addison-Wesley, 1995.

[11] Glodt, C., P. Kelsen, N. Amálio and Q. Ma, *From platform-independent to platform-specific models using democles*, in: *OOPSLA Companion 2009*, 2009, pp. 795–796.

[12] Glodt, C., P. Kelsen and E. Pulvermueller, *Democles: a tool for executable modeling of platform-independent systems*, in: *OOPSLA Companion 2007*, 2007, pp. 870–871.

[13] Haugen, Ø., B. Møller-pedersen, J. Oldevik and A. Solberg, *An mda-based framework for model-driven product derivation*, in: *SEA*, 2004.

[14] Holleis, P. and A. Schmidt, *Makeit: Integrate user interaction times in the design process of mobile applications*, in: *Pervasive Computing*, LNCS **5013**, 2008.

[15] Huebscher, M., N. Pryce, N. Dulay and P. Thompson, *Issues in developing ubicomp applications on symbian phones*, in: *Future Mobile Computing Applications (FUMCA)* (2006).

[16] Kelsen, P., *A declarative executable model for object-based systems based on functional decomposition*, in: *ICSOFT 2006*, 2006, pp. 63–71.

[17] Kelsen, P. and Q. Ma, *A lightweight approach for defining the formal semantics of a modeling language*, in: *Models 2008*, LNCS **5301**, 2008, pp. 690–704.

[18] Miller, J. and J. Mukerji, *MDA guide version 1.0.1*, Technical report, Object Management Group (OMG) (2003).

[19] Pinto, F., "From Platform-Independent EP Models to Platform-Specific Code via Templates," Master's thesis, University of Luxembourg (2010).

[20] Santos, A. L., K. Koskimies and A. Lopes, *A model-driven approach to variability in product-line engineering*, Nordic Journal of Computing **13** (2006), pp. 196–213.

[21] Schmidt, D. C., *Model-driven engineering*, IEEE Computer **39** (2006), pp. 25–31.

[22] Thompson, C., J. White, B. Dougherty and D. C. Schmidt, *Optimizing mobile application performance with model–driven engineering*, in: *SEUS 2009*, LNCS **5860** (2009), pp. 36–46.

[23] Warmer, J. and A. Kleppe, "The Object Constraint Language: getting your models ready for MDA," Addison-Wesley, 2003.

[24] Weigert, T. and F. Weil, *Practical experiences in using model-driven engineering to develop trustworth computing systems*, in: *SUTC'06* (2006).

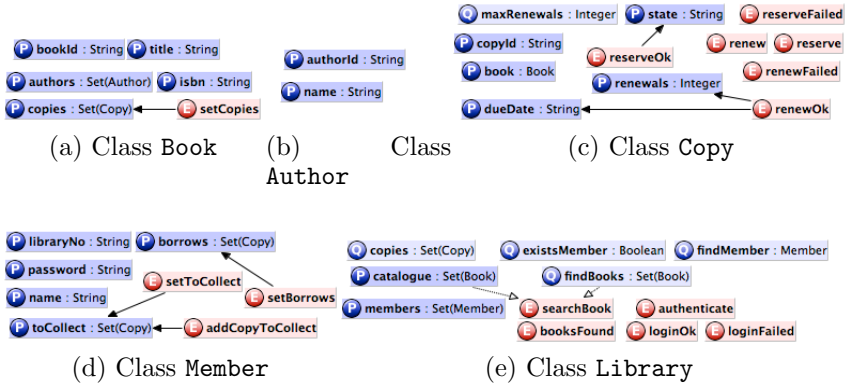
A The complete EP Model

A.1 The `Library` Domain

Figure A.1 presents the EP definitions of the classes that make the `Library` domain.

A.2 The `Application` Bridge

The EP definition of class `Main`, the sole class of the `Application` bridge is given in Fig. A.2.



(a) Class `Book` (b) Class `Author` (c) Class `Copy`

(d) Class `Member` (e) Class `Library`

Fig. A.1. EP definitions of the classes of the `Library` domain

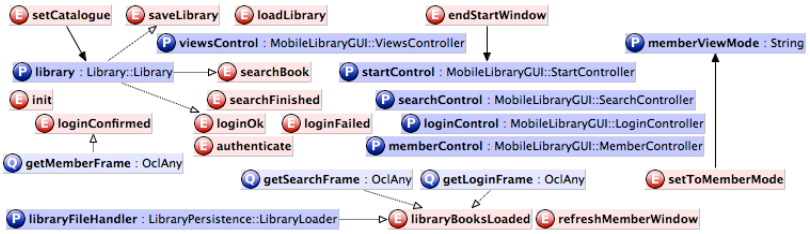


Fig. A.2. EP definition of class `Main` of the `Application` bridge

Fig. A.3. EP definition of class `LibraryLoader` of the `LibraryPersistence` bridge

A.3 The `LibraryPersistence` Bridge

The EP definition of class `LibraryLoader`, the sole class of the `LibraryPersistence` bridge is given in Fig. A.3.

