Formal Semantics of OMG's Interaction Flow Modeling Language (IFML) for Mobile and Rich-Client Application Model Driven Development

Carlo Bernaschina, Sara Comai and Piero Fraternali

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy, 20133.
E-mail: first.last@polimi.it

Abstract

Model Driven Engineering relies on the availability of software models and of development tools supporting the transition from models to code. The generation of code from models requires the unambiguous interpretation of the semantics of the modeling languages used to specify the application. This paper presents the formalization of the semantics of the Interaction Flow Modeling Language (IFML), a recent OMG MDA standard conceived for the specification of the front-end part of interactive applications. IFML constructs are mapped to equivalent structures of Place Chart Nets (PCN), which allows their precise interpretation. The defined semantic mapping is implemented in an online Model-Driven Development environment that enables the creation and inspection of PCNs from IFML, the analysis of the behavior of IFML specifications via PCN simulation, and the generation of code for mobile and web-based architectures.

Keywords: Mobile Applications, Rich-client applications, Model-Driven Development, Translational Semantics.

1. Introduction

Model Driven Engineering is the branch of Software Engineering that emphasizes the use of models in the development process <1>. 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 <2>. 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 <3>.

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 <4>.

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 paper, 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) <5>, 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 <6>, 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 paper is the formal specification of the se-

mantics 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.

1.1. Contribution of the paper

The contributions of the paper 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 <7>). 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.
- We 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.
  - 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.

The rest of the paper is organized as follows: Section 2 surveys the related work; Section 3 provides a concise overview of IFML; Section 4 presents the essential elements of PCN and the mapping rules for the basic IFML constructs; Section 5 elaborates on the PCN mapping rules that express the semantics of complex interfaces; Section 6 illustrates the implementation of the semantic mapping rules in the on-line MDD environment IFMLEdit.org, which also showcase two code generators based on the IFML semantics defined in the paper; Section 7 concludes the paper with a discussion of how the formal semantics helps formalize underspecified features of IFML and analyze existing code generators; it also summarizes the limitations of the proposed mapping rules and provides an outlook on the ongoing and future work.

2. Related Work

The literature on MDD is abundant; we focus the overview of the related work to the topics more closely related to the paper: the practical approaches for the MDD of application front-ends and the translational methods for the specification of the semantics of MDA models and DSLs.

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 <8>, and, more recently, for mobile applications <3>. 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 <9>, and, possibly combined with agile techniques, increment productivity and quality <10; 1; 11>.

2.1.1. MDD approaches based on MDA and UML

The OMG Interaction Flow Modeling Language (IFML) <5> 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 <6>, 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:

[1] http://ifmledit.org/

<12> employs UML class diagrams and sequence diagrams to represent mobile applications and to generate code for the Android and Windows Phone platforms; Arctis <13> adopts a small UML profile and UML activity diagrams and translates such inputs into a state machine to obtain an executable Android application; <14> 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 <15> 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.

2.1.2. *MDD approaches based on Domain Specific Languages*

Other MDD solutions exploit Domain Specific Languages (DSL). The authors of <16> 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 <17> 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 <18> 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 <19> supports GUI design of mobile applications, but only simple behaviors can be specified; the RAPPT tool <3> 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.

The approach described in this paper starts from an MDA standard language, to favor the interoperability between the modeling language used to specify the front-end and the other languages of the same family usable for the MDD of the back-end. Furthermore, besides the implementation of the model transformation rules for supporting generative development, the focus of the paper is primarily on the formal specification of the modeling language semantics.

2.1.3. *Industrial MDD tools for multi-platform application front-ends*

In the industrial sector, trends described in Forrester <20> and Gartner <21> research reports show an orientation towards "low-code" development platforms, with Mendix <22> and Outsystems <23> leading this segment of the software tool market. Mendix <22> 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 <23> 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 mentioned WebRatio platform <6> 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.

Although not comparable in robustness with industrial strength products, our on-line 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.

2.1.4. *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 <24>.

A common way of formalizing a modeling language is through *translation semantics* <4>: 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 <25>: the authors extend AMMA, a framework for defining DSLs, with the specification of behavioral semantics expressed by means of ASMs. In <26> the authors introduce a transformational semantic framework based on ASM for the expression of the executable se-

mantics 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, <27> 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 <29>, 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 <30> 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.

In this paper, 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 <31> offer several advantages: they have a precise semantics <32>, their integration with a a subset of UML modeling constructs can be expressed formally with fUML <28>, and the Alf language <33> 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 <34>. The capability of Petri Nets <35>, and of their extensions <36>, 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 <37>, and the workflows of client-server communication <38>. Among the generalizations of Petri Nets, Colored Petri Nets (CPNs) are the most widely used formalism; they incorporate data, hierarchy, and time <36> and are supported by CPN Tools <39>, 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 <38>. Place Chart Nets (PCN) <7> 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 asynchronicity 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.

2.1.5. Cross-Platform development framework

A different approach to improve the efficiency of application development, especially in multi-platform projects, relies on the so-called *cross-platform development tools*, which enable the creation and distribution of applications to multiple platforms. Examples of frameworks following this approach are GTK <40>, which focuses on the User Interface, and QT <41>, which provides abstractions for OS dependent primitives. In the mobile environment, specific solutions exploit web development skills and support cross-platform coding in languages such as JavaScript/Java/C#, CSS and HTML5. Examples of such tools include Appcelerator Titanium <42>, IBM MobileFirst Platform Foundation <43>, PhoneGap <44>, RhoMobile <45>, Salesforce <46>, Telerik AppBuilder <47>, Xamarin <48>, Flutter <49> 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 apps inside native containers that provide access to native platform features).

3. Background: the Interaction Flow Modeling Language

The Interaction Flow Modeling Language (IFML) <5> 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

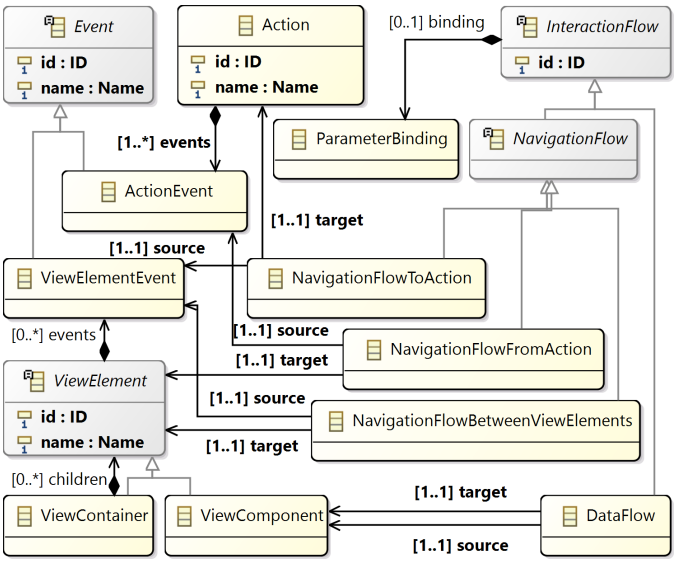


Figure 1: IFML Metamodel.

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 1 shows the essential elements of the IFML metamodel.

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 2a. Figure 2b shows an alternative organization, where the user interface is split into different independent ViewContainers corresponding to page templates.

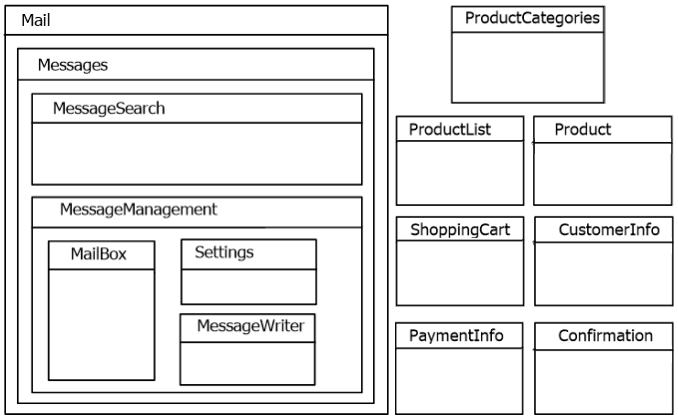


Figure 2: Different ViewContainers configurations expressing the interface organization

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 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).

The switch from one ViewContainer to another one can be

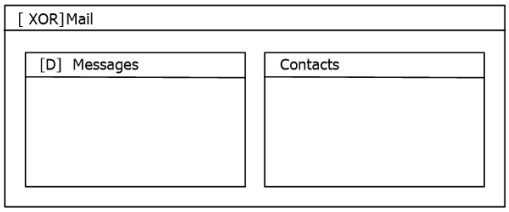


Figure 3: Example of mutually exclusive sub-containers

expressed implicitly or explicitly, as shown in Figure 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 4.

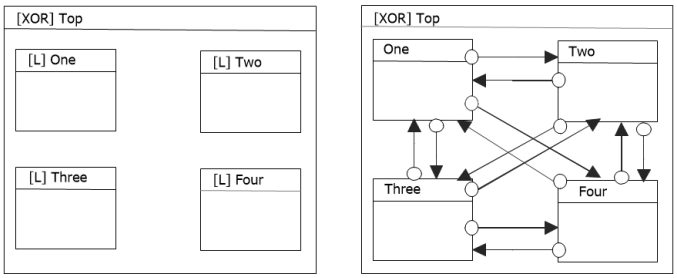


Figure 4: Landmark ViewContainers and explicit navigation

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 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 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.

