
POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMATICA E BIOINGEGNERIA
DOCTORAL PROGRAMME IN COMPUTER SCIENCE AND ENGINEERING

---

# TOOLS, SEMANTICS AND WORK-FLOWS FOR WEB AND MOBILE MODEL DRIVEN DEVELOPMENT

Doctoral Dissertation of:
**Carlo Bernaschina**

Supervisor:
**Prof. Piero Fraternali**

Tutor:
**Prof. Simone Garatti**

The Chair of the Doctoral Program:
**Prof. Andrea Bonarini**

2018 – XXX Cycle

# Acknowledgements

This work is the result of a long personal journey that eventually brought me here. During this journey I had the opportunity to walk with many amazing people. Some stayed with me from the beginning to the end, some joined along the way while others left. Each one has played a role, big or small, and I would like to thank all of them for their particular contributions.

To Piero Fraternali, for its guidance and enormous support during these years. He was present during all the steps of this work, encouraging me to go forward and expand my horizons.

To my parents and family, for their support, presence and kindness. They helped me going through happy and rough times, always showing me the bright side of every situation and helping me cope with stress.

To Roman Fedorov, for an amazing friendship and the never ending wisdom for more than 10 years.

To Fernando Sánchez-Figueroa and Jordi Cabot, for their timely and detailed reviews.

To John McCutchan, for the amazing opportunity he gave me and the career guidance.

To my friends and colleagues, for their everlasting friendship and patience.

To a special person who has recently joined the walk, for putting my life upside down and giving me the strength to finish writing.

Thank You.

# Abstract

WEB enabled mobile devices are becoming more an more ubiquitous in our lives. Application development for these devices opens newer and newer challenges. Model Driven Development was proposed as a solution able to reduce complexity and enhance productivity. This methodology was, and is still, not broadly adopted due to a proven, or perceived, high costs/advantages ratio making it difficult to reach a break-even point. The goal of the research presented in this thesis is to propose tools, semantics and work-flows aimed at reducing the costs of Model Driven Development, especially in the field of web and mobile applications. We will focus on tooling, by presenting an agile model transformation framework enabling the introduction of the Model Driven methodology in existing tools or the bootstrapping and rapid iterative development of new environments. We present a formal semantics for the Interaction Flow Modeling Language, focused on web and mobile applications, having as objective a simple tool independent interpretation of IFML models enabling tools interoperability. We present an on-line tool for the rapid prototyping of web and mobile applications, showing how the proposed framework and semantics can be easily integrated together to produce a production ready Model Driven environment. We eventually present a Model and Text co-evolution work-flow which facilitates the interaction between code generators and human developers, by treating the application source code as the central artifact and the code generator as a virtual developer, i.e., yet another member of the team. The experimental results show how the proposed methodology can reduce both the amount of work needed to obtain a production ready application and the level of expertise required in the process.

# Sommario

I dispositivi mobili con accesso al web stanno diventando sempre piú onnipresenti nelle nostre vite. Lo sviluppo di applicazioni per questi dispositivi apre sempre nuove sfide. Lo Sviluppo Model Driven venne proposto come soluzione capace di ridurre la complessitá e al tempo stesso incrementare la produttivitá. Questa metodologia non venne, e ancora oggi non viene, ampiamente adottata per il comprovato, o anche solo percepito, elevato rapporto costi/vantaggi che rende difficile raggiungere un punto di break-even. L'obiettivo della ricerca presentata in questa tesi é di proporre tool, semantica e work-flow focalizzati a ridurre i costi dello Sviluppo Model Driven, in particolare nel campo web e delle applicazioni mobili. Ci concentreremo sui tool, presentando un framework agile per lo sviluppo di trasformazioni di modello, il quale facilita l'introduzione della metodologia Model Driven in tool preesistenti o la rapida evoluzioni di nuovi ambienti di sviluppo. Presenteremo una semantica formale per l'Interaction Flow Modeling Language, focalizzata allo sviluppo web e di applicazioni mobili, la quale si pone come una interpretazione di IFML indipendente dagli specifici tool incrementandone la interoperabilitá. Presenteremo un tool on-line per lo sviluppo rapido di prototipi di applicazioni web e mobili, mostrando come il framework e la semantica proposta possano facilmente essere integrate per produrre un ambiente di sviluppo Model Driven professionale. Infine, presenteremo un work-flow per la co-evoluzioni di modelli e artefatti testuali, il quale facilita l'interazione tra generatori di codice e sviluppatori umani trattando il codice generato come l'artefatto al centro dello sviluppo e il generatore di codice come uno sviluppatore virtuale, ossia come un membro del team. I risultati sperimentali dimostrano come la metodologia proposta puó ridurre sia la quantitÃ  di lavoro richiesto per produrre un'applicazione e le competenze richieste durante il processo stesso.

# Contents

CHAPTER 1

# Introduction

The proliferation of web enabled mobile devices in the last decade has presented great opportunities and challenges. In particular we have seen a shift from general purpose complex desktop applications to specialized web and mobile applications with particular focus on user experience, development time and cost [138].

The industry addressed the challenge by introducing development tools specifically targeted at simplifying application development in these use-cases. Particular attention is devoted to Low Code/No Code Development Tools, which are a valuable, and cost effective, alternative to classical development techniques [81, 133, 137]. These tools are vendor specific and their selection is a critical step in the application development [107]. Projects can be delayed or even fail due to vendor lock-in.

In the academia Model Driven Engineering (MDE), and in particular Model Driven Development (MDD), has been adopted as a valuable approach able to address this challenge [52, 120, 128]. Model Driven Engineering [50, 51] is the branch of Software Engineering that emphasizes the use of models, i.e., simplified descriptions of the application that capture its essential aspects at a certain level of abstraction, e.g., independently of the platform for which the application will be designed and of the technologies with which it will be implemented. Model Driven Development is a subset of Model Driven Engineering specifically focused on supporting the development effort.

Abstraction is the most important aspect of MDE and MDD. It enables developers to validate high level concepts and introduce details, which accommodate complexity, at later stages in the process. This distinction gives importance to a proper balancing between what is considered a high level concepts and an implementation detail [75].

## 1.1 Problem Statement

Model Driven Development has a limited adoption in the industry due to social and technical factors making the benefits not outweigh the costs [87, 134], despite expectations and hope by the MDD community [61, 80, 98]. While the majority of research is focusing on increasing the benefits of Model Driven Development in order to reach a cost/benefits break even [55, 89], this thesis would like to address the problem by reducing costs.

***Problem Statement:*** *Reduce the costs of Model Driven Development by changing or relaxing some of its assumptions.*

We address the development and evolution of MDD tools, the semantics of modeling languages and the work-flow enabling the collaboration between developers and MDD experts. We specifically focus on the use-case of Web and Mobile applications development, given its unique set of requirements, i.e. details like styling of the User Interface (UI) or device sensors support are crucial. Differently from recent approaches, focused on the overloading of existing languages with newer and newer features [53, 115, 132], we focus on basic concepts, i.e., the flow of information in an application, and collaboration between developers and non MDD experts.

## 1.2 Research Questions

In this section, we formulate the research question that motivate the work of this thesis; we list the questions following the logical steps needed to move from a high level application description to a final product.

***Research Question 1.*** *Is it possible to simplify the development of model editors and model transformation in order to reduce the friction between developers and MDD?*

The high specialization required to develop and maintain a Model Driven Environment is highly perceived as a cost of Model Driven Development. Making it necessary the introduction of MDD experts in the development team or the interaction with external vendors. Complexity in the maintenance of these tools increase friction making their maintenance and evolution perceived as an additional cost of Model Driven Development.

***Research Question 2.*** *Is it possible to model an application at a high level ensuring a unique an intuitive interpretation of the model itself?*

While various modeling languages have been adapted to or explicitly developed for the description of user interaction, their general purpose nature can lead to different interpretations of the very same model. The introduction of newer and newer extensions to these languages, in order to cover new use-cases and final product requirements, can lead to an increase of complexity in the models, making them less and less intuitive from the prospective of a newcomer. Both these effects are perceived as an additional cost to Model Driven Development.

***Research Question 3.*** *Is it possible to integrate MDD into a preexisting development work-flow? And, in particular, is it possible for non MDD experts to contribute to the project without the direct involvement of experts?*

In its classical instances Model Driven Development follows the Forward Engineering approach, i.e., MDD based work-flows are model-centric giving no real value to the final code, which is seen as yet another artifact in a purely automated tool-chain of transformations which target is the product. Each detail of the final product must be described via extensions on the high level model and/or via model transformations. This mindset collides with the code-centric nature of development work-flows, where various developers work in parallel to achieve a set of small, generally independent, tasks targeting a common goal. The high level of expertise required and the inability for non MDD experts to experiment or apply persisting point-wise fixes in the produced code is perceived as an additional cost of Model Driven Development.

## 1.3 Contributions

In this thesis we explore the feasibility of reducing the friction between developers and the Model Driven Development approach, by proposing a set of tools, semantics and work-flow focused on the development of Web and Mobile applications via the high level definition and generation of the application front-end and the parallel manual introduction of requirements not captured by the high level definition. The contributions of this thesis are as follows:

- We present ALMOsT.js, an OpenSource agile model driven transformation framework for JavaScript, which enables non Model Driven Development experts to experiment with the MDD methodology and easily integrate the results in existing on-line tools.
- We define a Web and Mobile centric formal semantics for the Interaction Flow Modeling Language (IFML). Instead of focusing on its extension, required to describe all the details involved in Web and Mobile development, we focus just on high level core concepts in order to give a unique and precise interpretation to each valid model.
- We present IFMLEdit.org, an OpenSource IFML based Model Driven Development Environment which serves as a use-case for ALMOsT.js, as a reference implementation for the proposed IFML semantics and proposes the usage of IFML for the rapid prototyping of Web and Mobile applications. The focus of the tool is not the full generation of the final application, but instead easy experimentation of user critical journeys via rapid model iterations and configuration less evaluation, following the Agile methodology. The generated application can be easily customized to satisfy requirements not directly addressed by the IFML model, i.e., styling, data access layer and business logic.
- We define a conceptual work-flow for model and code co-evolution in which the code-generator is considered as yet another developer in the team. This work-flow enables seamless collaboration between MDD experts and the other developers of the team, by preserving manual contributions on the generated code without the need of MDD expertise. We present ALMOsT-Git a reference implementation of the proposed work-flow based on the Git version control system and evaluate its effects on the application development effort on a reference IFML-based tool from the industry and the proposed IFMLEdit.org.

While these contributions focus as a whole in reducing development costs, they can also be applied separately and mixed with other languages and tools. As an example, in Chapter 6 the defined conceptual work-flow for model and code co-evolution will be evaluated with both IFMLEdit.org and a tool which is not built upon neither the presented agile model driven transformation framework nor the proposed IFML semantics.

## 1.4 Thesis organization

The structure of the thesis follows the same logical flow presented in Section 1.2:

**Chapter 2** discusses the background of the work contained in this thesis, focusing on the concepts common to the remaining chapters, i.e., the development of Web and Mobile application, the proposed MDD based solutions and the Interaction Flow Modeling Language (IFML). Each one of the remaining chapters may introduce additional specific background.

**Chapter 3** discuss the relation between Model Driven Development an the Agile movement and present the ALMOsT.js framework.

**Chapter 4** presents the proposed Web and Mobile centric semantics for the Interaction Flow Modeling Language (IFML). Showing a set of small examples in which the proposed semantics is able to give a unique and well defined interpretation.

**Chapter 5** presents IFMLEdit.org a tool developed with the proposed methodology and framework presented in Chapter 3, which was key in the development and validation of the semantics proposed in Chapter 4 and provides different code-generators compliant with the proposed semantics which are used as starting points for the approach proposed in Chapter 6.

**Chapter 6** introduces a novel work-flow for the seamless collaboration between MDD tool-chains and manual development and evaluates its impact on the development effort of two use-case applications.

Finally **Chapter 7** concludes the thesis, discusses the current challenges and proposes the future direction of the research in this area.

This thesis includes the material from the following publications, co-authored by the candidate:

- Carlo Bernaschina, Sara Comai, Piero Fraternali. *"Formal Semantics of OMG's Interaction Flow Modeling Language (IFML) for Mobile and Rich-Client Application Model Driven Development"* [46]
- Carlo Bernaschina, Sara Comai, Piero Fraternali. *"Online Model Editing, Simulation and Code Generation for Web and Mobile Applications."* [45]
- Carlo Bernaschina. *"ALMOsT.js: An Agile Model to Model and Model to Text Transformation Framework."* [43]
  Also presented in the Tutorial:
  Carlo Bernaschina. *"How to cook an Agile Web Based Model Driven Environment in a night"*

- Rocio Nahime Torres, Carlo Bernaschina. *"ALMOsT-Trace: A Web Based Embeddable Tracing Tool for ALMOsT.js."* [125]
- Emanuele Falzone, Carlo Bernaschina. *"Model Based Rapid Prototyping and Evolution of Web Application"* [71]
  Extended in:
  Carlo Bernaschina, Emanuele Falzone, Piero Fraternali, Sergio Luis Herrera Gonzalez. *"The Virtual Developer: Integrating Code Generation and Manual Development with Conflict Resolution"* [Under evaluation at the time of submission of this work]
- Carlo Bernaschina, Sara Comai, Piero Fraternali. *"IFMLEdit.org: Model Driven Rapid Prototyping of Mobile Apps."* [44]

CHAPTER 2

# Background

Methodologies and tools involved in application development have always evolved in order to increase productivity and reduce costs. The introduction of high level languages (e.g. Fortran, C, ...), which enable the ability to support multiple hardware architectures, highly increased software re-usability and reduced development costs. The introduction of the so-called *cross-platform development tools*, which enable the creation and distribution of applications to multiple platforms/operating systems, increased software re-usability and reduced costs even further. Examples of frameworks following this approach are GTK [17], which focuses on User Interface, and QT [24], which provides abstractions for OS dependent primitives.

Similarly, in the mobile environment, specific solutions exploit web development skills and support cross-platform coding in languages such as JavaScript/Java/C#/Dart, CSS and HTML5. Examples of such tools include Appcelerator Titanium [5], IBM MobileFirst Platform Foundation [18], PhoneGap [2], RhoMobile [26], Salesforce [27], Telerik AppBuilder [29], Xamarin [33], Flutter [15] and many others: the developer writes code only once and the tool derives the implementation for different target platforms, including native applications, standard web applications (typically based on HTML5, JavaScript and CSS), and hybrid applications (e.g., embedding HTML5 applications inside native containers that provide access to native platform features).

Industry standard Low Code/No Code development tools [81, 133, 137] and Model Driven Development can be considered as the next step, as they enable even higher productivity, avoiding highly repetitive tasks.

The chapter is structured as follows: Section 2.1 presents the usage of Model Driven Development for application front-ends, Section 2.2 presents a set of Low Code/No Code industrial tool, some of which based on MDD and Section 2.3 will give a background in the Interaction Flow Modeling Language (IFML) [114], a standard modeling

language focused on front-end description, which is the use-case modeling language used in Chapter 3, the focus of the semantics proposed in Chapter 4, the chosen language for the tool presented in Chapter 5 and the use-case modeling language for the evaluation of the work-flow proposed in Chapter 6.

## 2.1 Model-driven development of application front-ends

Several model-driven development approaches have been proposed in the literature to address the generation of code for web and rich clients applications [73], and, more recently, for mobile applications [40]. The use of MDD techniques is reported to promote early detection of software defects, decrease the effort needed for development and maintenance, increase portability to new platforms [110], and, possibly combined with agile techniques, increment productivity and quality [50, 124, 140].

### 2.1.1 MDD approaches based on MDA and UML

The OMG Interaction Flow Modeling Language (IFML) [114] applies the MDA standards to the specification of interactive application front-ends, including mobile and rich client interfaces; model-driven development based on IFML is implemented by the commercial tool WebRatio [32], which supports IFML diagram editing and full code generation of ready-to-publish web and mobile applications.

Other MDA approaches adopt the UML standard diagrams for modeling the front-end of mobile and web applications: [115] employs UML class diagrams and sequence diagrams to represent mobile applications and to generate code for the Android and Windows Phone platforms; Arctis [103] adopts a small UML profile and UML activity diagrams and translates such inputs into a state machine to obtain an executable Android application; The application can be specified using a UML editor for activities with state machines as their external contracts; layout files are created with the Android SDK and linked to the blocks, which encapsulate their behavior. [72] employs UML state machine diagrams to specify GUIs, transitions, and data-flows among application screens, but the internal application logic needs to be coded in JavaScript. Also the work in [119] models and generates graphical interfaces for mobile cross-platforms applications using UML; it expresses the transformation rules in ATL (Atlas Transformation Language).

The above-mentioned approaches show the feasibility of adopting the MDA standards and model transformations to generate the code of the user interface for mobile and rich client applications; our work proceeds in the same line, but starts from an MDA standard expressly designed for modeling the front-end and focuses on the formal specification of the modeling language semantics, as a principled basis for the simulation of models and the generation of code.

[99] and [112] propose metamodels for specific mobile platforms such as Android and Windows Phone 7, respectively.

[49] describes a UML2 Profile for the platform-independent specification, where GUI layout and the transition between screens are described by a Class and a Statechart Diagram, while [99] reports a UML metamodel based approach for the development of Android applications.

In [129] a model-driven approach is used to generate mobile applications for multiple platforms, based on a subset of UML: namely, real use-cases for requirement gathering, class diagram for structural modeling, state machine for behavioral modeling. They also propose a UML profile for modeling mobile domain specific concepts and use Action Language for Foundational Subset of UML (ALF) to specify actions in the state machines. This allows the automatic generation of business logic code for multiple platforms.

### 2.1.2 MDD approaches based on Domain Specific Languages

Other MDD solutions exploit Domain Specific Languages (DSL). The authors of [74] survey model driven approaches specifically targeted at the development of cross-platform mobile applications: they show that proposals are mainly based on textual DSLs, most works are in a prototypical status and adopt a hybrid between coding and modeling, where MDD is used for simple applications and is extended with coding for the more complex functions. For example, $mD^2$ [84] is a prototypical framework where models are specified with a textual DSL, comprising two kinds of view elements: *individual content* and *container*; the former include abstract interaction widgets such as labels, form fields, and buttons, grouped inside containers, which can be organized in a hierarchy; to manage complex navigation scenarios, a workflow can be defined to specify the switch between view containers, possibly guarded with conditions to be fulfilled before the user is allowed to proceed with the navigation. AXIOM [92] is another textual DSL for the mobile application domain: it is based on it own abstract model, specifying at the one side the composition of the application's screen and of its logical UI controls, and at the other side the application's behavior in response to user and system events. Each view is seen as a state; transitions between states are defined by means of attributes on UI controls. Transitions may be optionally associated with guard conditions and actions. AXIOM is completely generative and for each native platform it produces complete implementations, without the need of programmers' intervention. Among the on-line model-driven environments, the system in [67] supports GUI design of mobile applications, but only simple behaviors can be specified; the RAPPT tool [40] generates only the scaffolding of a mobile application based on a high level description specified with a textual DSL, and requires the insertion of manually written code to express the application logic. MDD of native applications in Android and iOS is shown in [132].

## 2.2 Industrial tools for multi-platform application front-ends

In the industrial sector, trends described in Forrester [81] and Gartner [133, 137] research reports show an orientation towards "low-code" development platforms, with Mendix [19] and Outsystems [21] leading this segment of the software tool market. Mendix [19] adopts proprietary graphical models to build complex applications, relying on model interpretation; its applications exploit a hybrid web and mobile code, but can also leverage device functions, thus achieving a near-native user experience. Outsystems [19] supports visual modeling and generates standard Java or .NET applications, connected with a variety of back-end systems: specifications are built using the entity-relationship model for data and a flow-chart notation for the business logic; instead, a WYSIWYG approach is used for the user interface. Finally, the already men-

tioned WebRatio platform [32] adopts IFML as a modeling language, augmented with an own language for describing back-end business actions, and supports full code generation for web and cross platform mobile applications. Advanced GUI features can be manually programmed and incorporated into the IFML model, exploiting the extension mechanisms of IFML, and then integrated in the template-based rules of the code generator.

Some of these tools are effectively MDD based, showing that the approach has found traction in the industry, but the Model Driven Development methodology is a mere implementation detail and not a key aspect of the development work-flow.

Although not comparable in robustness with industrial strength products, our online implementation offers an open source, lightweight MDD environment, grounded on a formally specified and verifiable semantics, which can be used as-is for producing a reference implementation of IFML models, for prototyping and developing rich web and mobile cross-platform applications, and for supporting hands-on MDD education; the tool can be easily extended, in the input modeling language, in the semantic mapping rules, and in the code generation rules, to enable a generative approach to GUI development for any input language and execution platform.

## 2.3 The Interaction Flow Modeling Language

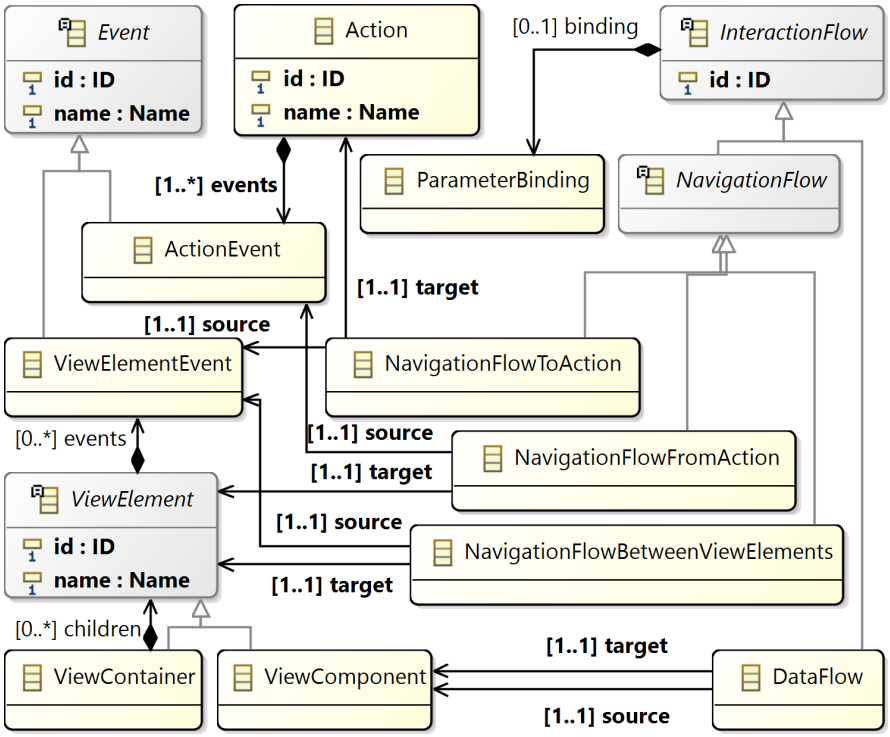



**Figure 2.1:** *IFML Metamodel.*

The Interaction Flow Modeling Language (IFML) [114] is an OMG standard that supports the abstract description of application front-ends for such devices as desktop computers, laptops, mobile phones, and tablets. IFML uses a single type of diagram, in which developers can specify the organization of the interface, the content to be

displayed, and the effect on the interface of events produced by the user interaction or from system notifications. IFML does not represent the business logic of the actions activated by the user interaction with the interface, which can be modeled with the most appropriate UML diagram (e.g., class diagrams for the object model structure, sequence and collaboration diagrams for the behavior).

Figure 2.1 shows the essential elements of the IFML metamodel.

### 2.3.1 Composing the Interface Structure

The essential IFML classifier for the specification of the application structure is the *ViewElement*, which specializes into *ViewContainer* and *ViewComponent*.

*ViewContainers* are the containers into which the interface content is allocated; they support the visualization of content and the interaction of the user. A ViewContainer can be internally structured in a hierarchy of sub-containers. For example, nested ViewContainers can be used to model a rich-client application, where the main window contains multiple tabbed frames, which in turn contain several nested panes, as shown in Figure 2.2a. Figure 2.2b shows an alternative organization, where the user interface is split into different independent ViewContainers corresponding to page templates.

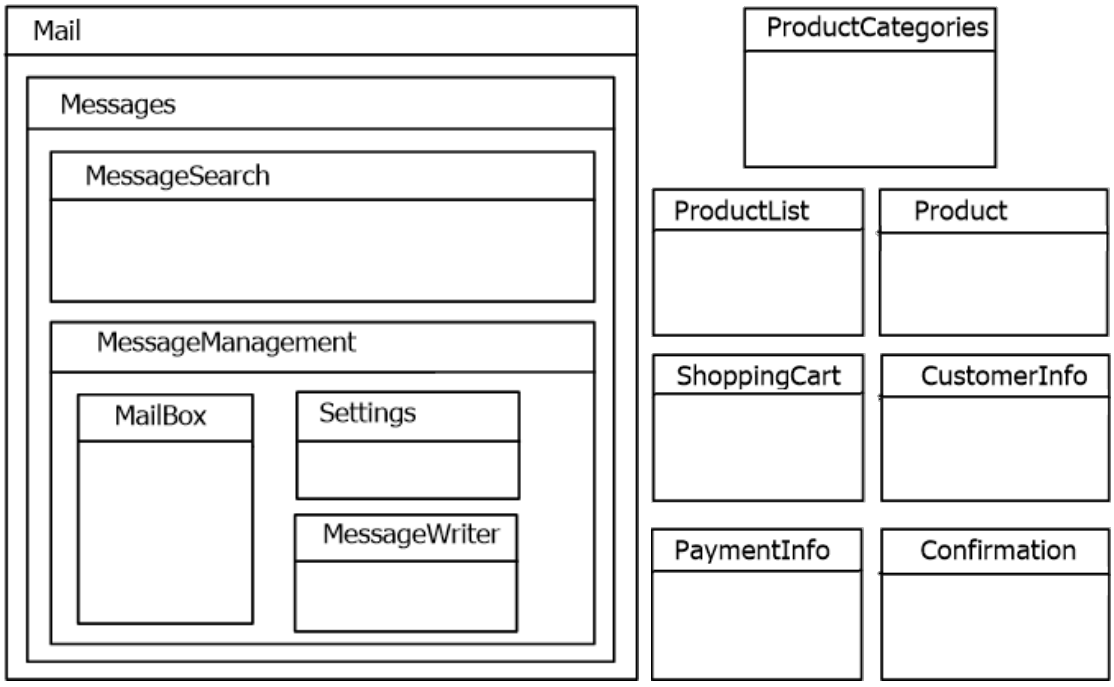



**Figure 2.2:** *Different ViewContainers configurations expressing the interface organization*

ViewContainers nested within a parent ViewContainer can be displayed simultaneously (e.g., an object pane and a property pane) or in mutual exclusion (e.g., two alternative tabs). The alternate visualization of interface portions can be represented by means of a specialization of the generic ViewContainer classifier, the *XOR ViewContainer*. Such ViewElement comprises two or more sub-ViewContainers, with the meaning that its children are visualized one at a time. To make the access to the parent XOR ViewContainer deterministic, one of its children can be tagged as the *default*

XOR child: this means that it is displayed by default when the parent container is accessed. In Figure 2.3, the Mail top-level ViewContainer, tagged with the XOR label, comprises two sub-containers, displayed alternatively: one for messages and one for contacts. When the top level ViewContainer is accessed, by default the interface displays the Messages ViewContainer (tagged with the *D* label).

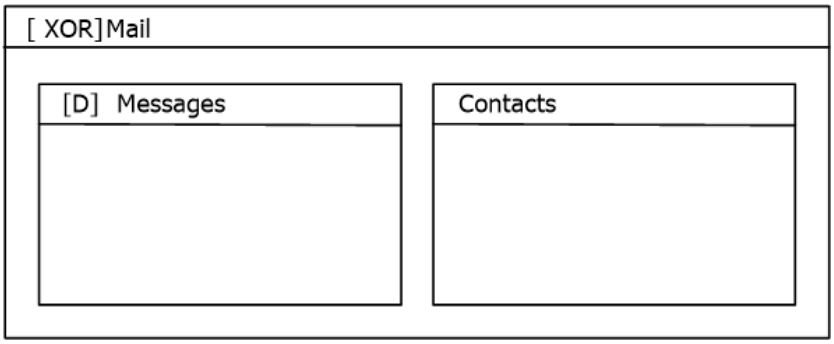



**Figure 2.3:** *Example of mutually exclusive sub-containers*

The switch from one ViewContainer to another one can be expressed implicitly or explicitly, as shown in Figure 2.4. A ViewContainer can be tagged as *Landmark*, denoted with an L label. This property means that the ViewContainer is reachable with a direct navigation step from all the others nested inside the same parent, as explicitly represented in the right part of Figure 2.4.

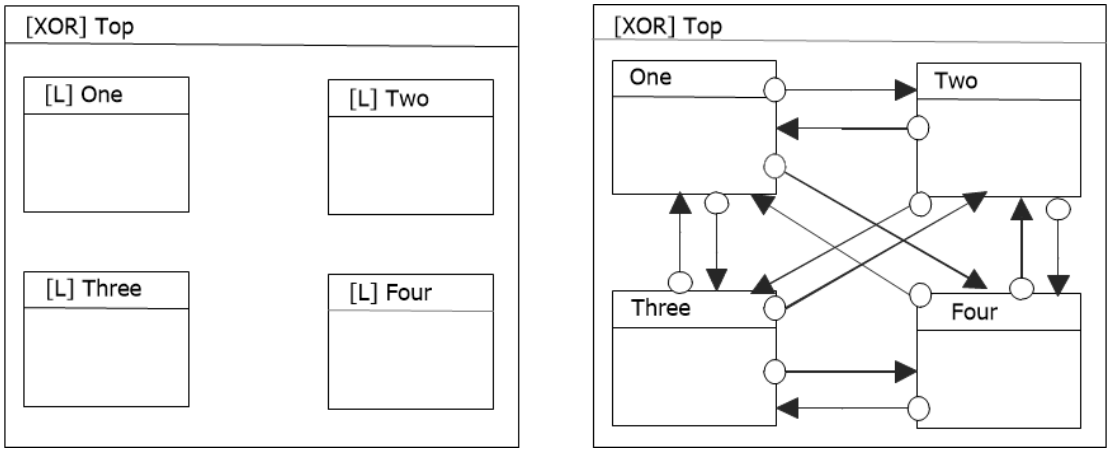



**Figure 2.4:** *Landmark ViewContainers and explicit navigation*

For example, in Figure 2.4, when the One ViewContainer is not in view, it can be switched into view from any other "sibling" ViewContainer currently in view. This behavior is equivalent to specifying the 12 explicit Events and NavigationFlows shown in the right part of Figure 2.4. An *Event*, denoted with a circle, and a *NavigationFlow*, denoted with an arrow, are the essential IFML constructs for specifying the user interaction, as further illustrated later in this Section.

By convention, the whole IFML diagram, which represents the entire application interface, is interpreted as a XOR ViewContainer: the *top-level ViewContainers* of the diagram are displayed one at a time, in alternative (Figure 2.2b provides an example).

A ViewContainer can include *ViewComponents*, which denote the actual content of the interface. The generic ViewComponent classifier can be stereotyped, to express different specializations, such as lists, object details, data entry forms, and more.

Figure 2.5 shows the notation for expressing ViewComponents, stereotyped and embedded within ViewContainers: Search comprises a MessageKeywordSearch *Form*, which represents a form for entering data; MailBox includes a MessageList *List*, which denotes a list of objects; finally, MessageViewer comprises a MessageContent *Details*, which displays the data of an object. ViewComponents can have input and output parameters. For example, a ViewComponent that shows the details of an object has an input parameter corresponding to the identifier of the object to display; a data entry form exposes as output parameters the values submitted by the user; and a list of items exports as output parameter the identifier of the selected item.

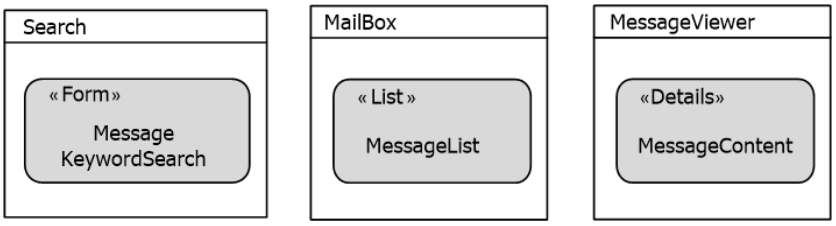


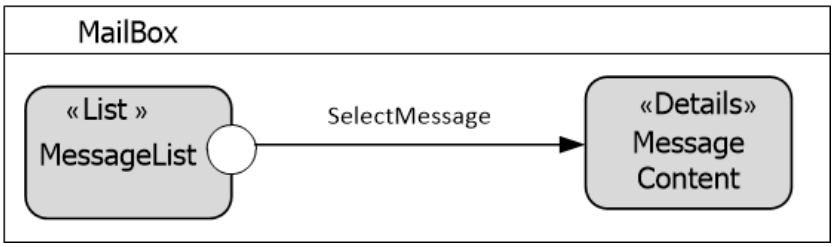

**Figure 2.5:** *Example of ViewComponents within view containers*

### 2.3.2 Events, Navigation and Data Flows

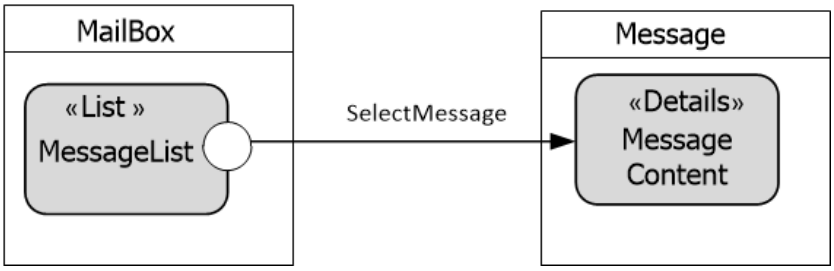
*ViewElements* (ViewContainers and ViewComponents) can be associated with *Events*, to express that they support the user interaction. For example, a ViewContainer can be associated with an Event to display or navigate to another ViewContainer (as in Figure 2.4), a List ViewComponent with an Event for selecting one or more items (as in Figure 2.6), and a Form ViewComponent with an Event for input submission. IFML Events are mapped to interaction widgets in the implemented application; the nature of such widgets depends on the specific platform and is not captured by IFML. The effect of an Event is represented by a *NavigationFlow*, represented with an arrow, which connects the Event to the ViewElement affected by it. For example, in an HTML web application an event can be the selection of one item from a list by means of a link, and its effect is the display of a new page with the details of the chosen object. A NavigationFlow is denoted as an arrow, connecting the Event associated with the source ViewElement to the target ViewElement (as shown in Figures 2.4 and 2.6). The NavigationFlow specifies a change of state of the user interface: the target ViewElement of the NavigationFlow is brought into view, after computing its content; the source ViewElement of the NavigationFlow may remain in view or switch out of view depending on the structure of the interface.

Figure 2.6 shows two examples of NavigationFlows between ViewComponents. In Figure 2.6a, the NavigationFlow associated with the SelectMessage Event connects its source (MessageList, which displays a list of objects), and its target (MessageContent, which displays the data of an object). When the Event occurs, the content of the target ViewComponent is computed so to display the chosen object, and the source remains in view since it is in the same ViewContainer. In Figure 2.6b the source and target

ViewComponents are positioned within distinct top-level ViewContainers (MailBox and Message); the triggering of SelectMessage causes the display of Message, with the enclosed ViewComponent, and the replacement of MailBox, which gets out of view.




**(a)** *NavigationFlow between ViewComponent in the same ViewContainer*




**(b)** *NavigationFlow between ViewComponent in different ViewContainers*

**Figure 2.6:** *Example of NavigationFlow between ViewComponents*

The specification of ViewComponents and NavigationFlows can be made more precise, by showing the objects from which the content of the ViewComponent derives, the inputs and outputs of ViewComponents, and the parameter passing from the source to the target ViewComponent.

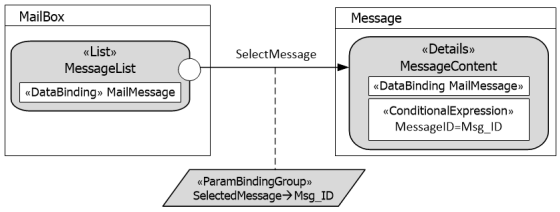



**Figure 2.7:** *Example of DataBinding, ConditionalExpression, and ParameterBindingGroup*

Figure 2.7 shows a refined specification: the ViewComponents comprise a *DataBinding* element that identifies the data source, which can be a persistent entity or an object class defined in the Domain Model of the application. Both the ViewComponents in Figure 2.7 extract their content from the MailMessage entity. The MessageContent Details ViewComponent also comprises a *ConditionalExpression* sub-element; this denotes a filter condition used to query the data source and extract the content relevant for publication. In Figure 2.7 the ConditionalExpression is parametric: it extracts the object whose MessageID attribute value equals the value of the Msg_ID parameter. The parametric ConditionalExpression defines an input dependency for the associated component. To be computed, the MessageContent Details ViewComponent requires an input value for the condition parameter; in Figure 2.7, such parameter value is supplied as a side effect of the triggering of the SelectMessage Event. The parameter

passing rule is represented with a *ParameterBindingGroup* element associated with the Navigation Flow, which couples an output parameter of the source ViewComponent (the object selected from the List ViewComponent) to an input parameter of the target ViewComponent (the object displayed by the Details ViewComponent).

The example of Figure 2.7 shows that NavigationFlows have a twofold role: they enable the expression of the effect of an Event in the interface and the specification of parameter passing rules. These two roles are orthogonal and can be separated. The model in the right part of Figure 2.4 illustrates the case in which the effect of the Event on the interface is expressed independently of a parameter passing rule: each NavigationFlow in the example has no parameters and only changes the ViewContainer currently in view.

The dual case occurs when a parameter passing rule is expressed independently of an interaction Event. This design pattern exploits the IFML *DataFlow* construct, which represents an input-output dependency between a source and a target ViewElement and is denoted as a dashed arrow.

Figure 2.8 shows the *DataFlow* construct, representing an input-output dependency between a source and a target ViewElement, denoted as a dashed arrow. In the example, MailViewer includes three ViewComponents: the MailMessages *List* is defined on the MailMessage entity, which is explicitly specified in this example, and shows a list of messages; the MessageContent *Details* is also defined on the MailMessage entity and displays the data of a message; the Attachments *List* is defined on the Attachment entity and shows a list of mail attachments.

To express data dependencies, links are associated with parameter binding elements that specify how data flow between components. Parameter bindings, denoted by the *ParamBinding* element in Figure 2.8, connect one or more parameters exposed by a source component with corresponding parameters accepted by a target component. In the example, the identifier of the selected message is passed from MailMessages to MessageContent. The latter has a parametric *ConditionalExpression*: it denotes a filter condition used to query the data source and extract the content relevant for publication; in the example, it extracts from the data source the message with the identifier provided as input to the component. Also Attachments has a parametric ConditionalExpression, used to select for display the attachments associated (through the AttachedTo relationship) with the mail message provided as input to it.

When the ViewContainer is accessed, the list of messages is displayed, which requires no input parameters. The DataFlow between MailMessages and MessageContent expresses a *parameter passing rule* between its source and target: even if the user does not trigger the Select Event, an object is randomly chosen from those displayed in the MailMessages *List* and supplied as input to MessageContent, which displays its data. Similarly, the DataFlow between the MessageContent and Attachments specifies an automatic parameter passing rule that supplies the parameter needed for computing the list of attachments, independently of the user interaction. By triggering the Select event associated with the MailMessages *List* the user can choose a specific message from the list and determine the display of its content and attachments, thus overriding the default content shown at startup.