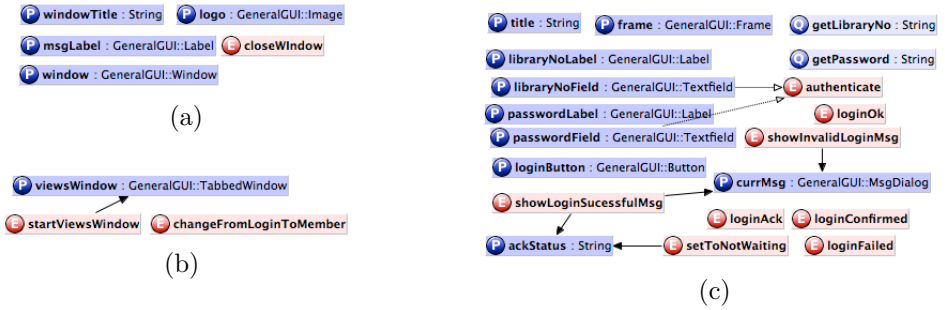


Fig. A.4. EP definition of classes (a) `StartController`, (b) `ViewsController` and (c) `LoginController` of bridge `MobileLibraryGUI`

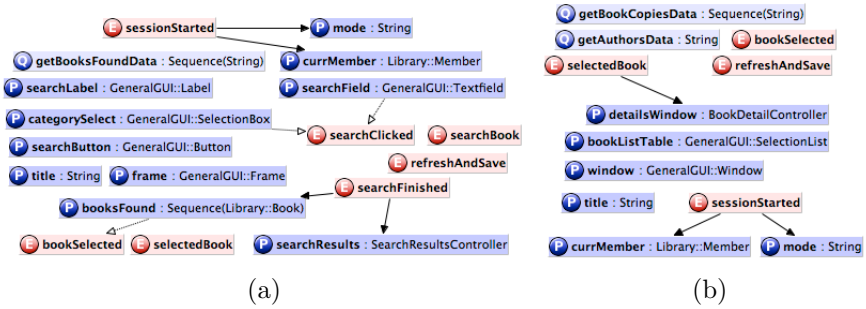


Fig. A.5. EP definition of classes (a) `SearchController` and (b) `SearchResultsController`

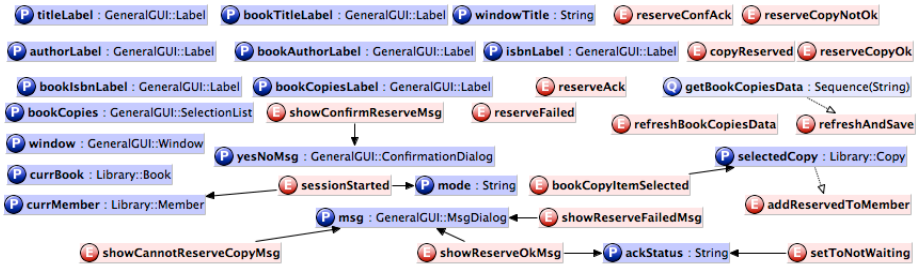
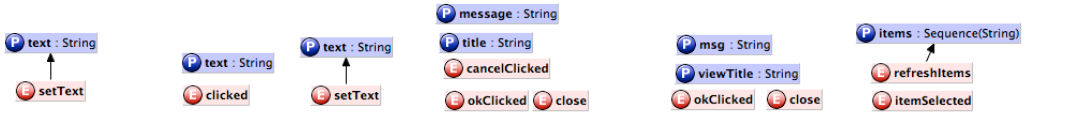
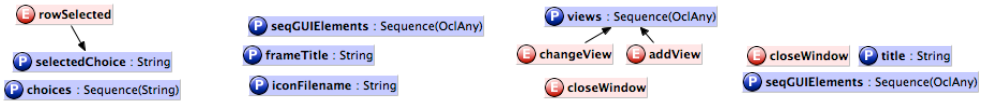


Fig. A.6. EP definition of classes (a) `BookDetailController` and (b) `MemberController` of bridge `MobileLibraryGUI`



(a) `TextField` (b) `Button` (c) `Label` (d) `ConfirmationDialog` (e) `MsgDialog` (f) `SelectionList`



(g) `SelectionBox` (h) `Frame` (i) `TabbedWindow` (j) `Window`

Fig. A.7. EP definition of the classes that make the binding domain `GeneralGUI`

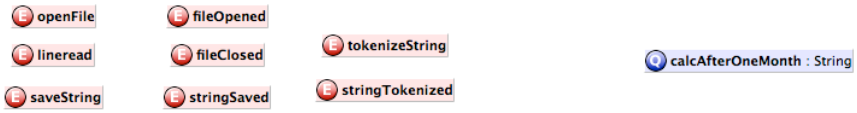
A.4 The `MobileLibraryGUI` Bridge

Figures A.4, A.5 and A.6 present the EP definitions of the classes that make the `MobileLibraryGUI` bridge.

A.5 The `GeneralGUI` Binding Domain

Figures A.7 present the EP definitions of the classes that make the `GeneralGUI` binding domain.

A.6 `Persistence`, `StringUtils` and `DateProcessing` Binding Domains



(a) Class `FileHandler` (b) Class `StringTokenizer` (c) Class `DateHandler`

Fig. A.8. EP definitions of classes (a) `FileHandler` (`Persistence` binding domain), (b) `StringTokenizer` (`StringUtils` binding domain) and (c) `DateHandler` (`DateProcessing` binding domain)

Figures A.8 present the EP definitions of the classes that make the `Persistence`, `StringUtils` and `DateProcessing` binding domains.