Events, Navigation and Data Flows. *ViewElements* (ViewContainers and ViewComponents) can be associated with *Events*, to express that they support the user interaction. The effect of an Event is represented by a *NavigationFlow*, represented with an arrow, which connects the Event to the ViewElement affected by it (as shown in Figures 4 and 6). The NavigationFlow specifies a change of state of the user interface: the target

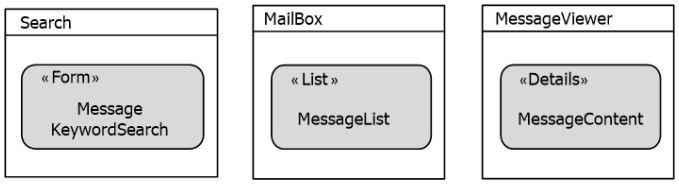
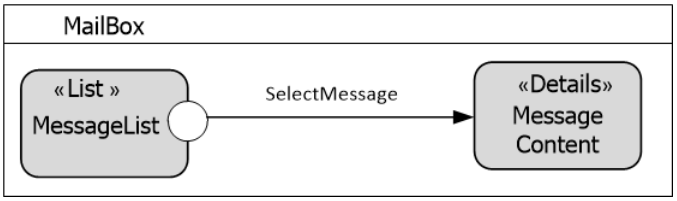


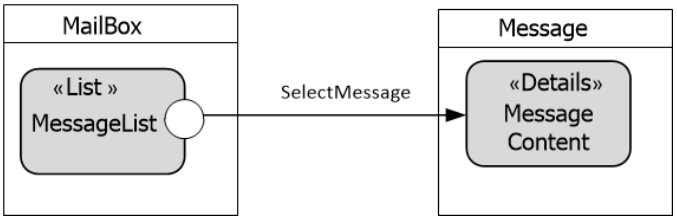
Figure 5: Example of ViewComponents within view containers

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 6 shows two examples of NavigationFlows between ViewComponents. In Figure 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 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 6: Example of NavigationFlow between ViewComponents

Figure 7 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 7, 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.

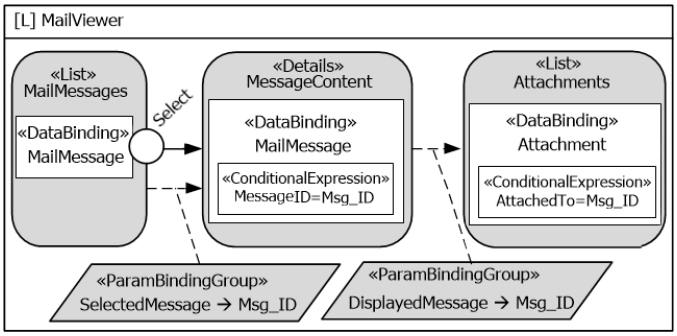


Figure 7: Example of DataFlows

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.

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 8, 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 8 shows an example of Action, for the creation of a new object.

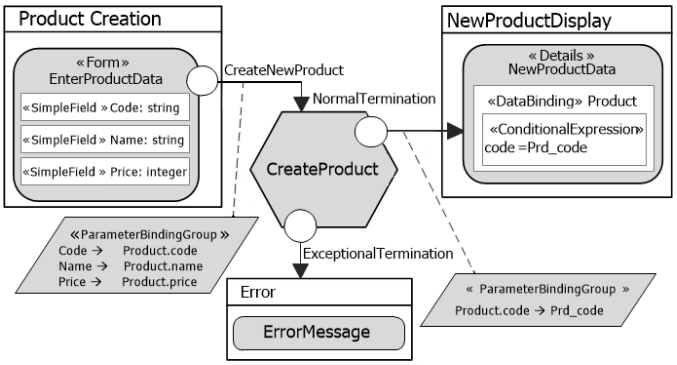


Figure 8: Example of Actions

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.

4. 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 paper, we exploit a generalization of Petri Nets, Place Chart Nets <7>, 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 4.5 addresses the basic structure of an application, which serves as a building block for all the other mapping rules. Section 4.6 concentrates on simple *ViewContainer*s and their interaction with *Event*s and *NavigationFlow*s. Section 4.7 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 4.8 introduces the semantic rules for mapping *Action*s, their triggering, and their life-cycle. Finally, Section 5 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.

4.1. Place Chart Nets

Place Chart Nets (PCN) <7> 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.

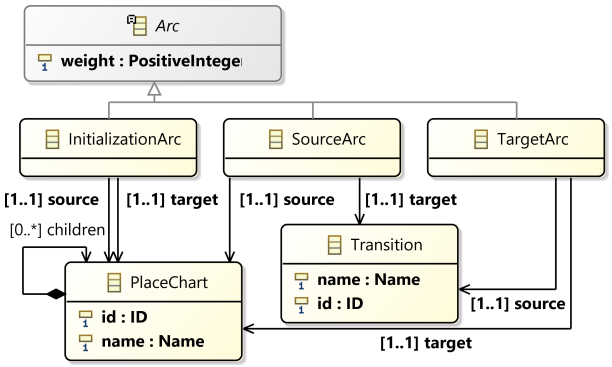


Figure 9: PCN Metamodel.

Figure 9 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 $Y_1 \dots Y_n$.

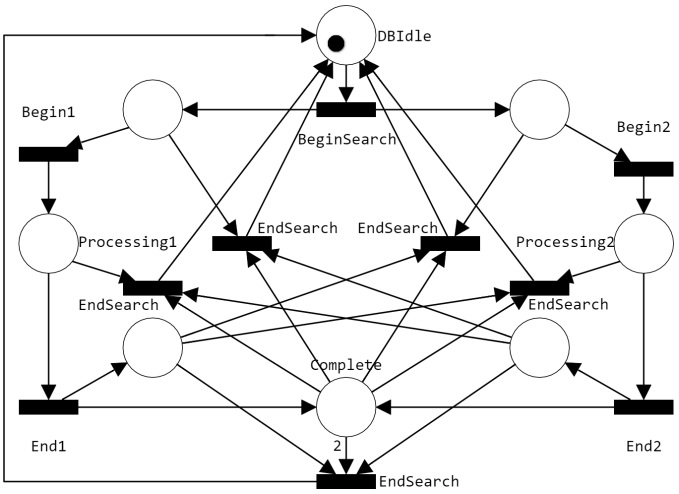
A formal definition of PCNs can be found in <7>. We illustrate the benefits of their usage by means of an example,

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.

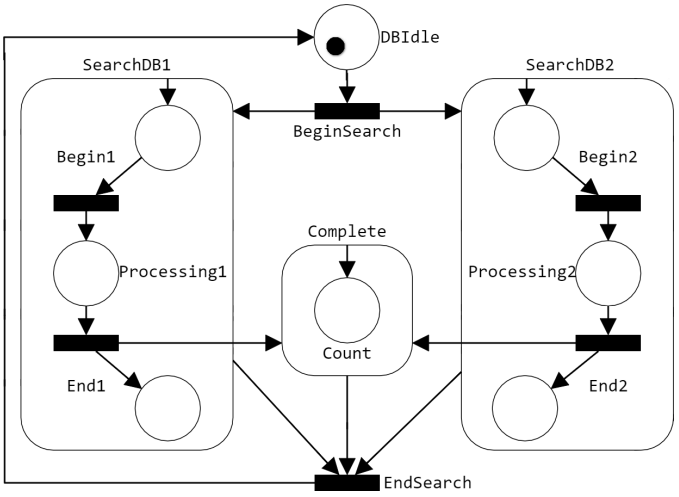
Figure 10a shows a Petri Net that describes such process, considering two database copies. The *BeginSearch* transition removes a token from *DBIdle* and initializes the parallel search, which proceeds asynchronously. The completion of a search is described by transitions *End1* or *End2*, which add 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 10a, 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 10b 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 system regardless of which database answered first.

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



(a) Petri Net



(b) Place Chart Net

Figure 10: Model complexity reduction of Place Chart Nets

4.2. 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 V, XOR ViewContainers as X, ancestor ViewContainers as A, child ViewContainers as C, source ViewContainers as S, target ViewContainers as T, transitions mapping events as e.

4.3. 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 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 B to $\overline{\mathsf{B}}$ or vice versa, changing the value of the variable.

4.4. 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.

4.5. 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 11 shows the PCN corresponding to an empty IFML model.

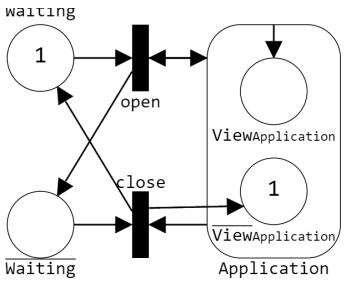


Figure 11: PCN of an empty application

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.

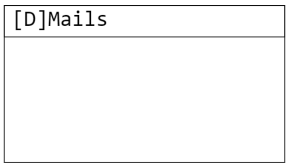
The example of Figure 11 defines the first mapping rule:

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 $\overline{\text{Application}}$, and a close transition, which moves a token from $\overline{\text{Waiting}}$ and $\overline{\text{Application}}$ to Waiting and $\overline{\text{View}_{\text{Application}}}$.

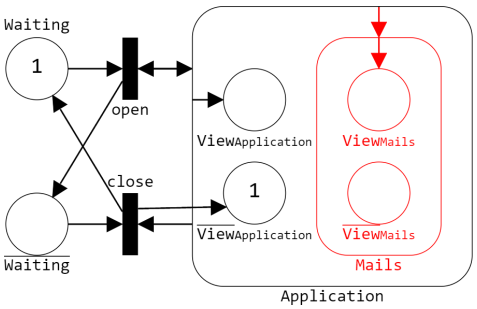
4.6. Mapping the structure: View Containers

Figure 12a 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 12b shows its mapping[2]: the PCN of Figure 11 is extended by introducing a place chart called Mails, child of



(a) IFML model



(b) Place Chart Net

Figure 12: Single, empty default ViewContainer Model

Application, which is initialized by the parent. The Mails place chart has two children bottom place charts ($\text{View}_{\text{Mails}}$, initialized by the parent, and $\overline{\text{View}_{\text{Mails}}}$). The Boolean variable $\text{View}_{\text{Mails}}$ represents whether the ViewContainer is, or is not, in view. The firing of the open transition now adds also a token to $\text{View}_{\text{Mails}}$, meaning that the ViewContainer is displayed.