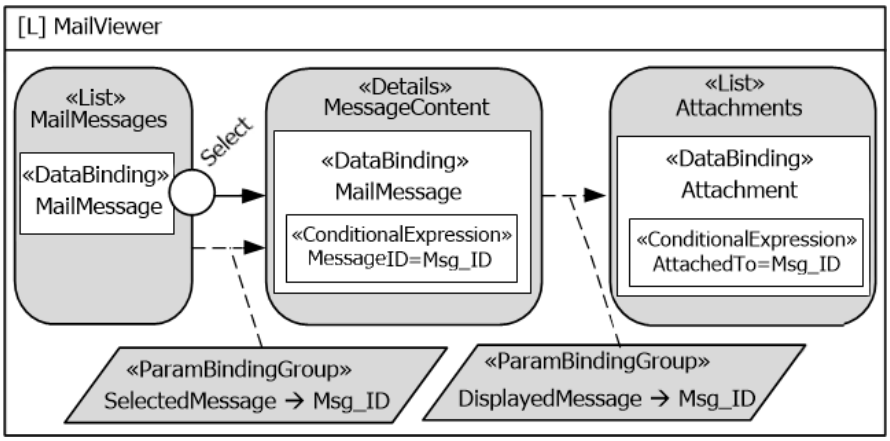



**Figure 2.8:** *Example of DataFlows*

### 2.3.3 Actions

An Event can also cause the triggering of a piece of business logic, which is executed prior to updating the state of the user interface; the IFML *Action* construct, represented by an hexagon symbol as shown in Figure 2.9, denotes the invoked program, which is treated as a black box, possibly exposing input and output parameters. The effect of an Event firing an Action and the possible parameter passing rules are represented by a NavigationFlow connecting the Event to the Action and possibly by DataFlows incoming to the Action from ViewElements of the interface. The termination of the Action may cause a change in the state of the interface and the production of input parameters consumed by ViewElements; this is denoted by termination events associated with the Action, connected by NavigationFlows to the ViewElements affected by the Action.

Figure 2.9 shows an example of Action, for the creation of a new object.

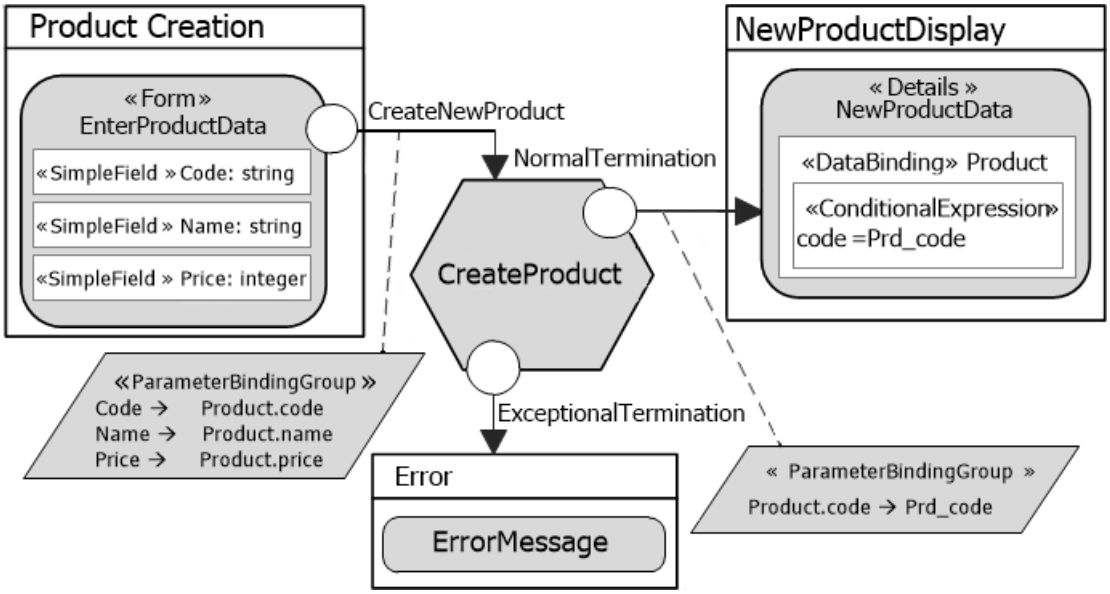



**Figure 2.9:** *Example of Actions*

ProductCreation includes a *Form* with *SimpleField* sub-elements for specifying the

data entry of a new product. The CreateNewProduct Event triggers the submission of the input and the execution of the CreateProduct Action. A ParameterBindingGroup is associated with the NavigationFlow from the CreateNewProduct Event, to express the parameter binding between the *Form* and the Action. The Action has two termination Events: upon normal termination, the code of the new product is emitted as an output parameter, used as input to calculate the NewProductData Details within NewProductDisplay; upon abnormal termination (e.g., a database connection error), a distinct Event and NavigationFlow pair specifies that an alternative ViewContainer is displayed, which contains an error message ViewComponent.

### 2.3.4 Extensions

Given the abstract nature of the IFMLs description of application front-ends, the defined models lack the level of details needed to fully generate a final implementation. It is important to notice that abstraction is a major feature of IFML and not a limitation, freeing the modeler from the necessity to define the final rendition of each element on screen.

Various works focused on the extension of IFML or the integration with other modeling languages and tools in order to produce models and implementations with a higher level of details.

In [106] an approach for the generation of Rich Internet Applications (RIA) using IFML and the second revision of the Web Ontology Language (OWL2) has been proposed. The high level IFML description is merged with a OWL Ontology via model transformations introducing layout and styling details.

In [118] a JavaFX specific extension of IFML is proposed. The enhanced model contains RIA specific details which can be exploited by subsequent Model-to-Text transformations to generate a full implementation.

In [52] a mobile specific, but platform independent, extension of IFML is proposed. Graphical concepts (e.g. *Screen*, *ToolBar*, ...), shared by the majority of mobile UI frameworks, are introduced by means of stereotypes over the ViewElements used for the structural description of the application. Particular attention has been given to the description of mobile specific interactions (e.g. *Swype*) and Events (e.g. *Notifications*, *GPS Location Update*).

In [53] the previous work has been extended with the introduction of concepts specific to the emerging field of IoT devices and systems. IoT specific Actions and Events are introduced by means of stereotypes on the model elements. The biggest contribution of this work is the definition and analysis of IFML patterns for Data Synchronization and IoT Interaction.

CHAPTER 3

# Agile MDD Tools Development

Model Driven Development (MDD) is the branch of software engineering that advocates the use of *models*, i.e., abstract representations of a system, and of *model transformations* as key ingredients of software development [98]. With MDD, developers use a general purpose (e.g. UML [30]) or domain specific (e.g., IFML [114]) modeling language to portrait the essential aspects of a system, under one or more perspectives, and use (or build) suitable chains of *transformations* to progressively refine the models into executable code.

Both general purpose and domain specific modeling languages pose their own challenges to the developers, which explain the alternate fortunes of MDD adoption in the industry [88, 134].

General purpose languages, such as UML, aim at spanning the entire spectrum of software applications under all possible perspectives and thus are large and complex. It is hardly the case that developers can afford building their own MDD development environment, including such aspects as model editing, verification and code generation, because such a task could easily exceed the effort of the actual application to be delivered. Therefore, general purpose modeling languages must be supported by dedicated tools, which in turn present, despite the claims of interoperability, problems of so-called vendor lock-in. On the other hand, domain specific modeling languages [96] have been proposed to overcome the complexity and steep learning curve of general purpose approaches; their abstractions do not pretend to be universal, but embody domain knowledge that make the language speak in terms closer to the intuition of the developer and of the application stakeholders. On the negative side, though, domain specific modeling languages have a more restricted user base, which prevents in many cases the development of industrial-strength supporting tools. Domain specific modeling language may be even used only by one organization, which tailors them to its

specific needs and development culture. For this class of languages, the need arises of creating and maintaining the tool chain necessary for bridging the gap between model specification and the real application deployment. Such a tool chain revolves around one or more domain specific *model transformations*. A such transformation is a procedure that takes in input a model, plus some additional domain or technical knowledge, and outputs a more refined model (yielding, at the end of the chain, the source code, which can be seen as the most concrete model). Creating model transformations is not itself an easy task. It is analogous to the construction of a compiler, which requires mastering very specific abilities, including meta-modeling (i.e., the specification of the grammar of the input language), definition of language semantics, and implementation of mapping rules. Presently, model transformation languages, such as ATL [94] and OMG QVT [25], and their support development environments, such as the ATL Integrated Environment (IDE) for Eclipse [7], are tools for the experts. They require specific skills regarding the language syntax and semantics and programming style. In many cases, organizations create and evolve a supporting MDD environment alongside their applications or product lines. Developers recognize recurring patterns of abstraction in their requirements and designs and tend to factor them out and promote them into abstractions of an emerging domain specific modeling language. Then, they try to boost reuse by generating code from the instantiation of their own abstractions. It is easy to see that this way of affording development requires the application of effective software engineering practices not only to the application, but also to its supporting MDD environment. Specifically, the well-known principles of agility should be seamlessly applied to both focuses of development, especially in the today's scenarios where scarcity of time and resources and high technological variability are the norm.

To address the agile development of MDD support tools, this chapter presents **ALMOsT.js**, an agile, in-browser framework for the rapid prototyping of MDD transformations, which lowers the technical skills required for developers to start be proficient with modeling and code generation. The contributions of this chapter can be summarized as follows:

- We describe ALMOsT.js, an MDD development framework that exploits well-know technical standards (JavaScript, Node.js and the JointJS GUI library) to help developers edit models in their formalism of choice and create transformations quickly and without complex meta-modeling steps.
- We discuss how ALMOsT.js can be used with an agile methodology, by rapidly prototyping, verifying, and evolving model editors and transformations.
- We illustrate how in ALMOsT.js developers specify transformations with a simple rule language that extends JavaScript naturally.
- We evaluate the effectiveness of ALMOsT.js in a case study in which the framework has been applied to a real world MDD scenario: the construction of an agile development environment for the OMG Interaction Flow Modeling Language (IFML) [114], called IFMLEdit.org[1] (presented in Chapter 5). ALMOsT.js has been used to create an IFML model editor, a model to model transformation mapping IFML to Place Chart Nets (PCNs) [97] for verifying the correctness of

[1]The environment is online and can be used at http://ifmledit.org/

models via analysis and simulation (presented in Chapter 4), and four model to text transformations for automatically generating Web and cross-platform mobile code (presented in Chapter 5).

## 3.1 Background

### 3.1.1 Agile model driven development

Agile software development [23] is an incremental and iterative approach based on principles that aim at increasing productivity and adherence to requirements, while keeping the process as lightweight as possible. Agile Model Driven Development has been advocated as a promising approach [127], which has not yet fully expressed its potential [82, 108]. Its idea is to organize the MDD process in ways that take advantage of the agile development principles:

1. Enabling an incremental and iterative development cycle by using toolchains able to test and validate even incomplete models [95].
2. Applying a Test-Driven Development, a distinctive feature of extreme programming [42], to MDD. [35, 124].
3. Merging agile workflows such as SCRUM [123] with MDD in novel methodologies, to achieve system-level agile processes [140].

The above mentioned approaches mainly focus on enhancing the development cycle of the final product, assuming a predefined set of modeling languages and transformations. The rapid evolution of new technologies and the short time-to-market required today highlight the need to integrate modeling languages and tools in the loop [64, 131], co-evolving iteratively the modeled final product and the MDD tools and languages. Accordingly, ALMOsT.js aims to foster agile processes in MDD not only when the models and transformations have already been chosen and are stable, but also when developers need a lightweight development environment that can be exploited to deliver minimum viable products rapidly, for the model concepts and transformations as well as for the target applications.

### 3.1.2 Model transformation languages and tools

Model transformations are supported by many languages [62, 100] and implemented by an even greater number of tools [57]. We limit the discussion to the works that are more relevant to our approach. Several works focus on domain specific languages and on different aspects of model transformation, as extensively discussed in [47, 56, 62]. The proposed transformation languages employ different styles:

1. **Declarative**: transformations exploit mappings between elements of the input and of the output metamodel. An example is EMF Henshin [39], which focuses on model-to-model transformation using triple graph grammars. **Query/View/-Transformation** (QVT) [105] is a standard model transformation language by the Object Management Group (OMG). It is actually based on three languages (QVT Relational, QVT Core, QVT Operational) which are used to define model transformation at different levels of abstraction. Two of the engine which are conform with the QVT standard are SmartQVT and ModelMorf.

2. **Imperative**: transformations are programmed as a sequence of operations. An example is Kermeta [70] an imperative programming language able to perform model transformations.

3. **Hybrid**: transformations mixing the declarative and imperative style. An example is ATLAS Transformation Language (ATL) [93] where declarative mappings can be extended by imperative constructs. The ATL transformation engine also automatically identifies the optimal rules execution order.

Some languages focus on particular features like Epsilon Transformation Language (ETL) [100], which is a model-to-model transformation language that can handle more than one source and target model at a time. Model-to-text transformation languages, such as Xpand, the imperative language part of OpenArchitectureWare, are generally template-based. Code is embedded inside plain text, making it easy to convert a source or configuration file into a template.

As illustrated in Section 3.2, with ALMOsT.js we aimed at quite a different objective, trading completeness for simplicity: reusing a popular, general purpose language, in which many developers are already proficient, to build a lightweight MDD environment whereby developers can design domain specific languages and the associated transformations rapidly and without resorting to external tools. The design decision of using a general-purpose language is further justified by experimentation performed in [83], showing that domain specific languages doesn't present a statistically significant advantage w.r.t. a general-purpose counterpart, at least the investigated use-cases.

## 3.2 The ALMOsT.js Framework

In this section, we describe the architecture of ALMOsT.js (AgiLe MOdel Transformations), its rule language, and showcase its usage for customizing model editors and for building model to model and model to text transformations. For concreteness, the examples of data objects and rules are drawn from a simplified running example, and further expanded in Chapter 5.

### 3.2.1 Requirements

Before introducing the architecture and use of ALMOsT.js, we pinpoint the requirements for its development that make the resulting environment amenable for use within an agile software development methodology. Here, we focus on the core characteristic of agile development methodologies, such as SCRUM [122], which commands developers to create minimum viable products rapidly and evolve them via frequent iterations (sprints). As a reference usage scenario for ALMOsT.js we imagine a software team, who has elicited the requirements for an application or product line and decides to exploit MDD in its development, by progressively factoring out domain abstractions from requirement and building a support environment for creating models, verifying them and generating code, in an iterative way. This scenario does not rule out the complementary case in which a tool company wants to implement rapidly and evolve over time a support environment for an existing standard or proprietary MDD language.

Under the above mentioned drivers, the requirements at the base of ALMOsT.js can be summarized as follows:

1. **No installation**. It must be possible for the team to use the framework instantly, with no installations.

2. **No new language**. It must be possible to start using the environment without learning languages that are not normally employed for application development.

3. **Fast start-up**. It must be possible to create a minimum viable model editor and model transformation in a very short time (e.g., less than one day).

4. **Parallel development**. It must be possible to work in team on different aspects of the same sprint, e.g., by adding a new concept to the model editor, and the corresponding generative rules to the model to model transformation and model to text transformation in parallel.

5. **Customized output.** There must be an easy and standard path to turn the (possibly prototypical) generated code into a complete version, by adding (manually) the missing aspects.

6. **Customized generation.** It must be possible to tailor the non-functional aspects of the generated code easily, e.g., by adding graphics and sample data collections for early validation of the product with stakeholders.

### 3.2.2 Architecture

In this section we show how we addressed requirements #1 (No installation) and #2 (No new language) in the design of the ALMOsT.js architecture. ALMOsT.js has the simple organization illustrated in Figure 3.1. The implementation has been realized entirely in JavaScript, which is also the only language needed by developers to plug their domain models and transformations into ALMOsT.js, in observance of the above mentioned requirements.

At the core of the framework, the **Data Model** comprises the domain model objects, encoded according to the minimalist structure explained in Section 3.2.3. Objects of the data model are created and manipulated by the **Model Editor**, a GUI component, running in the browser, with the usual functions for creating and modifying projects consisting of MDD diagrams, and for storing, retrieving, manipulating and annotating models and their constituents. Alternative Model Editors can be plugged-in, as far as they are able to create objects of the Data Model, making ALMOsT.js independent of the model editing GUI (more details on the Model Editor are provided in Section 3.3).

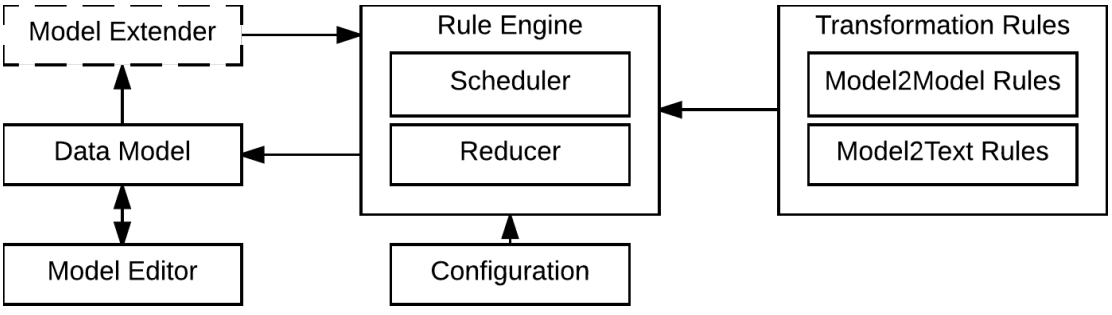



**Figure 3.1:** *Architecture of ALMOsT.js*

The objects of the data model are also read and written by the model transformations, encoded as **Transformation Rules** in the **Model2Model** and **Model2Text** components. To avoid inventing yet another transformation language, transformation rules are coded as pure JavaScript functions, according to the prototypical signatures illustrated in section 3.2.4. This choice also simplified the implementation of the **Rule Engine**, which employs the standard function call mechanism of JavaScript to orchestrate the execution of rules. The Rule Engine exploits the stateless and side effects free nature of model transformation and employs a functional approach. Once configured, it acts as a main loop iterating one simple function, which takes as input a model object (described in Section 3.2.3) and returns as output the aggregated results of applying all the relevant rules to it. The Rule Engine function is internally organized in two sub-functions:

#### Scheduler

The Scheduler is responsible of matching the input rules against the model objects (see Section 3.2.4), schedules rule execution, and collects their outputs. At each round of the main loop of the Rule Engine, the Scheduler manages the matching of the rules to the objects of the model and determines the rules to fire in the current round. The order of rule execution is pseudo-random: it depends on the order of definition of rules and on the order of creation of the model objects. All the values produced by the execution of the rules (e.g. partial models, partial file tree) are aggregated into a final result (e.g. a model, a file tree).

#### Reducer

The Reducer is the sub-function of the Rule Engine responsible of aggregating the results created by the execution of the Scheduler following the Map-Reduce approach [65].

The Reducer is fully customizable, allowing the developer to define the output structure most suitable for the problem at hand. A custom Reducer is composed of *reduce* function, which is responsible of merging pairs of elements, an optional *initialization* value, which is used as first element in the sequence, and an optional *finalize* function, which enables complex operations (e.g. computing the average of a sequence, dividing the sum of the elements by the number of element).

ALMOsT.js provides a set of predefined modular reduction policies, which can be used to easily configure complex aggregation schema by simple configurations.

- `none()` A Naïve policy which accepts no value. While it has no practical value on its own, it can be used in conjunction with compound policies.
- `single()` A Naïve policy which accepts one single value. While it has little practical value on its own, accept validating that just one rule is fired, it can be used in conjunction with compound policies.
- `first()` A Naïve policy which returns the first value encountered. Given the non-deterministic nature of the scheduler it should not be used on its own, but just in special compound cases (see `reduceBy()` and `groupBy()`.
- `concat()` All the aggregating values are concatenated into an array.

- `mergeOrSingle()` When a single value is provided it behaves like `single()`. When more values are provided they must be objects and are aggregated into a single one sharing all of their members. If more than one object share the same member key the related values are aggregated following the same policy.
- `flatten(policy)` It is a composite policy which enhances the behavior of another one. Values which are arrays are provided to the nested policy one item at a time. The default nested policy is `concat()`.
- `merge(policy, special)` It is a composite policy specific for object values. The nested policy is applied member-wise effectively merging the objects together. Optionally it is possible to specify custom policies for specific member keys via the `special` attribute, which is a key-policy map. The default nested policy is `mergeOrSingle()`.
- `reduceBy(key, policy)` It is a composite policy, specific for object values, which enhances the behavior of another one. Elements with the same `key` member value are aggregated together following the nested policy. It is responsibility of the developer to treat the `key` member accordingly in nested policies. The default policy is `merge(mergeOrSingle(), {key: first()})`, which is equivalent to a `mergeOrSingle()` preserving the reduction key.
- `groupBy(key, policy)` It is a composite policy, specific for object values, which enhances the behavior of another one. Elements with the same `key` member value are aggregated together following the nested policy. The result is a key-value map where values are the result of the nested policy. The default policy is `concat()`.
- `lazy(policy)` It is a composite policy, which enhances the behavior of another one. If no value is passed to the reducer the nested policy is not considered. After the first value has been provided the nested policy is invoked as expected. It can be used in conjunction with policies having a default value (e.g `concat()`, `merge()` or `groupBy()` to avoid to generate the default result if no value is passed.

As a small example we describe the reduction policy related to a directed graph like structure. The rules generate a set of nodes which have a **name** and a set of **neighbors**. Each rule generates a partial graph (one or more nodes) and it can be reduced by the following policy:

While the Rule Engine can be configured to obtain different types of transformations in Section 3.2.5 and Section 3.2.6 we will show three predefined configurations which describe Model-To-Model and Model-To-Text transformations.

### 3.2.3 Data Model

As shown in Figure 3.1, the data model underpins all the components of ALMOsT.js. The artifacts of the data model can be regarded as a minimalist subset of the entities of the Essential Meta Object Facility (EMOF) OMG metamodel standard [20]. To cope with requirements #2 (No new language) and #3 (Fast start-up) the concepts of the data model are exposed as plain JSON objects, which facilitates creation, storage,

```
flatten( // Allows rules to generate more than one node
  reduceBy( // Performs the aggregation at node level
      'name', // The key which identifies nodes
    merge(  // Aggregates the nodes
      none(), // Just the 'name' and 'neighbors' members are valid
      {
        'name': first(), // Preserve the name of the node
        'neighbors': flatten() // Concatenate all the neighbors
      }
    )
  )
)
```

**Listing 1:** *Reducer for graph-like structure*

manipulation, and migration between components. For example, developers can use model editors of their choice, provided that they can generate the JSON objects of the data model. Each MDD project must include a *model* object, which represents the entry point of the domain model; this is a root container object with two members:

1. **elements**: an array containing all the elements in the model.
2. **relations**: an array containing all the relations between elements in the model.

```
{
  "elements": [

  ],
  "relations": [

  ]
}
```

**Listing 2:** *The Model structure*

Developers can start from this minimal assumption to fill a model object with their own model concepts (elements and relations). Even if ALMOsT.js does not require any particular structure for elements or relations, we propose a simple format, which facilitates element type identification, relations navigation, and separation of concerns between the data that are a proper part of the elements (attributes) and the descriptive information associated with them (metadata). According to this proposed format, each entry in the **elements** array must be an object (in JSON) endowed with the following properties:

1. **id**: a string that uniquely identifies the element in the model. It is used as a reference by relations.
2. **type**: a string that identifies the type of the element. No formal type system is assumed, type names are completely user-defined. Specifying types for model elements helps rule condition expressions to locate matching elements by simply checking their type name (see the examples of rule conditions in Section 3.2.4).

3. **attributes**: an object, whose members are domain specific properties of the enclosing element.

4. **metadata**: an object, whose members are descriptive properties of the enclosing element. Rules (see Section 3.2.4) should not take any decision based on metadata and should use them just as optional information that helps the output generation. Examples of metadata are graphical information (e.g., position and size of a model element in the graphical model editor) or debug information.

The following JSON object illustrates the representation of a model element: the *Mails* object of type *ifml.ViewContainer*.

```
{
  "id": "mails",
  "type": "ifml.ViewContainer",
  "attributes": {
    "name": "Mails",
    "landmark": true,
  },
  "metadata": {
    "graphics": {
      "position": { "x": 100, "y": 50},
      "size": { "width": 160, "height": 90}
    }
  }
}
```

**Listing 3:** *Example of Element structure*

Each entry in the **relations** array is a (relation) object, endowed with the mandatory **type** member, which describes the semantics of the relation. As for elements, no assumption is made on the available types of relations, which can be defined by the developer. The other members of a relation object must be references to elements in the **elements** array.

The following JSON object represents a hierarchy relationship between two model elements, *mails* and *mails-list*.

```
{
  "type": "hierarchy",
  "parent": "mails",
  "child": "mails-list"
}
```

**Listing 4:** *Example of Relation structure*

### 3.2.4 Transformation Rules

As per requirement #2 (No new language), we did not create a custom rule language; the entire tool, including the rule language, is based on the JavaScript programming language. All transformations are expressed as Condition-Action (CA) rules [135],

defined as pairs of plain JavaScript functions. Rule definitions are stored in JavaScript source format in the **Model to Model Rules** and **Model to Text Rules** components of figure 3.1. A CA rule simply consists of:

1. **Condition**. A function verifying if the current rule can be applied. It must return a Boolean value (or anything that can be interpreted as a Boolean value) informing the Rule Engine that the rule is activated and can fire.
2. **Action**. A function responsible of mapping the inputs to a series of outputs. The output of an action function must be a JSON object. Each member of these objects is treated separately and passed to the **Reducer** component for conflict resolution and aggregation (as explained in Section 3.2.2).

The condition part of a rule may match specific elements or relations in the model, or a whole model. Three type of rules can be identified based on their input:

1. **Model rules**. The input is the whole model.

```
createRule(
  function (model) { /* Condition */ },
  function (model) { /* Action */ }
)
```

**Listing 5:** *Model Rule signature*

2. **Element rules**. The input is a specific element.

```
createRule(
  function (element, model) { /* Condition */ },
  function (element, model) { /* Action */ }
)
```

**Listing 6:** *Element Rule signature*

3. **Relation rules**. The input is a specific relation.

```
createRule(
  function (relation, model) { /* Condition */ },
  function (relation, model) { /* Action */ }
)
```

**Listing 7:** *Relation Rule signature*

#### Model rules

These rules can be used for setup purposes, e.g., for inserting fixed outputs into a model. The example below shows the declaration of a model rule.

The (model to text) rule contains a condition function that receives in input a model object and tests if it contains elements. The action part simply returns a piece of the JSON configuration that specifies a fixed-name project folder.

```
createRule(
  // Condition function
  function (model) {
    return model.elements.length > 0;
  },
  // Action function
  function (model) {
    return {
      project: { type: "folder", name: "myProject" }
    };
  }
);
```

**Listing 8:** *Example of Model Rule*

#### Element rules

An element rule has a condition function that matches a specific model element and maps it (and possibly other related ones) into others elements and/or relations. Note that element rules can be used to identify patterns in the source model, by writing a condition part that traverses the model graph starting from the matched element. The following declaration illustrates the structure of an element rule.

```
createRule(
  function (element, model) {
    return element.type === "ifml.ViewContainer";
  },
  function (element, model) {
    return {
      elements: [{
        id: element.id,
        type: "pcn.PlaceChart",
        attributes: { name: element.name }
      }],
      relations: []
    };
  }
);
```

**Listing 9:** *Example of Element Rule*

The (model to model) rule is activated by a match with an element of a given type (`ifml.ViewContainer`) and produces as output a model element of another type (`pcn.PlaceChart`), with the same Id and name of the input element.

#### Relation rules

A relation rule has a condition that matches a specific relation and can be used to map relations into other relations and/or elements. Also relation rules can be used to identify patterns in the model, by writing conditions that traverse the model graph starting from the matched relation. An example of relation rule is as follows.

The above rule matches an input relation object (of type `hierarchy`) and creates

```
createRule(
  function (relation, model) {
    return relation.type === "hierarchy";
  },
  function (relation, model) {
    var id = relation.child + "-init";
    return {
      elements: [{
        id: id,
        type: "pcn.Transition",
        attributes: {}
      }],
      relations: [{
        type: "source",
        transition: id, source: relation.parent
      }, {
        type: "target",
        transition: id, target: relation.child
      }]
    };
  }
);
```

**Listing 10:** *Example of Relation Rule*

as outputs one element (of type `pcn.Transition`) and two relations associated with it, one of type `source`, associated with the parent of the matched relations, and one of type `target`, associated with the child of the matched relation[2].

### 3.2.5 Model to Model transformation

In the proposed approach for model to text transformations, each rule is responsible of generating an output partial model, following the structure presented in Section 3.2.3 (i.e., possessing only the three *elements*, *relations* and *metadata* members). The predefined, and default, **m2m** (Model-To-Model) configuration for the Reducer component is provided to the developer. This simple configuration follows the minimal model definition of ALMOsT.js and so is not opinionated on the structure of neither *elements* nor *relations*.

```
merge( // The partial models are merged
  none(), {
  'elements': flatten(), // Elements are concatenated
  'relations': flatten() // Relations are concatenated
})
```

**Listing 11:** *M2M: Model To Model predefined Reducer*

The non opinionated nature of the default configuration gives a high degree of freedom to the developer at the cost of imposing that each output element must be created

[2]The rule transforms a hierarchical nesting relationship between IFML ViewContainers into an initialization PCN transition, which connects the place associated with the enclosing container to that associated to the enclosed one.

by a single rule. To enable the ability of generating partial *elements* in different rules and let the Reducer aggregate them the predefined **m2a** (Model-To-ALMOsT) configuration is provided. This more complex configuration follows both the minimal model

```
merge( // The partial models are merged
  none(), {
  'elements': flatten( // Allows rules to generated multiple elements
    reduceBy( // Elements with the same id are merged
      'id',
      merge(
        none(), { // The Elements should follow the suggested structure
        'id': first(), // Preserve ID
        'type': single(), // The type should be generated once
        'attributes': merge(), // Preserve all the attributes
        'metadata': merge() // Preserve all the metadata
      })
    )
  ),
  'relations': flatten() // Relations are concatenated
})
```

**Listing 12:** *M2A: Model To ALMOsT predefined Reducer*

definition of ALMOsT.js and the suggested *elements* and *relations* structure.

### 3.2.6 Model to Text transformations

In the proposed approach for model to text, each rule generates a subset of the folders and files that constitute the project.

The following piece of JSON shows the output of a model to text rule that creates a file (named **package.json**) inside a folder (named **project**). The predefined, **m2t**

```
{
  "project": {
    "name": "project",
    "isFolder": true,
    "children": ["package"]
  },
  "package": {
    "name": "package.json",
    "content": "...Source File Content..."
  }
}
```

**Listing 13:** *Model To Text example output*

(Model-To-Text) configuration for the Reducer component is provided to the developer. In this simple configuration the final aggregated result is an object where each member is a node of the file-system tree describing a *folder* or a *file*. The relation between parent folders and children files or folders is simply tracked as a *children* property of the parent. In this way, for example, each model to text rule can simply generate code dealing with a specific aspect of the matched element (e.g., Java code or JSON

```
merge(
  merge(
    none(), {
    'name': single(),
    'isFolder': single(),
    'content': single(),
    'children': flatten()
  })
)
```

**Listing 14:** *M2T: Model To Text predefined Reducer*

configuration) and the value of the *children* property of the folder that must contain it, without worrying about other aspects treated by distinct rules. This structure can be naïvely walked and saved to disk or converted into an archive (e.g. tar, zip, ...) for download.

## 3.3 Model editor

The starting point of the MDD practices is model editing, which can be useful also in absence of code generation, e.g., for documentation. We decided not to tie ALMOsT.js to a specific editor, but to make the architecture easily integrated with any editor of choice, provided that it respects the technical requirements of being executable in the browser and produce and consume JSON objects. We propose a reference Model Editor implementation called ALMOsT-Joint which is built on top of the JointJS diagram library (see Chapter 5 for a real world application developed using it). It manages models in the format compliant to the specifications illustrated in Section 3.2.3; inside the editor, model objects are enriched with metadata in order to fit JointJS internal representation requirements. Developers can customize the appearance and behavior of both types of elements managed by the Model Editor, **entities** and **links**. The introduction of a new element in the editor requires different workflows for entities and links.

For entity-type elements, customization typically requires to:

1. Define the graphical representation of the element (in the popular SVG format).
2. Define the element type and register it in the list of available types.
3. (optional) Define the editable attributes.
4. (optional) Define its list of valid super- and sub-element types.
5. (optional) Define the default outgoing link type and constraints.

For link-type elements:

1. Define the graphical representation of the link.
2. Define and register the link type.
3. (optional) Define the editable attributes.
4. (optional) Define the valid source and target element types.

Following the philosophy of ALMOsT.js, developers are free to maintain a 1 to 1 relation between graphical elements in the editor and elements in data the model, or specify a custom mapping rule.

## 3.4 Extending ALMOsT.js

To support requirement #6 (Custom generation), ALMOsT.js can be extended in a variety of ways. The **Model Extender** component, shown in Figure 3.1, can be optionally used to enrich the Data Model, described in Section 3.2.3, with custom utility functions that facilitate rule implementation. Template engines like EJS[3] or Pug[4] (formerly known as Jade) can be easily incorporated in the action part of model to text rules. This facility supports: requirement #2 (No new language), because developers can reuse their favorite web platforms; requirement #4 (Parallel development) because graph walking algorithms are separated from actual code generation; and requirement #6 (Custom generation), because rules can use different output standards adopting the most appropriate frameworks, libraries, web languages or platforms.

## 3.5 ALMOsT-Trace: Tracing ALMOsT.js

Online platforms like [41] can help reduce the complexity of models and model transformations management, execution and validation.

Traceability of artifacts during the evolution of a software project is a key aspect for regression testing and monitoring. In model transformations the ability to trace models between transformation steps [38,91] enables advanced analysis and rapid error detection.

Following the *keep it simple* and plug-in based nature of ALMOsT.js the introduction of ALMOsT-Trace in a project is completely optional and as for other plug-ins does not require any changes to the existing system.

ALMOsT-Trace is based on two components:

1. A drop-in replacement for the **createTransformer** creator function which enhances the created transformer with tracing abilities, as shown below. Tracing

```
1 // Traced transformer
2 var transformer = createTracedTransformer(rules, 'm2m');

3 // Events
4 transformer.on('begin', function (model) { ... });
5 transformer.on('skipped', function (rule, input) { ... });
6 transformer.on('executed', function (rule, input, output) { ... });
7 transformer.on('end', function (result) { ... });
```

**Listing 15:** *Traded Transformer as a Drop-In replacement*

   aware rules can add custom tracing data to the final report, via a **trace** function that is injected as last parameter in each one of them.

[3] http://www.embeddedjs.com/
[4] http://www.pugjs.org

2. A **dashboard** which listens for events on a transformer and allows the developer to analyze the traces recorded during every execution. It can be easily integrated inside any web based tool already using ALMOsT.js and enabled on demand.

```
// Create the dashboard
var dashboard = createDashboard(transformer);
// Show the dashboard
dashboard.show();
```

**Listing 16:** *Dashboard initialization*

At the end of the execution of each rule the tuple $< rule, input, output >$ is stored. This lookup table allows the dashboard to lookup all the tuples related to each *rule*, *input* element and relation or *output* object components.

While forward tuples lookups (from *rules* and *input* to the *output*) are naïve to implement, backward lookups (from *output* to *rules* and *input*) are not. If during the aggregation phase two or more objects are merged together it is not possible to identify them from their root reference, it is instead required to search for outputs in the tuples collection which contain a particular subpart that survived unaltered till the end of the aggregation.

While reference-based types (*Object*, *Array*, ...) are easy to identify, it is enough to compare the reference, primitive types (*number*, *string*, ...) cannot be identified by means of a simple comparison, it is not possible to distinguish a number from another which contains the same value.

Enhanced rules map all the primitive values to their *Object* wrapper counterpart (*Number*, *String*, ...) the wrappers will generate the same result during aggregation, but will be uniquely identifiable. The enhanced transformer is responsible to restore each one of the wrapper objects to their original form before returning the result to the caller, preserving though the drop-in nature of ALMOsT-Trace.

By analyzing the stored tuples in the lookup table the dashboard of ALMOsT-Trace generates four different reports:

1. **Rule execution statistics.** All the defined rules are listed and for each one of them it is possible to know: (a) the number of evaluations (b) the number of executions (c) the enabling elements or relations (d) the output related to the execution against each enabling element.

2. **Input Model statistics.** All the elements and relations defined in the input model are listed and for each of them it is possible to know: (a) the number of enabled rules (b) the output related to the execution of the enabled rules against the element or relation itself.

3. **Output Model statistics.**

   For *model to model* and *model to text* transformations all the elements, relations, folders or files defined in the output are listed and for each one of them it is possible to know: (a) the rules which generated them (b) the input against which the rules have been executed to generate the element, relation, folder or file itself.

4. **Output Object statistics.**

   For *custom* transformations (transformations with an output format defined by the developer) the final output JSON Object is presented in an expandable tree-view. For each *Object* attribute or *Array* item it is possible to know if they were generated directly from a rule or if they are the result of an aggregation step. For each attribute or item that can be identified in one of the rules outputs the corresponding $< rule, input >$ pair is shown.

## 3.6 Discussion and limitations

The experience of implementing IFMLEdit.org demonstrated the suitability of ALMOsT.js in an agile MDD context. The case study addressed an already existing MDD language (IFML), but we deem that the lessons learned apply also to the case in which the development team designs its own domain specific abstractions and generation rules.

The development of the IFMLEdit.org project followed a SCRUM iterative cycle. The element of IFML were added one at a time to the online tool, developing in parallel the Model Editor customization, the model to model rules, and the model to text rules.

The first sprint featured a zero application consisting of just an empty ViewContainer and was realized in 24 hours. Each subsequent sprint added further language elements, in the following order: multiple ViewContainers, NavigationFlows between ViewContainers, Landmark ViewContainers, Nested AND ViewContainers, ViewComponents, NavigationFlows between ViewComponents, DataFlows, XOR ViewContainers, arbitrarily nested ViewContainers and Actions. Each sprint took on average less than 2 days to complete. The most complex sprint (adding arbitrarily nested AND and XOR ViewContainers) took 5 days, mostly allocated to the model to model transformation rules, which were necessary to sort out the innumerable semantic issues that arise with such an advanced design patterns.

However, the objective of simplicity, prominent for achieving agility of use, also implied trade-offs with respect to the expressive power and capabilities of the rule language and engine. In particular, the following limitations can be observed:

1. No advanced mechanisms are offered to exercise finer control over the execution of rules, such as execution priorities.

2. No higher order transformations are supported, because rules cannot match other rules but only model objects.

3. Rule modularization is not supported, because rules of the same transformation cannot be divided into packages, e.g., clustering rules according to different purposes. If a complex transformation has to be split into separated stages or focuses, it must be formulated as a set of sub-transformations.

4. A basic knowledge of the JavaScript programming language is required to use the framework. The design principles of ALMOsT.js can be used to develop equivalent frameworks in other well known languages, making the approach suitable to developers with different backgrounds.

5. No guarantee is given about the side effect free nature of the transformations rules. This property of the transformation rules should be enforced via code linters and/or code-reviews.

6. Termination and Confluence of a generic transformation cannot be guaranteed by the framework. Termination is guaranteed if each rule guarantees it's completion in a finite amount of time regardless of it's inputs. Confluence is guaranteed if the reducer configuration is invariant to the order of execution.

CHAPTER 4

# Predictable Web and Mobile Semantics

Model Driven Engineering is the branch of Software Engineering that emphasizes the use of models in the development process [50]. Models are simplified descriptions of the application that capture its essential aspects at a certain level of abstraction, e.g., independently of the platform for which the application will be designed and of the technologies with which it will be implemented.

In the state of the practice, models are frequently used in the early stages of development, for reasoning about design alternatives and for documenting high level design decisions.

However, models can be used also to automate, at least in part, the production of the implementation. When the semantics of the language used to express the model is known, models can be automatically or semi-automatically transformed into implementation artifacts, by means of model transformations that progressively incorporate the details originally omitted from the input models. A well-known and popular example is the automatic mapping of Entity-Relationship conceptual diagrams into logical database models implemented with the SQL Data Definition Language [69]. Generative model-driven software development is especially useful when the application contains repetitive patterns, is expected to require frequent updates, or must be produced for multiple software platforms [40].

The key to enabling a generative approach to model-driven development is the availability of a rigorous semantics for the modeling language(s) employed. Whereas the use of models for software documentation may tolerate imprecision, their use as input to transformations requires that the syntactic validity and intended meaning of each model be understandable unambiguously by a software component. A common way to express the semantics of high level software modeling languages is to map them to a general-purpose, rigorous mathematical notation, whose semantics is determined

exactly [59].

Among the multiple perspectives under which an application can be modeled, the user's interaction aspect emerges as a particularly interesting one for two reasons. On the one side, the user interface is often the first concern tackled by developers, as it stems naturally from the requirement analysis performed with the stakeholders; thus, being able to model it and quickly prototype the model in the target platform(s) may help reduce the insurgence of early stage design errors. On the other side, the technologies for the implementation of the user interface have been proliferating with the advent of rich web and mobile applications, and it is often the case that the same application front-end must be developed and maintained for multiple, and technically quite diverse, platforms and implementation languages.

In this chapter, we concentrate on the model driven development of the front-end part of web and mobile applications. As a language for modeling this perspective, we consider OMG's Interaction Flow Modeling Language (IFML) [114], an UML-based language, part of the Model Driven Architecture (MDA), expressly conceived for representing the structure and behavior of the interface of interactive applications. The specifications of IFML define the execution semantics informally, by means of examples and natural language illustration. Model driven development environments that implement code generation from IFML models, e.g., WebRatio [32], embody the language semantics in the code generator, which makes it hard to check that a correct implementation is produced for every valid input model.

The focus of this chapter is the formal specification of the semantics of IFML, with the aim of exposing the behavior of applications specified with IFML models, enabling the derivation of a reference implementation of IFML models usable to verify model driven code generation tools. As a collateral objective, we also introduce an on-line Model Driven Development environment for web and mobile applications that puts to work the defined semantics.

The contributions of this chapter are as follows:

- We formalize the semantics of IFML by means of a rule-based mapping to a variant of Petri Nets (Place Chart Nets, PCN [97]). The semantic rules capture the organization of the front-end in terms of connected view containers comprising interdependent interface components, the triggering and management of the events caused by the user's interaction or by the system, including the firing of business actions and their effect on the status of the interface. The semantic rules can map a variety of real-world front-end configurations: the organization of the interface typical of classic web applications, where the user's interaction causes the update of the entire page; the front-end structure of desktop and native mobile applications, where the GUI is hosted in a top-level container, structured into nested sub-containers that are updated and displayed selectively; and a mix of these two configurations, which is typically found in Rich Internet Applications.

- We exploit the semantic mapping to expose underspecified aspects of the IFML language and to highlight tool-specific interpretations of IFML by commercial code generators.

In Chapter 5 we will showcase the power of the defined semantic rules by implementing two model transformations:

- A model-to-model transformation from IFML to PCN, which allows one to specify the model of the application interface in IFML, produce the equivalent PCN and inspect its behavior via event-driven simulation. While the current simulation is limited, advanced model verification and testing can be obtained via reachability and firebility analyses of the PCN model.
- A model-to-text transformation, which generates the executable implementation code from the IFML model. The transformation takes in input also the platform specification (thin or fat client) and delivers the code of a thin-client web application and/or of a cross-platform fat-client mobile application. The model-to-text transformation permits the customization of the generated prototype also by non-programmers, enabling early requirements validation. Note that the model-to-text transformation starts from the IFML model and thus does not directly exploit the model-to-model transformation from IFML to PCN. However, the code generation rules are an indirect result of the semantic mapping, because they reflect the precise understanding of the behavior of IFML constructs gained with the specification of the model-to-model rules.

Both transformations are implemented in an open source on-line tool[1] offered to the software engineering community, usable for model driven engineering education and for rapid prototyping of web and mobile applications based on IFML.

## 4.1 Formal semantics of modeling languages

The semantics of many DSLs and MDA-based models is described only informally; however, the automated analysis of models and the effective development of tools, such as model interpreters, debuggers, and testing environments, require the formal specification of the language semantics [54].

A common way of formalizing a modeling language is through *translation semantics* [59]: the abstract syntax of the modeling language is mapped into an existing formal language, with well-defined semantics, such as, e.g., Abstract State Machines (ASM). This technique is adopted in [66]: the authors extend AMMA, a framework for defining DSLs, with the specification of behavioral semantics expressed by means of ASMs. In [76] the authors introduce a transformational semantic framework based on ASM for the expression of the executable semantics of metamodel-based languages; they also discuss alternative methods, such as *weaving*, in which the execution semantics is embedded within the abstract syntax of the modeling language without resorting to an external formalism. In this line, [109] proposes to extend meta-modeling languages with the specification of behavioral semantics by means of the UML2 standard action language, foundational UML (fUML) [28]. Another example is presented in [101], which integrates formal methods with high-level notations (in particular UML), to enhance model quality, detect possible defects, and compute properties directly from models. Finally, the work in [104] addresses the problem of specifying transformations: transformation languages themselves can be modeled as DSLs. For each pair of domains, the metamodel of the rules can be (quasi-)automatically generated to create a language tailored to the desired transformation. The authors showcase the proposed approach on the mapping of Finite State Automata to Petri Nets.

[1] http://ifmledit.org/

In this work, we have adopted a rule-based translational approach to the specification of the IFML semantics.

Among the possible target formal models, the event-driven dynamics of the application front-end makes event and transition systems, such as UML State Machines and Petri Nets, a natural choice. UML State Machines [30] offer several advantages: they have a precise semantics [22], their integration with a a subset of UML modeling constructs can be expressed formally with fUML, and the Alf language [3] can be used for the description of specific behavior. Furthermore, several tools offer UML state machine execution. On the negative side, the synchronous nature of State Machines limits their ability to express asynchronous behavior typical of mobile and rich-client interfaces, and better fits the semantics of pure HTML-HTTP web interfaces [60]. The capability of Petri Nets [113], and of their extensions [90], to express asynchronous occurrences makes them particularly adequate for the representation of behavior patterns found in rich-client and mobile front-ends, such as the independent refresh of different parts of the view, the treatment of push notifications from the server or from the system [68], and the workflows of client-server communication [79]. Among the generalizations of Petri Nets, Colored Petri Nets (CPNs) are the most widely used formalism; they incorporate data, hierarchy, and time [90] and are supported by CPN Tools [10], which can be used for the design of complex processes and their simulation. However, CPNs support hierarchies at transition level, and not at place level. This allows a CPN diagram to be structured in reusable modules but, even for small applications, models tend to quickly become very complex [79]. Place Chart Nets (PCN) [97] are an alternative Petri Net extension, which incorporates some ideas from StateCharts: they add *hierarchy on places* and *preemptive transitions*: a transition empties not only its input places but also all descendant places of its input places. This feature enables the modeling of both asynchronism and of exception handling by means of preemption, a capability extremely useful in the modeling of mobile and rich-client applications, because it reduces the exponential explosion of the number of transitions needed to express the management of user's interactions that affect multiple parts of the interface. Furthermore, PCNs, like PNs, can be simulated, which allow designers to better understand their application front-end and to predict the behavior of the system produced from the models.

## 4.2 Semantic Mapping of IFML

The semantics of IFML can be defined by mapping IFML models to a language with known execution behavior. Given the event-driven/asynchronous nature of the computation represented by IFML diagrams, Petri Nets offer the most natural formalism for expressing their execution semantics. In this work, we exploit a generalization of Petri Nets, Place Chart Nets [97], which offers a modularization construct that reduces the combinatorial explosion of states and transitions induced by the mapping of IFML diagrams.

The proposed mapping focuses on structure and interaction. Section 4.2.5 addresses the basic structure of an application, which serves as a building block for all the other mapping rules. Section 4.2.6 concentrates on simple *ViewContainer*s and their interaction with *Event*s and *NavigationFlow*s. Section 4.2.9 addresses *ViewComponent*s, their life-cycle, their interaction with *Event*s, and the difference between the two types

of *InteractionFlow*s (*NavigationFlow*s and *DataFlow*s). Section 4.2.12 introduces the semantic rules for mapping *Action*s, their triggering, and their life-cycle. Finally, Section 4.4 extends the mapping to the case of arbitrarily nested *ViewContainer*s, enabling the representation of complex, real-life application structures.

The illustrated mapping rules do not consider the type and values of the data to be displayed, but express the logic that governs the display of an element in the interface. The mapping rules for *ViewComponent* apply uniformly to all classes of *ViewComponent*s, denoted by the different stereotypes, such as *Details*, *List* and *Form*, and are independent of the presence and values of such IFML elements as *DataBinding*s, *ConditionalExpression*s and *Field*s. Similarly, the mapping rules for *InteractionFlow*s are independent from the nature, and even from the presence, of *ParameterBinding*s. These assumptions make the semantic mapping data-independent, so that the behavior of the application is predictable just by observing the structure of the model.

### 4.2.1 Place Chart Nets

Place Chart Nets (PCN) [97] generalize Petri Nets (PN) to allow the representation of hierarchy and of preemption, while retaining the asynchronous nature of Petri Nets and their formal properties.

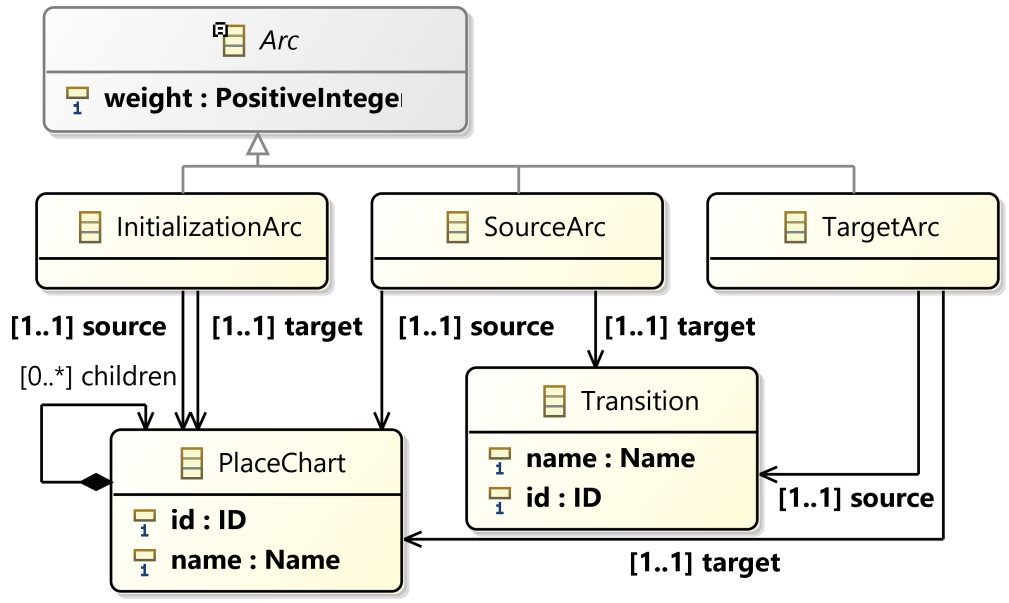



**Figure 4.1:** *PCN Metamodel.*

Figure 4.1 shows the essential elements of the PCN metamodel. The base construct of a PCN is a *place chart (PC)*, which represents a hierarchy of places. A PC with no parent nor children is called a *place*, because it is equivalent to a PN place. A place with a parent, but no children is called a *bottom place chart*. A place with children, but no parent is called a *top place chart*. The number of tokens in a place chart with no children is defined as in Petri Nets, while the number of tokens in a place chart with children is defined as the maximum of the number of tokens in its children. As in Petri Nets, transitions remove tokens from a group of place charts and add tokens to others. A transition is enabled when all the source place charts have at least the number

of tokens required. Removing a token from a place chart with no children decrements the number of tokens by one, while removing a token from a place chart with children empties all of them. To avoid non-determinism, it is possible to insert tokens only in place charts with no children. However, as a convenient way to reduce the number of arcs in a model, *default arcs* are introduced. A default arc connects a parent place chart to one or more of its descendants. If default arcs are defined from a place chart $X$ to (a subset of) its descendants $Y_1 \dots Y_n$, then an arc targeting $X$ is equivalent to a set of arcs targeting $Y_1 \dots Y_n$.

A formal definition of PCNs can be found in [97]. We illustrate the benefits of their usage by means of an example, which models the execution of a parallel search over replicated databases. The search process is started on all the independent copies and, when one database returns a result, query execution at all the other copies halts.

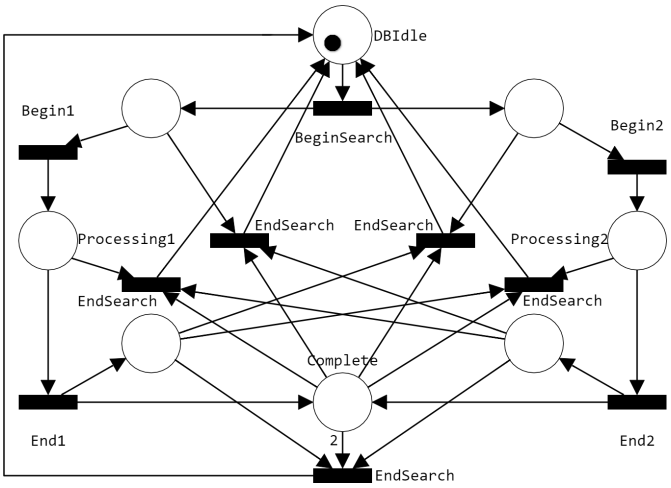



**Figure 4.2:** *Model complexity reduction - Petri Net*

Figure 4.2 shows a Petri Net that describes such process, considering two database copies. The *BeginSearch* transition removes a token from *DBIdle* and initializes the parallel search, which proceeds asynchronously. The completion of a search is described by transitions *End1* or *End2*, which add a token to the *Complete* place. The *EndSearch* transition restores the idle state of the system. To this end, it must remove all the tokens from the net; however, due to the asynchronous nature of the system, it is not known where the tokens are going to be exactly. Thus, the PN enumerates all the possible configurations and contains a transition that handles each possibility. As shown in Figure 4.2, even this simple scenario requires five different *EndSearch* transitions. The proliferation of transitions increases if the model must describe the execution of the query more accurately, by refining the *Processing* places with a more complex net, or if the system has more than two replicas.

Figure 4.3 describes the same process with a PCN. Each parallel process is enclosed in a top place chart. The *BeginSearch* transition initializes the two parallel top place charts *SearchDB1* and *SearchDB2*: a default arc connects them to the first place of the nested net. The *Complete* place of the PN is replaced by a *Complete* top place chart with a child (*Count*) targeted by a default arc. The *EndSearch* transition removes a token from *SearchDB1*, *SearchDB2* and *Complete* and restores the idle state of the

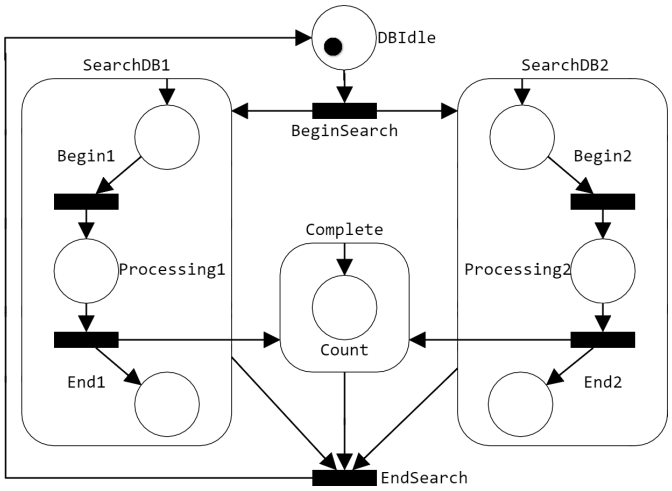



**Figure 4.3:** *Model complexity reduction - Place Chart Net*

system regardless of which database answered first.

As the (small) example shows, PCNs describe complex models with less elements.

### 4.2.2 Notations

In the following subsections we show how to map the essential IFML constructs to PCNs. We will adopt the following notations and naming rules:

- IFML elements: they are denoted with italicized labels or, if generic, with capital letters; the following naming conventions for generic elements are used: ViewContainers are represented as $V$, XOR ViewContainers as $X$, ancestor ViewContainers as $A$, child ViewContainers as $C$, source ViewContainers as $S$, target ViewContainers as $T$, NavigationFlows as $F$, and events as $e$. For the sake of brevity, sometimes we leave the NavigationFlow associated with an event implicit, as in the phrase "the target of Event $e$".
- PCN elements: they are denoted with sans-serifletters, according to a naming convention that links the PCN element to the IFML it maps. Place charts mapping generic ViewContainers are represented as $\mathsf{V}$, XOR ViewContainers as $\mathsf{X}$, ancestor ViewContainers as $\mathsf{A}$, child ViewContainers as $\mathsf{C}$, source ViewContainers as $\mathsf{S}$, target ViewContainers as $\mathsf{T}$, transitions mapping events as $\mathsf{e}$.

### 4.2.3 Mapping Boolean variables

A recurrent problem in the mapping of IFML to PCN is representing Boolean variables. A Boolean variable $B$ can be described by two places (or bottom place charts), labeled $\mathsf{B}$ and $\overline{\mathsf{B}}$; the presence of a token in one of them represents the positive or negative state of the variable. We can use transitions to move the token from $\mathsf{B}$ to $\overline{\mathsf{B}}$ or vice versa, changing the value of the variable.

### 4.2.4 Running Examples

Throughout this section we will use a very simple mail client application as a running example. Its interface shows the list of received mails, the details of a selected

email, and the list of its attachments. The underlying data model includes a Mail entity, associated with the Attachment entity representing mail attachments; emails are also associated with a User entity. The aim of this first example is to showcase the semantic mapping of different models for computing and displaying content and of several possible user interaction patterns. A second example, a music application, is also introduced to explain the semantic mapping of IFML actions. This application simply allows the user to play and stop a song.

### 4.2.5 Mapping the Application

The simplest IFML model is the empty diagram: it can be interpreted as an application that, as soon as opened, terminates, neither displaying any interface nor performing any action.

Figure 4.4 shows the PCN corresponding to an empty IFML model.

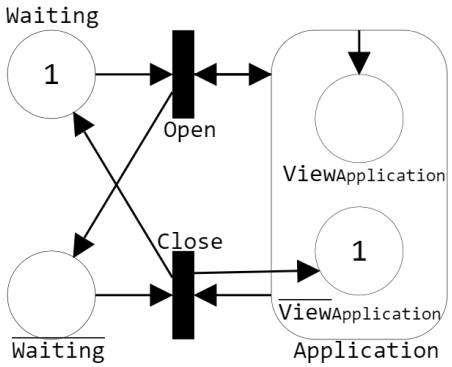



**Figure 4.4:** *PCN of an empty application*

It contains two places (Waiting and $\overline{\text{Waiting}}$), a top place chart (Application) with two children ($\text{View}_{\text{Application}}$ and $\overline{\text{View}}_{\text{Application}}$). The $\text{View}_{\text{Application}}$ bottom place chart is initialized by default from the parent. The PCN also contains two transitions called open and close, which move a token between the Waiting Boolean variable and Application. The initial marking comprises one token in Waiting and one token in $\overline{\text{View}}_{\text{Application}}$, describing that a user can open the application, which is initially not in view. The open transition removes and adds a token from Application and moves a token from Waiting to $\overline{\text{Waiting}}$, disabling itself and enabling close. The close transition moves a token from Application to $\overline{\text{View}}_{\text{Application}}$ and moves a token from $\overline{\text{Waiting}}$ to Waiting, disabling itself and enabling open, thus resetting the application state to the default initial marking.

The example of Figure 4.4 defines the first mapping rule:

Rule **1)** APPLICATION

The mapping of an IFML model produces a PCN that contains a Waiting and a $\overline{\text{Waiting}}$ place, an Application top place chart with two children $\text{View}_{\text{Application}}$ and $\overline{\text{View}}_{\text{Application}}$. $\text{View}_{\text{Application}}$ is initialized by default from the parent. The PCN also contains an open transition, which moves a token from Waiting and Application to $\overline{\text{Waiting}}$ and Application, and a close transition, which moves a token from $\overline{\text{Waiting}}$ and Application to Waiting and $\overline{\text{View}}_{\text{Application}}$.

### 4.2.6 Mapping the structure: View Containers

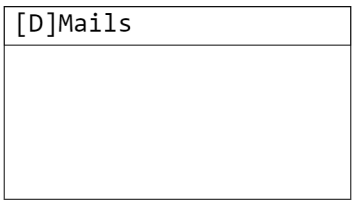



**Figure 4.5:** *Single, empty default ViewContainer Model - IFML model*

Figure 4.5 shows the second simplest scenario: an IFML model with one top-level default ViewContainer ($Mails$, in the example). This model corresponds to an application that shows a blank screen at start-up.

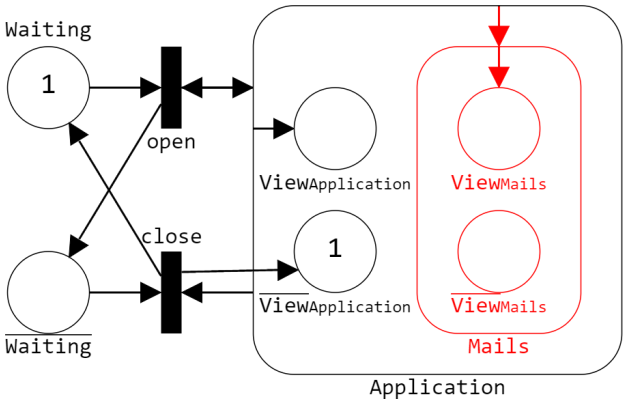



**Figure 4.6:** *Single, empty default ViewContainer Model - Place Chart Net*

Figure 4.6 shows its mapping[2]: the PCN of Figure 4.4 is extended by introducing a place chart called Mails, child of Application, which is initialized by the parent. The Mails place chart has two children bottom place charts ($\mathsf{View}_{\mathsf{Mails}}$, initialized by the parent, and $\overline{\mathsf{View}}_{\mathsf{Mails}}$). The Boolean variable $\mathsf{View}_{\mathsf{Mails}}$ represents whether the ViewContainer is, or is not, in view. The firing of the open transition now adds also a token to $\mathsf{View}_{\mathsf{Mails}}$, meaning that the ViewContainer is displayed.

This example of Figure 4.6 defines two mapping rules:

Rule **2)** TOPVIEWCONTAINER

A top-level ViewContainer $V$ maps to a place chart child of Application named V, with two children bottom place charts ($\mathsf{View}_{\mathsf{V}}$ and $\overline{\mathsf{View}}_{\mathsf{V}}$). $\mathsf{View}_{\mathsf{V}}$ is initialized by default from the parent.

[2]In the following, the red color identifies the portions of a PCN affected (inserted or updated) by the mapping rule that the example illustrates.

Rule **3)** DEFAULT TOPVIEWCONTAINER

The presence of the *default* property of a top-level ViewContainer $V$ maps to an initialization arc from Application to V, denoting that the child ViewContainer becomes visible by default when the parent application opens.

### 4.2.7 Mapping top-level navigation: Navigation Flows between View Containers

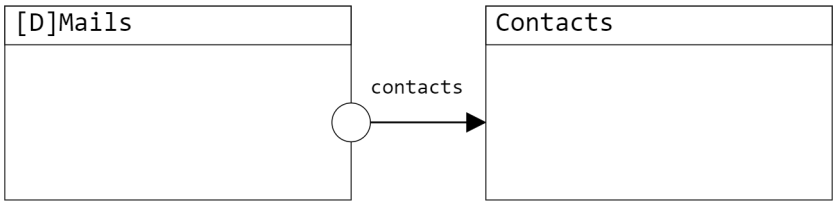



**Figure 4.7:** *Navigation between top-level ViewContainers - IFML model*

Figure 4.7 shows the elementary navigation step between top-level ViewContainers. The IFML model comprises two top-level ViewContainers ($Mails$ and $Contacts$), an Event called $contacts$ associated with the $Mails$ ViewContainer, and a NavigationFlow from such event, targeting the $Contacts$ ViewContainer.

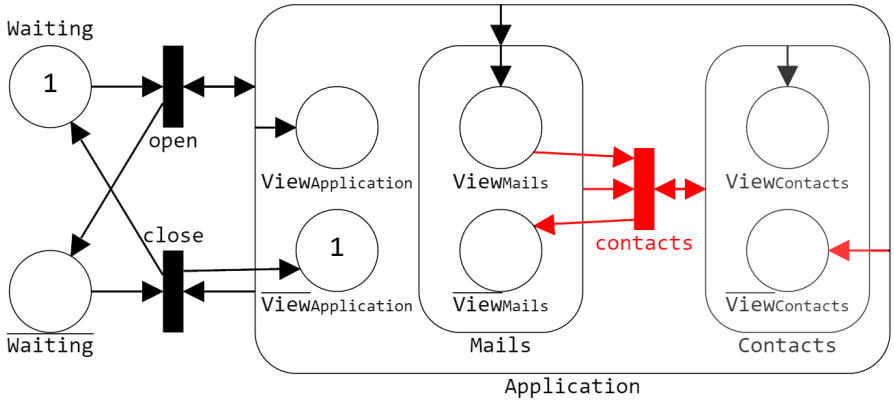



**Figure 4.8:** *Navigation between top-level ViewContainers - Place Chart Net*

Figure 4.8 shows the PCN that maps the IFML model of Figure 4.7. According to rule 2, the Application top place chart contains two place charts Mails and Contacts, each one with two children bottom place charts ($\mathsf{View}_{\mathsf{Mails}}$, $\overline{\mathsf{View}}_{\mathsf{Mails}}$, $\mathsf{View}_{\mathsf{Contacts}}$, $\overline{\mathsf{View}}_{\mathsf{Contacts}}$). Based on rule 3, Mails is initialized by Application, because $Mails$ is the *default* top-level ViewContainer; this causes the initialization by default of the $\mathsf{View}_{\mathsf{Mails}}$ place chart. Conversely, Contacts is not initialized as the default, but an initialization arc from Application targets the $\overline{\mathsf{View}}_{\mathsf{Contacts}}$ place chart, denoting that the $Contacts$ ViewContainer is not in view initially. The navigation between the two ViewContainers is represented by a transition named contacts, which denotes the change of the display status of the two ViewContainers. The transition moves a token from Mails, $\mathsf{View}_{\mathsf{Mails}}$ and Contacts to Contacts and $\overline{\mathsf{View}}_{\mathsf{Mails}}$. Note that the apparently redundant input place charts of the contacts transition (Mails, $\mathsf{View}_{\mathsf{Mails}}$) are necessary to ensure that: 1) to-

kens are consumed also for the possible children place charts of Mails; 2) the transition is enabled only when the $Mails$ ViewContainer is actually in view.

This example of Figure 4.7 defines the following mapping rules:

Rule **4)** NON DEFAULT TOPVIEWCONTAINER

The absence of the default property in a top-level ViewContainer $V$ maps to an initialization arc from Application to $\overline{\mathsf{View}_\mathsf{V}}$.

Rule **5)** TOP LEVEL NAVIGATIONFLOW

A NavigationFlow from an Event $e$ associated with a top-level ViewContainer $S$ and targeting a top-level ViewContainer $T$ maps to a transition e that moves a token from S, $\mathsf{View}_\mathsf{S}$ and T to T and $\overline{\mathsf{View}_\mathsf{S}}$.

### 4.2.8 Mapping landmark navigation: Landmark ViewContainers

ViewContainers with the *landmark* visibility property represent the target of implicit navigation flows from the other ViewContainers nested within their parent container.

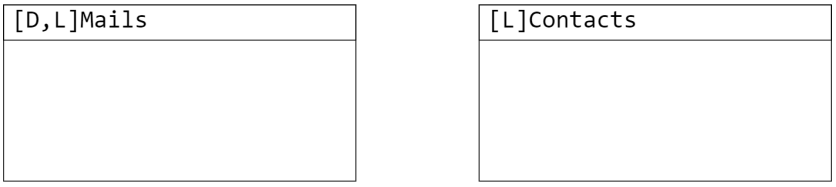



**Figure 4.9:** *Navigation using Landmarks - IFML model*

Figure 4.9 shows an example of landmark ViewContainers ($Contacts$ and $Mails$), meaning that from one ViewContainer it is possible to navigate to the other. The landmark visibility property is mapped into a set of transitions, according to the following rule:

Rule **6)** LANDMARK VIEWCONTAINER

The presence of the *landmark* property of a top-level ViewContainer V1 maps to a transition $\mathsf{landmark}_{\mathsf{V1}}$ that moves a token from Application and $\mathsf{View}_{\mathsf{Application}}$ to $\mathsf{View}_{\mathsf{Application}}$ and V1. For each top-level ViewContainer V2 different from V1, the $\mathsf{landmark}_{\mathsf{V1}}$ transition adds a token to $\overline{\mathsf{View}_{\mathsf{V2}}}$.

Note that the rule removes and adds a token to the parent of the landmark ViewContainers (i.e., Application for top-level ViewContainers); this is because the navigation can be originated by any of the sibling ViewContainers and thus the token must be consumed at the parent level, which will cause the removal of a token also from the place chart of *all* the ViewContainers within it, including the one that was previously in view.

The model in Figure 4.9 maps into the PCN of Figure 4.10: the two transitions called $\mathsf{landmark}_{\mathsf{Mails}}$ and $\mathsf{landmark}_{\mathsf{Contacts}}$ check that the application is currently visible,

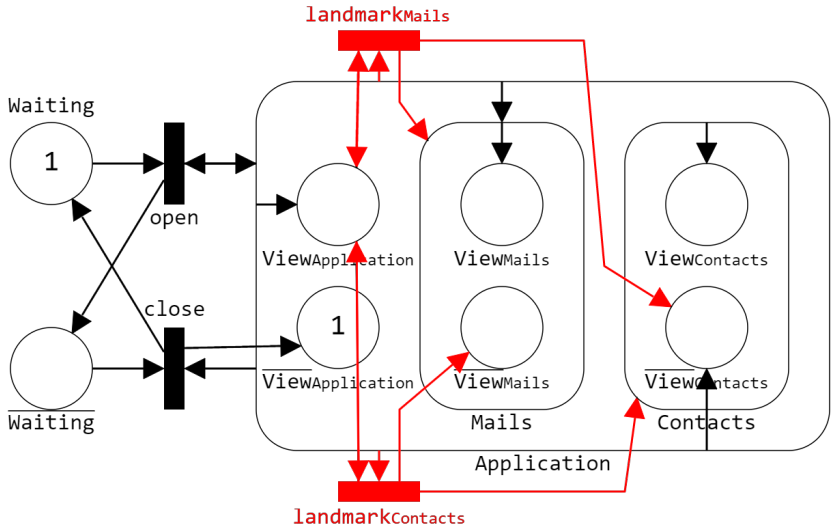



**Figure 4.10:** *Navigation using Landmarks - Place Chart Net*

by adding and removing a token from $\mathsf{View}_{\mathsf{Application}}$, and remove a token from it. Transition $\mathsf{landmark}_{\mathsf{Mails}}$ adds a token to Mails and $\overline{\mathsf{View}}_{\mathsf{Contacts}}$; symmetrically, transition $\mathsf{landmark}_{\mathsf{Contacts}}$ adds a token to Contacts and $\overline{\mathsf{View}}_{\mathsf{Mails}}$. The effect of each transition is to initialize its target top-level ViewContainer and set the status of all the other ones to not in view.

### 4.2.9 Mapping the content: View Components

The examples discussed so far specify only empty interfaces without content. This section discusses the mapping of models that include ViewComponents, whose content is computed and rendered in the interface, possibly based on the value of some input parameters.

The behavior of a ViewComponent can be regarded as the result of the interplay between two parts:

- the *model*, representing the status of the interaction with the data source providing content to the ViewComponent;
- the *view model*, representing the display of content in the interface.

For example, in a pure HTML web application, the *model* could be the data bean holding objects extracted from a database and the *view model* could be the HTML rendition of such objects. In an Android app, the *model* could be a Java object and the *view model* the GUI widget bound to it.

The *model* part of a ViewComponent can be modeled by the following states:

- *Clear*: the ViewComponent lacks some input in order to be computed (e.g., the values of the parameters appearing in its conditional expression), thus it cannot show any content and remains empty.
- *Ready*: the ViewComponent is ready to be computed; this happens in two cases: the ViewComponent does not require any input parameter or it has already received the needed parameter values.

- *Computed*: the content of the ViewComponent has been computed and the ViewComponent is ready to be displayed.

The change between these states are modeled by the following transitions:

- *Propagate*: it changes the state from *Clear* to *Ready* and represents the propagation of input parameters to the ViewComponent;
- *Compute*: it changes the state from *Ready* to *Computed* and represents the computation of the component's content using the possibly received inputs; the content becomes available to the view model.

To represent the states of the model of a ViewComponent, two Boolean variables $In$ and $Out$, i.e., four PCN bottom place charts, are used: In, $\overline{\text{In}}$, Out and $\overline{\text{Out}}$ (for an example see Figure 4.12). The $In$ variable denotes the availability of all the input data necessary for the computation of the ViewComponent; the $Out$ variable describes the completion of the computation, which makes the content available to the view model. The states of the model part are therefore represented by the following configurations:

- *Clear*: a token in $\overline{\text{In}}$ and a token in $\overline{\text{Out}}$;
- *Ready*: a token in In and a token in $\overline{\text{Out}}$;
- *Computed*: a token in $\overline{\text{In}}$ and a token in Out.

Notice that, the fourth configuration (a token in both In and Out) is not meaningful, since it represents the case where the content has been produced, but the inputs necessary for such computation have not been consumed.

Figure 4.12 shows the *Compute* transition, which is a transition internal to the ViewComponent. The *Propagate* transition will be exemplified later (in Figure 4.16): it is commanded by an event external to the ViewComponent, like, for example, a user interaction enabling parameter passing.

The *view model* part of a ViewComponent can be represented by two states:

- *Invalid*, denotes that the ViewComponent is not displayed.
- *Visible*, denotes that the ViewComponent has received data from the model and therefore can be displayed.

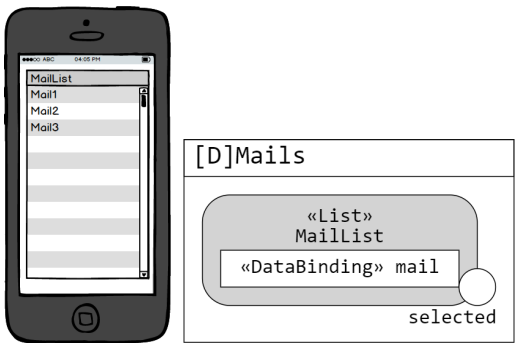
A Boolean Variable models the two states, represented by two bottom place charts View and $\overline{\text{View}}$, respectively, as exemplified in Figure 4.12.

The transition from the *Invalid* to the *Visible* state is modeled by the *Render* transition, shown in Figure 4.12: it represents the copy of the content from the model to the view model and the consequent rendering of the ViewComponent.

As an example, Figures 4.11a and 4.11b present a simple IFML model with a top-level ViewContainer comprising only one ViewComponent: an application that displays a list of mails.

Figure 4.12 shows the PCN mapping of the IFML model[3]. A place chart, child of Mails, named MailList represents the $MailList$ ViewComponent, initialized from the

[3]From now on, for brevity we omit the Application top place chart and the $Waiting$ Boolean variable from the PCN mapping of the IFML models.




**(a)** *Mockup* **(b)** *IFML model*

**Figure 4.11:** *Single ViewComponent - Mockup & IFML model*

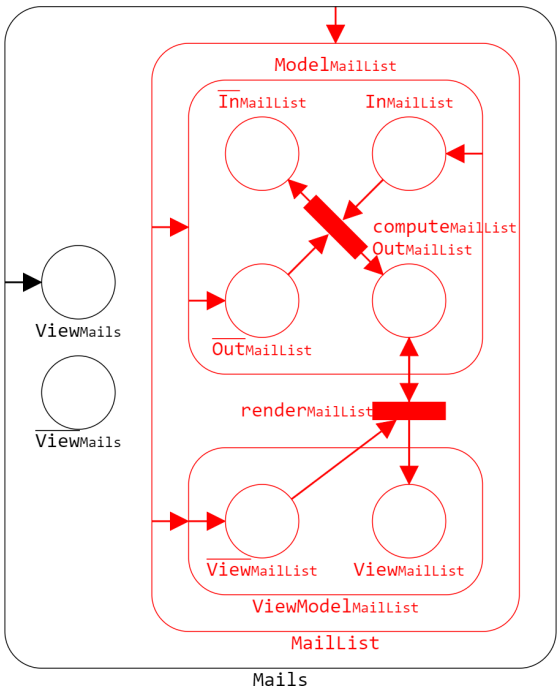



**Figure 4.12:** *Single ViewComponent model - Place Chart Net*

parent. It contains two child place charts $\mathsf{Model}_{\mathsf{MailList}}$ and $\mathsf{ViewModel}_{\mathsf{MailList}}$ representing the model and the view model part of the ViewComponent, respectively.

$\mathsf{Model}_{\mathsf{MailList}}$ contains the four bottom place charts defined earlier ($\mathsf{In}_{\mathsf{MailList}}$, $\overline{\mathsf{In}}_{\mathsf{MailList}}$, $\mathsf{Out}_{\mathsf{MailList}}$ and $\overline{\mathsf{Out}}_{\mathsf{MailList}}$). Transition $\mathsf{compute}_{\mathsf{MailList}}$ represents the computation of the content: it removes a token from $\mathsf{In}_{\mathsf{MailList}}$ and adds a token to $\overline{\mathsf{In}}_{\mathsf{MailList}}$, denoting the consumption of the inputs; it also removes a token from $\overline{\mathsf{Out}}_{\mathsf{MailList}}$ and adds a token to $\mathsf{Out}_{\mathsf{MailList}}$, denoting the availability of the model content.

$\mathsf{ViewModel}_{\mathsf{MailList}}$ contains the two bottom place charts $\mathsf{View}_{\mathsf{MailList}}$ and $\overline{\mathsf{View}}_{\mathsf{MailList}}$. MailList also contains a transition describing the rendering of the view model, named $\mathsf{render}_{\mathsf{MailList}}$, which removes a token from $\overline{\mathsf{View}}_{\mathsf{MailList}}$ and $\mathsf{Out}_{\mathsf{MailList}}$ and adds one token to $\mathsf{View}_{\mathsf{MailList}}$ and $\mathsf{Out}_{\mathsf{MailList}}$.

The $MailList$ ViewComponent is initialized by default from the $Mails$ ViewContainer as follows: the model is set to the ready state and the view model to the invalid state.