This example of Figure 12 defines two mapping rules:

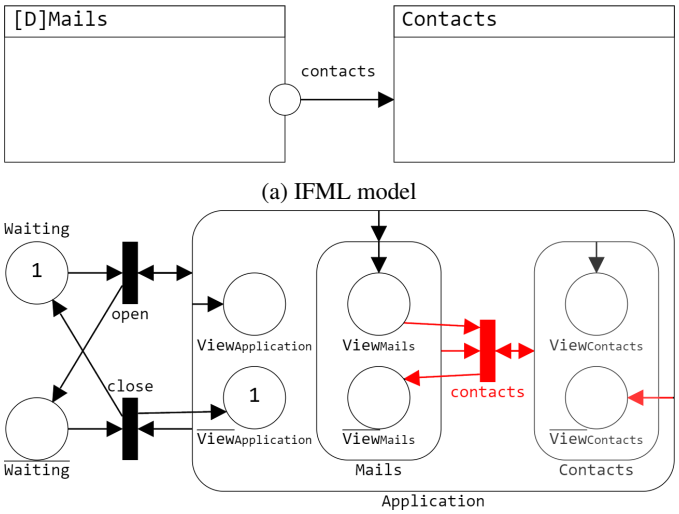
Rule **2)** TOPVIEWCONTAINER A top-level ViewContainer *V* maps to a place chart child of Application named V, with two children bottom place charts (View_{V} and $\overline{\text{View}_{\text{V}}}$). View_{V} is initialized by default from the parent.

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.

Navigation between ViewContainers. Figure 13a 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 13b shows the PCN that maps the IFML model of Figure 13a. According to rule 2, the Application top place chart contains two place charts Mails and Contacts, each one with two children bottom place charts ($\text{View}_{\text{Mails}}$, $\overline{\text{View}_{\text{Mails}}}$, $\text{View}_{\text{Contacts}}$, $\overline{\text{View}_{\text{Contacts}}}$). 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 $\text{View}_{\text{Mails}}$ place chart. Conversely, Contacts is not initialized as the default, but an initialization arc from Application targets the $\overline{\text{View}_{\text{Contacts}}}$ 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

[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.



(a) IFML model

(b) Place Chart Net

Figure 13: Navigation between top-level ViewContainers

transition moves a token from Mails, $\text{View}_{\text{Mails}}$ and Contacts to Contacts and $\overline{\text{View}_{\text{Mails}}}$. Note that the apparently redundant input place charts of the contacts transition (Mails, $\text{View}_{\text{Mails}}$) are necessary to ensure that: 1) tokens 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 13a 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 $\overline{\text{View}_{\text{V}}}$.

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, View_{S} and T to T and $\overline{\text{View}_{\text{S}}}$.

Landmark Navigation.

ViewContainers with the *landmark* visibility property represent the target of implicit navigation flows from the other ViewContainers nested within their parent container. Figure 14a 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 $\text{landmark}_{\text{V1}}$ that moves a token from Application and $\text{View}_{\text{Application}}$ to $\text{View}_{\text{Application}}$ and V1. For each top-level ViewContainer V2 different from V1, the $\text{landmark}_{\text{V1}}$ transition adds a token to $\overline{\text{View}_{\text{V2}}}$.

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 14a maps into the PCN of Figure 14b: the two transitions called $\text{landmark}_{\text{Mails}}$ and $\text{landmark}_{\text{Contacts}}$ check that the application is currently visible, by adding and removing a token from $\text{View}_{\text{Application}}$, and remove a token from it. Transition $\text{landmark}_{\text{Mails}}$ adds a token to Mails and $\overline{\text{View}_{\text{Contacts}}}$; symmetrically, transition $\text{landmark}_{\text{Contacts}}$ adds a token to Contacts and $\overline{\text{View}_{\text{Mails}}}$. 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.

4.7. Mapping the content: View Components

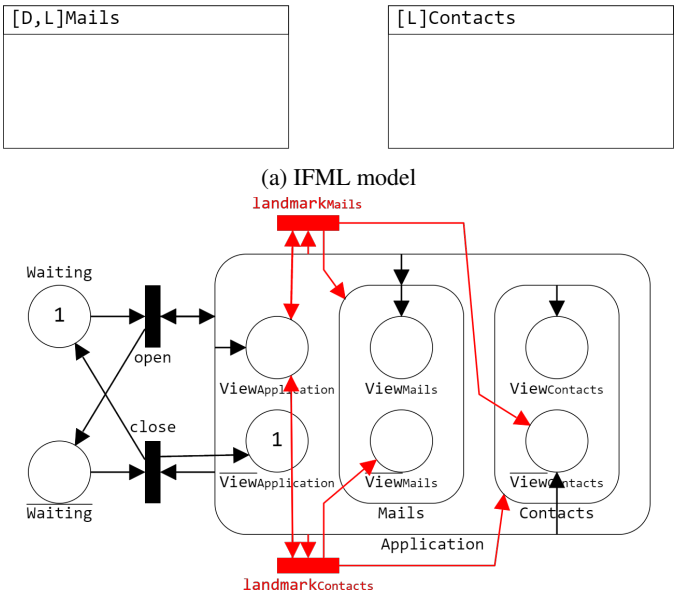
The examples discussed so far specify only empty interfaces without content. This section discusses the mapping of models that include ViewComponents, whose content is computed and rendered in the interface, possibly based on the value of some input parameters.

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:



(a) IFML model

(b) Place Chart Net

Figure 14: Navigation using Landmarks

- *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, $\overline{\mathsf{In}}$, Out and $\overline{\mathsf{Out}}$ (for an example see Figure 15c). 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 $\overline{\mathsf{In}}$ and a token in $\overline{\mathsf{Out}}$;
- *Ready*: a token in In and a token in $\overline{\mathsf{Out}}$;
- *Computed*: a token in $\overline{\mathsf{In}}$ 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 15c shows the *Compute* transition, which is a transition internal to the ViewComponent. The *Propagate* transition will be exemplified later (in Figure 17c): 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.

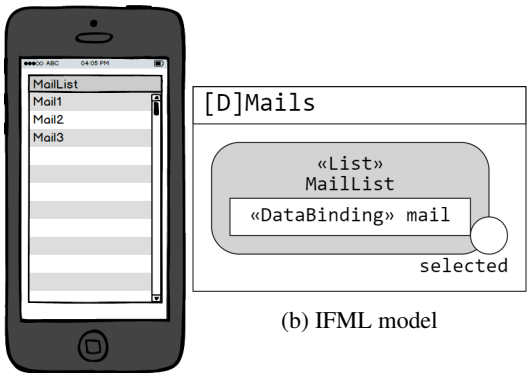
A Boolean Variable models the two states, represented by two bottom place charts View and $\overline{\mathsf{View}}$, respectively, as exemplified in Figure 15c.

The transition from the *Invalid* to the *Visible* state is modeled by the *Render* transition, shown in Figure 15c: 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 15a and 15b present a simple IFML model with a top-level ViewContainer comprising only one ViewComponent: an application that displays a list of mails. Figure 15c shows the PCN mapping of the IFML model[3]. A place chart, child of Mails, named MailList represents the *MailList* ViewComponent, initialized from the 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.

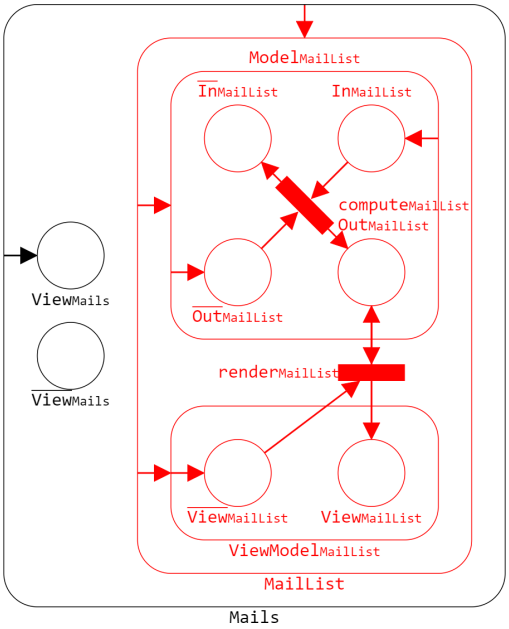
$\mathsf{Model}_{\mathsf{MailList}}$ contains the four bottom place charts defined earlier ($\mathsf{In}_{\mathsf{MailList}}$, $\overline{\mathsf{In}}_{\mathsf{MailList}}$, $\mathsf{Out}_{\mathsf{MailList}}$ and $\overline{\mathsf{Out}}_{\mathsf{MailList}}$). Transition $\mathsf{compute}_{\mathsf{MailList}}$ represents the computation of the content:

[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



(c) Place Chart Net

Figure 15: Single ViewComponent model

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.