The example of Figure 4.11 introduces the rules:

Rule **7)** BASE VIEWCOMPONENT

A ViewComponent $C$ child of a parent ViewContainer $P$ maps to:

1) a place chart C, child of P, initialized by default from P.
2) two children place charts of C: $\mathsf{Model}_{\mathsf{C}}$ and $\mathsf{ViewModel}_{\mathsf{C}}$, initialized by default from C.
3) four bottom place charts $\mathsf{In}_{\mathsf{C}}$, $\overline{\mathsf{In}}_{\mathsf{C}}$, $\mathsf{Out}_{\mathsf{C}}$ and $\overline{\mathsf{Out}}_{\mathsf{C}}$, children of $\mathsf{Model}_{\mathsf{C}}$.
4) a transition $\mathsf{compute}_{\mathsf{C}}$, which removes a token from $\mathsf{In}_{\mathsf{C}}$ and $\overline{\mathsf{Out}}_{\mathsf{C}}$ and inserts a token into $\overline{\mathsf{In}}_{\mathsf{C}}$ and $\mathsf{Out}_{\mathsf{C}}$.
5) two bottom place charts $\mathsf{View}_{\mathsf{C}}$ and $\overline{\mathsf{View}}_{\mathsf{C}}$, children of $\mathsf{ViewModel}_{\mathsf{C}}$. $\overline{\mathsf{View}}_{\mathsf{C}}$ is initialized by default from $\mathsf{ViewModel}_{\mathsf{C}}$.
6) a transition $\mathsf{render}_{\mathsf{C}}$, which removes a token from $\overline{\mathsf{View}}_{\mathsf{C}}$ and $\mathsf{Out}_{\mathsf{C}}$ and inserts a token into $\mathsf{View}_{\mathsf{C}}$ and $\mathsf{Out}_{\mathsf{C}}$.

Rule **8)** VIEWCOMPONENT INITIALIZATION - NO INCOMING DATA FLOWS

A ViewComponent $V$ without incoming data flows is initialized in the *ready* state, i.e., with defaults arcs from $\mathsf{Model}_{\mathsf{V}}$ to $\mathsf{In}_{\mathsf{V}}$ and $\overline{\mathsf{Out}}_{\mathsf{V}}$.

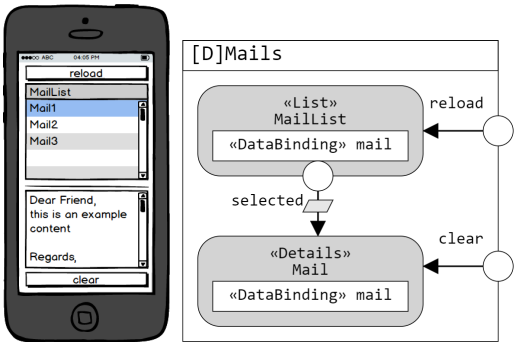
Rule 8 specifies the initialization for all the ViewComponents, except those that have one or more input *DataFlows*, that will be treated later by rule 10.

Note that the example of Figure 4.12 addresses the case of a ViewComponent without input. The next example elaborates on the mapping of ViewComponents with input parameters.

### 4.2.10 Events and Navigation Flows

Figure 4.13 extends the previous example with different events and navigation flows:

- the mail list is interactive, enabling message selection from the list and display in another interface component. When the application starts, only the list is displayed; after the user selects one item from the list, its details are shown.
- The $reload$ event allows the user to refresh the list of mails;
- The $clear$ event enables flushing the Mail message visualization.




**(a)** *Mockup* **(b)** *IFML model*

**Figure 4.13:** *Navigation between ViewComponents: navigation flow - Mockup & IFML model*

Figure 4.13b illustrates the corresponding IFML model: a NavigationFlow from $MailList$ to $Mail$ has a parameter binding that associates the currently selected mail in the list with an input parameter in the filter condition of the $Mail$ ViewComponent[4]. Each time the user chooses a mail message, the $selected$ event is fired, the ID of the chosen message is made available as the new value of the input parameter of the $Mail$ ViewComponent, and the new model content is computed based on such value; then, the view model part of the $Mail$ ViewComponent is updated and displays the newly selected message.

The PCN of Figure 4.14 illustrates the mapping of the IFML diagram of Figure 4.13b. Now the place chart of the $Mails$ ViewContainer comprises two children place charts, corresponding to the $MailList$ and $Mail$ ViewComponents, inserted based on rule 7 and 8 (in particular, they are initialized to the ready state, as per rule 8). Note that although the place chart associated with the $Mail$ ViewComponent is initialized to the ready state, its input parameter initially has a null value because the user has not selected a message yet; thus, the computation and rendering transitions fire but the view model displays an empty ("null") content. The navigation flow from the $selected$ event maps into a transition ($selected$) that affects the $Mail$ ViewComponent as follows: the model place chart resets to the ready state and the view model place chart resets to the invalid state; the computation and rendering transitions fire, causing the refresh of the model content, based on the new value of the input parameter (the message selected by the user) and the display of the view model.

The $reload$ and $clear$ events do not have any parameter binding, but their behavior is very similar: they reset the model place chart to the ready state and the view model place chart to the invalid state: as an effect, the previous content is invalidated and the model and view model are recomputed. Note that the NavigationFlow of the $clear$

[4]For brevity, in the IFML diagram we omit the SelectorCondition of ViewComponents and the parameters in the ParameterBinding of the InteractionFlows, described in the text.

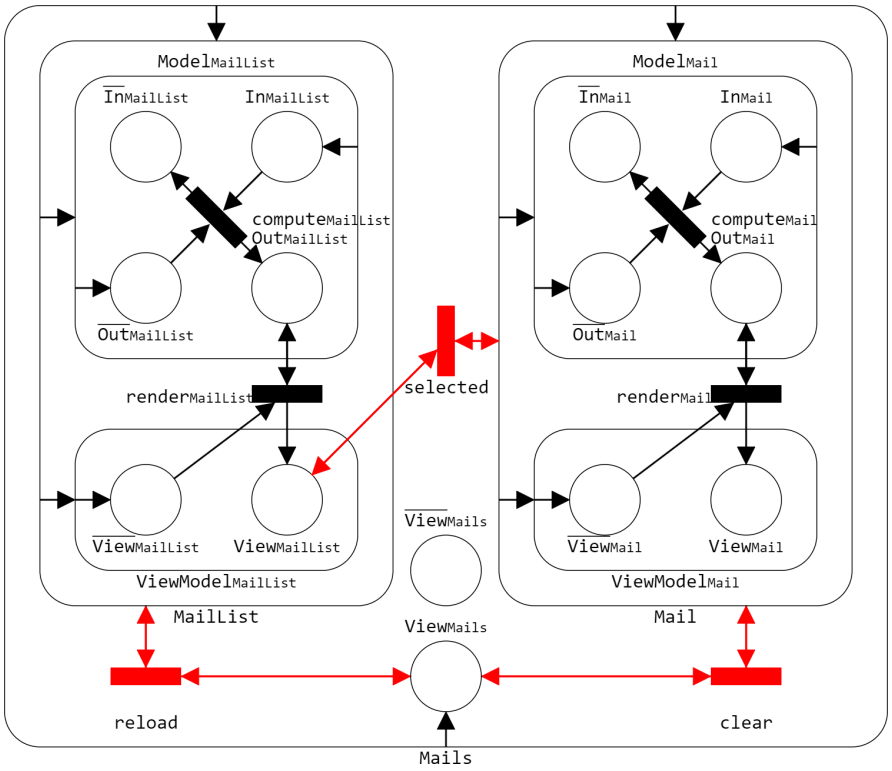



**Figure 4.14:** *Navigation between ViewComponents: navigation flow - Place Chart Net*

event does not provide the required input parameter to the $Mail$ ViewComponent and thus the "null" content is displayed, with the effect of clearing the message interface area.

The example of Figure 4.13 introduces the rule:

Rule **9)** SIMPLE NAVIGATIONFLOW

A NavigationFlow from an Event $e$, associated with a source ViewElement $S$ and pointing to a target ViewElement $T$, such that $S$ and $T$ are children of the same ViewContainer or $S$ is the parent of $T$, maps to a transition e that:

1) removes a token from the place chart of T and adds a token to it.

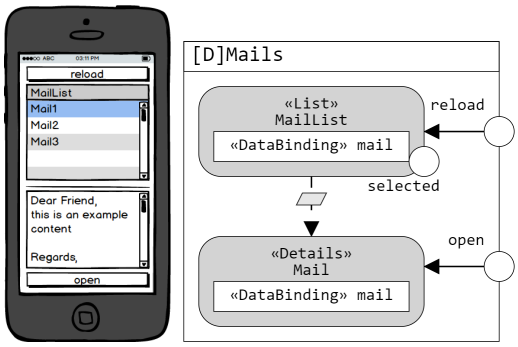
2) removes a token from the place chart $\text{View}_S$ and adds a token to it.

Note that rule 9 applies to ViewElements, i.e., to both ViewContainers and ViewComponents. A further example of NavigationFlow with a ViewContainer as target element appears in Figure 4.25.

### 4.2.11 Events and Data Flows

Figure 4.15 modifies the example of Figure 4.13, showing an alternative design pattern. In Figure 4.13 the selection of a mail message causes the immediate (i.e., synchronous) display of the mail content.

Conversely, in the IFML diagram of Figure 4.15b, Mail selection occurs in two




**(a)** *Mockup* **(b)** *IFML model*

**Figure 4.15:** *Navigation betweenViewComponents: data flow - Mockup & IFML model*

steps: first the user can (repeatedly) choose the mail message he wants to access, as represented by the $select$ event, which is local to the $MailList$ ViewComponent and has the sole effect of changing its output parameter; then he can trigger the $open$ event, which fetches the present value of the output parameter, associated with the DataFlow, and displays the currently selected message. Upon such event, the $Mail$ ViewComponent shows the content of the message identified by the input parameter. In other terms, the $open$ event triggers the *propagation* of the parameter(s) supplied to the target ViewComponent by its incoming DataFlow(s). In this case, the model of the $Mail$ ViewComponent is initially in the *clear* state, waiting for input data from all the ViewComponents on which it depends (in this example only from $MailList$, but in the general case it may receive inputs from several components); then, if all such ViewComponents are in the visible state, the propagate transition can fire and changes the $Mail$ status from clear to ready, enabling the computation and rendering transitions.

Instead the $reload$ event behaves like in the previous case.

The mapping of the model of Figure 4.15 introduces the rule:

Rule **10)** VIEWCOMPONENT INITIALIZATION - INCOMING DATAFLOWS

A ViewComponent $V$, target of a non empty set $\mathcal{F}_V$ of DataFlows, maps into:

1) initialization arcs from the parent that set the *clear* state of its model place charts (one arc adds a token to $\overline{\mathsf{In}_\mathsf{V}}$ and one arc adds a token to $\overline{\mathsf{Out}_\mathsf{V}}$).

2) a transition $\mathsf{propagate}_\mathsf{V}$ that removes a token from $\overline{\mathsf{Out}_\mathsf{V}}$ and $\overline{\mathsf{In}_\mathsf{V}}$ and adds a token to $\overline{\mathsf{Out}_\mathsf{V}}$ and $\mathsf{In}_\mathsf{V}$; for each DataFlow $F_i$ in $\mathcal{F}_V$, $\mathsf{propagate}_\mathsf{V}$ also removes and adds a token into $\mathsf{View}_{\mathsf{S_i}}$, where $S_i$ is the source ViewComponent of $F_i$.

### 4.2.12 Mapping Actions

The last basic IFML element to map is the Action, which models a piece of business logic invoked by the user's interaction or by a system-generated event. Figure 4.17 shows an example of usage.

A music application allows the user to play songs: the interface consists of two top ViewContainers $Playing$, containing a ViewComponent ($PlayerPlaying$) that shows the status of the application when a song is playing; and the default $Stopped$ View-

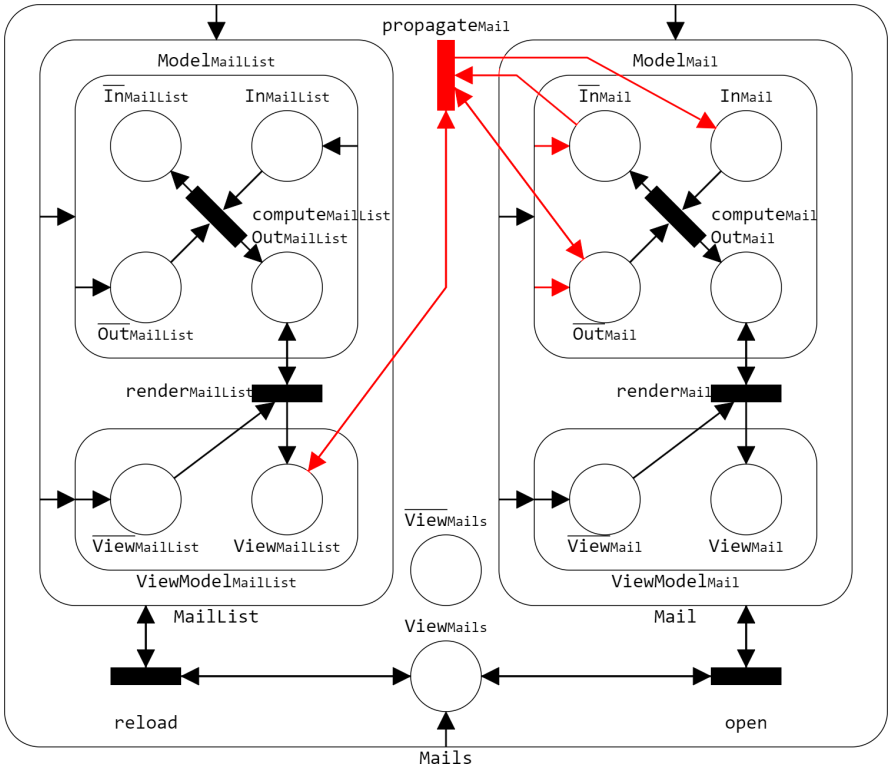



**Figure 4.16:** *Navigation betweenViewComponents: data flow - Place Chart Net*

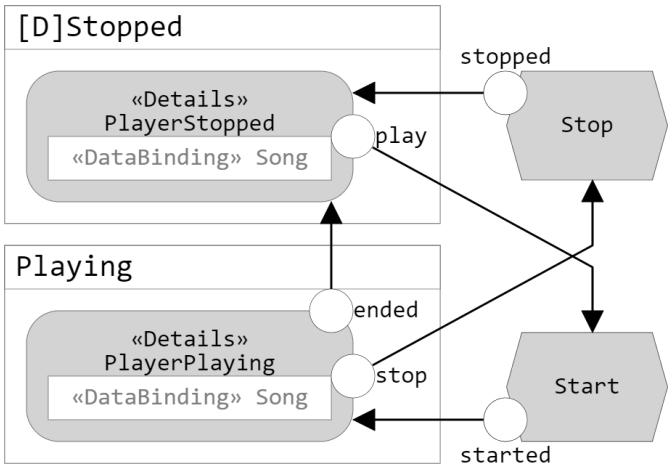



**Figure 4.17:** *Action interaction - IFML model*

Container comprising a ViewComponent ($PlayerStopped$) that shows the status of the application when no song is playing. Three events control the evolution. The $play$ event, associated with the $PlayerStopped$ ViewComponent, starts the song; it invokes the $Play$ Action, which, upon termination, raises the $started$ event and visualizes the $Playing$ ViewContainer. The $stop$ event, associated with the $PlayerPlaying$ ViewComponent, stops the song; it invokes the $Stop$ Action, which, upon termination, raises the $stopped$ event and visualizes the $Stopped$ ViewContainer. Finally, the $ended$ event, associated with the $PlayerPlaying$ ViewComponent, signals that the song has finished and visualizes the $Stopped$ ViewContainer.

The mapping of Actions in FigureFigure 4.18 reuses some of the rules for mapping NavigationFlows and adds new ones for handling the specificity of Action execution.

Definition 1 identifies the smaller ViewContainer that supports the triggering of an Action. This ViewContainer can be considered as the ancestor of the Action, in the definition of the mapping rules.

**Definition 1.** ACTION ORIGIN

Given an Action $K$, with an incoming NavigationFlow $F$ triggered by an event $e$ associated with a source ViewElement $S$, the Origin of $K$ ($O_K$) is $S$, if $S$ is a ViewContainer, or the ViewContainer enclosing $S$, if $S$ is a ViewComponent.

In Figure 4.17, the Origin of the $Play$ Action is the $Stopped$ ViewContainer and of the $Stop$ Action is the $Playing$ ViewContainer.

The execution of an Action is mapped by the following rule:

Rule **11)** ACTION EXECUTION

Given an Action $K$, with an incoming NavigationFlow $F$, let $O_K$be the Origin of $K$. $K$ maps to:

1) a place chart $\mathsf{K}$ child of $\mathsf{O_K}$.
2) Two place charts $\mathsf{Running_K}$ and $\overline{\mathsf{Running}}_\mathsf{K}$ children of $\mathsf{K}$. $\mathsf{Running_K}$ is initialized by default from the parent.
3) An initialization arc from $\mathsf{O_K}$to $\overline{\mathsf{Running}}_\mathsf{K}$.

Rule 11 maps the Action as a sub-state of its Origin, initialized to the not in execution sub-state.

The activation and termination of an Action is mapped by the following two rules:

Rule **12)** ACTION ACTIVATION

Given an Action $K$, a NavigationFlow $F$ targeting $K$ and starting from an Event $e$ related to the ViewElement $V$. $F$ maps to a transition $\mathsf{e}$ which removes and adds a token from $\mathsf{K}$ and $\mathsf{View_V}$.

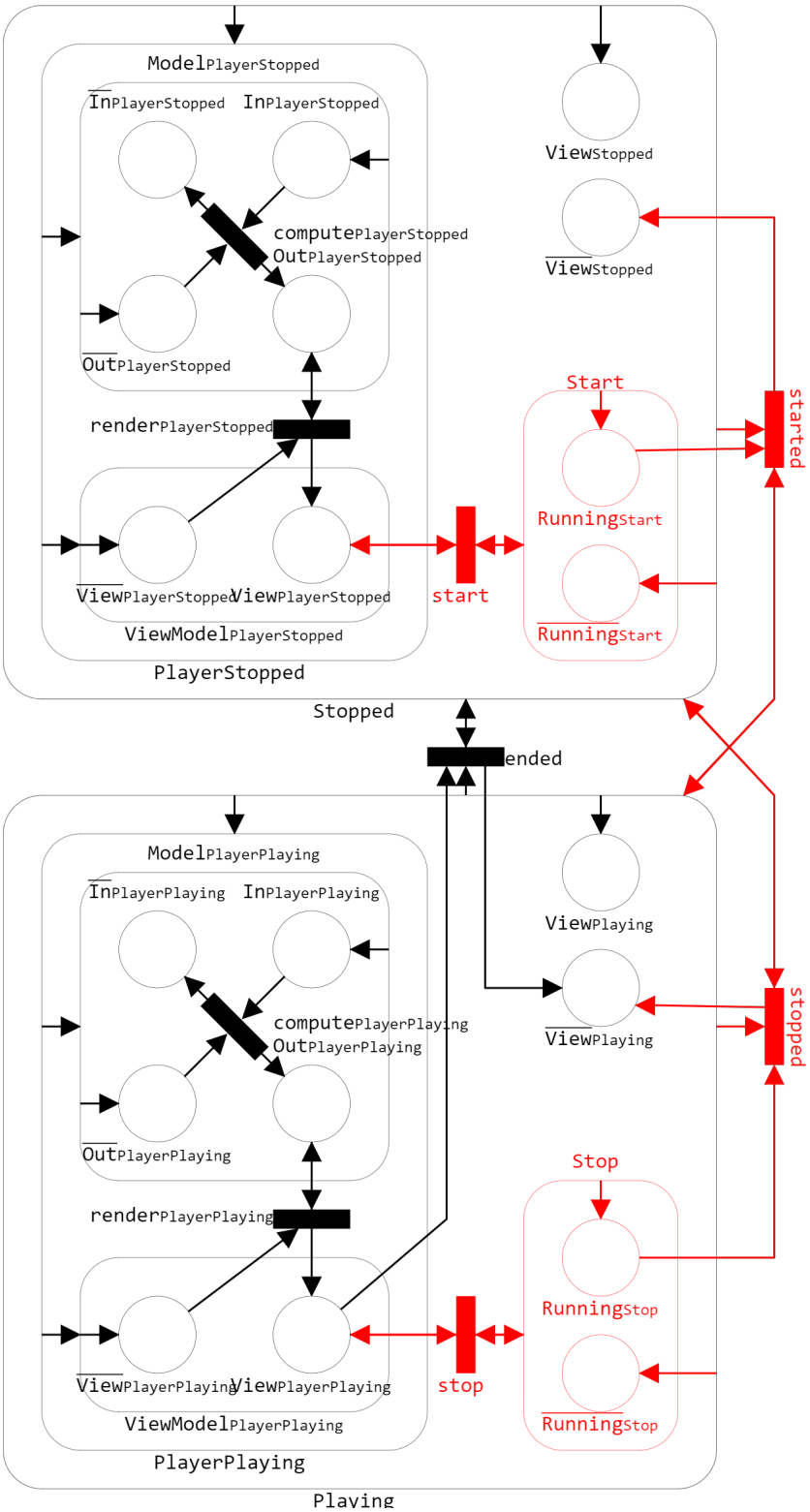



**Figure 4.18:** *Action interaction - Place Chart Net*

Rule **13)** ACTION TERMINATION

Given an Action $K$, a NavigationFlow $F$ targeting a top-level ViewContainer $V$ and starting from an Event $e$ related to $K$, let $O_K$ be the Origin of $K$. $F$ maps to:

1) a transition e that removes a token from $\mathsf{Running_K}$ and V and adds one token to $\overline{\mathsf{Running_K}}$ and V.

2) If $V$ is not $O_K$, e removes also a token from $\mathsf{O_K}$ and adds one token to $\overline{\mathsf{View_{O_K}}}$.

## 4.3 Mapping basic IFML elements: a complete example

To recap the mapping rules of the basic IFML elements, Figures 4.20, 4.21, and 4.22 illustrate an application for accessing a song library, with three alternative design patterns. The content model of the application is shown in Figure 4.19: entity *Song* is associated with a N:M relationship to entity *Artist* and to a N:1 relationship with entity *Genre*.


**Figure 4.19:** *Content model of the song library application*

All the IFML models contain three ViewComponents: $ArtistList$ shows all the artists in the library, $GenreList$ presents the available genres, and $SongList$ displays a list of songs, which can be filtered by genre and/or by author. The $SonglList$ ViewComponent has a SelectorCondition that filters songs based on the ID of a genre and artist, which are the parameters associated with the Navigation/DataFlow incoming to the ViewComponent.

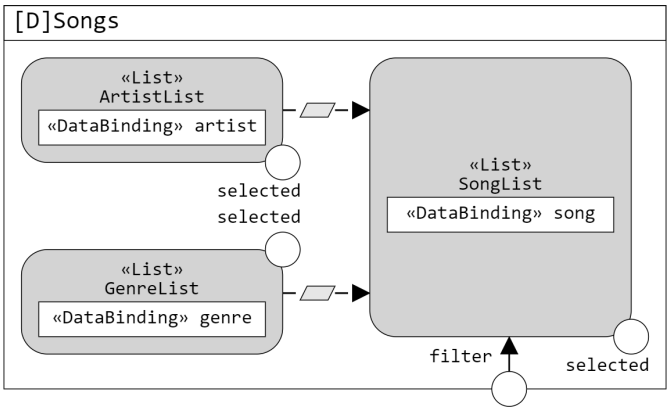



**Figure 4.20:** *Filter on request*

Figure 4.20 shows a first design pattern that lets the user set and change the selection criteria, and then apply them in one shot to the song list: the ID of the genre and artist, used for filtering songs, are parameters of *DataFlows* exiting the $ArtistList$ and $GenreList$ ViewComponents. The $selected$ events are applied only locally to the $ArtistList$ and $GenreList$, and their parameters will be propagated only when the $filter$ event is triggered by the user.

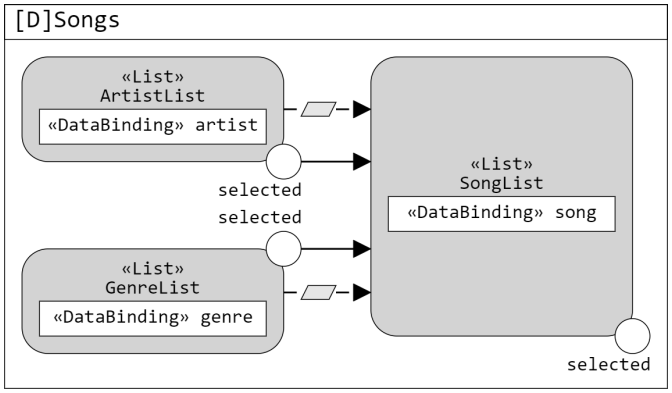



**Figure 4.21:** *Multiple filters*

Figure 4.21 presents an alternative design pattern, where the user can filter songs first by one criterion (e.g., by artist) and then restrict the resulting list by the other criterion (e.g., by genre), or vice versa.

The $ArtistList$ and $GenreList$ ViewComponents are connected to the $SongList$ ViewComponent with *both a NavigationFlow and a DataFlow*, and parameters are associated with the DataFlows. As a result, when the user changes the currently selected item either in the $ArtistList$ or in the $GenreList$ (triggering the $selected$ event), the content of the $SongList$ is updated immediately due to the presence of the NavigationFlow and all the current input parameters are used for the computation of the $SongList$.

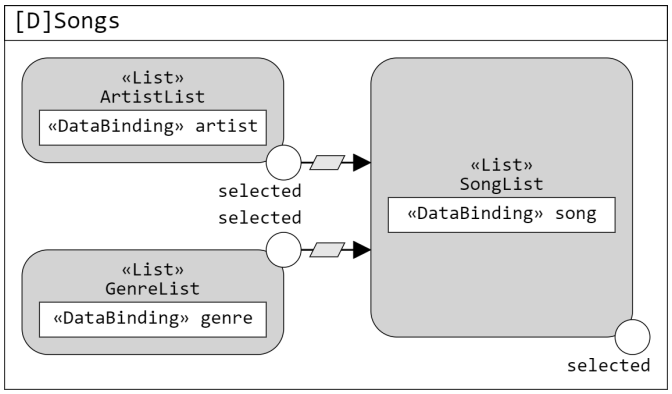



**Figure 4.22:** *Single filter*

Finally, the diagram in Figure 4.22 contains only *NavigationFlows*. In this case, when the user selects an artist or a genre, the $SongList$ is computed based on the value of the parameter associated with the NavigationFlow associated with the triggering event. Indeed, each time one of the $selected$ events triggers, the filters in $SongList$

are reset and just the selected item in the current list is propagated, allowing the user to filter the songs just by artist or by genre, not by both of them.

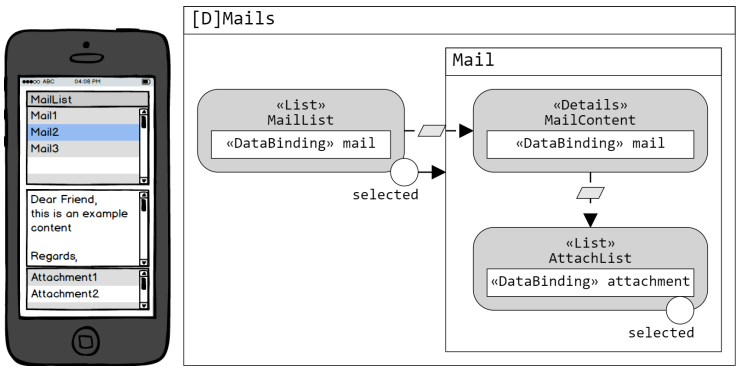
As shown by these examples, it is possible to express different behaviors by mixing events, navigation and data flows; the semantics of each design pattern is formally captured by the PCN mapping, which allows the designer to predict the behavior of each configuration exactly. We refer the reader to the IFMLedit.org online tool for the generation and simulation of the PCNs corresponding to the examples of this section.

These examples are particularly important because they show how their semantics is solely related to the elements in the model and their connections. No tool specific or application specific assumptions are made in the process. The mapping rules are not opinionated on the type of information managed by the application or their actual values. The triggering of an event leads always to the same result, i.e., the complete invalidation of the target. The proposed examples, and other variants, have been implemented in other IFML based tools during the validation process. Even these simple models exposed how current IFML dialects partially base their semantics on external configurations like the *Invalidation Targets* of the NavigationFlow (i.e., an external configuration specifying which ViewElements are affected by the activation of a NavigationFlow) or on the current state of the application (i.e., the same NavigationFlow produces different effects based on the interaction history of the application).

## 4.4 Mapping Complex Interfaces

### 4.4.1 Mapping Interface Composition: Nested ViewContainers

IFML allows the description of complex interface composition patterns, in which ViewContainers are nested and displayed selectively. To illustrate this capability, and its mapping from IFML to PCN,




**(a)** *Mockup* **(b)** *IFML model*

**Figure 4.23:** *Nested ViewContainers - Mock & IFML model*

Figure 4.23 extends the mail application example. Now the user can access both the content and the list of attachments of the currently selected email simultaneously. Figure 4.23b presents the IFML model, which exploits a $Mail$ ViewContainer nested within $Mails$. The $selected$ event has an outgoing NavigationFlow that originates from the $MailList$ ViewComponent and targets the nested $Mail$ ViewContainer, which expresses a navigation between ViewElements *of different types* directly enclosed in the

same ViewContainer; the parameter passing that enables the computation of the ViewComponents is expressed by parameter bindings associated with the DataFlows that connect $MailList$ to $MailContent$ and $MailContent$ to $AttachList$. The specific aspect of the example is that the occurrence of the $selected$ event invalidates and recomputes the whole content of the target ViewContainer and causes the display of all the ViewComponents comprised in it, i.e., of the mail content and of the list of its attachments.

Figure 4.24 illustrates the PCN mapping and exemplifies the treatment of nested ViewContainers.

The top-level $Mails$ ViewContainer is mapped to the Mails place chart, according to rule 2 and the $MailList$ ViewComponent is mapped to the MailList place chart (rule 7 and rule 8). The nested ViewContainer $Mail$ maps into the Mail place chart, embedded within Mails. The parent place chart (Mails) initializes by default its child place chart (Mail); specifically, the initialization arc targets the $\mathsf{View}_{\mathsf{Mail}}$ bottom place chart, denoting that the nested ViewContainer gets into view at the same time as its parent.

The content of the $Mail$ ViewContainer is mapped as per rules 7 and 10: two place charts named MailContent and AttachList map the corresponding view components, and the transitions $\mathsf{propagate}_{\mathsf{MailContent}}$ and $\mathsf{propagate}_{\mathsf{AttachList}}$ map the DataFlows incoming to the $MailContent$ and $AttachList$ ViewComponents, respectively. The $selected$ Event and the NavigationFlow are mapped according to rule 9: a transition (selected) removes and adds a token from/to $\mathsf{View}_{\mathsf{MailList}}$ and Mail: it resets Mail and the underlying PCN to its default configuration, clearing the model and invalidating the view model of both the $MailContent$ and $AttachList$ ViewComponents.

The example of Figure 4.23 introduces the rule:

Rule **14)** NESTED VIEWCONTAINERS

A ViewContainer $C$ child of another ViewContainer $P$ maps to:

1) A place chart C child of the place chart P.
2) Two bottom place charts $\mathsf{View}_{\mathsf{C}}$ and $\overline{\mathsf{View}}_{\mathsf{C}}$ within C.
3) An initialization arc from C to $\mathsf{View}_{\mathsf{C}}$.

Rule **15**) NON XOR PARENT

A ViewContainer $C$ child of a non XOR ViewContainer $P$ maps to an initialization arc from the place chart P to the place chart C.

Rule 15 expresses the fact that a child ViewContainer is displayed automatically when its parent gets into view. This may not be the case when the parent is a XOR ViewContainer, as illustrated in the next section.

### 4.4.2 Advanced composition: XOR View Containers

IFML provides the concept of XOR ViewContainer, which allows one to design interfaces that display different pieces of content in alternative. XOR ViewContainers

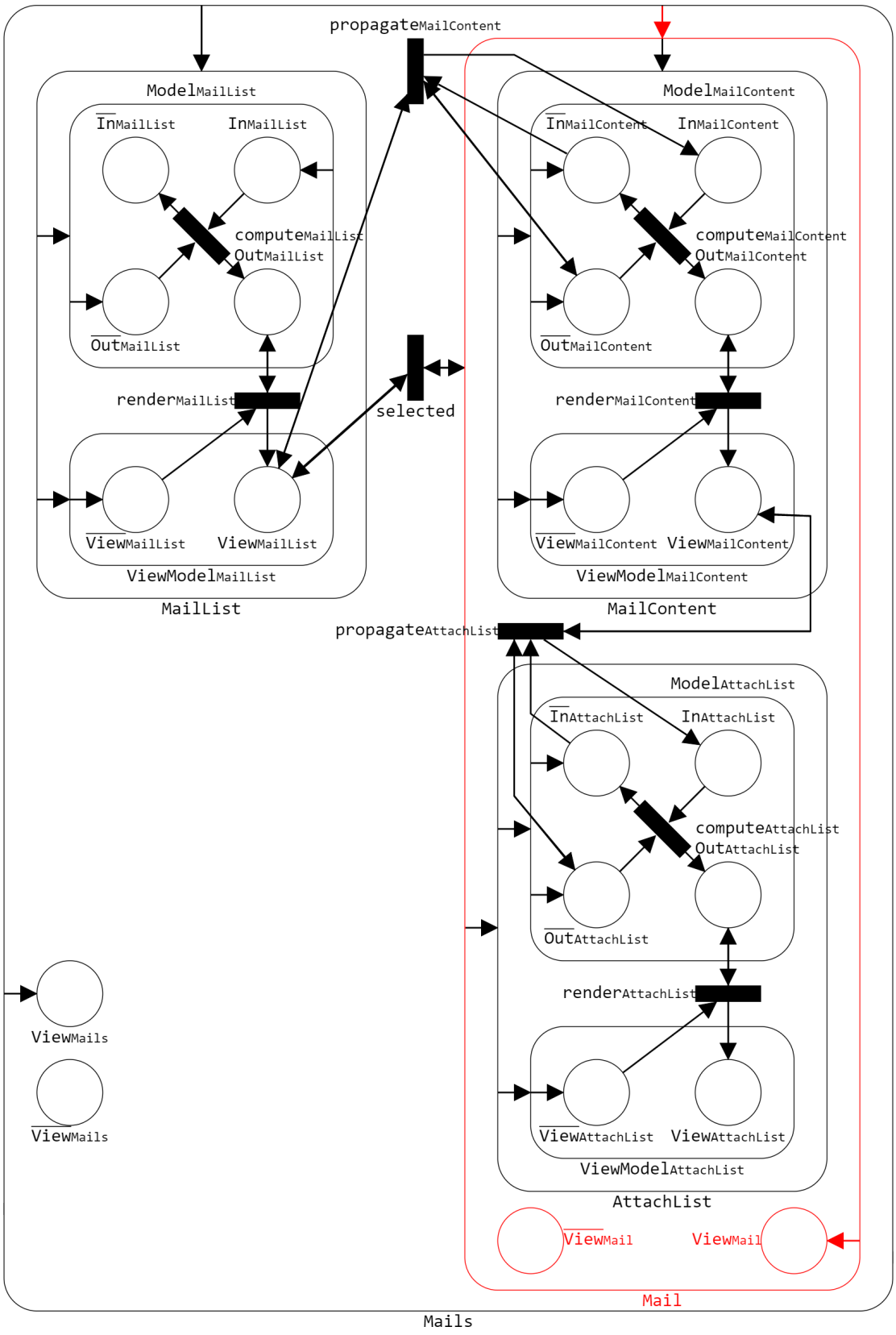



**Figure 4.24:** *Nested ViewContainers - Place Chart Net*

comprise other sub-ViewContainers, of which at most one at a time is in view. The combination of nested ViewContainers and XOR ViewContainers enables the representation of the complex composition of rich-client applications, in which the structure of the interface is a hierarchy of views and components, displayed based on the user events and according to complex visibility rules.

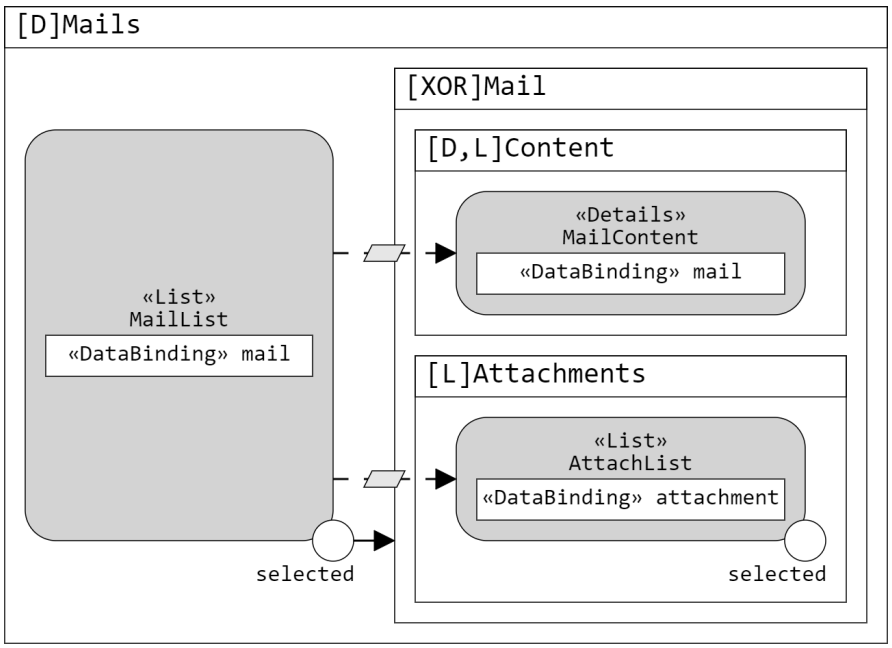



**Figure 4.25:** *Nested XOR ViewContainers - IFML model*

Figure 4.25 shows an example of XOR ViewContainers. The mail application of Figure 4.23 is changed so that the $MailContent$ and $AttachList$ ViewComponents are computed and displayed one at a time. S ch design may be convenient e.g., for small devices, to economize the screen space. To specify this behavior, the IFML model embeds the $MailContent$ and $AttachList$ ViewComponents within ViewContainers ($Content$ and $Attachments$) *nested in a XOR parent ViewContainer*.

Note that the *default* child of a XOR ViewContainer ($Content$ in the example) is the one shown when the parent is accessed; if no default child is specified, no child ViewContainer is displayed initially and the choice of what to access first is left to the user. A *landmark* XOR child ViewContainer is reachable with one navigation step from any of the XOR sibling ViewContainers: in the example, the interface lets the user toggle $Content$ and $Attachments$ into view.

Figure 4.26 shows the mapping of the IFML diagram of Figure 4.25. Two place charts, nested within the Mail place chart, map the $Content$ and $Attachments$ XOR children ViewContainers. According to Rule 14, they comprise the place charts $\mathsf{View}_{\mathsf{Content}}$, $\overline{\mathsf{View}}_{\mathsf{Content}}$, $\mathsf{View}_{\mathsf{Attachments}}$ and $\overline{\mathsf{View}}_{\mathsf{Attachments}}$. The $Content$ ViewContainer is the default child of $Mail$, therefore a default initialization arc is inserted from Mail to $\mathsf{View}_{\mathsf{Content}}$. Conversely, the $Attachments$ ViewContainer is not the default one and thus an initialization arc is inserted from the Mail place chart to the $\overline{\mathsf{View}}_{\mathsf{Attachments}}$ place chart.

Both $Content$ and $Attachments$ are landmark ViewContainers, therefore two tran-

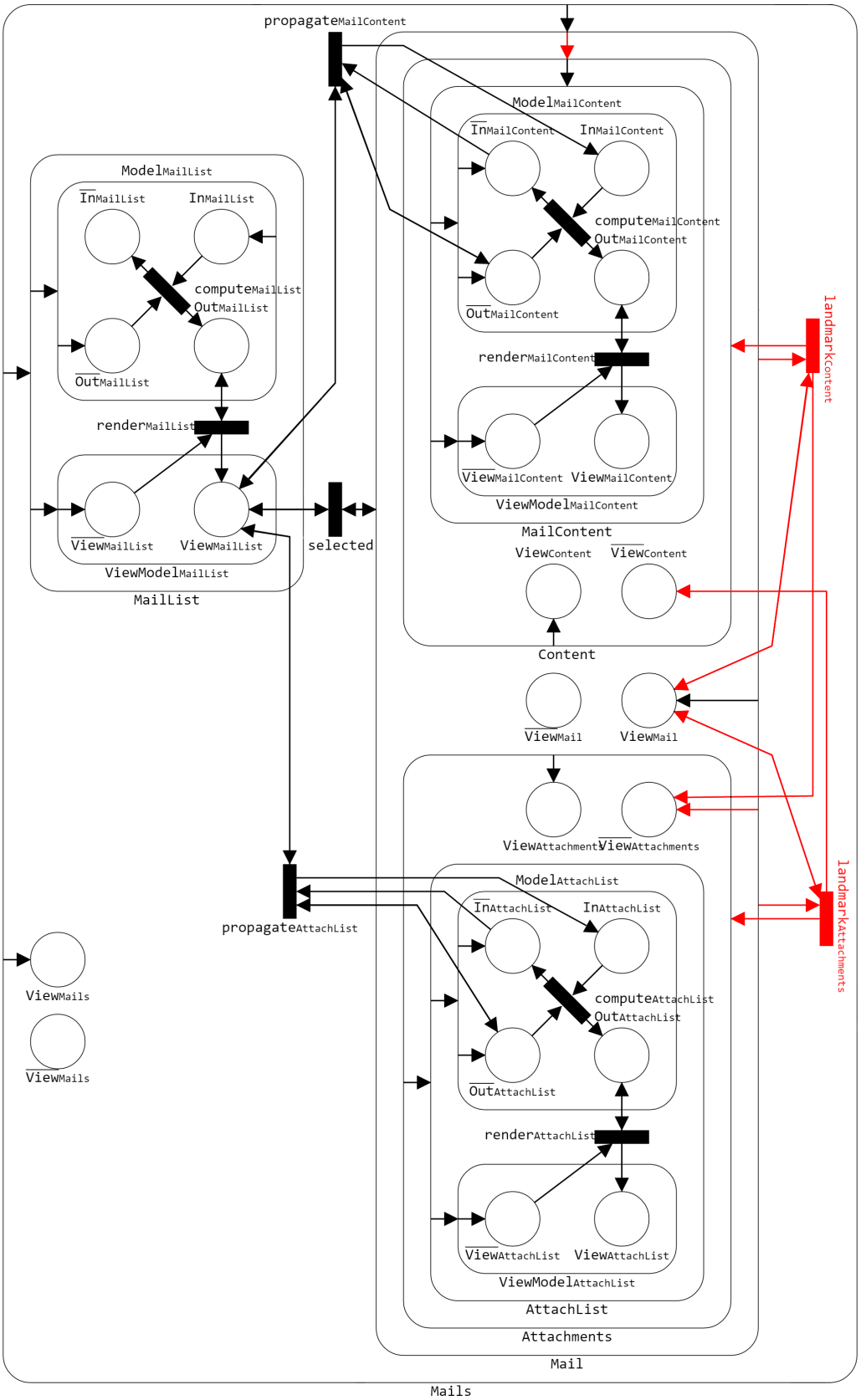



**Figure 4.26:** *Nested XOR ViewContainers - Place Chat Net*

sitions ($\mathsf{landmark}_{\mathsf{Content}}$ and $\mathsf{landmark}_{\mathsf{Attachments}}$) are inserted, which follow the same configuration as for the rule of top-level landmarks (Rule 6): they remove a token from Mail and $\mathsf{View}_{\mathsf{Mail}}$ and add a token to $\mathsf{View}_{\mathsf{Mail}}$. Furthermore, transition $\mathsf{landmark}_{\mathsf{Content}}$ adds a token to Content and $\overline{\mathsf{View}}_{\mathsf{Attachments}}$ and transition $\mathsf{landmark}_{\mathsf{Attachments}}$ adds a token to Attachments and $\overline{\mathsf{View}}_{\mathsf{Content}}$.

The example of Figure 4.26 introduces the rules:

Rule **16)** XOR DEFAULT CHILD

A default ViewContainer $C$ child of a XOR ViewContainer $P$ maps to an initialization arc from the place chart P to the place chart C.

In Figure 4.26 the rule inserts the arc from Mail to Content.

Rule **17)** XOR NON DEFAULT CHILD

A non default ViewContainer $C$ child of a XOR ViewContainer $P$ maps to an initialization arc from the place chart P to $\overline{\mathsf{View}}_{\mathsf{C}}$.

In Figure 4.26 the rule inserts the arc from Mail to $\overline{\mathsf{View}}_{\mathsf{Attachments}}$.

Rule **18)** XOR LANDMARK CHILD

The presence of the *landmark* property of a ViewContainer $C$ child of a XOR ViewContainer $X$ maps to a transition $\mathsf{landmark}_{\mathsf{C1}}$ that moves a token from X and $\mathsf{View}_{\mathsf{X}}$ to $\mathsf{View}_{\mathsf{X}}$ and C. For each ViewContainer $C_i$ child of $X$ different from $C$, the $\mathsf{landmark}_{\mathsf{C}}$ transition adds a token to $\overline{\mathsf{View}}_{\mathsf{C_i}}$.

In Figure 4.26 the rule inserts the transitions $\mathsf{landmark}_{\mathsf{Content}}$ and $\mathsf{landmark}_{\mathsf{Attachments}}$.

**Mapping navigation in deeply nested structures**

The examples discussed so far illustrate the mapping of navigation in quite simple interfaces, with at most one level of structural nesting. Many real world applications, though, have a more articulated composition; for example, the rich-client web version of a popular mail program organizes content within an interface comprising four levels of nested containers. As a further example, we illustrate the mapping of IFML models featuring arbitrarily nested interface structures. The rules we are going to introduce generalize the previous ones, from the case of flat or two-level composition structures to any mix of nested ViewContainers and XOR ViewContainers. For the sake of uniformity in the specification of the rules, we consider the application itself as a XOR ViewContainer, because it is the topmost (albeit implicit) element of the model and its children top-level ViewContainers are displayed alternatively. The aspect that is affected by the presence of deeply nested structures is navigation, i.e., a NavigationFlow from a source ViewElement $S$ to a target ViewElement $T$, with $S$ and $T$ positioned in arbitrary locations of the hierarchically nested structure of the application.

Figure 4.27 exemplifies a multi-level interface structure. The application contains two top-level ViewContainers $TL0$ and $TL1$. $TL1$ contains two XOR ViewContainers $X0$ and $X1$, which comprise two children ViewContainers each: $X0C0$, $X0C1$, $X1C0$

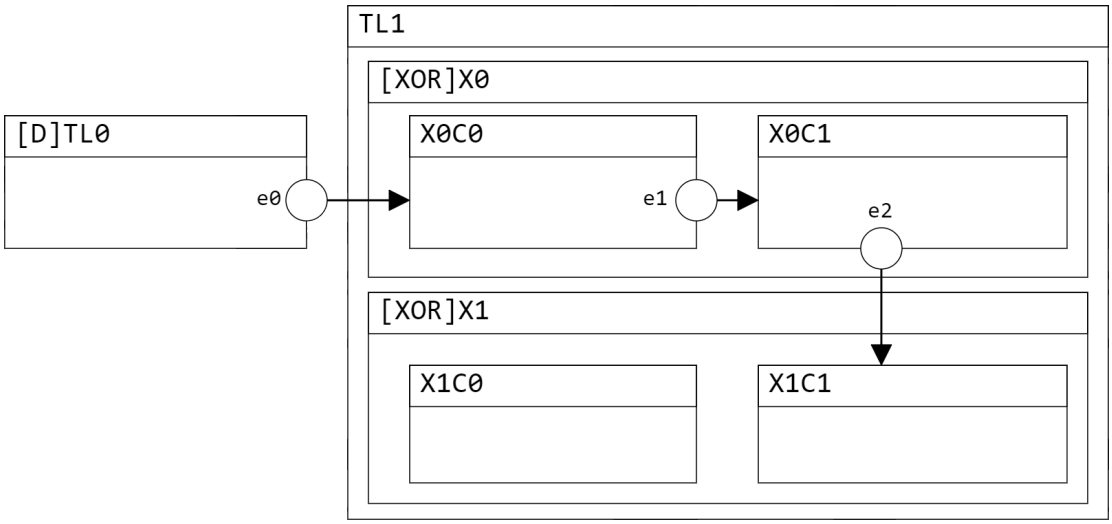



**Figure 4.27:** *Deeply Nested ViewContainers navigation - IFML model*

and $X1C1$. The model also comprises three events $e0$, $e1$ and $e2$, with associated NavigationFlows.

**Preliminary definitions**

Before proceeding to the illustration of the mapping, we introduce a number of auxiliary definitions.

Definition 2 identifies the smallest ViewContainer in which an interaction, denoted by an Event and its associated NavigationFlow, operates.

**Definition 2.** INTERACTION CONTEXT

Given a NavigationFlow $F$, with source $S$ and target $T$, the Interaction Context of $F$ ($I_F$) is the ViewContainer $V$ such that:

- $V$ = $S$, if $S$ is ancestor of $T$ and is not a XOR ViewContainer;
- $V$ = $T$, if $T$ is ancestor of $S$ and is not a XOR ViewContainer;
- $V$ is the ancestor of $S$ and $T$ such that no descendant of $V$ is ancestor of $S$ and $T$, in all the other cases.

Intuitively, the interaction context of an event and of its associated NavigationFlow is the smallest portion of the application that contains the source and the target of the interaction. In the example of Figure 4.27, the interaction context of the NavigationFlow associated with $e0$ is $Application$, of the NavigationFlow associated with $e1$ is $X0$, and of the NavigationFlow associated with $e2$ is $TL1$.

Definition 3 identifies the largest sub-elements of a ViewContainer that comprise interface elements displayed in alternative.

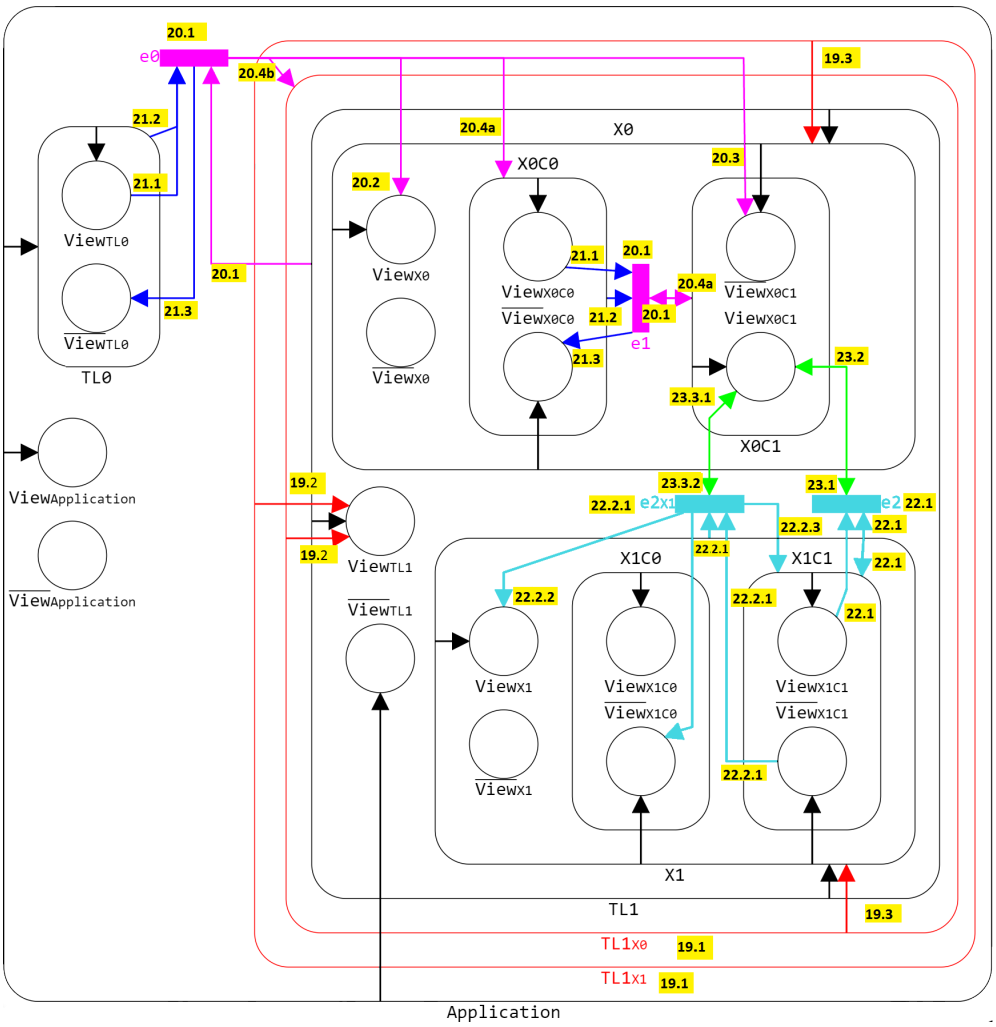



**Figure 4.28:** *Deeply Nested ViewContainers navigation*

**Definition 3.** TOPMOST XOR DESCENDANTS

Given a non-XOR ViewContainer $V$, the set of topmost XOR descendants of $V$ is the set of its descendant XOR ViewContainers that have no ancestor XOR ViewContainers that are descendants of $V$.

Intuitively, given a portion of an interface, the topmost XOR descendants identify the largest independent regions enclosed in it that may display alternate content. In the example of Figure 4.27, the sets of topmost XOR descendants of $TL0$, $X0C0$, $X0C1$, $X1C0$ and $X1C1$ are empty, whereas the set of topmost XOR descendants of $TL1$ contain $X0$ and $X1$.

When an interaction causes a ViewElement to be displayed, the nested structure of the interface may require other elements to get into view as well. Definition 4 identifies the portion of an interface element that gets into view when one of its sub-elements is displayed.

**Definition 4.** CO-DISPLAYED ANCESTOR

Given a ViewElement $V$, let $X$ be a XOR ViewContainer ancestor of $V$; the co-displayed ancestor of $V$ inside $X$ is the ViewContainer $A$ such that:

- $A = V$, if $V$ is direct child of $X$.
- $A$ is the child of $X$ ancestor of $V$, otherwise.

Intuitively, the co-displayed ancestor of a ViewElement inside a region of the interface, or inside the whole application interface, identifies the largest container that gets into view when the ViewElement is displayed. In the example of 4.27, the co-displayed ancestor of $X0C0$ inside $Application$ is $TL1$ and the co-displayed ancestor of $X0C0$ inside $X0$ is $X0C0$ itself.

The presence of XOR ViewContainers in the structure of the interface produces another side-effect of navigation: the display of a set of target elements may require hiding other elements, which are replaced by the newly displayed ones. Definitions 5 and 6 help identify the portions of an interface element that are activated by an interaction event and comprise content displayed alternatively.

**Definition 5.** XOR TARGETS SET

Given a NavigationFlow $F$, with target $T$, and a ViewContainer $A$ ancestor of $T$, the XOR targets set of $F$ inside $A$ ($\mathcal{T}_F^A$) contains the XOR ViewContainers ancestors of $T$ descendants of $A$.

**Definition 6.** EXTENDED XOR TARGETS SET

Given a NavigationFlow $F$ with target $T$ and a ViewContainer $A$ ancestor of $T$, the extended XOR targets set of $F$ inside $A$ ($\mathcal{T}^{\star A}_F$) is defined as:

- $\mathcal{T}^{\star A}_{F} = \mathcal{T}^{A}_{F} \cup \{A\}$, if $A$ is a XOR ViewContainer
- $\mathcal{T}^{\star A}_{F} = \mathcal{T}^{A}_{F}$ otherwise.

Intuitively, the XOR targets set of a NavigationFlow comprises all the ViewContainers that may display alternate content after the interaction; such ViewContainers must be initialized correctly, distinguishing their sub-elements that must be displayed or hidden.

In the example of 4.27, the XOR targets set of $e0$ inside $Application$ contains $X0$ and the extended XOR targets set contains $X0$ and $Application$; the XOR targets set of $e1$ inside $X0$ is empty and the extended XOR targets set contains $X0$; the XOR targets set of $e2$, inside $TL1$ contains $X1$ and the extended XOR targets set is the same; finally, the XOR targets set of $e2$ inside $X1$, is empty and the extended targets set contains $X1$.

Definitions 7 and 8 identify the portions of an interface element that transition into view or out of view as a consequence of an interaction.

**Definition 7.** DISPLAY SET

Given a NavigationFlow $F$, with target $T$, and a ViewContainer $A$ ancestor of $T$, let $\mathcal{T}^{\star A}_{F}$ be the extended XOR targets set of $F$, inside $A$. The display set of $F$, inside $A$, ($\mathcal{D}^{A}_{F}$) contains all the co-displayed ancestors of $T$ inside each element in $\mathcal{T}^{\star A}_{F}$.

Intuitively, after an interaction targeting a ViewElement $T$, the target $T$ becomes in view. Such activation propagates upwards in the interface hierarchy of containers to the relevant co-displayed ancestors of $T$.

In the example of 4.27, the display set of $e0$, inside its interaction context ($Application$), contains $X0C0$ (the target) and $TL1$ (a co-displayed ancestor in the interaction context); the display set of $e1$, inside its interaction context ($X0$), contains $X0C1$ (the target); the display set of $e2$, inside its interaction context ($TL1$), contains $X1C1$ (the target). Finally, the display set of $e2$, inside $X1$, contains $X1C1$ (because the *extended* targets set inside $X1$ contains $X1$ itself).

**Definition 8.** HIDE SET

Given a NavigationFlow $F$, with target $T$, and a ViewContainer $A$ ancestor of $T$, let $\mathcal{T}^{\star A}_{F}$ be the extended XOR targets set of $F$ inside $A$ and let $I_F$ be the interaction context of $F$. The hide set of $F$, inside $A$, ($\mathcal{H}^{A}_{F}$) contains all the ViewContainers not ancestors of $T$ and children of an element $X_i$ in $\mathcal{T}^{\star A}_{F} \backslash \{I_F\}$.

Intuitively, for an interaction targeting a ViewElement T, the hide set identifies the interface elements that are displayed *in alternative* to T and to its co-displayed ancestors; after the interaction, these elements are out of view.

In the example of 4.27, the hide set of $e0$, inside its interaction context ($Application$), contains $X0C1$ (both the source $TL0$ and $X0C1$ get, or remain, out of view); the hide set of $e1$, inside its interaction context ($X0$), is empty; the hide set of $e2$, inside its interaction context ($TL1$), contains $X1C0$. Finally, the hide set of $e2$, inside $X1$, contains $X1C0$.

### Mapping initialization

Based on the definitions above, the mapping from the IFML model of a nested interface to an equivalent PCN can be defined. The main difference with the case of flat interfaces occurs when the XOR targets set of a NavigationFlow, within its interaction context, is non empty, which means that there are ViewContainers, other from the direct target, which are affected by the interaction and may get into view; in this case, the PCN mapping must initialize the children of XOR targets set ViewContainers properly, so that the right ones are displayed. Specifically, the default initialization should *not* be applied indiscriminately to the ViewContainers of the XOR targets set, but selectively to their children, not to override the effect of the user's navigation. The correct policy is to initialize by default the ViewContainers in the display set, which should be computed and rendered, remove from view the ViewContainers in the hide set, and put into view the ViewContainers of the XOR targets set. In the example of Figure 4.27, the XOR target set inside the interaction context of $e0$ contains the $X0$ ViewContainer. When $e0$ occurs, the correct behavior is *not* to initialize by default $X0$, because this may override the user's choice of accessing a specific child ViewContainer ($X0C0$). Instead, $X0$ should be set to the visible state, its explicitly targeted child $X0C0$ should be initialized by default, and the other XOR children in the hide set (just $X0C1$ in the present case) should be set to the not in view status. Conversely, the default initialization should be applied to the $X1$ XOR ViewContainer, because this is not directly affected by the NavigationFlow (it does not belong to the XOR targets set).

In other terms, as the example of Figure 4.27 shows, when a ViewContainer (e.g.,$TL1$) comprises one or more XOR descendants (e.g., $X0$ and $X1$), there may be multiple ways to initialize the XOR ViewContainers and their children, depending on the actual target of an interaction; this behavior must be captured by the mapping rules, as illustrated in the PCN of Figure 4.28.

The alternative ways to initialize the XOR ViewContainers and their children are specified by the following rule:

Rule **19)** SELECTIVE INITIALIZATION

Given a non XOR ViewContainer $V$, each topmost XOR descendant $X_i$ of $V$ maps to:

1) a place chart $\mathsf{V/X_i}$ enclosing the place chart $\mathsf{V}$; this denotes that the ViewContainer $V$ may be accessed by a navigation that targets $X_i$ or one of its descendants; in Figure 4.28, this clause inserts the place charts $\mathsf{TL1/X0}$ and $\mathsf{TL1/X1}$.

2) An initialization arc from $\mathsf{V/X_i}$ to $\mathsf{View_V}$; this means that being in $\mathsf{V/X_i}$ implies being in $\mathsf{V}$; this clause inserts the arc from $\mathsf{TL1/X0}$ and $\mathsf{TL1/X1}$ to $\mathsf{View_{TL1}}$.

3) For all the children $C_j$ of $V$, such that $C_j \neq X_i$ and $C_j$ is not an ancestor of $X_i$, an initialization arc from $\mathsf{V/X_i}$ to $\mathsf{C_j}$; this denotes that the ViewElements not affected by the navigation are initialized by default; this clause inserts the arc from $\mathsf{TL1/X0}$ to $\mathsf{X1}$ and from $\mathsf{TL1/X1}$ to $\mathsf{X0}$.

4) If an ancestor $A_i$ of $X_i$ child of $V$ exists, an initialization arc from $\mathsf{V/X_i}$ to $\mathsf{A_i}$; since Rule 19 applies to all non XOR ViewContainers that enclose the XOR ViewContainer $X_i$, this clause denotes that if $X_i$ is nested inside $V$ through a set of non XOR ViewContainers $\{A_{i,1}, \ldots A_{i,n}\}$, where $A_{i,j}$ is child of $V$, $A_{i,j+1}$ is child of $A_{i,j}$, and $A_{i,n}$ is father of $X_i$, then there is a place chart $\mathsf{A_{i,j}/X_i}$ for each ancestor of $X_i$, with an arc from $\mathsf{A_{i,j}/X_i}$ to $\mathsf{A_{i,j+1}/X_i}$, for $j = 1 \ldots n-1$. This clause ensures that initialization is correctly propagated along nested non XOR ViewContainers and "stops" at the XOR ViewContainer.

The effect of rule 19 is to insert a place chart that makes explicit the access to a specific XOR-child of a ViewContainer (e.g., the place chart $\mathsf{TL1/X0}$ denotes that $TL1$ is accessed through $X0$); such place chart is used to specify that the default initialization of the ViewContainer (e.g, of $TL1$) does not apply to one of its descendant XOR children (e.g., to $X0$).

Figure 4.28 shows the PCN mapping the IFML diagram of Figure 4.27[5]. As per rule 19, the place chart $\mathsf{TL1}$ is enclosed within the two auxiliary place charts $\mathsf{TL1/X0}$ and $\mathsf{TL1/X1}$, with the initialization arcs inserted by the rule.

**Mapping NavigationFlows**

The rules that map a NavigationFlow from S to T, positioned arbitrarily within a nested interface structure, distinguish two cases:

1 The interaction context of the NavigationFlow is a XOR ViewContainer (as in the case of the NavigationFlows associated with $e0$ and $e1$): this means that S and T cannot be visible at the same time. Specifically, S must be in view, otherwise the event cannot be triggered, and T must be out of view. The effect of the interaction does not depend on the actual status of the interface: S gets out of view and T becomes visible.

2 The interaction context of the NavigationFlow is a non XOR ViewContainer, as in the case of $e2$: this means that S and T may or may not be visible at the same time. Specifically, S must be in view, otherwise the event cannot be triggered, while T can be in view or not depending on the current state of the interface, which depends on the past interactions affecting T and possibly the ancestors of T. Thus, the interaction can have different effects depending on the current visibility status of the target and of its ancestors.

**XOR Interaction Context**. The case in which the interaction context of a NavigationFlow is a XOR ViewContainer generalizes the rules for the navigation between top-level ViewContainers described in Section 4.2.6. In this situation, the source of the interaction is in view and the target is out of view. Therefore, the interaction must set the target to the visible state and, differently from the flat interface case, also update the visibility of the correct children of its XOR ancestors.

In the place chart of Figure 4.28, the transition $\mathsf{e0}$, instead of adding a token to $\mathsf{TL1}$, adds a token to the auxiliary place chart $\mathsf{TL1/X0}$, thus avoiding to initialize by default $\mathsf{X0}$. It also adds a token to $\mathsf{View_{X0}}$, $\overline{\mathsf{View}}_{\mathsf{X0C1}}$ and $\mathsf{X0C0}$, completing the initialization of $X0$ with the correct active child. Note that the transition mapping the NavigationFlow

[5]In the figure, we show by means of labels the specific rule or rule clause responsible of inserting each relevant portion of the PCN.

associated with the $e1$ event follows the rule for top-level ViewContainers (rule 5), as illustrated in Figure 4.8, because there is no XOR ViewContainer between the common ancestor $X0$ and the target $X0C1$ (the XOR target set of $e1$ is empty).

The mapping of events such as $e0$ introduces the following rules:

Rule **20)** GENERIC NESTED NAVIGATION (XOR CONTEXT)

Given a NavigationFlow $F$ associated with an event $e$ and with target $T$, whose interaction context $I_F$ is a XOR ViewContainer, $F$ maps to:

1) A transition e that removes a token from the place chart of the co-displayed ancestor of $T$ child of $I_F$; this empties all the place charts contained within $I_F$. In Figure 4.28 this clause inserts the transitions e0 and e1, and the arcs from TL1 to e0 and from X0C1 to e1.

2) For each ViewContainer $X_i$ in the XOR targets set of $F$ inside $I_F$, an arc from e to $\mathsf{View}_{\mathsf{X_i}}$; this sets to visible all the affected XOR ViewContainers; this clause inserts the arc from e0 to $\mathsf{View}_{\mathsf{X0}}$ (the XOR target set of $e1$ is empty).

3) For each element $H_i$ in the hide set of $F$ inside $I_F$, an arc from e to $\overline{\mathsf{View}}_{\mathsf{H_i}}$; this sets the ViewContainers of the hide set to invisible; this clause inserts the arc from e0 to $\overline{\mathsf{View}}_{\mathsf{X0C1}}$ (the hide set of $e1$ is empty).

4) For each element $D_i$ in the display set of $F$ inside $I_F$:

   4.a) an arc from e to $\mathsf{D_i}$, if there does not exist a ViewContainer topmost XOR descendant of $D_i$ and ancestor of $T$; this maps the case in which activation does not propagate upward along the container hierarchy; this clause inserts the arc from e0 to X0C0 and from e1 to X0C1.

   4.b) An arc from e to $\mathsf{D_i}/\mathsf{A_j}$, if there exists a ViewContainer $A_j$ topmost XOR descendant of $D_i$ and ancestor of $T$; this maps the case in which other ViewContainers at higher levels of the container hierarchy are activated; this clause inserts the arc from e0 to TL1/X0.

Rule **21)** NESTED NAVIGATION FROM VIEWELEMENT (XOR CONTEXT)

Given a NavigationFlow $F$ from an event $e$ with target $T$ associated with a ViewElement $S$, whose interaction context $I_F$ is a XOR ViewContainer, let $A$ be the co-displayed ancestor of $S$ inside $I_F$. Then $e$ (additionally) maps to:

1) An arc from $\mathsf{View_S}$ to e; this ensures that the source is in view; in Figure 4.28 this clause inserts the arc from $\mathsf{View}_{\mathsf{TL0}}$ to e0 and from $\mathsf{View}_{\mathsf{X0CO}}$ to e1.

2) An arc from A to e; this ensures that the co-displayed ancestor of $S$ is emptied; this clause inserts an arc from TL0 to e0 and from X0C0 to e1.

3) An arc from e to $\overline{\mathsf{View}}_{\mathsf{A}}$, if $T$ is not an ancestor of $S$; this denotes that the

co-displayed ancestor of the source gets out of view, unless it is contained in the navigation target; this clause inserts the arc from e0 to $\overline{\text{View}}_{\text{TL0}}$ and from e1 to $\overline{\text{View}}_{\text{X0C0}}$.

Rule 20 and 21 insert in the PCN of Figure 4.28 the transitions e0 and e1; both the mapped events have a XOR ViewContainer as navigation context.

Note that rule 20 addresses the general case, whereas rule 21 extends the PCN with arcs specific to the configuration in which the event is associated with a ViewElement; a similar rule (omitted for brevity, but implemented in the on-line system) is defined for the case in which the event is associated with an Action triggered by a deeply nested interface.

**Non-XOR Interaction Context**. This scenario is a generalization of the ones described for events and NavigationFlows associated with ViewContainers (in Section 4.2.6) and ViewComponents (in Section 4.2.9). When the interaction context is a non XOR ViewContainer, the source and the target may or may not be in view at the same time and thus the interaction may have different effects depending on the visibility status of the target when the event occurs. More precisely, the effect of the interaction does not depend on the current status of the application if there are no XOR ViewContainers ancestor of the target within the interaction context. If such XOR ViewContainers do exist, the effect of the interaction depends on the currently in view child in each of them. If all the "right" children are in view, the target is already in view too, and needs only to be recomputed. If instead a co-displayed ancestor of the target is out of view, it must be activated, together with its descendants that are children of a XOR ancestor of the target. $e2$ is an example of this situation; the interaction context is a non XOR ViewContainer ($TL1$) and there is a XOR ViewContainer ($X1$) ancestor of the target $X1C1$. If $X1C1$ is already in view when the interaction occurs, then it is just recomputed; if not, it must switch into view, replacing its sibling ViewContainer $X1C0$. Therefore, the mapping from IFML to PCN inserts two different transitions: the former changes the active child of $X1$ if $X1C1$ is not in view; the latter recomputes $X1C1$, if it is already in view.

The case of non-XOR interaction context is addressed by the following rule:

Rule **22)** CONDITIONAL NAVIGATION

Given a NavigationFlow $F$, associated with an event $e$ and with target $T$, whose interaction context $I_F$is a non XOR ViewContainer, $F$ maps to:

1) a transition e that removes/adds a token from/to T, and for each ViewContainer $A_i$ ancestor of T and child of an element in the XOR targets set of $F$ inside $I_F$, an arc from $\text{View}_{\text{A}_\text{i}}$ to e; and an arc from e to $\text{View}_{\text{A}_\text{i}}$; the "refresh" transition e maps the case in which the target is already in view: it causes the target to be recomputed and leaves the rest of interface unchanged. In Figure 4.28, this clause inserts the transition e2 with the arc to/from X1C1, and the arc from $\text{View}_{\text{X1C1}}$ to e2.

2) For each ViewContainer $X_i$ in the XOR targets set of $F$ inside $I_F$:

   2.a) a transition e ▶ $\text{X}_\text{i}$ that removes a token from $\text{X}_\text{i}$ and $\overline{\text{View}}_{\text{A}_\text{i}}$ and adds a

token to $A_i$, where $A_i$ is the co-displayed ancestor of $T$ inside $X_i$; each transition e ▶ $X_i$ maps the case in which a relevant sub-ViewContainer of $X_i$ is not in view: it causes the computation and switch into view of the ancestors of the target inside $X_i$, or of the target itself if this is a direct child of $X_i$. This clause inserts the transition e2 ▶ X1, the arc from X1 to e2 ▶ X1, from $\overline{\text{View}}_{\text{X1C1}}$ to e2 ▶ X1, and from e2 ▶ X1 to X1C1.

2.b) For each ViewContainer $X_{i,j}$ in the extended XOR targets set of $F$ inside $X_i$, an arc from e ▶ $X_i$ to $\text{View}_{X_{i,j}}$; this denotes that each element $X_i$ and the XOR ViewContainers nested inside it become in view. This clause inserts the arc from e2 ▶ X1 to $\text{View}_{\text{X1}}$.

2.c) For each element $D_{i,j}$ in the display set of $F$ inside $X_i$ such that there exists a ViewContainer $D_{i,k}$ that is the topmost XOR descendant of $D_{i,j}$ and ancestor of $T$, an arc from e ▶ $X_i$ to $D_{i,j}/D_{i,k}$; if such $D_{i,k}$ does not exist, the arc goes from e ▶ $X_i$ to $D_{i,j}$; this maps the computation of the elements of the display set inside $X_i$; this clause inserts the arc from e2 ▶ X1 to X1C1.

2.d) For each element $H_{i,j}$ of the Hide Set of $F$ inside $X_i$, an arc from e ▶ $X_i$ to $\overline{\text{View}}_{H_{i,j}}$: this maps the switch out of view of the elements of the hide set inside $X_i$; this clause has no effect because the hide set of $e2$ inside $X1$ is empty.

Rule **23)** CONDITIONAL NAVIGATION FROM VIEWELEMENT

Given a NavigationFlow $F$ from an Event $e$ associated with a ViewElement $S$and with target $T$, whose Interaction Context $I_F$is not a XOR ViewContainer. $F$ maps to:

1) an arc from $\text{View}_S$ to e; this denotes that the source must be in view; this clause inserts the arc from $\text{View}_{\text{X0C1}}$ to e2.

2) an arc from e to $\text{View}_S$, if $T$ is not an ancestor of $S$; this denotes that the source remains visible; this clause inserts the arc from e2 to $\text{View}_{\text{X0C1}}$.

3) For each element $X_i$ in the XOR targets set of $F$ inside $I_F$, $F$ also maps to:

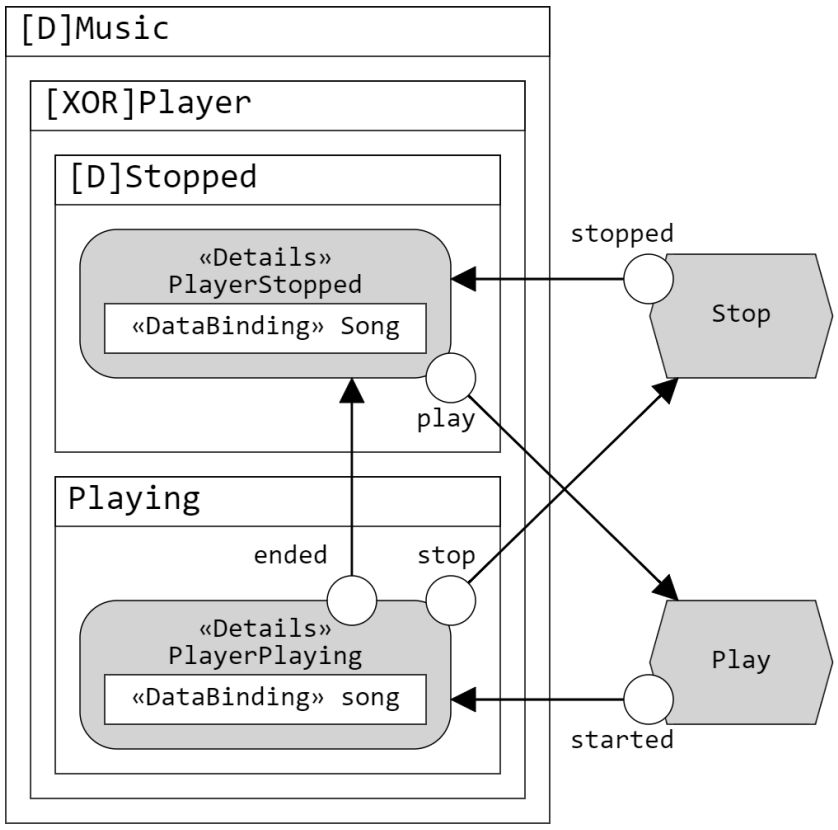
3.a) an arc from $\text{View}_S$ to e ▶ $X_i$; this clause inserts the arc from $\text{View}_{\text{X0C1}}$ to e2 ▶ X1.

3.b) an arc from e ▶ $X_i$ to $\text{View}_S$, if $T$ is not an ancestor of $S$; this clause inserts the arc from e2 ▶ X1 to $\text{View}_{\text{X0C1}}$.

### 4.4.3 Mapping Actions with Nested ViewContainers

Figure 4.29a extends the example of Figure 4.17, by specifying the $Playing$ and $Stopped$ ViewContainers as XOR-child of an external $Music$ ViewContainer.

Two new rules (rule 24 and 25) need to be added, as counterparts of Rules 21 and 23 that were defined for the case of ViewElements.




**(a)** *IFML model*

**Figure 4.29:** *Actions in nested structures*

Rule 24 addresses the case in which the effect of triggering the Action does not depend on the current status of the interface, because the Interaction Context of the NavigationFlow that triggers the Action is a XOR viewContainer.

Rule **24)** NESTED NAVIGATION FROM ACTION

Given a NavigationFlow $F$, from an Event $e$ associated with an Action $K$ and with target $T$, whose Interaction Context is a XOR ViewContainer $I_F$; let $O_K$be the Origin of the Action $K$ and $C$ be the child of $I_F$that is co-displayed ancestor of $O_K$inside $I_F$; $F$ maps to:

1) an arc from $\mathsf{Running_K}$ to $\mathsf{e}$. This clause specifies that the termination event transition requires the Action to be running to fire.
2) An arc from $\mathsf{C}$ to $\mathsf{e}$ and from $\mathsf{e}$ to $\overline{\mathsf{View_C}}$, if $\mathsf{T}$ is not an ancestor of $\mathsf{O_K}$; this propagates upwards in the ViewContainer hierarchy the switch out of view of the Interaction Context of the Action.

Rule 25 addresses the case in which the effect of triggering the Action depends on the current status of the interface, because the Interaction Context of the Navigation-Flow that triggers the Action is a non XOR viewContainer and thus the target and some of its co-displayed ancestors may be already in view.

Rule **25)** CONDITIONAL NAVIGATION FROM ACTION

Given a NavigationFlow $F$ from an Event $e$ associated with an Action $K$ and with target $T$, whose Interaction Context $I_F$is a non XOR ViewContainer, let $O_K$be the Origin of $K$. $K$ maps to:

1) an arc from $\mathsf{Running_K}$ to $\mathsf{e}$; this clause ensures that the termination event is fired when the Action is in the running state.
2) An arc from $\mathsf{e}$ to $\overline{\mathsf{Running_K}}$, if $T$ is not an ancestor of $O_K$; this clause maps the termination of the Action execution.
3) For each element $X_i$ in the XOR targets set of $F$ inside $I_F$, $F$ also maps to:
   3.a) an arc from $\mathsf{Running_K}$ to $\mathsf{e} \blacktriangleright \mathsf{X_i}$; this clause ensures that also the auxiliary transitions that handle the update of the XOR targets set elements are enabled when the Action is running.
   3.b) An arc from $\mathsf{e} \blacktriangleright \mathsf{X_i}$ to $\overline{\mathsf{Running_K}}$, if $T$ is not an ancestor of $O_K$.