$\mathsf{ViewModel}_{\overline{\mathsf{MailList}}}$ 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 15 introduces the rules:

Rule **7)** BASE VIEWCOMPONENT A ViewComponent *C* child of a parent ViewContainer *P* maps to:
- a place chart C, child of P, initialized by default from P.
- two children place charts of C: $\mathsf{Model}_\mathsf{C}$ and $\mathsf{ViewModel}_\mathsf{C}$, initialized by default from C.
- four bottom place charts In_C, $\overline{\mathsf{In}}_\mathsf{C}$, Out_C and $\overline{\mathsf{Out}}_\mathsf{C}$, children of $\mathsf{Model}_\mathsf{C}$.
- a transition $\mathsf{compute}_\mathsf{C}$, which removes a token from In_C and $\overline{\mathsf{Out}}_\mathsf{C}$ and inserts a token into $\overline{\mathsf{In}}_\mathsf{C}$ and Out_C.
- two bottom place charts View_C and $\overline{\mathsf{View}}_\mathsf{C}$, children of $\mathsf{ViewModel}_\mathsf{C}$. $\overline{\mathsf{View}}_\mathsf{C}$ is initialized by default from $\mathsf{ViewModel}_\mathsf{C}$.
- a transition $\mathsf{render}_\mathsf{C}$, which removes a token from $\overline{\mathsf{View}}_\mathsf{C}$ and Out_C and inserts a token into View_C and Out_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 In_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.

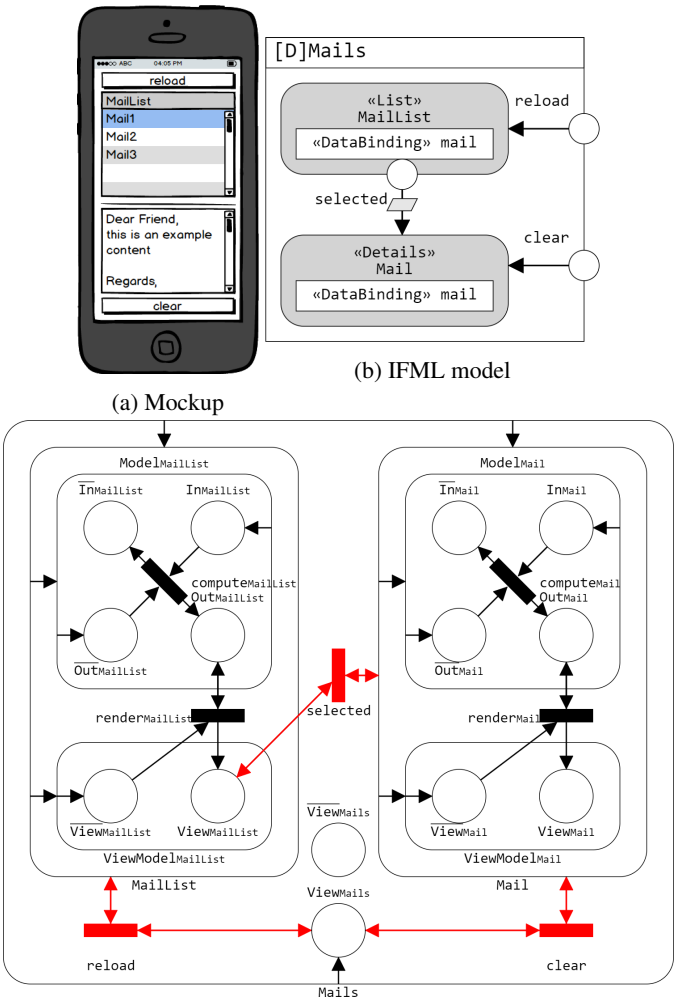
Note that the example of Figure 15c addresses the case of a ViewComponent without input. The next example elaborates on the mapping of ViewComponents with input parameters.

Events and Navigation Flows

Figure 16 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.

Figure 16b 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 16c illustrates the mapping of the IFML diagram of Figure 16b. 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) Mockup (b) IFML model

(c) Place Chart Net

Figure 16: Navigation betweenViewComponents: navigation flow

[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.

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* 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 16 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:
- removes a token from the place chart of T and adds a token to it.
- removes a token from the place chart Views and adds a token to it.

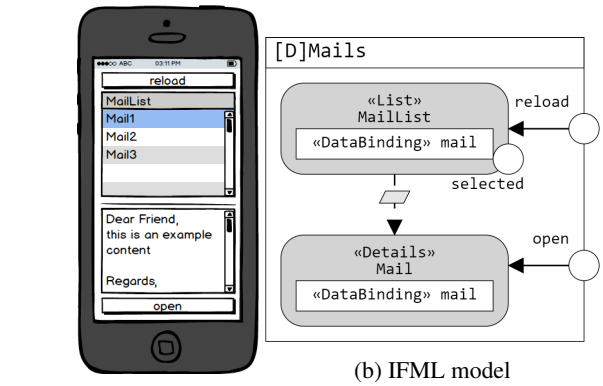
Note that rule 9 applies to ViewElements, i.e., to both ViewContainers and ViewComponents. A further example of NavigationFlow with a ViewContainer as target element appears in Figure 20.

Events and Data Flows

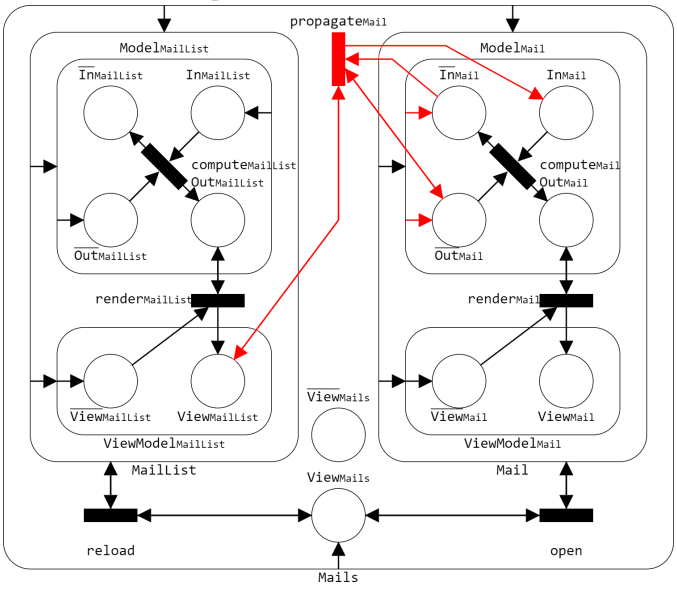
Figure 17 modifies the example of Figure 16, showing an alternative design pattern. In Figure 16 the selection of a mail message causes the immediate (i.e., synchronous) display of the mail content. Conversely, in the IFML diagram of Figure 17b, Mail selection occurs in two steps: first the user can (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 17 introduces the rule:



(a) Mockup (b) IFML model



(c) Place Chart Net

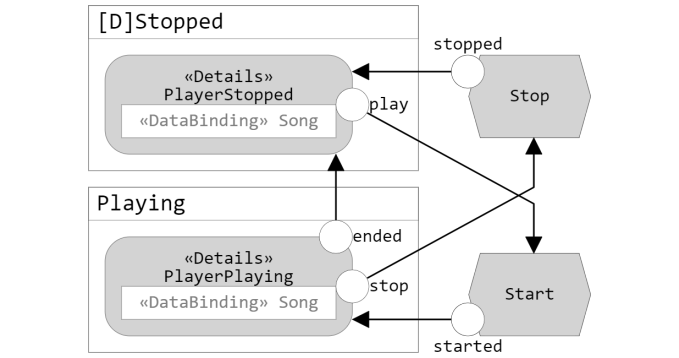
Figure 17: Navigation betweenViewComponents: data flow

Rule **10)** VIEWCOMPONENT INITIALIZATION - INCOMING DATAFLOWS A ViewComponent V, target of a non empty set $\mathcal{F}_V$ of DataFlows, maps into:
- initialization arcs from the parent that set the *clear* state of its model place charts (one arc adds a token to $\overline{\mathsf{In}_V}$ and one arc adds a token to $\overline{\mathsf{Out}_V}$).
- a transition $\mathsf{propagate}_V$ that removes a token from $\overline{\mathsf{Out}_V}$ and $\overline{\mathsf{In}_V}$ and adds a token to $\overline{\mathsf{Out}_V}$ and In_V; for each DataFlow F_i in $\mathcal{F}_V$, $\mathsf{propagate}_V$ also removes and adds a token into Views_{S_i}, where S_i is the source ViewComponent of F_i.