## 4.5 Adherence to the Standard

We conclude the presentation of the mapping of IFML to PCN by underlying that the adherence of the mapping rules to the intended semantics of IFML 1.0 has been verified, albeit only informally; indeed, the IFML 1.0 specifications describe the behavior of the language only by means of examples described narratively. For verifying the

adherence of the proposed mapping rules to the intended meaning of IFML constructs, we have submitted them to an expert team led by the principal author of the IFML specifications [114]. The evaluation has been conducted on 13 test cases[6]. The models used for verification were designed as building blocks with increasing complexity, so to progressively incorporate the features of IFML 1.0 addressed by the mapping rules. Each model is small in size and is focused on a specific aspect of the IFML specification to enable in depth analysis of both the IFML and the generated PCN. The expert team verified the correspondence between the behavior expressed by the IFML model and that embodied in the corresponding PCN chart, by manually testing sequences of interaction events on both models.

[6]These sample models are published in the on-line tool IFMLEdit.org.

CHAPTER 5

# IFMLEdit.org: A reference implementation

The software development process is a refinement loop that iteratively transforms requirements into a final product. Iteration is fundamental to address incomplete, loosely defined, or rapidly changing functional and non-functional requirements. In web and mobile applications development, the wide range of device screens and coding platforms makes the ability to rapidly evolve and evaluate new releases even more critical.

In this Chapter we propose IFMLEdit.org[1] an open-source online tool for rapid prototyping of web and mobile applications. The tool was developed following an agile approach based on ALMOsT.js (proposed in Chapter 3), by the iterative introduction of new features in small runs. The tool will be further evaluated in Chapter 6, showing how it can be integrated in a developer centric work-flow.

IFMLEdit.org features an IFML visual editor that allows the developer to insert (by means of drag&drop) elements in the model, edit their property and connect them. The tool also features a PCN visual editor that was used during the development of the formal semantics presented in Chapter 4, enabling fast iterations and experimentation of the proposed rules. As a support to verification, IFMLEdit.org also offers a function to simulate the generated PCN, easing the inspection of its dynamics in reaction to an automatically created sequence of events. The IFML to PCN semantic transformation is described in Section 5.1. IFMLEdit.org features 4 code generators targeting 3 different architectures: Web Server, Web Client and Mobile. The IFML to code transformations are described in Section 5.2. Rapid iterations on the model are enabled thanks to in-browser emulation, which enables the developer to directly evaluate the effects of the model changes on a working prototype, without the need to download, build and deploy the generated code. For each target architecture the tool provides a specific emulator. All the emulators allow the developer to edit the information stored in the data access

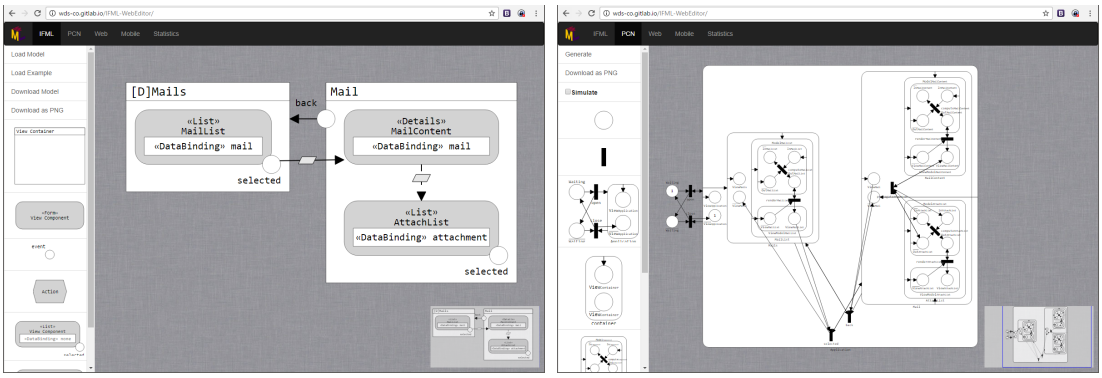
[1] https://www.ifmledit.org

layer to better understand the effects on the generated application.

IFMLEdit.org is not the only free and open source IFML editor. Another open source project[2], based on the Eclipse Modeling Framework (EMF) [12] is available. Both tools are based on IFML, but with different goals. IFMLEdit.org aims at facilitating the approach of new users to the IFML ecosystem, by providing an online tool which allows the developer to learn IFML by examples and generate a functional prototype of their application without programming and software installation. The project at `ifml.github.com` aims at providing a specification-compliant building block, limited to model editing, which MDD software companies can exploit as a starting point for the realization of a complete IFML IDE.

## 5.1 Model to Model transformation

IFMLEdit.org allows tool builders to analyze and simulate the semantics of their models, thanks to the mapping rules from IFML to PCN, proposed in Chapter 4, implemented with ALMOsT.js. Figure 5.1 shows the input IFML model (left) and the corresponding output PCN model (right), generated by the transformation rules. The PCN diagram expresses formally the dynamics of the application interface specified by the IFML diagram, using the classic concepts of Petri Nets, i.e., places, transitions, arcs and tokens[3].




**(a)** *IFML model* **(b)** *PCN model*

**Figure 5.1:** *IFML (left) to PCN (right) transformation*

Each rule of the model to model transformation is responsible of generating a part of the PCN diagram and thus creates an output partial model, following the structure presented in Section 3.2.3 (i.e., possessing only the two *elements* and *relations* members). The Reducer is configured using the predefined **m2a** (Model-To-ALMOsT) configuration, presented in Section 3.2.5.

The implemented transformation rules are in a one to one mapping with their formal definition proposed in Chapter 4. While the result of these rules contains the whole semantics of the application and can be used for simulation, it is missing any metadata required for its rendition on screen. In order to obtain a comprehensive graphical representation of the model, the purely semantic rules are paired with a purely graphical

[2] `http://ifml.github.com`

[3] PCNs add to Petri Nets advanced modularization constructs, which make complex diagrams particularly readable

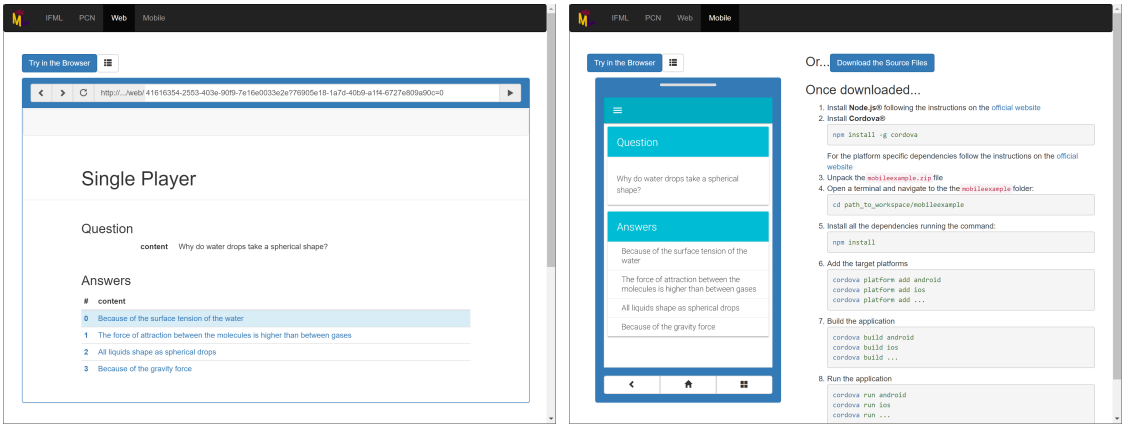
counterpart. This second set of rules is responsible to infer high level graphical meta-data of the PCN model, by exploiting the graphical meta-data of the original IFML model. This goal is accomplished by generating a second model where each *element* is a node a layout tree which defines constraints of the final graphical rendition, some of these nodes have a *relation* with a specific *element* in the original PCN model. A very simple layout algorithm is run on top of the layout tree converting the layout constrains in graphical meta-data (i.e, coordinates and sizes) which are attached to the related *elements* in the PCN model.

Overall, the transformation from IFML to PCN consists of 38 rules, 19 of which address all the aspects of the language while the remaining 19 are dedicated to the generation of the layout tree.

This transformation shows the power of ALMOsT.js, by exploiting different target logical meta-models in the same transformation without the need of any specific configuration.

## 5.2 Model to Text transformations

IFMLEdit.org present 4 different model to text transformations, implemented with ALMOsT.js, targeting different platforms and technologies. All these transformations exploit the output structure and *m2t* (Model-To-Text) Reducer configuration presented in Section 3.2.6 and use EJS [14] template language. The output of these rules is converted into a ZIP file which can be easily extracted by the developer.




**(a)** *Web* **(b)** *Mobile*

**Figure 5.2:** *Examples of generated applications*

### 5.2.1 Web Server

IFMLEdit.org features a model to text transformation targeting Web Servers, i.e. a full HTTP request response cycle is performed after each user interaction. The output of this transformation is a Node.js project following the MVC design pattern. Each model to text rule generates a subset of the folders and files that constitute the project. Model rules generate constant folders and files (e.g., system folders, configurations files, HTTP server initialization scripts, etc...) and textual files containing sample data

collections for filling the interface with content (repositories). Element rules generate the source code of a controller (JavaScript code) and of a view (Pug template). Controllers and views of nested IFML elements (e.g, nested ViewContainers and ViewComponents within ViewContainers) are included and configured during the server initialization phase. This model to text transformations consist of 12 rules.

The generated project can be downloaded as a zip file or run inside the browser. To support client-side execution of the project, a web worker is instantiated when IFMLEdit.org is accessed, which emulates a Node.js environment and the HTTP protocol.

### 5.2.2 Web Client

IFMLEdit.org features a model to text transformation targeting Web Clients, i.e. application running in the browser and communicating with remote resources just when the task cannot be performed locally. The output of this transformation is a JavaScript project following the MVVM design pattern. Each model to text rule generates a subset of the folders and files that constitute the project. Model rules generate constant folders and files (e.g., system folders, configurations files, WebComponents initialization scripts, etc...) and textual files containing sample data collections for filling the interface with content (repositories). Element rules generate the source code of Web Component [31] composed of a viewmodel (JavaScript code) and of a view (HTML template). Web components of nested IFML elements (e.g, nested ViewContainers and ViewComponents within ViewContainers) are connected by simple inclusion of the child in the parent view. This model to text transformations consist of 10 rules.

The generated project can be downloaded as a zip file or run inside the browser. To support client-side execution of the project, an IFrame is instantiated with custom APIs, which emulates packages support.

### 5.2.3 Mobile - Cordova

IFMLEdit.org features a model to text transformation targeting Mobile platforms. The output of this transformation is a Cordova project following the MVVM design pattern, similarly to the Web Client transformation. Each model to text rule generates a subset of the folders and files that constitute the project. Model rules generate constant folders and files (e.g., system folders, configurations files, WebComponents initialization scripts, etc...) and textual files containing sample data collections for filling the interface with content (repositories). Element rules generate the source code of Web Component [31] composed of a viewmodel (JavaScript code) and of a view (HTML template). Web components of nested IFML elements (e.g, nested ViewContainers and ViewComponents within ViewContainers) are connected by simple inclusion of the child in the parent view. This model to text transformations consist of 10 rules.

The generated project can be downloaded as a zip file or run inside the browser. To support client-side execution of the project, an IFrame is instantiated with custom APIs, which emulates packages support and device specific resources and functionalities which are not available in a desktop browser.

### 5.2.4 Mobile - Flutter

IFMLEdit.org features another model to text transformation targeting Mobile platforms. The output of this transformation is a Flutter [15] following the Reactive principles of the target framework. Each model to text rule generates a subset of the folders and files that constitute the project. Model rules generate constant folders and files (e.g., system folders, configurations files, main application scaffolding, etc...) and textual files containing sample data collections for filling the interface with content (repositories). Element rules generate the source code of Flutter Widgets [15] (Dart [11] code). Widgets of nested IFML elements (e.g, nested ViewContainers and ViewComponents within ViewContainers) are connected by simple inclusion of the child widget in the parent layout tree. This model to text transformations consist of 10 rules.

The generated project can be downloaded as a zip file. IFMLEdit.org does not provide emulation support for this generator due to the impossibility to directly execute the generated code in a JavaScript sandbox.

CHAPTER 6

# Seamless Model and Text Co-Evolution

## 6.1 Introduction

Model Driven Development (MDD) is the branch of software engineering that advocates the use of *models*, i.e., abstract representations of a system, and of *model transformations* as key ingredients of software development. Developers use a general purpose (e.g. UML [30]) or domain specific (e.g. IFML [114]) modeling language to specify systems under one or more perspectives, and employ (or build) suitable chains of transformations, to refine the models into a final product (e.g., executable code).

Abstraction is the most important aspect of MDD. It enables developers to validate high level concepts and introduce low-level details at later stages in the process. In the *forward engineering* approach [51, 116, 126], details are introduced via model transformations, which iteratively refine the model, eventually getting to the final product. After a model change, the process is reiterated to produce a new version. Model to Model (M2M) and Model to Text (M2T) transformations are exploited to achieve this goal.

In this chapter we focus on transformations that refine a platform-independent model by introducing details specific for a target platform (e.g., system, implementation or rendition languages). This specialization is obtained via M2T transformations having a high-level meta-model as the source and a platform specific meta-model as the target. An example is a M2T transformation that maps a state machine representing GUI interactions to the source code of a specific GUI framework.

The source meta-model may not contain enough information to drive the production of a unique output, due to its abstraction level that omits details of secondary importance. A pure forward engineering approach requires such unknowns to be solved by enhancing both the source meta-model and the transformations, to remove uncertainties and maintain a unidirectional flow from model to code. However, such an approach can

lead to loss of abstraction in the source model and diminishes the benefits of the MDD methodology. User Interfaces (UI), and in particular web based ones, are a relevant example of this situation; model abstraction enables reasoning about the organization of the front-end and about the essential interactions, regardless of presentation details; however, the latter aspects, such as content layout, fonts, colors, or gestures, must be addressed after the core GUI design is complete, because their quality greatly influences the final user experience.

For transformations to generate specific features of the output not inferable from the input model, **template-based** approaches can be used [116]; commercial tools (e.g. WebRatio [32], Mendix [19], Outsystems [21] and Zoho Creator [34]) require developers to annotate high-level models with ad hoc attributes, so that the M2T transformation can select the proper implementation template and create the desired output. This approach has been successfully adopted in the industry, but introduces an additional source of complexity: the construction of templates could demand more work than the manual coding of the final artifact, especially for applications with very specific low-level requirements.

The alternative approach to the use of template-based transformations is to exploit MDD tools for the creation of the first prototype of the application, which is then extended manually to incorporate the aspects abstracted by the input model and by the transformations that apply to it. The obvious downside of this method is the misalignment between the model and the code, which hinders the use of transformations in the whole life-cycle of the application.

In this chapter, we concentrate on the development of projects involving MDD tool chains, with particular attention to the problem of co-evolving the model and the code. We propose a work-flow, based on well established developer-centric tools and methodologies, whereby MDD tools and developers are considered as equals and can both update the source code of the application, in a way that preserves the modifications introduced by both human and automated developers.

The contributions of this chapter can be summarized as follows:

- We propose a **model and code co-evolution** work-flow, which considers an MDD tool as a **Virtual Developer**, who works together with human developers seamlessly. The work-flow captures the similarities between distributed development and model and code co-evolution and helps resolve conflicts between the code produced by human developers and by MDD tools, in a general way that does not depend on the specific characteristics of the model transformation frameworks.

- We formalize the notion of conflicts between the manually and automatically created revisions of the source code and propose two strategies to address them: a *resolution strategy* exploits the functionality of common Version Control Systems (VCSs) to identify collisions and either solve them automatically or point them out to the developer for manual resolution; a *prevention strategy* formulates a few, simple rules for the design of code generators, which reduce the occurrence of unnecessary conflicts substantially.

- We provide a reference implementation of a MDD tool-chain (called ALMoST-Git [4]) that supports the proposed model and code co-evolution process, built on top of the popular Git [16] VCS.

- We evaluate the impact of the proposed model and code co-evolution process, under two viewpoints:

  - We develop two use-case applications, limited in size but realistic in the technical requirements, in four ways: with the proposed model and code co-evolution process and with a template-based forward engineering process, with two different MDD platforms: a prototyping tool (IFMLEdit.org [44]) and an industrial tool (WebRatio [32]). The comparison focuses on the amount of work required to integrate the manually developed features into the MDD model-and-generate loop and its impact on the total development effort. The assessment shows that in IFMLEdit.org model and code co-evolution process requires, in the worst case scenario, **83%** less code updates and impacts **84%** less lines than the template based approach, which leads to **27%** less code updates and **27%** impacted less lines during the whole application development; in WebRatio model and code co-evolution process requires, in the worst case scenario, **80%** less code updates and impacts **95%** less lines than the template based approach, which leads to **45%** less code updates and **38%** impacted less lines during the whole application development.
  - We evaluate the utility of the proposed conflict prevention strategy by re-implementing the code generator of IFMLEdit.org, so to incorporate the design guidelines; the improved version provides a further reduction of the integration work with respect to the original IFMLEdit.org implementation: in the worst case scenario, **60%** less code updates impacting **27%** less lines of code than the original code generator without the conflict prevention rules, which leads to a reduction of **7%** of the total development effort.

The rest of the chapter is organized as follows: Section 6.2 surveys the relevant literature on model-to-text transformations and the status of the practice in Version Control Systems for distributed development. Section 6.3 discusses the work-flow that allows developers and MDD tool-chains to work together, by co-evolving the code-base of a software project. Section 6.4 focuses on the critical aspect of the work-flow introduced in Section 6.3: the management of conflicts between manually programmed and automatically generated code; it shows how the proposed work-flow can accommodate industry standard MDD practices, such as template-based forward engineering and the use of protected areas, but can also enable less constrained, semi-automatic procedures for integrating the manually developed features into the MDD mode-and-generate loop. Section 6.5 presents simple guidelines for the design of model transformations that aim at reducing the need of manual intervention during conflict resolution between manually programmed and automatically generated code. Section 6.6 present an open-source implementation of the proposed approach, obtained by integrating the IFMLEdit.org MDD environment and the the Git Version Control System. To show the generality of the proposed approach, a second implementation has been performed, integrating Git with a commercial MDD tool (WebRatio). Section 6.7 evaluates the impact of the proposed work-flow during the development of two use cases with both the template-based forward engineering process and the model and code co-evolution process; it also assesses the effectiveness of the proposed transformation design guidelines.

## 6.2 Background

The research problem tackled in this chapter is the integration between manually programmed and generated code, which are the two ingredients of applications developed with MDD processes. The coexistence of the two types of code is unavoidable because automatically generated code represents only a fraction of the application requirements and thus the non-modeled aspects must be programmed manually and integrated in the mode-and-generate cycle. All the integration approaches aim at promoting the manually programmed code to a "higher" status, mapping it either into a *template* or into a *model feature*. Such a promotion can be attained, or at least supported, with different techniques: templatization, transformation inversion, and traceability links. After promotion, the part of the application that was originally coded manually can be regenerated automatically, with standard forward engineering.

The approach discussed in this chapter differs. No promotion is pursued: the generated and manual parts of the application are considered as having equal status and their inconsistencies are resolved semi-automatically. In this way, the model can be evolved and the code generated as if no manual modifications were necessary; the manual modifications, occurred in all the iterations of the project, are incorporated in the code base seen by both the human programmers and by the MDD tools and automatically "dragged" from one application version to the next.

The previous relevant work spans the following areas:

- **Transformation evolution and maintenance**: Model-to-text transformations are at the base of MDD. Most transformation frameworks exploit a template-based approach [117], i.e., they rely on partially developed examples of the output, which are incorporated during generation to produce the application code; this approach guarantees the needed flexibility, to the price of the complexity entailed in the creation and maintenance of templates [75, 117]. Relevant work concentrate on simplifying transformation design and on supporting the extraction of templates from reference implementations.

- **Round-trip engineering and co-evolution** Works in this category focus on the identification, maintenance, and even inversion, of the links between the input and the output of the transformation. Bidirectional approaches pursue the objective of maintaining consistency between two or more artifacts, by propagating changes in both the directions of a transformation. Traceability approaches aim at identifying the derivation path from one artifact to another, supporting change impact analysis and the back-propagation of properties from the code to the model.

### 6.2.1 Transformation Evolution and Maintenance

Various approaches and tools have been devised to enhance, or even to replace, M2T transformations. Protected areas are a simple mechanism to integrate manually programmed and automatically generated code; they define portions of the application for which the code is provided by the developer and is not overwritten by the code generator. For example, the industry standard transformation framework Acceleo [1] offers protected areas as well identifiable safe islands of code, which are preserved between executions of the model transformation.

In [48] the automatic production of code generators from interpreters has been proposed to avoid the need of manually coding M2T transformations. In [102] an approach is introduced that shows how modularization and dynamic invocation of templates can simplify the maintenance of transformations.

The need of evolving transformations can arise from changes in both the source meta-model and in the target technologies. In [86] an approach to simplify M2T transformation evolution after a meta-model change is discussed. In [75] a survey of possible approaches to organize model transformations is conducted, analyzing the effects of moving rapidly evolving aspects of the architecture from the M2M transformations to the M2T transformations, and even outside the MDD work-flow into an abstraction layer managed manually; the study focused on SQL queries.

The work [117] addresses the problem of template maintenance: the authors acknowledge the complexity and overhead of template programming and explore a mechanism (called *templatization*) for automating the update of templates after changes in the manually programmed code of the application from which the templates are extracted. The technique applies only to template maintenance, because the initial definition of the templates entails the design of their logic, a process which is difficult to automate.

The above mentioned approaches exploit specific features of the input and output meta-models and of the transformation infrastructure, to either simplify transformation development or to support the promotion of manual code into generative templates; the approach discussed in this chapter differs in that it is agnostic with respect to the input and output languages and on the model transformation technologies and addresses model and code co-evolution by relaxing the separation between the generated and the manually programmed code, supporting the identification and resolution of inconsistencies a posteriori, after each generation step.

### 6.2.2 Round trip engineering and model and code co-evolution

Round-trip engineering [37] refers to the capacity of synchronizing related software artifacts, such as models and source code. Model and code co-evolution techniques have been explored to support round-trip engineering.

The proposed approaches fall in two categories: bi-directional transformation and trace-based techniques.

#### Bidirectional Transformations

Bidirectionality, in software development, aims at maintaining consistency among a set of interdependent artifacts [77]. In MDD, bidirectional transformations support reversing the mapping from a model to another model [85]; when applied between a model and the code, they enable the automatic reconstruction or enrichment of model features from source code.

In [36] a bidirectional M2T transformation approach based on Triple Graph Grammar (TGG) is discussed. The Abstract Syntax Tree (AST) representation of the target language is used in a bidirectional M2M transformation defined via TGG. The AST is structured with particular attention to supporting extra chunks of text that can be introduced during manual modifications, but are not directly managed by the transformation. In general, addressing code-level changes by lifting them at the model level can reduce

the complexity of modeling and transformation, at the cost of defining a parser specific for the target language and a reverse mapping from low level code to the high level concepts; this approach normally works well for a limited class of code-level patterns.

#### Traceability of model transformations

In MDD, methods based on traceability links enrich the model transformation framework with additional information that relates source model elements to corresponding target model elements [121, 130, 136]. This mechanism can be used to track the derivation of model element, for purposes such as debugging, testing and performance tuning of model transformations. Traceability, when applied to a model-to-text transformation, can also support the analysis of the impact of a model update onto the generated code and thus support the reconciliation between manually and automatically generated code.

In [58] the authors integrate traceability links into an MDD code generation framework, to better monitor non-functional requirements throughout the life-cycle of a project. The idea is to exploit traceability links to enable the back-porting of the values of non functional properties, evaluated on the source code, to the modeling level, so that successive generations of the implementation can be checked for the preservation of the desired code-level properties.

In [78] an approach is presented, based on a tracing framework for change retainment, which helps tracing back code-level modifications to the model.

The recent work [63] pursues the goal of supporting collaborative MDD at both the modeling and coding level; a trace-based approach is exploited to detect modifications in both the model and the source code. After every update, a trace model is produced and used to merge manual code changes into code produced from application models and ensure that the shared code base reflect both model and manual code refinements. The collaborative development environment can also detect violations, i.e., manual edits of forbidden code segments, and signal them to the developers. The approach requires an ad hoc template engine, capable of producing the needed trace models, and limits the intervention of the programmers to fine-grained protected segments of the source code.

Both bidirectional transformations and traceability systems are bound to the target languages and transformation frameworks. The support of different languages or of new features of a target language requires updating the transformation and tracking chains.

The approach proposed in this chapter does not require the bidirectionality of the transformation nor the maintenance of an ad hoc traceability framework and poses minimal requirements on the collaborative development process, M2T transformation, input and output languages. In Section 6.4 we discuss the advantages and drawbacks of this approach w.r.t pure template-based MDD and show how the proposed work-flow and tools can help integrate manually programmed and automatically generated code, with no assumptions about the code organization and no dependency on the involved tools and languages.

### 6.2.3 Software merging and conflict resolution

Software merging is the process of integrating multiple versions of a system, by supporting developers in doing changes to the code and removing inconsistencies. Merging is managed by *version control mechanisms*, implemented by *Version Control Systems (VCS)*. These tools allow developers to work on private local copies of the code, which are then reconciled to produce a new shared version. Merging and conflict resolution procedures can be used to help integrate manually programmed and automatically generated code.

Merging techniques has been studied for a long time [111]. They can be grouped into categories, with increasing conflict detection and resolution power but also complexity: *text-based* approaches consider a software artifact as a plain file and apply merging at the level of the individual lines of the text; *syntactic* techniques also consider the syntax of the language and are able to avoid the production of unnecessary conflicts, e.g., those arising within comments; *semantic* techniques move a step forward by considering the semantics of the language, which enable the detection of conflicts that a purely syntactic check would not identify; finally, an orthogonal category of approaches comprises *operation-based* techniques, which consider not only the artifacts but also the operations made by developers on them.

For the evaluation of the impact of the proposed model and code co-evolution workflow, which detects the conflicts between the code created manually and that generated by the MDD tool chain, we have integrated two MDD environments with the popular Git VCS, which uses standard text-based, line-level merging. Even with such a basic technique, which cannot resolve all conflicts, promising results are achieved: in Section 6.7 we show that the number of conflicts that require human intervention to integrate manually programmed code and automatically generated code is less than the number of changes at the model transformation level needed to incorporate the manually programmed features into templates of the code generator. Since the proposed approach is orthogonal to the conflict detection and resolution methods, it is possible to integrate it with more sophisticated merge algorithms, such as fine grained/language specific [48] automatic resolution technique, leaving the manual intervention of the developer as a fall-back.

## 6.3 Approach

In this Section we first analyze how the software developers' community deals with parallel development of software features (Section 6.3.1), then we show similarities and differences between parallel development and model and code co-evolution (Section 6.3.2), and finally we propose a solution to apply the parallel development methodologies to the co-evolution of model and code, by treating the MDD tool-chain as yet another developer in the team (Section 6.3.3).

Before proceeding, we introduce the concepts and notations used in the rest of the paper.

- **Developer**: $D_i$ denotes the i-th member of the development team.
- **Local/central code base**: $C_i$ denotes the version of the code-base edited by developer $D_i$. $C_C$ identifies the central code-base shared among developers.

- **Revision**: $R_{i,j}$ denotes the j-th revision of code-base $C_i$; it is the full textual artifact stored in $C_i$ at a particular point in time. $R_{C,j}$ denotes a revision in the central code base.
- **Equivalence**: Two revisions $R_{i,j}$ and $R_{m,n}$ are equivalent, if the content of the textual artifact stored in them is the same.
- **Difference**: the difference $R_{i,j} - R_{m,n}$ of two revisions is the set of changes that need to be applied to $R_{m,n}$ to produce $R_{i,j}$.
- **Delta**: the delta introduced by $R_{i,j}$ ($\Delta_{i,j} = R_{i,j} - R_{i,j-1}$) is the difference between $R_{i,j}$ and the previous revision $R_{i,j-1}$.
- **Conflict**: let $n$ be the number of revisions in the central code-base $C_C$; a revision $R_{i,j}$ is said to be in conflict with $C_C$ if $R_{i,j-1}$ in not equivalent to $R_{C,n}$, or if $j$ is equal to 1. In other terms, a conflict signals that a new revision has been produced starting from a newly initialized code-base or from a version different from the last one consolidated in the central code base.
- **Submission**: let $n$ be the number of revisions in the central code-base $C_C$; the submission $R_{i,j} \rightarrow C_C$ is the act of creating a shared revision $R_{C,n+1}$ in $C_C$ which is equivalent to the local revision $R_{i,j}$. A submission is always performed from a local code-base $C_i$ to the central code-base $C_C$. If $R_{i,j}$ is not in conflict with $C_C$, the submission $R_{i,j} \rightarrow C_C$ generates $R_{C,n+1}$, in such a way that $\Delta_{C,n+1}$ is equivalent to $\Delta_{i,j}$. Note that $R_{i,j}$ is equivalent to $R_{C,n+1}$, by the definition of submission, and $R_{i,j-1}$ is equivalent to $R_{C,n}$, by the definition of conflict. The submission of a non-conflicting revision $R_{i,j}$ is always accepted. Conversely, the submission of a revision in conflict is by default **rejected**, to avoid the possible loss of local changes and prompt the developer to solve the conflict. A conflicting submission can be **forced**, overriding the default.
- **Conflict Resolution**: Let $n$ and $m$ be the number of revisions in the local code-base $C_i$ and in the shared code-base $C_C$ respectively; the conflict resolution $R_{C,m} \mapsto C_i$ is the act of creating a (local) revision $R_{i,n+1}$ based on both $R_{C,m}$ and $R_{i,n}$. Note that $R_{i,n+1}$ may be equivalent neither to $R_{C,m}$ nor to $R_{i,n}$. The objective of conflict resolution is to enable a submission from a developer even if he has worked on a version that is out of synch with respect to the central code base. Solving a conflict between $R_{C,m}$ and $C_i$ allows a submission $R_{i,n+1} \rightarrow C_C$ to be performed, even if $R_{i,n} \rightarrow C_C$ was previously rejected due to a conflict.

  When a revision $R_{i,n}$ is in conflict with $C_C$ the developer can solve the conflict by generating a new revision $R_{i,n+1}$. Performing the operation on $C_i$ instead of $C_C$ allows other developers to continue their work without interference. After the resolution is performed, the developer can submit $R_{i,n+1}$ to $C_C$. More than one conflict resolution step may be needed if $C_C$ gets concurrently updated during the development process.
- We refer to a revision of the code-base that results from a manual change as $R^M_{i,j}$ (**manual revision**), to a revision containing generated artifacts as $R^G_{i,j}$ (**generated revision**), and to a revision that results from conflict resolution as $R^R_{i,j}$ (**resolution revision**).

### 6.3.1 Parallel & Distributed Development

During day to day operations, teams must deal with the problem of parallel development, whereby different subjects introduce distinct new features into the code-base independently. A common development work-flow on an ongoing version of a system is the following:

**Work-flow: development without conflicts**

1. Developer $D_1$ starts the current development sprint with an empty local code base $C_1$.
2. She aligns to the current status of the project by initializing her local copy $C_1$ to contain $n$ revisions imported from the central code base $(R_{1,j} = R_{C,j})\ \forall_{j\in\{1..n\}}$.
3. She introduces a new feature by applying changes on top of $R_{1,n}$, creating a new revision $R^M_{1,n+1}$.
4. She makes a submission to the central code-base generating a new revision: $R^M_{1,n+1} \rightarrow C_C = R^M_{C,n+1}$. $R^M_{C,n+1}$ is accepted, because $R_{C,n} = R_{1,n}$.
5. She deletes her local copy of the code-base $C_1$ returning to the initial state. Her work is safely stored in $C_C$.

After such a work-flow, the revision history shown in Figure 6.1 is achieved.

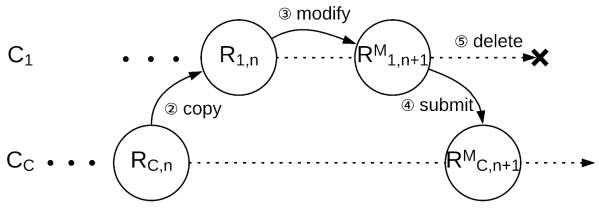



**Figure 6.1:** *Development without conflicts*

When two developers $D_1$ and $D_2$ work in parallel on the same system, the work-flow may look like the following:

**Work-flow: development with conflict**

1. Developers $D_1$ and $D_2$ create their local copy $C_1$ and $C_2$ of the code-base from the current content of the central code base $C_C$, which comprises $n$ revisions; then $\forall_{j\in\{1..n\}} R_{1,j} = R_{C,j} = R_{2,j}$.
2. $D_1$ introduces a new feature by applying changes to $R_{1,n}$, creating a new revision $R^M_{1,n+1}$.
3. At the same time and independently $D_2$ introduces another feature, by applying changes on top of $R_{2,n}$, creating a new revision $R^M_{2,n+1}$.
4. $D_1$ submits $R^M_{1,n+1}$ to the central code-base generating a new revision $R^M_{1,n+1} \rightarrow C_C = R^M_{C,n+1}$. $R^M_{C,n+1}$ is accepted because $R_{C,n} = R_{1,n}$.

5. $D_1$ deletes her local copy of the code-base $C_1$ returning to the initial state. Her work is safely stored in $C_C$.

6. $D_2$ tries to submit $R^M_{2,n+1}$ to the centralized code-base. The operation is rejected because $R^M_{2,n+1}$ creates a conflict with $C_C$.

7. $D_2$ performs conflict resolution between $R^M_{2,n+1}$ and $C_C$: $R^M_{C,n+1} \mapsto C_2$; this step generates a new local revision $R^R_{2,n+2}$, which integrates the feature locally developed by $D_2$ and the current state of the central code base (which, in this case, comprises the feature developed and submitted by $D_1$).

8. $D_2$ submits $R^R_{2,n+2}$ to the centralized code-base $R^R_{2,n+2} \rightarrow C_C$; the submission now is accepted because $R^R_{2,n+2}$ is the result of the conflict resolution $R_{C,n+1} \mapsto C_2$. This step produces a new shared revision $R^R_{C,n+2}$.

9. $D_2$ deletes her local copy of the code-base $C_2$ returning to the initial state. Her work is safely stored in $C_C$.

After such a work-flow, the revision history shown in Figure 6.2 is achieved.

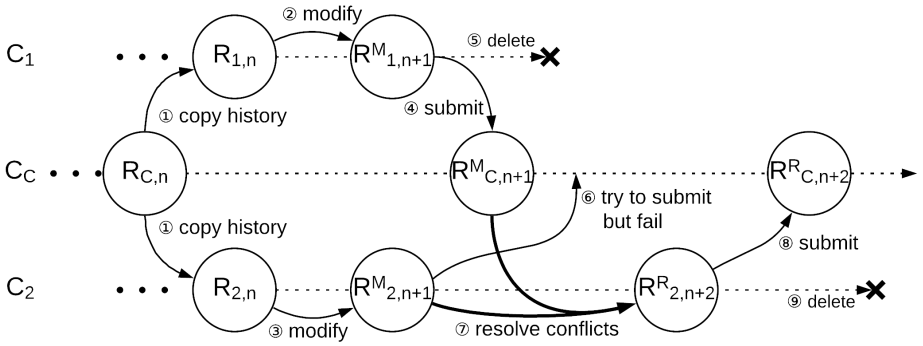



**Figure 6.2:** *Development with conflicts*

This approach can scale to a large number of developers who, before submitting their revision to the central code-base, are responsible to ensure that conflicts are resolved.

### 6.3.2 Model and Code Co-Evolution

In MDD, model-to-text transformations produce textual artifacts (e.g., code, configuration files, documentation, ...) from models. By allowing manual editing on the generated output, it is possible to introduce conflicts with subsequent transformation executions that start from an evolved version of the model. A naïve approach to the resolution of such conflicts is the one realized by the following work-flow:
**Work-flow: development with naïve conflict management**

1. Development starts with a phase of modeling and code generation. Thus, the central code-base initially contains a single revision $R^G_{C,1}$ which is the result of the first execution of the transformation.

2. Developer $D_1$ creates a local copy $C_1$ of the code-base $C_C$, which contains one revision. $R^G_{1,1} = R^G_{C,1}$

3. She introduces a manual change on top of $R^G_{1,1}$ producing a new revision $R^M_{1,2}$;

4. She performs a submission to the centralized code-base $R^M_{1,2} \rightarrow C_C$. The new revision $R^M_{C,2}$. is accepted because $R_{C,1} = R_{1,1}$.

5. She deletes her local copy of the code-base $C_1$ returning to the initial state. Her work is safely stored in $C_C$.

6. She evolves the original model and is ready to execute the transformation again.

7. She executes the transformation and stores the result into an a new empty code-base $C_1$ initializing it with the new purely generated revision $R^G_{1,1}$ (different from $R^G_{C,1}$), which reflects the current state of the model at the code level. Note that manual modifications of the code introduced at step 3 are not reflected in the newly initialized local code-base, because the code generator is blind to manual modifications and would overwrite them anyway.

8. She tries to submit $R^G_{1,1}$ to the centralized code-base. The operation is rejected because $R^G_{1,1}$ is in conflict with $C_C$; rejection occurs due to the way code generation works: $C_1$ contains just one revision, generated from the current model, and is not created by evolving a preceding shared revision stored in the central code base.

9. She solves the conflict between $R^G_{1,1}$ and $C_C$ generating an updated local copy that reconciles the manual changes stored in the central code-base and the new code generated after the model update ($R^M_{C,2} \mapsto C_1 = R^R_{1,2}$).

10. She submits the new revision to the centralized code-base generating a new revision $R^R_{1,2} \rightarrow C_C = R^R_{C,3}$. The submission is accepted because $R^R_{1,2}$ is the result of the conflict resolution $R^M_{C,2} \mapsto C_1$.

11. She deletes her local copy of the code-base $C_1$ returning to the initial state. Her work is safely stored in $C_C$.

After such a work-flow, the revision history shown in Figure 6.3 is achieved.

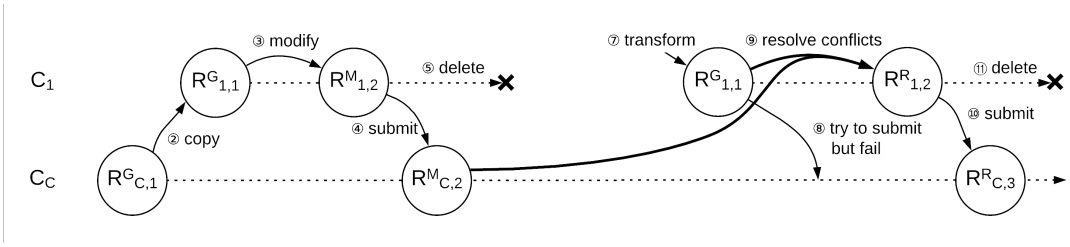



**Figure 6.3:** *Development with naïve conflict management*

### 6.3.3 The Virtual Developer

The developer $D_2$ in Section 6.3.1 works on an outdated version of the code-base (see step 3 and 6 of the parallel work-flow illustrated in figure 6.2), which requires the conflict to be solved. The model-to-text transformation in Section 6.3.2 generates a revision $R^G_{1,1}$ (see step 7 and 8 of the work-flow illustrated in figure 6.3) in conflict with the central code base due to the previous manual update of the generated code. These two scenarios are different with respect to the agent whose changes generate the conflict, but in both cases the evolution of the textual artifact follows a similar path.

In the work-flow of figure 6.3, after a model update, the newly generated revision $R^G_{1,1}$ (created at step 7) is used to initialize the local code-base $C_1$ afresh; this procedure is due to the fact that the model-to-text transformation does not proceed by evolving the code-base incrementally, as real developers do, but always regenerates the whole textual artifact from scratch. At every model transformation step, it would be as if an update on the entire artifact is checked in to the repository regardless of the past history of the code.

To make the behavior of the code generator more similar to that of the human developer, redundant information can be inserted in the code base history, simulating the incremental production of code by the model transformation. Figure 6.4 shows the result of such an approach: the work-flow is equivalent to the one of figure 6.4, but with redundant information; at each model update and code generation step, the developer creates a fresh local copy $C_1$ of the code-base $C_C$, up to the latest generated revision $R^G_{1,1} = R^G_{C,1}$, before overwriting it with the new generated revision $R^G_{1,2}$.

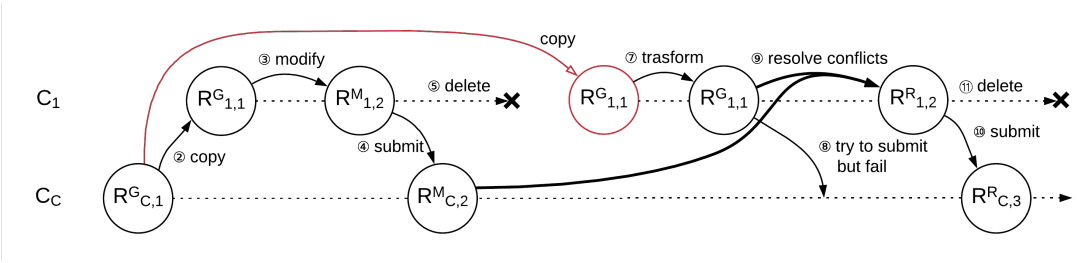



**Figure 6.4:** *Work-flow with redundant information to compute the delta after model update and code generation*

The initialization of the local code base with the latest generated revision before a code generation step makes it possible to compute the delta $\Delta^G_{1,2}$ introduced by $R^G_{1,2}$. Even though the model-to-text transformation overwrites the entire artifact, $\Delta^G_{1,2}$ singles out exactly the *incremental* changes that the model-to-text transformation induces over the previously generated revision $R^G_{1,1}$ after the model update. Besides the difference in the initialization of the code base, the manual and automatic generation of the artifacts now follow the same path, as visible by comparing the work-flow of figure 6.2 and figure 6.4. Revisions $R^M_{2,n+1}$, of figure 6.2, and $R^G_{1,2}$, of figure 6.4, are comparable: both generate a delta ($\Delta^M_{2,n+1}$ and $\Delta^G_{1,2}$ respectively), which embodies the changes w.r.t. a preceding revision ($R^M_{2,n}$ and $R^G_{1,1}$) in an outdated code-base ($C_2$ and $C_1$ respectively) generating a conflict with $C_C$.

Given such similarity, a model-to-text transformation can be considered as a *Virtual Developer*, who always generates a conflict with the shared code base $C_C$. Treating

the model-to-text transformation as an additional developer potentially simplifies the management of manual updates in the forward engineering MDD life cycle. Tools and methodologies that are normally applied for conflict resolution among developers can be applied to both human and virtual developers. Multiple human developers and a single Virtual developer can work in parallel, in a work-flow such as the following:

**Work-flow: the Virtual Developer**

1. $C_C$ contains $n$ revisions, the latest revision $R^G_{C,n}$ is generated.
2. A human developer $D_H$ creates a local copy $C_H$ of the code-base $C_C$. $\forall_{j\in\{1..n\}}(R_{H,j} = R_{C,j})$
3. $D_H$ introduces a new feature by applying changes to $R^G_{H,n}$, creating a new revision $R^M_{H,n+1}$.
4. $D_H$ submits the new revision to the centralized code-base generating a new revision $R^M_{H,n+1} \rightarrow C_C = R^M_{C,n+1}$. The submission is accepted because $R_{C,n} = R_{H,n}$.
5. $D_H$ deletes her local copy of the code-base $C_H$ returning to the initial state. Her work is safely stored in $C_C$.
6. The model is updated and the transformation is ready to be executed.
7. The Virtual Developer $D_V$ creates a local copy $C_V$ of the code-base $C_C$ *up to the latest generated revision*: $\forall_{j\in\{1..k\}}(R_{V,j} = R_{C,j})$, where $k$ is the index of the last *generated* revision ($k = n$ in the current example). In this way, the Virtual Developer aligns its local code base to the status that reflects the previous version of the model.
8. $D_V$ executes the transformation and stores the result into $C_V$ generating a new revision $R^G_{V,n+1}$, which is a replacement of the entire textual artifact.
9. $D_V$ tries to submit $R^G_{V,n+1}$ to the centralized code-base. The operation is rejected because $R^G_{V,n}$ is not equivalent to $R^M_{C,n+1}$, due to the intervening manual update of $D_H$.
10. $D_V$ solves the conflict between $R^M_{C,n+1}$ and $C_V$ generating $R^M_{C,n+1} \mapsto C_V = R^R_{V,n+2}$. In this step, the $\Delta^G_{V,n+1}$ is used to identify the modifications introduced by the latest round of code generation, which simplifies, and even automates in some cases, the identification and resolution of collisions with the manual modifications of the code, as explained in Section 6.4
11. In order to safely store the latest generated revision in the central code base for future alignment, $D_V$ *forces* the submission of $R^G_{V,n+1}$ to $C_C$, generating $R^G_{C,n+2}$.
12. $D_V$ submits the conflict resolution $R^R_{V,n+2}$ to the central code-base, generating a new shared revision $R^R_{V,n+2} \rightarrow C_C = R^R_{C,n+3}$. The submission is accepted because $R^G_{C,n+2}$ is equivalent to $R^G_{V,n+1}$. It is important to notice that $\Delta_{C,n+3}$ is identical to $\Delta_{V,n+2}$, due to the previous forced submission, which saved in the central code base also the latest generated revision.

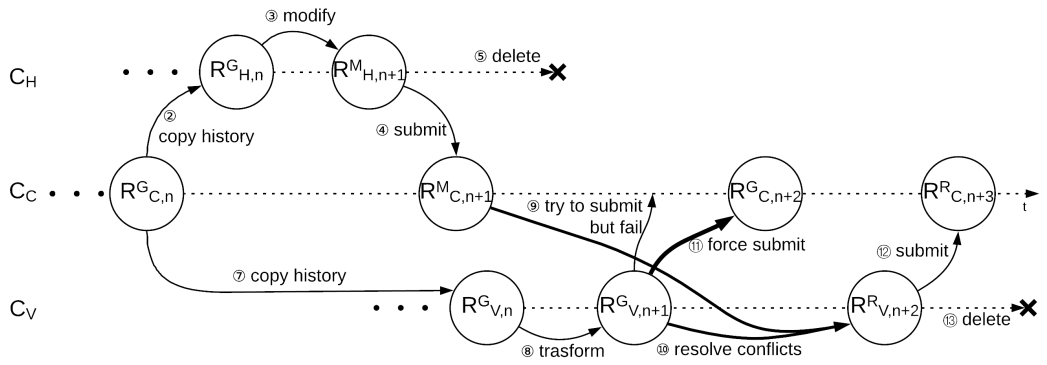



**Figure 6.5:** *Work-flow with the Virtual Developer; the revision history contains redundant information.*

13. $D_V$ deletes its local copy of the code-base $C_V$ returning to the initial state. Its work is safely stored in $C_C$.

Figure 6.5 shows the revision history that results from considering the model-to-text transformation as a Virtual Developer. The central code-base now contains a twofold sequence of revisions: automatically generated ($R^G_{C,j}$) and conflict-resolved ($R^R_{C,j+1}$). The human developer always updates the latest revision, whereas the Virtual Developer is always out-of-sync and applies changes on the latest *generated* revision $R^G_{C,j}$.

The organization of the work-flow of human and virtual developers described so far relies on three assumptions:

1. The work-flow contains at least one manual update by the human developer.
2. The work-flow contains at most one Virtual Developer. Parallel model updates and executions of the model-to-text transformations by multiple virtual developers require conflict resolution also between generated revisions; after conflict resolution, the i-th Virtual Developer would see a local code base containing a revision that mixes changes introduced by both human and other virtual developers. Parallel model updates would introduce different versions of the model at the same time. To cope with such a situation, the i-th Virtual Developer should check the central code base for the existence of a generated revision successive the one used to initialize its local code base at the previous code generation step. In this case, it should restart its work-flow assuming the last shared generated revision $R^G_{C,n+k}$ as the starting point.
3. The conflict resolution step, such as step 10 of the work-flow in figure 6.5 is executed atomically, i.e., no manual revision intervene in $C_C$ during a conflict resolution step. This assumption can be relaxed by starting a new conflict resolution step if a new manual revision is submitted to the central code-base in parallel to a conflict resolution step.

## 6.4 Automating conflict resolution

A key aspect of the work-flow proposed in Section 6.3.3 is conflict resolution. Based on the definition of conflict introduced in Section 6.3, conflict resolution is any automated,

semi-automatic, or manual procedure that takes in input the history of the code-base $C_C$ up to latest manual revision and of $C_V$ up to the latest generated revision and produces a new revision that integrates the newly generated version with all the previous manual modifications.

Purely manual approaches to the resolution of conflicts are impractical, as they cannot scale with the size of the code-base, requiring manual inspection of the entire artifact, or of a large portion thereof, after each model-to-text transformation. In this Section, first we show how the proposed work-flow can accommodate, with appropriate conflict resolution procedures, industry-standard approaches such as template-based forward engineering and protected areas; then, we show how the revision deltas can be exploited to relax the constraints of code generation with template-based transformations and with protected areas.

### 6.4.1 Template-based forward engineering

Template-based forward engineering incorporates all possible customizations of the output in templates, which the generator exploits to produce the code. No manual modifications on the generated code are allowed; if the application requires specific features not captured by the abstract model and by the current code generation rules, these must be provided to the generator in the form of new templates.

In a template-based work-flow, after each model change, the entire artifact is regenerated and used to replace the previous version: the revision produced by the conflict resolution $R^M_{C,n+1} \mapsto C_V = R^R_{V,n+2}$ is always equivalent to $R^G_{V,n+1}$, i.e., manual changes introduced in the code-base $C_C$ are ignored.

While conflict resolution can be completely automated and no human intervention is required, the price to pay is the need of creating and maintaining templates, a task that requires a kind of meta-programming, i.e., the programming of partial examples and code skeletons that the generator must fill-in to produce the executable code. This effort requires non-standard programming skills and is tied to the specificities of the MDD tool.

### 6.4.2 Protected areas

In template-based forward engineering, manual modifications are lifted at the level of code generation rules and the update of the generated code is disallowed. Instead, with protected areas, manual changes are confined to special portions of the project. Protected areas are defined with tool-specific techniques, compatible with the target language, to single out regions of the code that the generator cannot overwrite. With protected areas, conflict resolution can be automated, by using the content of revision $R^G_{V,n+1}$ as the base artifact to build $R^R_{V,n+2}$. Each area is identified by a unique signature, preserved between successive executions of the model-to-text transformation. If an area is present in both $R^M_{C,n+1}$ and $R^G_{V,n+1}$, its content is extracted from the central code base $R^M_{C,n+1}$ and inserted into the matching protected area of the local revision $R^R_{V,n+2}$, possibly replacing the generated code. If an area is present in $R^M_{C,n+1}$, but not in $R^G_{V,n+1}$, e.g., due to the cancellation of the part of the model that contained it, its content is not reinstalled in $R^G_{V,n+2}$.

With protected areas, no back-porting of features into templates is needed; but the

aspects where human intervention is required must be effectively separable from the rest of the code, an assumption that is hardly verified in modern web and mobile applications, where the presentation is still entangled with the structure of the front-end and new non-functional requirements are frequently introduced during the project lifetime.

Another assumption of protected areas is the possibility of storing the signature of an area inside the code. Various tools exploit features of the target language, such as comments. The following example shows the use of comments to delimit the portion of the code inserted by the developer. Such language features are generally available,

```
/**
* @param driver
* @return
*/
public Boolean canDrive(Person driver) {
  // Start of user code for operation canDrive
  // TODO should be implemented
  return null;
  // End of user code
}
```

```
/**
* @param driver
* @return
*/
public Boolean canDrive(Person driver) {
  // Start of user code for operation canDrive
  return driver.hasLicence();
  // End of user code
}
```

but this assumption makes protected areas language-dependent.

### 6.4.3 Exploiting deltas

The focus of this chapter is to relax the assumptions of MDD based on templates and protected areas, to support the resolution of conflicts between manually and automatically produced code in a broader spectrum of situations. By exploiting the redundant information stored in the history of revisions, it is possible to integrate manual changes in the MDD loop, limiting human intervention just to those cases in which a conflict resolution strategy cannot be inferred.

At any given point in the history of a project, the shared code-base $C_C$ and the local code base of the Virtual Developer $C_V$ may have diverged, due to the introduction of updates by human developers; however, thanks to the way in which the work-flow is managed, the two code-bases have an equivalent history up to the revisions $R^G_{C,n}$ and $R^G_{V,n}$ (see step 7 of the work-flow depicted in Figure 6.5). These revisions contain the *latest shared* output of the model-to-text transformation.

After such "synchronization point", the centrally shared delta $\Delta^M_{C,n+1}$ contains all the changes introduced by human developers posterior to $R^G_{C,n}$ and the local delta

$\Delta^G_{V,n+1}$ contains all the changes introduced by the Virtual Developer in the last (yet not shared) execution of the model-to-text transformation. It is precisely in those two deltas that the interference between the automatically generated and the manually written code must be identified and resolved.

To support the resolution of a conflict, each delta must be decomposed into the individual changes it contains. The following definitions characterize the content of a delta:

- **Line**: a tuple consisting of a unique identifier, a content and a position.
- **Artifact**: an ordered set of lines.
- **Atomic update**: an elementary modification of the artifact. Allowed atomic updates are:
  1. Insertion: creation of a new line with given content at a position successive to an existing line or at the beginning of the artifact. The position of the lines located after the new line are incremented.
  2. Deletion: the removal of a line, with the consequent decrement of all the lines positioned after it.
- **Update**: a set of atomic updates affecting distinct lines at consecutive positions.
- **Collision**: two updates $u_i$ and $u_j$ are in collision if there exist an atomic update $a_i \in u_i$ and an atomic update $a_j \in u_j$ affecting the same line.
- **Update Graph**: an undirected graph (henceforth denoted by $U$) in which vertexes are updates and edges, if present, denote the collision between them.
- **Collision Group**: a collision group is a connected component of the update graph having size greater than one, i.e., comprising at least one collision. An update graph with no collision groups is called a **Collision-free graph** (henceforth denoted by $U_F$); the updates contained in a collision-free graph can be applied independently. An update graph with at least one collision group is called a **Colliding graph** (henceforth denoted by $U_C$); the updates contained in a colliding graph cannot be applied independently, requiring collision resolution.
- **Collision Resolution**: a procedure that takes in input a colliding graph $U_C$ and produces in output a new collision free graph $U_F$.

Conflict resolution can be achieved with the following steps:

1. $\Delta^M_{C,n+1}$ is decomposed into its constituent (collision-free) updates $U^M_F$.
2. $\Delta^G_{V,n+1}$ is decomposed into its constituent (collision-free) updates $U^G_F$.
3. The conflict resolution update graph $U^R$ is formed, which contains both the updates of both $U^M_F$ and $U^G_F$.
4. If $U^R$ is a collision free graph, we can define $U^R_F = U^R$. Otherwise, if $U^R$ is a colliding graph, collision resolution is applied, so to create a new collision-free graph $U^R_F$.

5. The updates in $U_F^R$, which are guaranteed non to collide, are applied to $R_{V,n+1}^G$ generating $R_{V,n+2}^R$.

The above mentioned steps are supported in all industry standard VCSs: they automatically identify the set $U$, by decomposing $\Delta_{C,n+1}^M$ and $\Delta_{V,n+1}^G$, automatically produce $R_{V,n+2}^R$, if no collision groups are identified, or support the developer during collision resolution, if one or more collision groups are identified.

Different VCSs, or the same VCS with different configurations, may decompose $\Delta_{C,n+1}^M$ and $\Delta_{V,n+1}^G$ into different, yet equivalent, update sets.

## 6.5 Guidelines for the design of transformation rules

The effort of human intervention in the conflict resolution phase is proportional to the number of collision groups. Supposing that all the changes made by the human developers are needed, the delta $\Delta_{C,n+1}^M$ cannot be reduced. The only way to reduce human intervention during conflicts resolution is to reduce the number of collision groups. This goal can be accomplished in two ways:

- By enforcing separation of concerns between human and virtual developers at file and even at line level.
- By decreasing the size of $\Delta_{V,n+1}^G$ improving the design of model-to-text transformations.

As an aside, we present a use-case for collisions, which makes them a useful addition to the work-flow, as guards against changes with undesired collateral effects.

### 6.5.1 Separation of concerns

The possibility of identifying lines, areas, or entire files that are pure responsibility of the Virtual Developer or of the human developers can reduce the number of colliding updates.

Parts of the artifact that are meant to be edited only by human developers may be generated by the model-to-text transformation, e.g., as code skeletons to enable the execution of a prototype. Such mock-ups should be generated following a fixed and recognizable template, so that the changes to their content appears only in manual revisions $\Delta_{C,n+1}^M$. They are introduced and deleted in generated revisions and edited only in manual revisions, thus the resolution of the conflict they produce can be automated.

Separation of concerns can be pushed to the level of groups of lines or even of individual lines. Lines that are responsibility of both the human and the virtual developer should be split, exploiting language invariance rules. Line splitting is particularly relevant in the MDD of Web and mobile GUIs, based on HTML and CSS. HTML whitespace invariability can be exploited to achieve line-level separation of concerns. HTML tags can be split from a single line to multiple lines containing different attributes. At-

```
1 <span class="title" data-bind="text: item()['content']" ></span>
```

tributes related to styling, which is typically not modeled and thus responsibility of the

human developer, can be separated from attributes related to functional aspects or to data binding, which are typically modeled and thus responsibility of the Virtual Developer.

For example, the following HTML fragment could be generated to display a piece of content.

```
<span class="title"
   data-bind="text: item()['content']" >
</span>
```

To improve the user interface, the human developer may change the class attribute, e.g., adding a custom class. After a change in the model, the Virtual Developer regen-

```
<span class="title beautiful"
   data-bind="text: item()['content']">
</span>
```

erates the code. For example, the data-bind attribute could be updated in order to add custom behavior in response to a user click on the span.

```
<span class="title"
   data-bind="text: item()['content'], click: $parent.select">
</span>
```

When the two changes are merged, updates on the class and on the data-bind attribute are separated, collisions are prevented, and no human intervention is required for conflict resolution.

### 6.5.2 Reducing the size of changes

Reducing the number of lines modified by the code generator decreases the likelihood of collisions. Simple design guidelines help building model-to-text transformations that do not introduce unnecessary changes.

- *Avoiding non-determinism*: the transformation should produce always the same output (source code) from a given input (the model of the system); non-determinism in the code generation may produce (unnecessary) inconsistencies even between the code generated by two runs of the transformation on the same input model. Non-deterministic effects typically stem from the non-deterministic iteration over model elements during code generation. A simple design rule is to make the transformation exploit a fixed criterion for model navigation, even if such criterion is not part of the modeling language definition. For example a model-to-text transformation generating code from an IFML DataFlow model element can generate different artifacts depending on the order in which the parameter bindings are traversed. For example, the following JavaScript code could be produced from an IFML data flow associated with two parameter bindings denoting the title and the author of a song. If the order in which the parameter binding sub-elements of the

```
<span class="title beautiful"
   data-bind="text: item()['content'], click: $parent.select">
</span>
```

```
var packet = {
   'title' : data['song'],
   'author' : data['author_name']
};
```

IFML data flow are traversed is not deterministic, the model-to-text transformation can produce inconsistencies even between the code generated by two runs of the transformation on the same input model.

```
var packet = {
   'author' : data['author_name'],
   'title' : data['song']
};
```

- *Avoiding dependency on non-semantic model features*. Non-semantic aspects of a model may comprise the internal IDs of the model elements, their position in the model layout, etc; they may produce different outputs (source code) from input models that are syntactically or visually different, but semantically equivalent. For example, the model-to-text transformation could navigate the IFML ViewComponents comprised within a ViewContainer element depending on the order or position in which they are inserted in the diagram, which has no meaning in the language. Supposing that the *Question* ViewComponent "comes before" the *Answers* ViewComponent within the *Card* ViewContainer (see the IFML model of figure 6.6), the generated code would look like the following fragment. If the order of the two ViewComponents is reversed, the IFML model has the same meaning but the model-to-text transformation could produce different output.

### 6.5.3 Collisions that help

Not all collisions are harmful; some collisions may actually be artificially introduced as guards against model changes with impacts on the manually written implementation code. For example, the model may specify an abstract operation, listing its input and output parameters; this specification dictates the prototype of the concrete implementation written by the human developer. In this case, the model-to-text transformation can enrich with a comment the generated code, as shown below, specifying the expected inputs and output of an abstract operation, which is implemented by the human developer.

By deleting such comments, the human developer introduces in its delta an update that does not generate a collision with the delta of the code generator, which ignores the human modifications and repeatedly generates the same comment lines and thus does not introduce updates on them. Conversely, when the interface of the abstract operation

```
<span>
    <details-question params="context: context"></details-question>
    <list-answers params="context: context"></list-answers>
</span>
```

```
<span>
    <list-answers params="context: context"></list-answers>
    <details-question params="context: context"></details-question>
</span>
```

is altered in the model, the delta introduced by the model-to-text transformation contains a new update, which collides with the manual modification made by the human developer, prompting her to update the code implementing the operation.

This approach can be considered as a kind of *conditional protected area*: manually written code is preserved between generations while a condition persists; when the condition fails, due to a model update, the work-flow calls for manual intervention.

## 6.6 Implementation

The approach to MDD described in the previous sections has been implemented and evaluated with two different development environments: WebRatio [32], a product for the development of industrial Web and mobile application, and IFMLEdit.org [44], an open-source, on-line environment for the rapid prototyping of Web and mobile applications.

Given the open source nature of IFMLEdit.org, which enables the replacement of the code generator, this tool has been also used for the evaluation of the proposed model transformation design guidelines.

### 6.6.1 IFMLEdit.org

IFMLEdit.org (presented in Chapter 5) is an online environment for the specification of IFML models and the generation of rapid prototypes of Web and mobile architectures. In IFMLEdit.org, the model of the front-end is defined with IFML, the domain model

```
function run(parameters) {
  // REMOVE THESE LINES START
    // Inputs:
    // - username
    // - password
    // Outputs:
    // - user_id
    // REMOVE THESE LINES END

  return {
    event: "event-id",
        data: {}
  };
}
```

is inferred from the IFML diagram, and actions are treated as abstract black-boxes. The tool supports the generation of prototypes, which can be evaluated directly in the tool, via in-browser emulation, or downloaded as executable source code. The automatically generated prototype is turned into a real application, by customizing the look & feel and by replacing the mock-up data access and skeleton APIs calls with use-case specific implementations.

#### The code generation framework

IFMLEdit.org is built on top of the ALMOsT.js (presented in Chapter 3) transformation framework, which allows the developer to specify model transformations with a simple rule-based extension of JavaScript. ALMOsT.js supports the construction of template-based code generators, via the integration of templating languages, such as EJS [14]. A rule identifies a specific sub-structure in the input model graph (e.g., single model elements, specific connections between elements, or global information) and generates code from one or more templates. In the following example, the rule checks if the IFML element is an Event and, in that case, calls an EJS template that generates the corresponding HTML code.

```
createRule(
  // Activation Function
  function (element, model) { return model.isEvent(element) },
  // Rule Body
  function (event, model) {
      var template = require('./templates/system-event-v.html.ejs');
    return {
      name: 'index.html',
            content: template({event: event})
    };
  }
)
```

#### Structure of the generated code

IFMLEdit.org features a Web and a mobile code generator, both based on HTML, CSS and JavaScript. The former produces a client-side application, which can be connected to any preexisting REST service back-end. The latter produces a Cordova [9] application. The generated JavaScript code follows the CommonJS [8] specification, an approach to code modularization that, together with Asynchronous Module Definition (AMD) [6], inspired the standardization of modules in ECMAScript 6 [13]. The generated HTML code is separated into components following the Web Components [31] specification, an approach which inspired custom components in HTML5 [139].

### 6.6.2 WebRatio

WebRatio is an Eclipse-based commercial tool for the development of Web and mobile applications. In WebRatio, the user interaction is defined via IFML, the domain model via UML Class Diagrams (or Entity-Relationship) and the business logic via a proprietary Action Definition Language inspired by WebML [60]. The tool enables the

generation of the full code of the application at each model iteration, with a forward-engineering approach. Details that cannot be modeled directly (e.g., UI look & feel, application specific operations that implement IFML abstract Actions) are incorporated using a template-based approach.

#### The code generation framework

The generated textual artifact can be customized via templates, programmed with the Groovy scripting language. The code generator tackles templates selection thanks to an extension of IFML, which allows the developer to tag each IFML element with the custom template to use for code generation.

#### Structure of the generated code

WebRatio features both a Web and a mobile code generator. The former produces an application based on Java, JSP, HTML, CSS and JavaScript. The latter produces a Cordova [9] application based on HTML, CSS and JavaScript. In both cases, the generated code exploits APIs exposed by a custom runtime framework, which simplifies the mapping between the IFML concepts and the runtime components that implement them.

### 6.6.3 VCS integration

After a model update, the developer has to follow the work-flow described in Section 6.3 to produce a revision that incorporates the newly generated code and all the previous manual changes. As a reference implementation for the proposed work-flow and conflict resolution techniques, we have developed ALMOsT-Git [4], a middleware system that exposes the work-flow of the Virtual Developer and realizes its macro-operations (such as code base initialization, submission, and conflict resolution), masking their low-level implementation in the Git [16] VCS. In ALMOsT-Git the high-level concepts introduced in Section 6.3 and 6.4.3 are mapped to concrete Git primitives as follow:

- *code-base*s are mapped to Git *branches*, i.e., parallel histories inside a single Git repository. This enables the proposed work-flow to be followed by groups of developers sharing a central Git repository and also by individual developers identifying a branch in their repository as the centralized code-base.
- *revision*s are mapped to Git *commit*s.
- the act of copying the central code-base is mapped to the *clone* or *branch* Git operations, depending on the location of the central code-base; the former in the case of a centralized repository, the latter in the case of a local branch.
- *submission* is mapped to the Git *push* operation, which copies commits from the current branch to another local or remote one. As in the proposed work-flow, the *push* operation may fail if the latest commit in the remote branch (i.e., the *HEAD* of the branch) is not identified in the local branch.
- *collision resolution* is mapped to the *pull* operation in Git, which can be configured to try and resolve the conflict between the two branches and, in case of collisions, to stop and ask the developer to solve them.

ALMOsT-Git, given as input the location of the central repository and of the latest generated artifact, implements the work-flow of Section 6.3. It stops in case of collisions, highlighting the affected lines in the source code and asking the developer to solve them. When collision resolution is completed, the developer restarts the tool, which resumes and completes the work-flow.

### 6.6.4 Case studies

The model and text co-evolution work-flow and the collision prevention guidelines have been evaluated during the development of two use-case applications.

Both applications are sufficiently limited in size to be fully described, but feature non trivial requirements: access to remote resources, use of the multimedia and hardware capabilities of the device, and a customized presentation.

#### A Quiz Game

The use-case application is a mobile phone Quiz Game, designed to extend the play of a real card game. The application requires interaction with the device camera, to scan a QR code on a card of the real game, and with remote resources accessed through the network, to download the questions and the correct answers.

The development, tracked in a public repository[1], followed four sprints, each one addressing new functionalities and/or non-functional requirements:

1. **Proof of Concept**. The first version allows the user to scan a QR code on a special card of the real game, as required by the game rules. After scanning the QR code, the game proposes the player a multi-choice question on a subject related to the card (in the real game, water and energy conservation are the subjects of the cards). Once the user selects an option, the application shows the correct answer and returns to the initial state.

2. **Styling and Explanation**. A custom presentation style is applied to the user interface, to give the application a professional look&feel. The application allows the user to investigate the explanation relative to the correct answer for each proposed question.

3. **Secondary Game Mechanics**. While the previous game mechanics remains available, a new one is introduced whereby the player can skip the QR scanning and play in standalone mode, receiving a series of question of increasing difficulty. The skill level of the player is tracked and questions are adjusted accordingly.

4. **Internationalization**. The game mechanics is kept unchanged, but the ability to select the language is introduced.

The final IFML model for the application is visible in Figure 6.6. Figure 6.7 shows the difference between the default UI generated by IFMLEdit.org and the final result. This application has been released as part of the card game Funergy[2].

[1] `http://github.com/emanuele-falzone/almostjs-git-demo-game`
[2] `http://play.google.com/store/apps/details?id=com.eu.funergy`

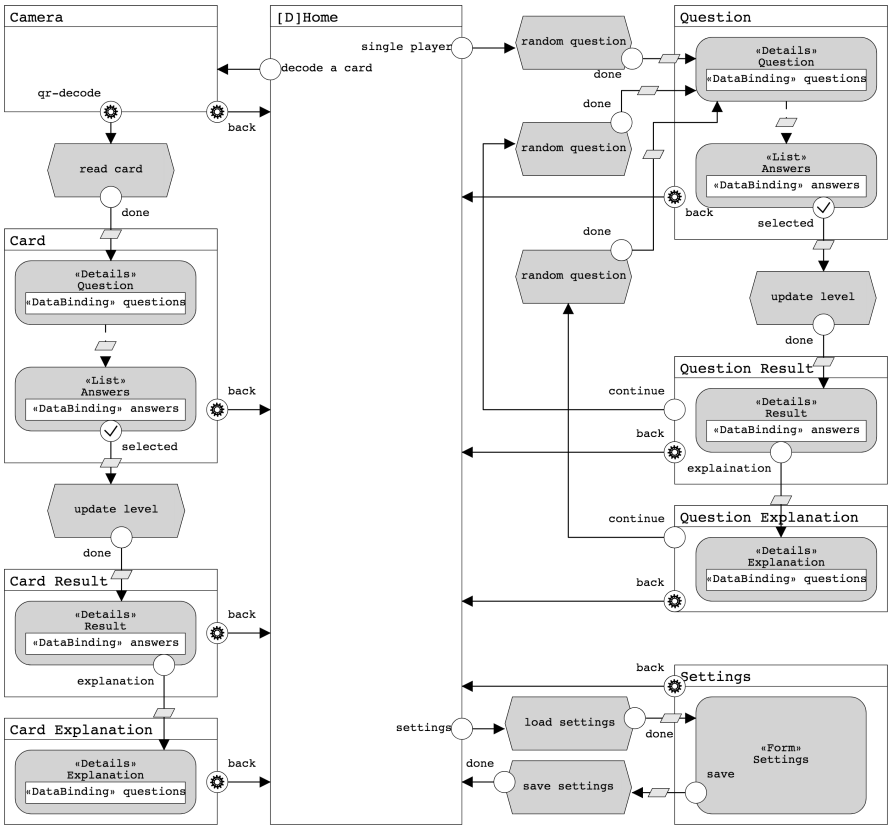



**Figure 6.6:** *Quiz Game: IFML model*

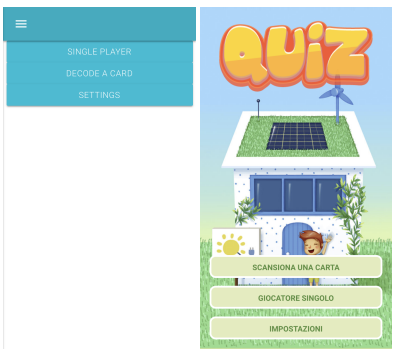



**Figure 6.7:** *Quiz Game: from prototype to final product*

**Media Player**

The second use-case application is a web-based Media Player, requiring interaction with the media capabilities of the device and the access to remote resources over the network.

The development, tracked on a public repository[3], followed 4 steps:

1. **Proof of Concept**. The application loads a list of songs, allows the user to select the song to play, and to pause/restart the currently playing song.
2. **Styling**. A custom presentation style is applied to the user interface and a system event is added to catch the end of a song.
3. **Songs Filtering**. An author filter is added, whereby the user can filter the song list to simplify selection.
4. **Song Cover**. The interface of the application is enhanced by showing the cover of the song currently playing.

The final IFML model for the application is visible in Figure 6.8. Figure 6.9 shows the

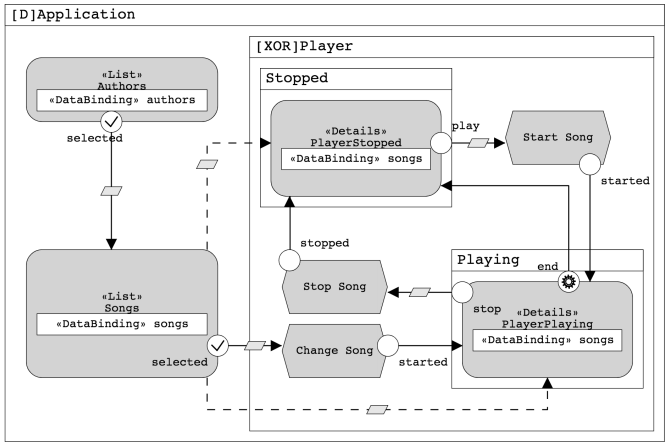



**Figure 6.8:** *Media Player: IFML model*

difference between the default UI generated by IFMLEdit.org and the final result.

## 6.7 Evaluation

In this section we compare the model and code co-evolution approach with classical template-based forward engineering, in both WebRatio and IFMLEdit.org; we also assess the effect of the transformation design guidelines proposed in Section 6.5, by comparing the original code generator of IFMLEdit.org with a new version that implements conflict prevention.

### 6.7.1 Template-based forward engineering vs Model & code co-evolution

The use-case applications have been developed using WebRatio and IFMLEdit.org, with both model and code co-evolution and the template-based forward engineering

[3] `http://github.com/emanuele-falzone/almostjs-git-media-player`

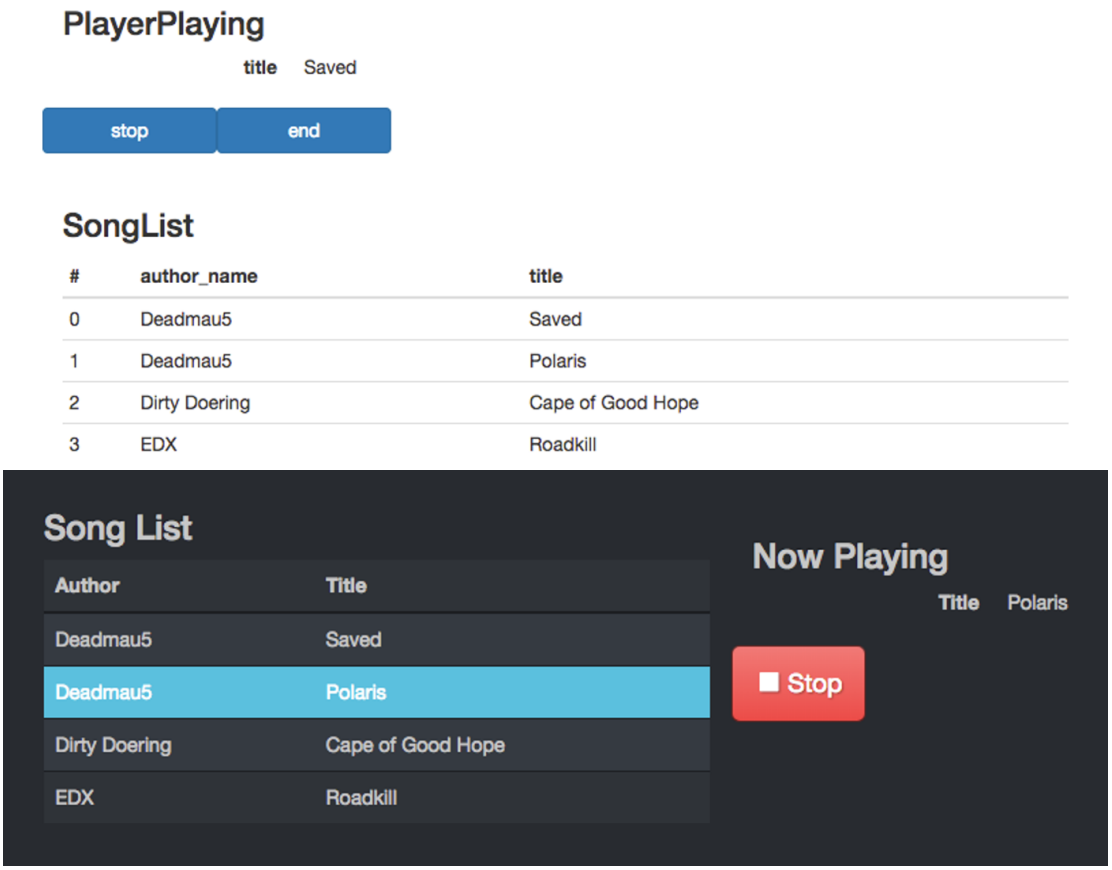



**Figure 6.9:** *Media Player: from prototype to final product*

processes. The two processes share the phases of model editing and feature development: in the former, the developer creates the model of the application and generates the code; in the latter, she programs the additional features that cannot be automatically generated from the model. The two processes diverge in the way in which the manually developed features are incorporated in the MDD loop, and in the effort required for this task. The template-based process requires the *back-porting* of the manually programmed features into the templates of the code generator, so that such features are automatically reproduced in the subsequent code generation steps. The model and code co-evolution process does not require the creation of additional artifacts: manually developed features are added on top of the generated code and automatically "dragged" from one version to the next; the developer must work only on the parts of the code that produce collisions. In other terms, the work to integrate manually developed features in the template-based forward engineering process is proportional to the code-level size of the whole feature, whereas in the model and code co-evolution process it is proportional to the size of the part of the code where the automatically generated and the manually programmed code interfere: if the manually programmed feature is perfectly isolated from the automatically generated code, the integration work is null.

The tables in the rest of this Section compare the extra work needed in the two processes for integrating manual features in the MDD loop, in two applications and with two MDD development environments. They show both the manually developed features (quantified in terms of the updates needed to build them and of the lines of code affected) and the integration work performed in the two processes. The integration work is evaluated as follows:

- In template-based forward engineering, we evaluate the effort of extending the existing templates of the MDD environment or of creating new ones, to back-port code-level features into the code generator; we compute the number of macro-updates (e.g., the implementation of an IFML abstract operation or of the services for data access, the modification of the visual style of an interface element, etc.) required to obtain the templates and the number of lines deleted or inserted to perform such interventions.
- In model and code co-evolution, we evaluate the effort required for collision resolution, quantified by the number of collision groups, i.e., the number of developer's macro-interventions needed to perform conflict resolution, and by the number of atomic updates involved in the resolution of the collision groups, i.e., the number of lines requiring manual revision.

In counting updates and the affected lines, the following rules have been followed:

- When a new template is produced by modifying a copy of an existing one, we only consider the lines that have been actually modified, not to penalize the template-based approach.
- When a code-level feature occurs multiple times in the project (e.g., the same data update operation), its implementation is counted only once in the template-based forward engineering approach, because we assume that the developer implements the feature, back-ports it into a template, and then generates the other occurrences of the same features automatically, reusing the template.