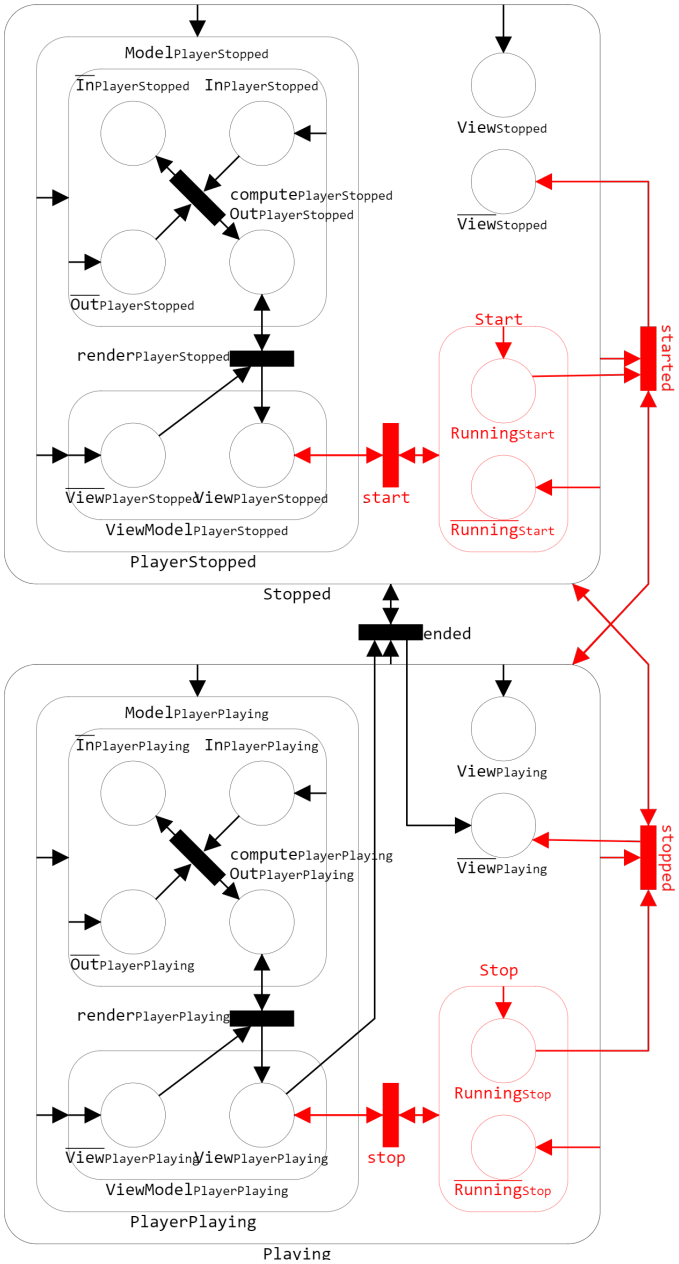
4.8. Mapping Actions

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 18a 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



(a) IFML model



(b) Place Chart Net

Figure 18: Action interaction

application when a song is playing; and the default *Stopped* ViewContainer 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 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 18a, 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 K child of O_K.
2) Two place charts Running_K and $\overline{\text{Running}_K}$ children of K. Running_K is initialized by default from the parent.
3) An initialization arc from O_K to $\overline{\text{Running}_K}$.

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 e which removes and adds a token from K and View_V.

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 Running_K and V and adds one token to $\overline{\text{Running}_K}$ and V.
2) If V is not O_K, e removes also a token from O_K and adds one token to $\overline{\text{View}_{O_K}}$.

5. Mapping Complex Interfaces

5.1. 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, Figure 19 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 19b 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 19c 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 $\text{View}_{\text{Mail}}$ bottom place chart, denoting that the nested ViewContainer gets into view at the same time as its parent.

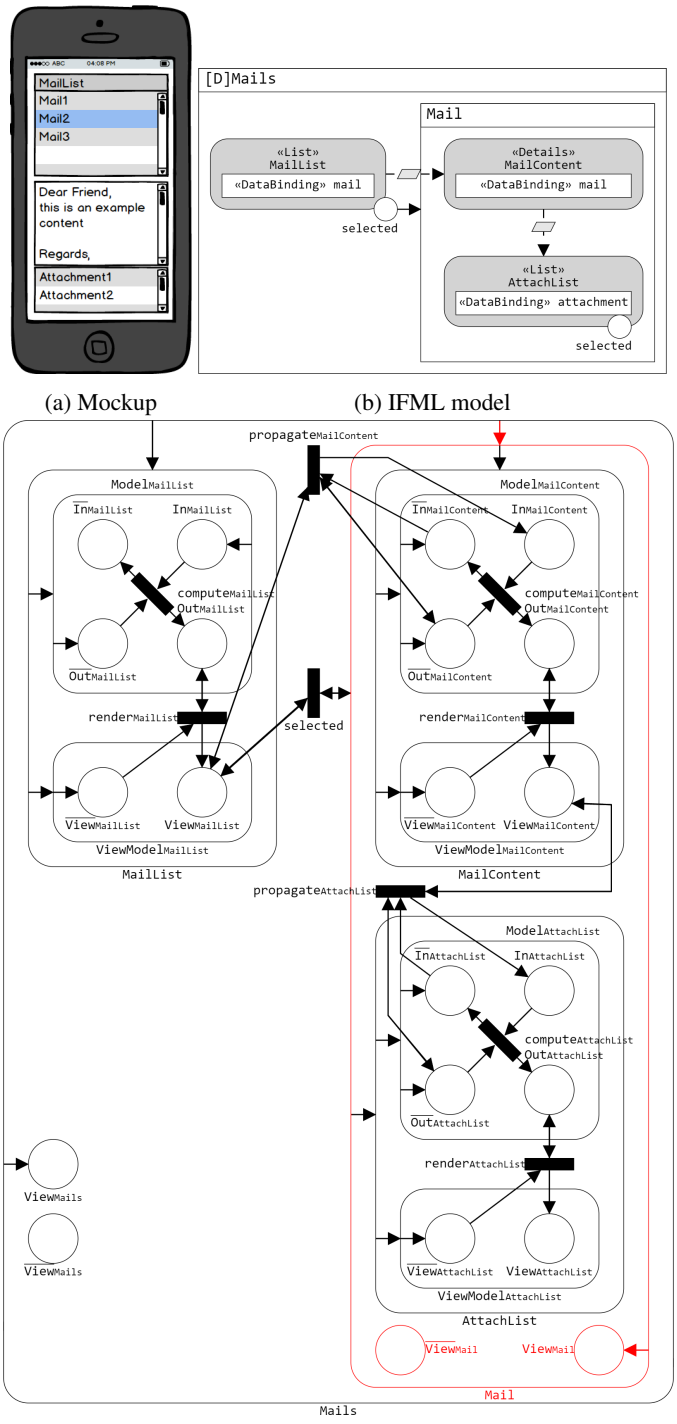
The content of the *Mail* ViewContainer is mapped as per rules 7 and 10: two place charts named MailContent and AttachList map the corresponding view components, and the transitions $\text{propagate}_{\text{MailContent}}$ and $\text{propagate}_{\text{AttachList}}$ map the DataFlows incoming to the *MailContent* 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 $\text{View}_{\text{MailList}}$ 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 19 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 View_{C} and $\overline{\text{View}_{\text{C}}}$ within C.
3) An initialization arc from C to View_{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.



(a) Mockup (b) IFML model

(c) Place Chart Net

Figure 19: Nested ViewContainers

5.2. 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 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 20 shows an example of XOR ViewContainers. The mail application of Figure 19 is changed so that the *MailContent* and *AttachList* ViewComponents are computed and displayed one at a time. Such 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 20b shows the mapping of the IFML diagram of Figure 20a. 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.

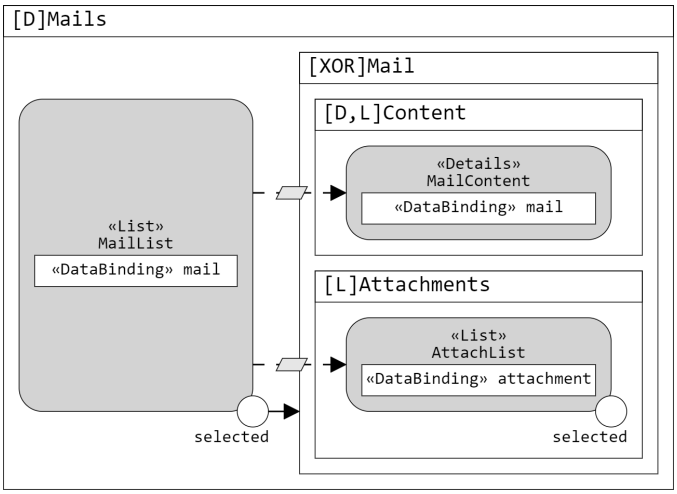
Both *Content* and *Attachments* are landmark ViewContainers, therefore two transitions ($\mathsf{landmark}_{\mathsf{Content}}$ and $\mathsf{landmark}_{\mathsf{Attachments}}$) are inserted, which follow the same configuration as for the rule of top-level landmarks (Rule 6): they remove a token from Mail 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 20 introduces the rules:

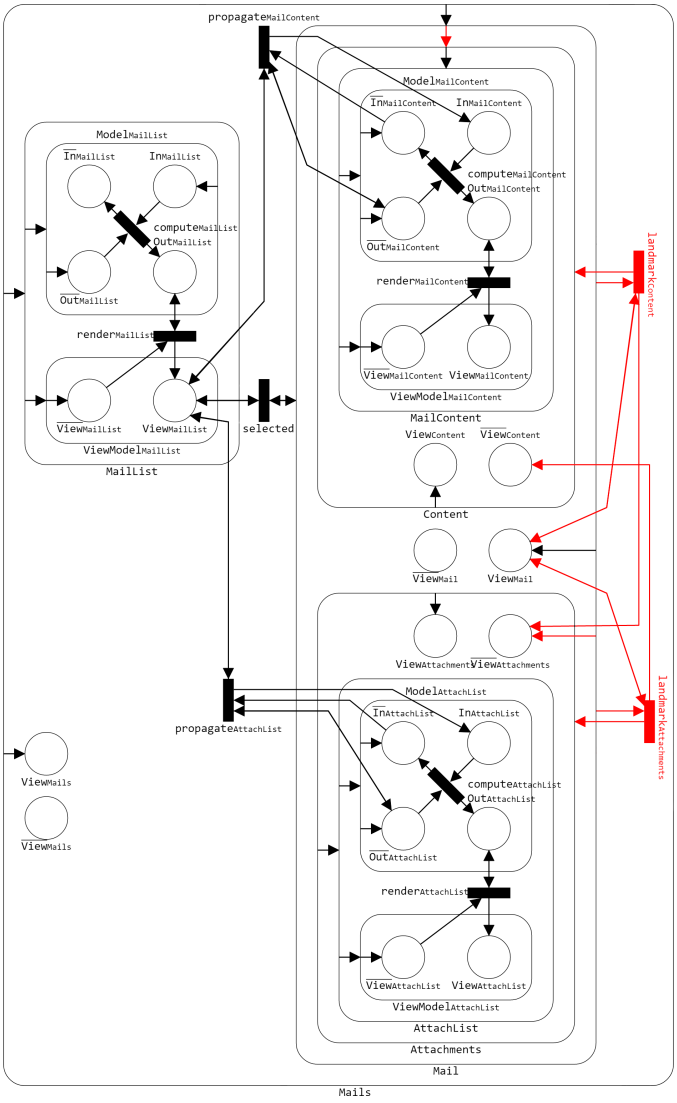
Rule **16)** XOR Default Child A default ViewContainer C child of a XOR ViewContainer P maps to an initialization arc from the place chart P to the place chart C. In Figure 20b the rule inserts the arc from Mail to Content.