**IFMLEdit.org**

Table 6.1 and 6.2 show the results of comparing the template-based and the model and code co-evolution processes, implemented with the original version of the IFMLEdit.org model editor and code generator.

Both the distribution and the amount of work differ in the two processes. The distribution of effort varies because the template-based process requires that customizations are performed *during the implementation of a sprint*, in order to generate the fully functional code; conversely, in the model and code co-evolution process, updates are applied to the generated code and thus the collisions are resolved *at the next sprint*, when the code generator is invoked again. This explains why the Proof of concept sprints have zero (collision detection) effort.

The **integration** work due to collision resolution is less than 20% of that required by back-porting of features into templates: the difference in the number of updates is 83% for the Quiz Game and 84% for the Media Player; the difference in the number of lines affected is 84% for the Quiz Game and 91% for the Media Player. To put such difference in context, Table 6.1 and 6.2 show also the total application **development** work: the model and code co-evolution process required less than 75% of the effort required by the template-based forward engineering process: the number of updates required was reduced by 27% in the Quiz Game and by 28% in the Media Player, while the number of lines of code was reduced by 28% in the Quiz Game and by 27% in the Media Player.

| | Model & Code Co-evolution | | | | Template-based fw engineering | | | |
|---|---|---|---|---|---|---|---|---|
| | Feature Devel. | | Conflict res. | | Feature Devel. | | Back-porting | |
| Sprint | Up. | At. Up. | Up. | At. Up. | Up. | At. Up. | Up. | At. Up. |
| Proof of Concept | 27 | 1154 | 0 | 0 | 28 | 1110 | 27 | 1102 |
| Styling and Expl. | 9 | 111 | 4 | 77 | 6 | 50 | 6 | 66 |
| $2^{nd}$ Mechanics | 29 | 492 | 3 | 20 | 18 | 255 | 15 | 127 |
| I18n | 33 | 237 | 5 | 139 | 27 | 181 | 24 | 179 |
| Total | 98 | 1994 | 12 | 236 | 79 | 1596 | 72 | 1474 |
| | Total effort (devel+integration) | | | | Total effort (devel+integration) | | | |
| | Updates | | Atomic updates | | Updates | | Atomic updates | |
| | 110 | | 2230 | | 151 | | 3070 | |

**Table 6.1:** *Quiz Game: development statistics with IFMLEdit.org*

The difference in the total development effort is due to the overhead of template programming, which requires extra code for integrating the feature into the code generator. This is especially true in the development of GUI templates, where achieving the desired visual effect is easier in the actual implementation code than in the "meta-level" code of the template. Note that the extra work of template-based forward engineering may be paid back, if the developed templates are reusable in other projects.

The model and code co-evolution process is amenable to further improvement. Table 6.1 and 6.2 show that conflict resolution is accountable for 11% of the total application development effort (10.9% for the Quiz Game and 11.4% for the Media Player). We were able to identify various collisions that cold be avoided; to do so, we revised the original code generator of IFMLEdit.org following the simple collision prevention

| Sprint | Model & Code Co-Evolution | | | | Template-based fw engineering | | | |
|---|---|---|---|---|---|---|---|---|
| | Feature Devel. | | Conflict res. | | Feature Devel. | | Back-porting | |
| | Up. | At. Up. | Up. | At. Up. | Up. | At. Up. | Up. | At. Up. |
| Proof of Concept | 16 | 144 | 0 | 0 | 12 | 100 | 15 | 148 |
| Styling | 5 | 32 | 1 | 3 | 5 | 42 | 5 | 32 |
| Songs Filtering | 4 | 47 | 1 | 4 | 3 | 17 | 4 | 16 |
| Songs Cover | 6 | 34 | 2 | 10 | 3 | 14 | 2 | 9 |
| Total | 31 | 257 | 4 | 17 | 23 | 173 | 26 | 205 |
| | Total effort (devel+integration) | | | | Total effort (devel+integration) | | | |
| | Updates | | Atomic updates | | Updates | | Atomic updates | |
| | 35 | | 274 | | 49 | | 378 | |

**Table 6.2:** *Media Player: development statistics with IFMLEdit.org*

guidelines discussed in Section 6.5. Such (avoidable) collision are responsible for the majority of the updates (and affected lines) involved in collision resolution. The evaluation of the improved IFMLEdit.org code generator is discussed in Section 6.7.2.

#### WebRatio

Table 6.3 and 6.4 contrast the amount of work required by back-porting and collision resolution. The **integration** work due to collision resolution is less than 20% of that required by back-porting of features into templates: the difference in the number of updates is 100% for the Quiz Game and 83% for the Media Player; the difference in the number of lines affected is 100% for the Quiz Game and 96% for the Media Player. To put such difference in context, Table 6.3 and 6.4 show also the total application **development** work: the model and code co-evolution process required less than 65% of the effort required by the template-based forward engineering process: the number of updates required was reduced by 45% in the Quiz Game and by 35% in the Media Player, while the number of lines of code was reduced by 38% in the Quiz Game and by 78% in the Media Player. It is important to notice that back-porting required the involvement of a tool expert, due to the proprietary nature of the template architecture; conversely, collision resolution was performed by a Java developer. Given the complexity (and expressive power) of the WebRatio template language, the same feature required a considerably higher work for back-porting than for conflict resolution. Ultimately, the effort for back-porting is comparable to the whole effort required by model and code co-evolution, i.e., feature development plus conflict resolution.

### 6.7.2 Evaluation of Collision Prevention Guidelines

As shown in Section 6.7.1, collision resolution is accountable for a non-negligible fraction of the total development effort; thus, any improvement of the code generator able to reduce the number of collisions can have a significant impact. To evaluate the effect of the collision prevention guidelines introduced in Section 6.4, the original code generator of IFMLEdit.org has been compared to a new version implementing the guidelines. The new version reduces non-determinism, enhances line-level separation of concerns, and introduces artificial collisions to guard against non backward-compatible changes.

| Sprint | Mode & Text Co-evolution | | | | Template-based fw engineering | | | |
|---|---|---|---|---|---|---|---|---|
| | Feature Devel. | | Conflict res. | | Feature Devel. | | Back-porting | |
| | Up. | At. Up. | Up. | At. Up. | Up. | At. Up. | Up. | At. Up. |
| Proof of Concept | 5 | 142 | 0 | 0 | 5 | 136 | 13 | 137 |
| Styling and Expl. | 4 | 29 | 0 | 0 | 4 | 24 | 4 | 22 |
| $2^{nd}$ Mechanics | 10 | 42 | 0 | 0 | 4 | 11 | 4 | 8 |
| I18n | 5 | 15 | 0 | 0 | 5 | 16 | 5 | 16 |
| Total | 24 | 228 | 0 | 0 | 18 | 187 | 26 | 183 |
| | Total Effort (devel+integration) | | | | Total Effort (devel+integration) | | | |
| | Up. | | At. Up. | | Up. | | At. Up. | |
| | 24 | | 228 | | 44 | | 370 | |

**Table 6.3:** *Quiz Game: development statistics with WebRatio*

| Sprint | Mode & Text Co-evolution | | | | Template-based fw engineering | | | |
|---|---|---|---|---|---|---|---|---|
| | Feature Devel. | | Conflict res. | | Feature Devel. | | Back-porting | |
| | Up. | At. Up. | Up. | At. Up. | Up. | At. Up. | Up. | At. Up. |
| Proof of Concept | 2 | 21 | 0 | 0 | 2 | 21 | 4 | 25 |
| Styling | 6 | 47 | 0 | 0 | 5 | 48 | 6 | 153 |
| Songs Filtering | 2 | 4 | 2 | 7 | 0 | 0 | 0 | 0 |
| Songs cover | 1 | 2 | 0 | 0 | 1 | 2 | 2 | 4 |
| Total | 11 | 74 | 2 | 7 | 8 | 71 | 12 | 182 |
| | Total Effort (devel+integration) | | | | Total Effort (devel+integration) | | | |
| | Up. | | At. Up. | | Up. | | At. Up. | |
| | 13 | | 81 | | 20 | | 253 | |

**Table 6.4:** *Media Player: development statistics with WebRatio*

As discussed in Section 6.4, the amount of work for conflict resolution is proportional to the number of collision groups in the collision graph containing the updates of the Virtual Developer ($\Delta^{G}_{V,n+1}$) and of the human developer ($\Delta^{M}_{C,n+1}$). Table 6.5 and Table 6.6 show, for the original and for the enhanced version of the IFMLEdit.org code generator, the following data: the number of updates and of atomic updates of the human developer, the number of updates and of atomic updates of the Virtual Developer, the number of updates and of atomic updates required for collision resolution. The number updates in $\Delta^{G}_{V,n+1}$ depends on the magnitude of the changes on the model at each sprint and is independent from previous sprints; the number of updates of the human developer grows, because at each sprint the new manual changes add up to the ones of previous sprints.

It can be noted that the implementation of the conflict prevention guidelines in the code generator reduces the number of collision groups by 75% in the Quiz Game and by 60% in the Media Player; the number of lines affected is reduced by 79% in the Quiz Game and by 27% in the Media Player.

For evaluating the impact of the improved code generator on the total development effort, only the number of updates can be used, because the number of lines increases when the line separation rules are added to the model transformation. Recalling from Table 6.1 and 6.2 that collision resolution is accountable for 10.9% of the total develop-

ment effort for the Quiz Game and 11% of the Media Player, then collision prevention guidelines reduce the total development effort by 8% (75% of 10.9%) in the Quiz Game and 7% (60% of 11%) in the Media Player.

These results show that even very simple design rules for the code transformation, such as a better separation of concerns at line level, can increase automatic resolution and thus reduce the collision resolution effort. All the remaining collisions detected by the VCS were actual conflicts between the manually implemented and automatically generated code, which required manual intervention and decision making by the developer.

| | Original Implementation | | | | | | Conflict Prevention | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Manual | | Generated | | Collisions | | Manual | | Generated | | Collisions | |
| Sprint | Up. | A.U. | Up. | A.U. | Up. | A.U. | Up. | A.U. | Up. | A.U. | Up. | A.U. |
| Proof of Concept | - | - | - | 1442 | - | - | - | - | - | 1510 | - | - |
| Styling and Expl. | 27 | 1124 | 21 | 287 | 4 | 77 | 27 | 1054 | 16 | 274 | 1 | 37 |
| $2^{nd}$ Mechanics | 30 | 1196 | 45 | 1168 | 3 | 20 | 30 | 1235 | 44 | 1209 | 1 | 6 |
| I18n | 47 | 1612 | 27 | 457 | 5 | 139 | 47 | 1709 | 25 | 404 | 1 | 6 |
| | | | | Total | 12 | 236 | | | | Total | 3 | 49 |

**Table 6.5:** *Quiz Game: comparison of collisions without and with conflict prevention rules*

| | Original Implementation | | | | | | Conflict Prevention | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Manual | | Generated | | Collisions | | Manual | | Generated | | Collisions | |
| Sprint | Up. | A.U. | Up. | A.U. | Up. | A.U. | Up. | A.U. | Up. | A.U. | Up. | A.U. |
| Proof of Concept | - | - | - | 1202 | - | - | - | - | - | 1249 | - | - |
| Styling | 14 | 141 | 9 | 93 | 1 | 3 | 16 | 144 | 6 | 71 | 0 | 0 |
| Songs Filtering | 18 | 178 | 14 | 258 | 1 | 4 | 20 | 176 | 11 | 238 | 0 | 0 |
| Songs Cover | 22 | 217 | 12 | 303 | 2 | 10 | 24 | 223 | 5 | 268 | 2 | 14 |
| | | | | Total | 5 | 17 | | | | Total | 2 | 14 |

**Table 6.6:** *Media Player: comparison of collisions without and with conflict prevention rules*

Note also that the conflict prevention guidelines reduce the size of the delta produced by the Virtual Developer: the total number of updates contained in the Virtual Developer's deltas in all sprints is reduced by 19% in the Quiz Game and by 47% in the Media Player. Eliminating sources of non-determinism removes unnecessary changes applied by the Virtual Developer.

CHAPTER 7

# Conclusions and Future Work

In this thesis we have addressed the high costs/advantages ratio involved in applying Model Driven Development to Web and Mobile applications development.

We have presented ALMOsT.js, a JavaScript framework for the agile development of model transformations which can be integrated inside web servers, web clients or any JavaScript based execution environment. It can be integrated in preexisting tools or paired with other general purpose (e.g. template engines) or specifically built (e.g. tracing plug-ins) tools in order to iteratively introduce the Model Driven methodology in an existing work-flow or build a custom Model Driven environment. Our future work will be focused on further evaluation of the proposed framework and the development of plug-ins which can further simplify interoperability with existing tools or introduce enhanced functionalities without undermining the agile nature of the tool itself. Our future work will also focus on the development of guidelines, patterns and best practices for transformations rules an the orchestration of complex graphs of transformations.

We have presented a formal semantics for the Interaction Flow Modeling Language specifically targeted to Web and Mobile applications. This semantics is tool independent and solely based on the model structure, making it suitable for inter tools compatibility and reducing the level of understanding required to properly identify the behavior of different models. Our future work will be focused on the exploitation of the formal properties of the proposed semantics in order to facilitate the identification of problematic configurations (e.g. unreachable states or dependency deadlock).

We have presented IFMLEdit.org a web based tool for the rapid prototyping of Web and Mobile applications. The tool implements the previously defined formal semantics and support different target architectures and languages. It provides both in browser emulation, for rapid concepts iterations and validation, and prototype download, for customization and final release. Our future work will focus on introducing of new code

generators featuring different underlying languages and frameworks and enhancing interoperability with other tools based on the Interaction Flow Modeling Language.

We have presented a formal work-flow for the cooperation of Model Driven code generators and human developers via off-the-shelf version control systems. The work-flow treats the application source code as the source of truth and the model, and code generator, as a Virtual Developer, i.e., yet another member of the team. We have also presented ALMOsT-Git, a reference implementation of the proposed work-flow based on the popular Git Version Control System, and evaluated its effect on the development of use-case applications w.r.t. the most common template based forward engineering approach and both the proposed IFMLEdit.org and an industrial tool. We have shown how applying the proposed methodology reduces the work required to obtain a working application and in particular how developers can collaborate in the process without the need of any Model Driven expertise. Our future work will be focused on the evaluation of the proposed methodology in an industry environment and the experimentation of hybrid approaches exploiting both the proposed work-flow, for point-wise fix and experimentation, and template based forward engineering that can be delayed and limited to recurrent modifications that justify the increased complexity.

All of these contributions play a role in reducing the costs of Model Driven Development from different prospective; further investigations will focus on quantifiable impacts on the development time and the different impacts that can be produced on teams of different size, expertise and organizational structure.

# List of Figures

# List of Tables

# List of Definitions

# List of Mapping Rules

# Bibliography


[1] Acceleo. https://www.eclipse.org/acceleo/. Accessed: 2018-02-20.

[2] Adobe Phonegap. http://phonegap.com/. Accessed: 2017-11-06.

[3] Alf 1.0 Specification. http://www.omg.org/spec/ALF/. Accessed: 2017-11-06.

[4] ALMOsT-Git a tool for model and text co-evolution. https://npmjs.com/package/almost-git/. Accessed: 2018-07-20.

[5] Appcelerator platform. http://www.appcelerator.com/. Accessed: 2017-11-06.

[6] Asynchronous Module Definition (AMD). https://github.com/amdjs/amdjs-api/wiki/AMD. Accessed: 2018-02-20.

[7] ATL Transformation Language. http://www.eclipse.org/atl/. Accessed: 2017-01-10.

[8] CommonJS Modules. http://wiki.commonjs.org/wiki/Modules/1.1.1. Accessed: 2018-02-20.

[9] Cordova. https://cordova.apache.org/. Accessed: 2018-07-20.

[10] CPNTools. http://cpntools.org/. Accessed: 2017-06-26.

[11] Dart. https://www.dartlang.com/. Accessed: 2018-07-20.

[12] Eclipse modeling framework (emf). https://www.eclipse.org/modeling/emf/. Accessed: 2017-11-06.

[13] ECMAScript 6. http://www.ecma-international.org/ecma-262/6.0/.

[14] Embedded javascript templates. https://ejs.co/. Accessed: 2018-02-20.

[15] Flutter project. https://www.flutter.io/. Accessed: 2017-11-06.

[16] Git. https://git-scm.com/. Accessed: 2018-02-20.

[17] GTK project. https://www.gtk.org/. Accessed: 2017-11-06.

[18] IBM MobileFirst platform foundation. https://www.ibm.com/support/knowledgecenter/SSNJXP/welcome.html Accessed: 2017-11-06.

[19] Mendix platform. https://www.mendix.com/. Accessed: 2017-11-06.

[20] Meta Object Facility Version 2.5.1. http://www.omg.org/spec/MOF/2.5.1/. Accessed: 2017-1-10.

[21] Outsystems platform. https://www.outsystems.com/. Accessed: 2017-11-06.

[22] Precise semantics of UML state machines (PSSM), beta version. http://www.omg.org/spec/PSSM/. Accessed: 2017-11-06.

[23] Principles behind the Agile Manifesto. http://agilemanifesto.org/principles.html.

[24] Qt project. https://www.qt.io/. Accessed: 2017-11-06.

[25] Query View Transformation V 1.3. http://www.omg.org/spec/QVT/1.3/. Accessed: 2017-01-10.

[26] Rhomobile suite. http://rhomobile.com. Accessed: 2017-11-06.

[27] Salesforce platform. https://www.salesforce.com. Accessed: 2017-11-06.

[28] Semantics of a foundational subset for executable UML models (FUML). http://www.omg.org/spec/FUML/. Accessed: 2017-11-06.

[29] Telerik appbuilder. http://www.telerik.com/platform/appbuilder. Accessed: 2017-11-06.

[30] Unified Modeling Language (UML). http://www.omg.org/spec/UML/. Accessed: 2017-11-06.

[31] Web Components. https://www.webcomponents.org/. Accessed: 2018-02-20.

[32] WebRatio platform. http://www.webratio.com/. Accessed: 2017-11-06.

[33] Xamarin platform. https://www.xamarin.com/. Accessed: 2017-11-06.

[34] Zoho Creator. https://www.zoho.com/creator/. Accessed: 2018-07-20.

[35] Gürkan Alpaslan and Oya Kalipsiz. Model driven web application development with agile practices. *CoRR*, abs/1610.03335, 2016.

[36] Anthony Anjorin, Marius Paul Lauder, Michael Schlereth, and Andy Schürr. Support for bidirectional model-to-text transformations. *ECEASST*, 36, 2010.

[37] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 692–706. Springer, 2006.

[38] Vincent Aranega, Jean-Marie Mottu, Anne Etien, and Jean-Luc Dekeyser. Using Trace to Situate Errors in Model Transformations. 50, April 2011.

[39] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010.

[40] Scott Barnett, Rajesh Vasa, and John Grundy. Bootstrapping mobile app development. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 657–660. IEEE Computer Society, 2015.

[41] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. Mdeforge: an extensible web-based modeling platform. In Richard F. Paige, Jordi Cabot, Marco Brambilla, Louis M. Rose, and James H. Hill, editors, *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS 2014, Valencia, Spain, September 30, 2014.*, volume 1242 of *CEUR Workshop Proceedings*, pages 66–75. CEUR-WS.org, 2014.

[42] Kent L. Beck. *Extreme programming explained - embrace change*. Addison-Wesley, 1990.

[43] Carlo Bernaschina. Almost.js: An agile model to model and model to text transformation framework. In Jordi Cabot, Roberto De Virgilio, and Riccardo Torlone, editors, *Web Engineering - 17th International Conference, ICWE 2017, Rome, Italy, June 5-8, 2017, Proceedings*, volume 10360 of *Lecture Notes in Computer Science*, pages 79–97. Springer, 2017.

[44] Carlo Bernaschina, Sara Comai, and Piero Fraternali. Ifmledit.org: Model driven rapid prototyping of mobile apps. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 207–208. IEEE, 2017.

[45] Carlo Bernaschina, Sara Comai, and Piero Fraternali. Online model editing, simulation and code generation for web and mobile applications. In *9th IEEE/ACM International Workshop on Modelling in Software Engineering, MiSE@ICSE 2017, Buenos Aires, Argentina, May 21-22, 2017*, pages 33–39. IEEE, 2017.

[46] Carlo Bernaschina, Sara Comai, and Piero Fraternali. Formal semantics of omg's interaction flow modeling language (IFML) for mobile and rich-client application model driven development. *Journal of Systems and Software*, 137:239–260, 2018.

[47] Matthias Biehl. Literature Study on Model Transformations, July 2010.

[48] Klaus Birken. Building code generators for dsls using a partial evaluator for the xtend language. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 407–424. Springer, 2014.

[49] G. Botturi, Emad Samuel Malki Ebeid, Franco Fummi, and Davide Quaglia. Model-driven design for the development of multi-platform smartphone applications. In *Proceedings of the 2013 Forum on specification and Design Languages, FDL 2013, Paris, France, September 24-26, 2013*, pages 1–8. IEEE, 2013.

[50] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[51] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.

[52] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. Extending the interaction flow modeling language (IFML) for model driven development of mobile applications front end. In Irfan Awan, Muhammad Younas, Xavier Franch, and Carme Quer, editors, *Mobile Web Information Systems - 11th International Conference, MobiWIS 2014, Barcelona, Spain, August 27-29, 2014. Proceedings*, volume 8640 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2014.

[53] Marco Brambilla, Eric Umuhoza, and Roberto Acerbis. Model-driven development of user interfaces for iot systems via domain-specific components and patterns. *J. Internet Services and Applications*, 8(1):14:1–14:21, 2017.

[54] Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, and Gabor Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Comput. Sci. Inf. Syst.*, 8(2):225–253, 2011.

[55] Jordi Cabot, Robert Clarisó, Marco Brambilla, and Sébastien Gérard. Cognifying model-driven software engineering. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers*, volume 10748 of *Lecture Notes in Computer Science*, pages 154–160. Springer, 2017.

[56] Daniel Calegari and Nora Szasz. Verification of model transformations: A survey of the state-of-the-art. *Electr. Notes Theor. Comput. Sci.*, 292:5–25, 2013.

[57] Deniz Cetinkaya and Alexander Verbraeck. Metamodeling and model transformations in modeling and simulation. In S. Jain, Roy R. Creasey Jr., Jan Himmelspach, K. Preston White, and Michael C. Fu, editors, *Winter Simulation Conference 2011, WSC'11, Phoenix, AZ, USA, December 11-14, 2011*, pages 3048–3058. WSC, 2011.

[58] Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. Round-trip support for extra-functional property management in model-driven engineering of embedded systems. *Information & Software Technology*, 55(6):1085–1100, 2013.

[59] Tony Clark, Paul Sammut, and James S. Willans. Applied metamodelling: A foundation for language driven development (third edition). *CoRR*, abs/1505.00149, 2015.

[60] Sara Comai and Piero Fraternali. A semantic model for specifying data-intensive web applications using webml. In Isabel F. Cruz, Stefan Decker, Jérôme Euzenat, and Deborah L. McGuinness, editors, *Proceedings of SWWS'01, The first Semantic Web Working Symposium, Stanford University, California, USA, July 30 - August 1, 2001*, pages 566–585, 2001.

[61] Diarmuid Corcoran. The good, the bad and the ugly: Experiences with model driven development in large scale projects at ericsson. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier, editors, *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings*, volume 6138 of *Lecture Notes in Computer Science*, page 2. Springer, 2010.

[62] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[63] Peter de Lange, Petru Nicolaescu, Thomas Winkler, and Ralf Klamma. Enhancing MDWE with collaborative live coding. In Ina Schaefer, Dimitris Karagiannis, Andreas Vogelsang, Daniel Méndez, and Christoph Seidl, editors, *Modellierung 2018, 21.-23. Februar 2018, Braunschweig, Germany*, volume P-280 of *LNI*, pages 199–214. Gesellschaft für Informatik e.V., 2018.

[64] Juan de Lara and Esther Guerra. *A posteriori* typing for model-driven engineering: concepts, analysis, and applications. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, page 1136. ACM, 2018.

[65] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[66] Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Apr.

[67] Chi-Kien Diep, Quynh-Nhu Tran, and Minh-Triet Tran. Online model-driven IDE to design guis for cross-platform mobile applications. In Huynh Quyet Thang, Binh Nguyen Thanh, Tien Van Do, Marc Bui, and Son Ngo Hong, editors, *4th International Symposium on Information and Communication Technology, SoICT '13, Danang, Viet Nam - December 05 - 06, 2013*, pages 294–300. ACM, 2013.

[68] Junhua Ding, Wei Song, and Dongmei Zhang. An approach for modeling and analyzing mobile push notification services. In *IEEE International Conference on Services Computing, SCC 2014, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 725–732. IEEE Computer Society, 2014.

[69] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman, 2000.

[70] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. Towards a Traceability Framework for Model Transformations in Kermeta. In J. Oldevik J. Aagedal, T. Neple, editor, *ECMDA-TW'06: ECMDA Traceability Workshop*, pages 31–40, Bilbao (Spain), July 2006. Sintef ICT, Norway.

[71] Emanuele Falzone and Carlo Bernaschina. Model based rapid prototyping and evolution of web application. In Tommi Mikkonen, Ralf Klamma, and Juan Hernández, editors, *Web Engineering - 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings*, volume 10845 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2018.

[72] Rita Francese, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Model-driven development for multi-platform mobile applications. In Pekka Abrahamsson, Luis Corral, Markku Oivo, and Barbara Russo, editors, *Product-Focused Software Process Improvement - 16th International Conference, PROFES 2015, Bolzano, Italy, December 2-4, 2015, Proceedings*, volume 9459 of *Lecture Notes in Computer Science*, pages 61–67. Springer, 2015.

[73] Piero Fraternali, Sara Comai, Alessandro Bozzon, and Giovanni Toffetti Carughi. Engineering rich internet applications with a model-driven approach. *TWEB*, 4(2):7:1–7:47, 2010.

[74] Lamia Gaouar, Abdelkrim Benamar, and Fethi Tarik Bendimerad. Model driven approaches to cross platform mobile development. In *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, IPAC '15, pages 19:1–19:5, New York, NY, USA, 2015. ACM.

[75] Jokin García, Oscar Díaz, and Jordi Cabot. An adapter-based approach to co-evolve generated SQL in model-to-text transformations. In Matthias Jarke, John Mylopoulos, Christoph Quix, Colette Rolland, Yannis Manolopoulos, Haralambos Mouratidis, and Jennifer Horkoff, editors, *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*, volume 8484 of *Lecture Notes in Computer Science*, pages 518–532. Springer, 2014.

[76] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages. *Autom. Softw. Eng.*, 16(3-4):415–454, 2009.

[77] Jeremy Gibbons and Perdita Stevens, editors. *Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures*, volume 9715 of *Lecture Notes in Computer Science*. Springer, 2018.

[78] Thomas Goldschmidt and Axel Uhl. Retainment policies - A formal framework for change retainment for trace-based model transformations. *Information & Software Technology*, 55(6):1064–1084, 2013.

[79] Volker Gruhn and AndrÃ¨ Koehler. Modeling communication behavior of mobile applications. In *Proceedings of the 11th International Workshop on Exploring Modeling Methods for Systems Analysis and Design (EMMSAD'06) held in conjunctiun with the 18th Conference on Advanced Information Systems (CAiSE'06), Luxembourg, 5-9 June, 2006*, pages 147–158, 2006.

[80] Brent Hailpern and Peri L. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–462, 2006.

[81] Jeffrey S. Hammond, Christopher Mines, Sara Sjoblom, Allison Vizgaitis, and Andrew Reese. The Forrester Wave: Mobile Low-Code Development Platforms, Q1 2017, Mar 2017.

[82] Regina Hebig and Reda Bendraou. On the need to study the impact of model driven engineering on software processes. In He Zhang, LiGuo Huang, and Ita Richardson, editors, *International Conference on Software and Systems Process 2014, ICSSP '14, Nanjing, China - May 26 - 28, 2014*, pages 164–168. ACM, 2014.

[83] Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej Wasowski. Model transformation languages under a magnifying glass: A controlled experiment with Xtend, ATL, and QVT. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2018, Lake Buena Vista, Florida, November 4-9, 2018*. ACM, 2018.

[84] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. Cross-platform model-driven development of mobile applications with $md^2$. In Sung Y. Shin and José Carlos Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 526–533. ACM, 2013.

[85] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-based classification of bidirectional transformation approaches. *Software and System Modeling*, 15(3):907–928, 2016.

[86] Bernhard Hoisl and Stefan Sobernig. Towards benchmarking evolution support in model-to-text transformation systems. In Jürgen Dingel, Sahar Kokaly, Levi Lucio, Rick Salay, and Hans Vangheluwe, editors, *Proceedings of the 4th Workshop on the Analysis of Model Transformations co-located with the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 28, 2015.*, volume 1500 of *CEUR Workshop Proceedings*, pages 16–25. CEUR-WS.org, 2015.

[87] John Edward Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.*, 89:144–161, 2014.

[88] John Edward Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 471–480. ACM, 2011.

[89] Javier Luis Cánovas Izquierdo, Valerio Cosentino, and Jordi Cabot. An empirical study on the maturity of the eclipse modeling ecosystem. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, pages 292–302. IEEE Computer Society, 2017.

[90] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1*. EATCS Monographs on Theoretical Computer Science. Springer, 1992.

[91] Álvaro Jiménez, Juan M. Vara, Verónica Andrea Bollati, and Esperanza Marcos. Metagem-trace: Improving trace generation in model transformation by leveraging the role of transformation models. *Sci. Comput. Program.*, 98:3–27, 2015.

[92] Christopher Jones and Xiaoping Jia. The AXIOM model framework - transforming requirements to native code for cross-platform mobile applications. In Joaquim Filipe and Leszek A. Maciaszek, editors, *ENASE 2014 - Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering, Lisbon, Portugal, 28-30 April, 2014*, pages 26–37. SciTePress, 2014.

[93] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[94] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[95] James Kirby Jr. Model-driven agile development of reactive multi-agent systems. In *30th Annual International Computer Software and Applications Conference, COMPSAC 2006, Chicago, Illinois, USA, September 17-21, 2006. Volume 2*, pages 297–302. IEEE Computer Society, 2006.

[96] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.

[97] Michael Kishinevsky, Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Alexander Taubin, and Alexandre Yakovlev. Coupling asynchrony and interrupts: Place chart nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings*, volume 1248 of *Lecture Notes in Computer Science*, pages 328–347. Springer, 1997.

[98] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003.

[99] Minhyuk Ko, Yongjin Seo, Bup-Ki Min, Seung Hak Kuk, and Hyeon Soo Kim. Extending UML meta-model for android application. In Huaikou Miao, Roger Y. Lee, Hongwei Zeng, and Jongmoon Baik, editors, *2012 IEEE/ACIS 11th International Conference on Computer and Information Science, Shanghai, China, May 30 - June 1, 2012*, pages 669–674. IEEE Computer Society, 2012.

[100] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2008.

[101] Fabrice Kordon and Yann Thierry-Mieg. Experiences in model driven verification of behavior with UML. In Christine Choppy and Oleg Sokolsky, editors, *Foundations of Computer Software. Future Trends and Techniques for Development, 15th Monterey Workshop 2008, Budapest, Hungary, September 24-26, 2008, Revised Selected Papers*, volume 6028 of *Lecture Notes in Computer Science*, pages 181–200. Springer, 2008.

[102] Gábor Kövesdán, Márk Asztalos, and László Lengyel. Polymorphic templates: A design pattern for implementing agile model-to-text transformations. In Davide Di Ruscio, Juan de Lara, and Alfonso Pierantonio, editors, *Proceedings of the 3rd Workshop on Extreme Modeling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, XM@MoDELS 2014, Valencia, Spain, September 29, 2014.*, volume 1239 of *CEUR Workshop Proceedings*, pages 32–41. CEUR-WS.org, 2014.

[103] Frank Alexander Kraemer. Engineering android applications based on UML activities. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, volume 6981 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2011.

[104] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In Sudipto Ghosh, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2009.

[105] Ivan Kurtev. State of the art of QVT: A model transformation language standard. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, volume 5088 of *Lecture Notes in Computer Science*, pages 377–393. Springer, 2007.

[106] Naziha Laaz and Samir Mbarki. A model-driven approach for generating RIA interfaces using IFML and ontologies. In Mohammed El Mohajir, Mohamed Chahhou, Mohammed Al Achhab, and Badr Eddine El Mohajir, editors, *4th IEEE International Colloquium on Information Science and Technology, CiSt 2016, Tangier, Morocco, October 24-26, 2016*, pages 83–88. IEEE, 2016.

[107] Adrian Leow, Van Baker, Jason Wong, and Marty Resnick. Critical Capabilities for Mobile App Development Platforms, Sep 2018.

[108] Reza Matinnejad. Agile model driven development: An intelligent compromise. In *9th International Conference on Software Engineering Research, Management and Applications, SERA 2011, Baltimore, MD, USA, August 10-12, 2011*, pages 197–202. IEEE Computer Society, 2011.

[109] Tanja Mayerhofer. Using fuml as semantics specification language in model driven engineering. In Yan Liu, Steffen Zschaler, Benoit Baudry, Sudipto Ghosh, Davide Di Ruscio, Ethan K. Jackson, and Manuel Wimmer, editors, *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013.*, volume 1115 of *CEUR Workshop Proceedings*, pages 87–93. CEUR-WS.org, 2013.

[110] Niklas Mellegård, Adry Ferwerda, Kenneth Lind, Rogardt Heldal, and Michel R. V. Chaudron. Impact of introducing domain-specific modelling in software maintenance: An industrial case study. *IEEE Trans. Software Eng.*, 42(3):245–260, 2016.

[111] Tom Mens. A state-of-the-art survey on software merging. *IEEE Trans. Software Eng.*, 28(5):449–462, 2002.

[112] Bup-Ki Min, Minhyuk Ko, Yongjin Seo, Seunghak Kuk, and Hyeon Soo Kim. A uml metamodel for smart device application modeling based on windows phone 7 platform. In *TENCON 2011 - 2011 IEEE Region 10 Conference*, pages 201–205, Nov 2011.

[113] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[114] OMG. Interaction flow modeling language (IFML), version 1.0. `http://www.omg.org/spec/IFML/1.0/`, 2015.

[115] Abilio G. Parada, Milena Rota Sena Marques, and Lisane B. de Brisolara. Automating mobile application development: Uml-based code generation for android and windows phone. *RITA*, 22(2):31–50, 2015.

[116] Andreas Pleuss, Stefan Wollny, and Goetz Botterweck. Model-driven development and evolution of customized user interfaces. In Peter Forbrig, Prasun Dewan, Michael Harrison, and Kris Luyten, editors, *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'13, London, United Kingdom - June 24 - 27, 2013*, pages 13–22. ACM, 2013.

[117] Marcos Antonio Possatto and Daniel Lucrédio. Automatically propagating changes from reference implementations to code generation templates. *Information & Software Technology*, 67:65–78, 2015.

[118] Sarra Roubi, Mohammed Erramdani, and Samir Mbarki. Extending graphical part of the interaction flow modeling language to generate rich internet graphical user interfaces. In Slimane Hammoudi, Luís Ferreira Pires, Bran Selic, and Philippe Desfray, editors, *MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016.*, pages 161–167. SciTePress, 2016.

[119] Ayoub Sabraoui, Mohammed El Koutbi, and Ismail Khriss. Gui code generation for android applications using a mda approach. In *2012 IEEE International Conference on Complex Systems (ICCS)*, pages 1–6, 2012.

[120] Daniel Sanchez and Hector Florez. Model driven engineering approach to manage peripherals in mobile devices. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Elena N. Stankova, Carmelo Maria Torre, Ana Maria A. C. Rocha, David Taniar, Bernady O. Apduhan, Eufemia Tarantino, and Yeonseung Ryu, editors, *Computational Science and Its Applications - ICCSA 2018 - 18th International Conference, Melbourne, VIC, Australia, July 2-5, 2018, Proceedings, Part IV*, volume 10963 of *Lecture Notes in Computer Science*, pages 353–364. Springer, 2018.

[121] Iván Santiago, Álvaro Jiménez, Juan Manuel Vara, Valeria de Castro, Verónica Andrea Bollati, and Esperanza Marcos. Model-driven engineering as a new landscape for traceability management: A systematic literature review. *Information & Software Technology*, 54(12):1340–1356, 2012.

[122] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[123] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*. Prentice Hall, 2002.

[124] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development - technology, engineering, management*. Pitman, 2006.

[125] Rocio Nahime Torres and Carlo Bernaschina. Almost-trace: A web based embeddable tracing tool for almost.js. In Jordi Cabot, Roberto De Virgilio, and Riccardo Torlone, editors, *Web Engineering - 17th International Conference, ICWE 2017, Rome, Italy, June 5-8, 2017, Proceedings*, volume 10360 of *Lecture Notes in Computer Science*, pages 554–558. Springer, 2017.

[126] Axel Uhl. Model-driven development in the enterprise. *IEEE Software*, 25(1):46–49, 2008.

[127] Axel Uhl and Scott W. Ambler. Point/counterpoint: Model driven architecture is ready for prime time / agile model driven development is good enough. *IEEE Software*, 20(5):70–73, 2003.

[128] Eric Umuhoza and Marco Brambilla. Model driven development approaches for mobile applications: A survey. In Muhammad Younas, Irfan Awan, Natalia Kryvinska, Christine Strauss, and Do Van Thanh, editors, *Mobile Web and Intelligent Information Systems - 13th International Conference, MobiWIS 2016, Vienna, Austria, August 22-24, 2016, Proceedings*, volume 9847 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2016.

[129] Muhammad Usman, Muhammad Zohaib Z. Iqbal, and Muhammad Uzair Khan. A model-driven approach to generate mobile applications for multiple platforms. In Sungdeok (Steve) Cha, Yann-Gaël Guéhéneuc, and Gihwon Kwon, editors, *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, pages 111–118. IEEE, 2014.

[130] Juan Manuel Vara, Verónica Andrea Bollati, Álvaro Jiménez, and Esperanza Marcos. Dealing with traceability in the mddof model transformations. *IEEE Trans. Software Eng.*, 40(6):555–583, 2014.

[131] Steffen Vaupel, Daniel Strüber, Felix Rieger, and Gabriele Taentzer. Agile bottom-up development of domain-specific ides for model-driven development. In Davide Di Ruscio, Juan de Lara, and Alfonso Pierantonio, editors, *Proceedings of the Workshop on Flexible Model Driven Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2015), Ottawa, Canada, September 29, 2015.*, volume 1470 of *CEUR Workshop Proceedings*, pages 12–21. CEUR-WS.org, 2015.

[132] Steffen Vaupel, Gabriele Taentzer, René Gerlach, and Michael Guckert. Model-driven development of mobile applications for android and ios supporting role-based app variability. *Software and System Modeling*, 17(1):35–63, 2018.

[133] Paul Vincent, Van L. Baker, Yefim V. Natis, Kimihiko Iijima, Mark Driver, and Rob Dunie. Magic Quadrant for Enterprise High-Productivity Application Platform as a Service, Apr 2017.

[134] Jon Whittle, John Edward Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.

[135] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[136] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling*, 9(4):529–565, 2010.

[137] Jason Wong, Van Baker, Adrian Leow, and Marty Resnick. Magic Quadrant for Mobile App Development Platforms, Jul 2018.

[138] Jason Wong and Daniel Rotigel. Best Practices for Selecting Mobile Development Tools and Technologies at a Midsize Enterprise, Jan 2018.

[139] World Wide Web Consortium (W3C). HTML5. `https://www.w3.org/TR/html5/`.

[140] Yuefeng Zhang and Shailesh Patel. Agile model-driven development in practice. *IEEE Software*, 28(2):84–91, 2011.