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 20b 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 20b the rule inserts the transitions $\mathsf{landmark}_{\mathsf{Content}}$ and $\mathsf{landmark}_{\mathsf{Attachments}}$.



(a) IFML model



(b) Place Chart Net

Figure 20: Nested XOR ViewContainers

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 21a 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: $X0C0$, $X0C1$, $X1C0$ 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.

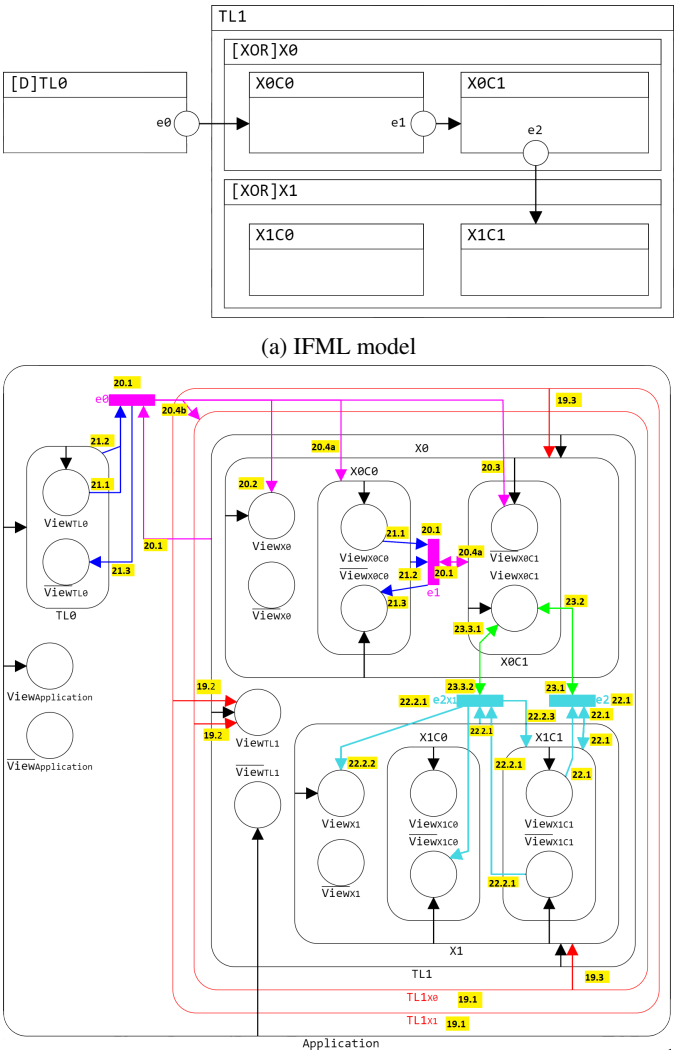
Definition 2. INTERACTION CONTEXT Given a NavigationFlow F, with source S and target T, the Interaction Context of F (I_F) is the ViewContainer V such that:
- $V = S$, if S is ancestor of T and is not a XOR ViewContainer;
- $V = T$, if T is ancestor of S and is not a XOR ViewContainer;
- V is the ancestor 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 21a, 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.

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 21a, 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 $X1$.



(a) IFML model

(b) Place Chart Net

Figure 21: Deeply Nested ViewContainers navigation

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 21a, 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:
- $\mathcal{T}^{\star A}_F = \mathcal{T}_F^A \cup \{A\}$, if A is a XOR ViewContainer
- $\mathcal{T}^{\star A}_F = \mathcal{T}_F^A$ 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 21a, 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, ($\mathcal{D}_F^A$) 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 21a, 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, ($\mathcal{H}_F^A$) 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 21a, 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 21a, 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 21a 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 21b.

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 V; this denotes that the ViewContainer V may be accessed by a navigation that targets X_i or one of its descendants; in Figure 21b, 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 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 V/X_i to C_j; this denotes that the ViewElements not affected by the navigation are initialized by default; this clause inserts the arc from TL1/X0 to X1 and from TL1/X1 to X0.
4) If an ancestor A_i of X_i child of V exists, an initialization arc from V/X_i to 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 $A_{i,j}/X_i$ for each ancestor of X_i, with an arc from $A_{i,j}/X_i$ to $A_{i,j+1}/X_i$, 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 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 21b shows the PCN mapping the IFML diagram of Figure 21a[5]. As per rule 19, the place chart TL1 is enclosed within the two auxiliary place charts TL1/X0 and 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 4.6. 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 21b, the transition e0, instead of adding a token to TL1, adds a token to the auxiliary place chart TL1/X0, thus avoiding to initialize by default X0. It also adds a token to $View_{X0}$, $\overline{View}_{X0C1}$ and X0C0, completing the initialization of $X0$ with the correct active child. Note that the transition mapping the NavigationFlow associated with the $e1$ event follows the rule for top-level ViewContainers (rule 5), as illustrated in Figure 13b, 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 like $e0$ 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 21b 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 $View_{X_i}$; this sets to visible all the affected XOR ViewContainers; this clause inserts the arc from e0 to $View_{X0}$ (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 $\overline{View}_{H_i}$; this sets the ViewContainers of the hide set to invisible; this clause inserts the arc from e0 to $\overline{View}_{X0C1}$ (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 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 D_i/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 $View_S$ to e; this ensures that the source is in view; in Figure 21b this clause inserts the arc from $View_{TL0}$ to e0 and from $View_{X0C0}$ 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 $\overline{View}_A$, if T is not an ancestor of S; this denotes that the co-displayed ancestor of the source gets out of

[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.

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 21b 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 4.6) and ViewComponents (in Section 4.7). 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:

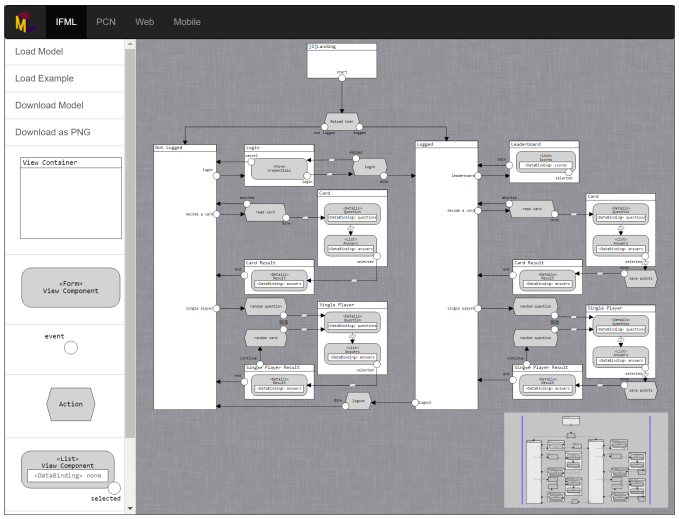
Rule **22)** Conditional Navigation Given a NavigationFlow F, associated with an event e and with target T, whose interaction context $I_F$is a non XOR ViewContainer, F maps to:
1) a transition e that removes/adds a token from/to T, and for each ViewContainer A_i ancestor of T and child of an element 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 21b, 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:
2.1) a transition e ▶ X_i that removes a token from X_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.
2.2) For each ViewContainer $X_{i,j}$ in the extended XOR targets set of F inside X_i, an arc from e ▶ X_i to $\text{View}_{\text{X}_{\text{i,j}}}$; 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 View_{X1}.
2.3) 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 $\text{D}_{\text{i,j}}/\text{D}_{\text{i,k}}$; if such $D_{i,k}$ does not exist, the arc goes from e ▶ X_i to $\text{D}_{\text{i,j}}$; this maps the computation of the elements of the display set inside X_i; this clause inserts the arc from e2 ▶ X1 to X1C1.
2.4) For each element $H_{i,j}$ of the Hide Set of F inside X_i, an arc from e ▶ X_i to $\overline{\text{View}}_{\text{H}_{\text{i,j}}}$: 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[6] and with target T, whose Interaction Context $I_F$is not a XOR ViewContainer. F maps to:
1) an arc from 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 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:
3.1) an arc from View_S to e ▶ X_i; this clause inserts the arc from $\text{View}_{\text{X0C1}}$ to e2 ▶ X1.
3.2) an arc from e ▶ X_i to View_S, if T is not an ancestor of S; this clause inserts the arc from e2 ▶ X1 to $\text{View}_{\text{X0C1}}$.

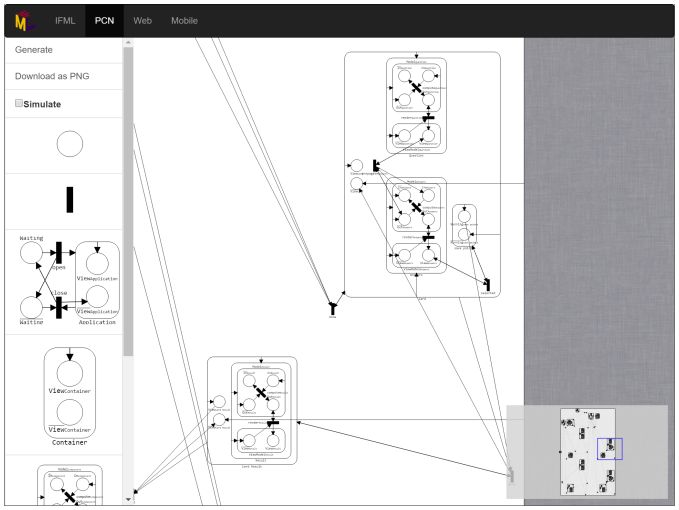
We conclude the presentation of the mapping of IFML to PCN by underlying that the adherence of the mapping rules to the intended semantics of IFML 1.0 has been verified, albeit only informally; indeed, the IFML 1.0 specifications describe the behavior of the language only by means of examples described 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 <5>. The evaluation has been conducted on 13 test cases.[7] 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. 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] A similar rule, omitted for brevity but implemented in the on-line system, is defined for events associated with Actions.

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

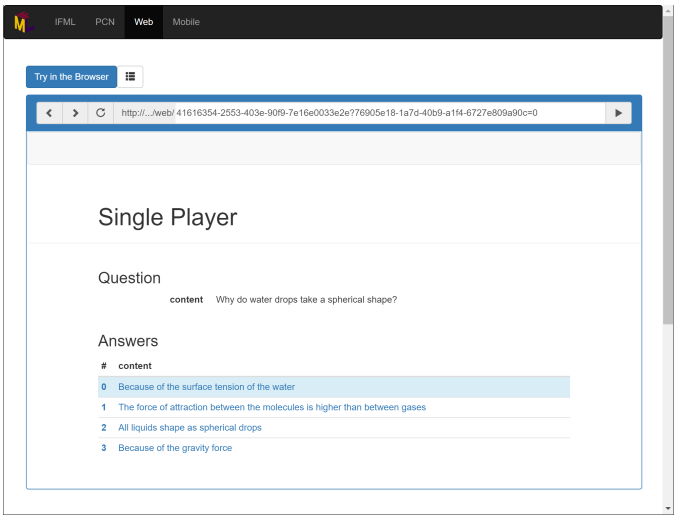


(a) IFML model Editor

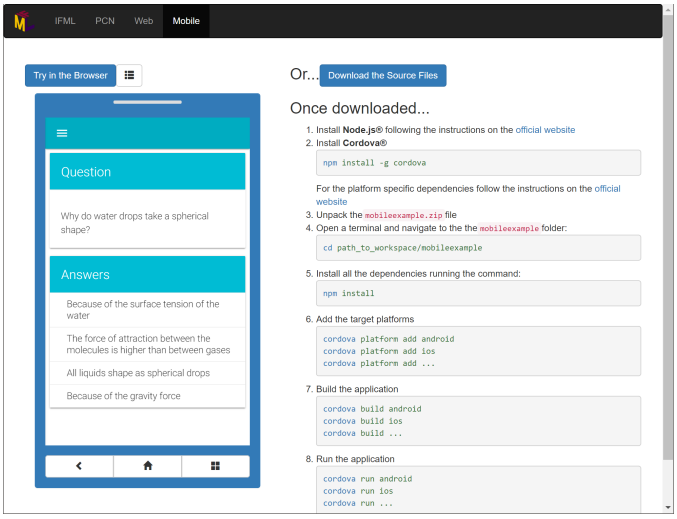


(b) IFML model semantic mapping to PCN

Figure 22: Modeling



(a) Web



(b) Mobile

Figure 23: Examples of generated applications

6. Tool Support

The described semantic mapping is implemented in IFMLEdit.org <50; 51>, an on-line web tool whereby developers can create an IFML model and derive the equivalent PCN according to the rules defined in Section 4. 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.

From the IFML model developers can generate the implementation code for the web or for a cross-platform mobile language, execute the obtained prototype, verify its compliance to requirements and possibly perform multiple model & generate iterations, until a satisfactory version is produced. As a last step, the automatically generated code can be turned into a real application, by customizing the look&feel and by replacing the mock-up data access and action calls with real API calls.

Figure 23a shows the generated web prototype, running in the web server emulated inside the browser. Figure 23b shows the generated mobile prototype, running in the mobile emulator embedded within the browser. In-browser emulation allows the developer to test the current web or mobile release without installing any web server and also in absence of the Internet connection.

The generated implementation uses sample data collections automatically created, to mimic the data binding of ViewComponents and populate the UI, allowing the developer to test the interface without the need of loading content. For more realistic tests, the developer can edit, add or remove entities from the automatically generated sample database.

IFMLEdit.org can be used iteratively to evaluate alternative application interface designs and interaction patterns. The code of the generated prototype can be downloaded and refined to produce the final application.

IFMLEdit.org is not the only free and open source IFML editor. Another open source project[8], based on the Eclipse

[8] http://ifml.github.com

Modeling Framework (EMF) <52> 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.

7. Discussion, Conclusions, and Future Work

Generative MDD approaches rely on the availability of modeling languages supported by effective tools. A prerequisite for code generation is that the language semantics be formally specified, so that the outcome of code generation can be anticipated by developers and models can be ported across tools ensuring that the behavior of the generated implementation remains consistent to the language semantics. This is not always the case with commercial MDD tools, which either employ proprietary notations whose semantics is not publicly available, or, when they adopt a standard yet informally defined language such as IFML, embed the interpretation of the language in the code generation rules.

To put our definition of the IFML semantics to work, we have examined two cases: 1) aspects of IFML underspecified in the standard documentation <5>; 2) interpretations of IFML constructs in the WebRatio commercial tool that embody unnecessary restrictions or assumptions about the language semantics.

The PCN mapping rules defined in this paper formalize and refine the informal semantics described in <5> by means of examples and of an informal ViewElement computation algorithm (contained in Section 9 of the specifications). The following scenarios are clarified by the mapping from IFML to PCN:

- Navigation of nested interfaces: the IFML specifications do not prescribe exactly what happens after a navigation event (due to a NavigationFlow, a Landmark, or the termination of an Action) in nested ViewContainers. The rules in Section 5 map the application behavior in response to every possible event affecting an interface structure made of arbitrarily nested ViewContainers and XOR ViewContainers. Specifically, the rules dictate which ViewElement must be brought in view and out of view; for those that are in view after the event, they show if their content must be recomputed or just redisplayed.

- ViewComponent computation: Section 9 of the IFML specifications provides an informal algorithm that describes how the computation of an IFML-specified application should proceed along linked ViewElements; the mapping rules of Section 4.7 expose precisely the workflow for computing the model and the view model parts of any combination of connected or standalone ViewComponents; such workflow is described explicitly in the PCN by means of the *Compute* and *Propagate* transitions.

- Local events: local events are events that affect only the ViewComponent to which they are associated, such as the Select event of Figure 17c. Our semantic mapping shows that such events do not map to a state changing transition, but only affect the output parameter of the ViewComponent.

The analysis of the code generator of WebRatio permitted us to identify a few scenarios where the tool takes unnecessarily restrictive interpretations of the language.

- DataFlow and NavigationFlow: WebRatio somehow blurs the distinction between the two constructs and uses a non standard feature (called automatic NavigationFlows) to express patterns in which a ViewContainer comprises one ViewComponent (e.g., a List ViewComponent) connected to another dependent ViewComponent (e.g., a Details to show an object from the list). WebRatio automatic NavigationFlows are used to express that at the first access to the ViewContainer the Details ViewComponent displays a (randomly chosen) object from the List ViewComponent. This pattern can be expressed using only the standard IFML constructs DataFlow and NavigationFlow, as shown e.g., in Figure 19. This is possible because the semantics of the two constructs has been better separated: the DataFlow expresses the input-output data dependency between the source and the target ViewElements, whereas the NavigationFlow expresses the triggering of the (re)computation of the target ViewElement.

- Nested containers: in WebRatio a ViewContainer nested within another ViewContainer has solely a presentation purpose (e.g, the code generator may produce a separate interface module for the embedded ViewContainer); conversely, the semantic mapping clarifies that nested ViewContainers can be used as first-class citizens of the IFML model, to denote that the ViewElements contained in them are selectively invalidated and recomputed after the occurrence of an event.

7.1. Limitations

The semantic mapping discussed in Section 4 and 5 focuses on the essential features of IFML that determine the execution behavior in reaction to events generated by the user, by the system, or by the termination of business actions. Several elements of the language, which are of practical use but do not influence the way events are handled, have been omitted. These include, e.g., several details about the ViewComponents, such as the different types of fields that can be added to a Form ViewComponent, the DataBinding element, which specifies the Domain Model element used to retrieve the content displayed in a ViewComponent, or the ConditionalExpression element, which is a query associated with a DataBinding for retrieving the actual content to display. Such omitted elements can be specified in the IFMLEdit.org model editor and are exploited for the web and mobile code generation, but do not influence the semantic mapping from the IFML model to the equivalent PCN.

Furthermore, the algorithm that describes the IFML computation in Section 9 of the standard also considers two specific aspects of the IFML execution semantics that depend on the actual values of the parameters: 1) the policy for resolving non determinism when a ViewComponent receives multiple values of the same parameter from different DataFlows or NavigationFlows; 2) the policy for deciding if the user's choices accumulated in previous interactions should be reused when re-computing the content of a ViewElement. Also these aspects impact the values shown in the interface, but not the way in which the application reacts to event. Therefore, they do not affect the PCN mapping, but only the generation of the code. The web and mobile code generators of IFMLEdit.org adopt the following choices, among those declared as possible in the specifications:

- Parameter propagation adopts the *non-deterministic choice*: one input parameter is chosen non-deterministically at run-time among the available ones.
- Navigation history preservation adopts the policy to re-compute the content of ViewComponents considering input parameter values determined by previous interactions with the same component.

7.2. Future work

The ongoing and planned future work addresses the following improvements:

- Semantic mapping: we will extend PCN with value-based tokens and a guard language, to allow the explicit representation of IFML aspects not treated so far, such as the behavioral properties of ViewComponents depending on their DataBinding and on the value of input parameters.
- Model checking: we plan to apply PCN analysis methods, such as reachability and liveness analysis, to enable the verification of IFML models thanks their semantic mapping to a formal model.
- Simulation: we plan to lift the level of simulation from the PCN to the IFML model, allowing developers not interested in the semantic mapping to be able to test the behavior of the front end on sequences of events automatically generated by the underlying semantic PCN representation.
- Code Generation: we plan to better encapsulate the parts of the code that are mocked-up by the code generator and thus require manual refinement, such as the API calls for content retrieval and for executing the business logic of Actions. The ultimate goal is to have the generator automatically merge the portions of code created automatically and programmed manually, so that the regeneration of the project preserves all the modifications made by the developer.
- Case studies and practical validation: we have already started using IFMLEdit.org in MDD educational programs and we plan to conduct a case study comparing the types of development activities and the effort distribution across the activities during the development of a sample application by two groups of randomly allocated developers using either the manual coding or the generative MDD approach.

Acknowledgments

The authors would like to thank WebRatio for the fruitful discussions on the tool code generation rules. This work has been partially supported by the FESR Project Empower of Regione Lombardia and of the European Community.

Biography

Carlo Bernaschina is a PhD Candidate of Web Technologies at DEIB, Politecnico di Milano, Italy. His main research interests concern software engineering, methodologies and tools for Mobile and IoT application development, with a focus on code generation from software models and agile methodologies.

Sara Comai is associate professor at the Dipartimento di Elettronica e Informazione of Politecnico di Milano since March 2007. She received her Ph.D. in Ingegneria Informatica e Automatica from Politecnico di Milano on January 13, 2000. Her recent research interests include the specification, design and automatic generation of complex Web applications.

Piero Fraternali is full professor of Web Technologies at DEIB, Politecnico di Milano, Italy. His main research interests concern software engineering, methodologies and tools for Web and mobile application development, with a focus on code generation from software models, multimedia content processing, and augmented reality mobile applications.

References

[1] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, 1st Edition, Morgan & Claypool Publishers, 2012.

[2] R. Elmasri, S. Navathe, Fundamentals of Database Systems, 6th Edition, Addison-Wesley Publishing Company, USA, 2010.

[3] S. Barnett, R. Vasa, J. Grundy, Bootstrapping mobile app development, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, IEEE Press, 2015, pp. 657–660.

[4] T. Clark, P. Sammut, J. Willans, Applied metamodelling: a foundation for language driven development, arXiv preprint arXiv:1505.00149.

[5] OMG, Interaction flow modeling language (IFML), version 1.0, http://www.omg.org/spec/IFML/1.0/ (2015).

[6] WebRatio platform, http://www.webratio.com/, accessed: 2017-11-06.

[7] M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, A. Yakovlev, Coupling asynchrony and interrupts: Place chart nets, in: P. Azéma, G. Balbo (Eds.), Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings, Vol. 1248 of Lecture Notes in Computer Science, Springer, 1997, pp. 328–347.

[8] P. Fraternali, S. Comai, A. Bozzon, G. T. Carughi, Engineering rich internet applications with a model-driven approach, ACM Trans. Web 4 (2) (2010) 7:1–7:47.

[9] N. Mellegård, A. Ferwerda, K. Lind, R. Heldal, M. R. V. Chaudron, Impact of introducing domain-specific modelling in software maintenance: An industrial case study, IEEE Trans. Software Eng. 42 (3) (2016) 245–260.

[10] Y. Zhang, S. Patel, Agile model-driven development in practice, IEEE Softw. 28 (2) (2011) 84–91.

[11] T. Stahl, M. Voelter, K. Czarnecki, Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2006.

[12] A. G. Parada, M. R. S. Marques, L. B. de Brisolara, Automating mobile application development: UML-based code generation for Android and Windows Phone, RITA 22 (2) (2015) 31–50.

[13] F. A. Kraemer, Engineering Android applications based on UML activities, in: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems, MODELS'11, Springer-Verlag, 2011, pp. 183–197.

[14] R. Francese, M. Risi, G. Scanniello, G. Tortora, Model-driven development for multi-platform mobile applications, in: Product-Focused Software Process Improvement - 16th International Conference, PROFES 2015, Bolzano, Italy, December 2-4, 2015, Proceedings, 2015, pp. 61–67.

[15] A. Sabraoui, M. E. Koutbi, I. Khriss, GUI code generation for Android applications using a mda approach, in: Complex Systems (ICCS), 2012 International Conference on, 2012, pp. 1–6.

[16] L. Gaouar, A. Benamar, F. T. 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, ACM, New York, NY, USA, 2015, pp. 19:1–19:5.

[17] H. Heitkötter, T. A. Majchrzak, H. Kuchen, Cross-platform model-driven development of mobile applications with md2, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, ACM, New York, NY, USA, 2013, pp. 526–533. doi:10.1145/2480362.2480464.
URL http://doi.acm.org/10.1145/2480362.2480464

[18] C. Jones, X. Jia, The AXIOM model framework - transforming requirements to native code for cross-platform mobile applications, in: ENASE 2014 - Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering, Lisbon, Portugal, 28-30 April, 2014, 2014, pp. 26–37. doi:10.5220/0004882100260037.

[19] C.-K. Diep, Q.-N. Tran, M.-T. Tran, Online model-driven IDE to design GUIs for cross-platform mobile applications, in: Proceedings of the Fourth Symposium on Information and Communication Technology, SoICT '13, ACM, New York, NY, USA, 2013, pp. 294–300.

[20] F. Inc., The Forrester Wave: Mobile Low-Code Development Platforms, Q1 2017, Tech. rep. (03 2017).

[21] G. Inc., Magic Quadrant for Enterprise High-Productivity Application Platform as a Service, Tech. rep. (04 2017).

[22] Mendix platform, https://www.mendix.com/, accessed: 2017-11-06.

[23] Outsystems platform, https://www.outsystems.com/, accessed: 2017-11-06.

[24] B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France, G. Karsai, Challenges and directions in formalizing the semantics of modeling languages., Comput. Sci. Inf. Syst. 8 (2) (2011) 225–253.

[25] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, A. Pierantonio, Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs (Apr 2006).
URL https://hal.archives-ouvertes.fr/hal-00023008

[26] A. Gargantini, E. Riccobene, P. Scandurra, A semantic framework for metamodel-based languages., Autom. Softw. Eng. 16 (3-4) (2009) 415–454.

[27] T. Mayerhofer, Using fUML as semantics specification language in model driven engineering, in: 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., 2013, pp. 87–93.

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

[29] F. Kordon, Y. Thierry-Mieg, Experiences in model driven verification of behavior with UML, in: Proceedings of the 15th Monterey Conference on Foundations of Computer Software: Future Trends and Techniques for Development, Monterey'08, 2010, pp. 181–200.

[30] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, M. Wimmer, Explicit transformation modeling, in: Models in Software Engineering, Workshops and Symposia at MODELS 2009, Revised Selected Papers, Vol. 6002, 2010, p. 240–255.

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

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

[33] Alf 1.0 Specification, http://www.omg.org/spec/ALF/, accessed: 2017-11-06.

[34] S. Comai, P. Fraternali, A semantic model for specifying data-intensive web applications using WebML, in: SWWS, 2001, pp. 566–585.

[35] T. Murata, Petri nets: Properties, analysis and applications., Proceedings of the IEEE 77 (4) (1989) 541–580.

[36] K. Jensen, Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1, Springer Publishing Company, Incorporated, 2010.

[37] J. Ding, W. Song, D. 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, 2014, pp. 725–732.

[38] V. Gruhn, A. Koehler, Modeling communication behaviour 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, 2006, pp. 147–158.

[39] CPNTools, http://cpntools.org/, accessed: 2017-06-26.

[40] GTK project, https://www.gtk.org/, accessed: 2017-11-06.

[41] Qt project, https://www.qt.io/, accessed: 2017-11-06.

[42] Appcelerator platform, http://www.appcelerator.com/, accessed: 2017-11-06.

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

[44] Adobe Phonegap, http://phonegap.com/, accessed: 2017-11-06.

[45] Rhomobile suite, http://rhomobile.com, accessed: 2017-11-06.

[46] Salesforce platform, https://www.salesforce.com, accessed: 2017-11-06.

[47] Telerik appbuilder, http://www.telerik.com/platform/appbuilder, accessed: 2017-11-06.

[48] Xamarin platform, https://www.xamarin.com/, accessed: 2017-11-06.

[49] Flutter project, https://www.flutter.io/, accessed: 2017-11-06.

[50] C. Bernaschina, S. Comai, P. Fraternali, IFMLEdit.Org: Model driven rapid prototyping of mobile apps, in: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 207–208.

[51] C. Bernaschina, S. Comai, P. Fraternali, Online model editing, simulation and code generation for web and mobile applications, in: Proceedings of the 9th International Workshop on Modelling in Software Engineering, MISE '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 33–39.

[52] Eclipse modeling framework (emf), https://www.eclipse.org/modeling/emf/, accessed: 2017-11-06.