Bridging IFML and Elm Applications via a Normalized Systems Expander

Jan Slifka[a] and Robert Pergl[b]
Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00 Prague 6, Czech Republic
fi

Keywords: Normalized Systems Theory, Interaction Flow Modeling Language (IFML), Model-Driven Code Generation.

Abstract: Web front-end applications are essential for delivering smooth user experiences across a multitude of platforms and devices. However, these applications often face difficulties maintaining long-term evolvability as user demands and stakeholder expectations continue to shift. In this paper, we propose using the Interaction Flow Modeling Language (IFML) to design applications and then generating source code in Elm, a statically typed, pure functional programming language tailored for web frontends. By applying Normalized Systems Theory, we aim to ensure long-lasting stability in two key ways: first, by defining how the resulting source code should align with the theory's principles; second, by employing expanders to generate code and incorporating a harvesting mechanism that allows custom modifications to the generated source without losing the connection to the original model. We demonstrate the practical application of our approach by designing an application using IFML models, introducing custom code, and regenerating the application from an updated model while preserving those customizations. Our contribution is a novel methodology that integrates IFML, Elm, and Normalized Systems Theory to improve the stability and maintainability of web front-end applications.

1 INTRODUCTION

In today's digital landscape, web front-end applications are an integral part of the ecosystem, enabling interactive and engaging user experiences on multiple platforms. However, their maintenance and scalability often become significant challenges as they become more complex. User expectations evolve rapidly, and the effort required to keep up with these changes can lead to cumbersome code management. Over time, the difficulty of integrating new features or adapting to shifting requirements can require extensive revisions or even complete code overhauls (Dvořák & Pergl, 2022).

We propose using Interaction Flow Modeling Language (Brambilla & Fraternali, 2015) models to design applications and then generate source code in Elm (Czaplicki, 2020), a statically typed, pure functional programming language tailored for web frontends. However, this approach extends beyond a simple code generator. By applying Normalized Systems Theory, our objective is to ensure long-term stability in two key ways: first, by defining how the resulting source code should align with the theory's principles; second, by leveraging expanders to generate code and incorporating a harvesting mechanism that allows custom modifications in the generated source without losing the connection to the original model.

2 METHODOLOGY

Our methodology is grounded in design science (Hevner et al., 2004). First, in Section 1, we investigate the problem domain and identify the issues we aim to address – this forms the awareness cycle. Next, in Section 3, we explore potential approaches by leveraging Normalized Systems and related research, marking the suggestion cycle. In Section 4, we then engage in a series of design cycles, incrementally improving the transformation process from IFML to Elm to develop a functional solution. In Section 5, we enter the evaluation cycle, demonstrating how our solution operates in practice. Finally, in Section 6, we summarize our contributions.

[a] https://orcid.org/0000-0002-4941-0575
[b] https://orcid.org/0000-0003-2980-4400

Slifka, J. and Pergl, R.
Bridging IFML and Elm Applications via a Normalized Systems Expander.
DOI: 10.5220/0013470800003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 63-74
ISBN: 978-989-758-757-3; ISSN: 2184-2833

3 BACKGROUND

In this section, we examine the concept of Normalized Systems and their use of expanders to generate software applications from structured specifications. Next, we explore the Interaction Flow Modeling Language (IFML), a modeling language for user interfaces, and Elm, a programming language and framework for developing web front-end applications, as we aim to integrate these within the framework of Normalized Systems. Finally, we review related work on this topic to provide context and highlight the novelty of our approach.

3.1 Normalized Systems

Normalized Systems (NS) theory (Mannaert et al., 2016) offers general guidelines, grounded in rigorous mathematical proofs, for constructing evolvable systems. It applies software engineering principles, such as separation of concerns and data version transparency, to create a highly modular structure. Although applicable across various domains, its most significant impact is in software engineering. The so-called NS expanders are employed to generate enterprise information systems from specifications, ensuring that the resulting systems adhere to NS Theory (Oorts et al., 2014).

3.1.1 Normalized Systems Theory

Normalized Systems Theory (NST) defines system structures to ensure evolvability by advocating the creation of fine-grained modules. Evolvability issues often stem from tightly coupled components with numerous dependencies, where a small change in one module triggers widespread modifications in others, commonly referred to as combinatorial effects. These effects become more pronounced as the system grows. NST aims to eliminate or control these effects by adhering to the following principles:

- **Separation of Concerns:** This involves dividing a program into distinct sections, each addressing a specific concern. Each component should have a single purpose and only one reason for modification. Examples include multi-tier architecture, separation of cross-cutting concerns, and the use of messaging or integration buses.
- **Data Version Transparency:** Data entities exchanged with action entities must exhibit version transparency, ensuring that updates to data entities do not affect the actions using them, thereby eliminating combinatorial effects.
- **Action Version Transparency:** Action entities invoked by other actions must also exhibit version transparency, ensuring updates to actions do not impact dependent actions, further mitigating combinatorial effects.
- **Separation of States:** Interactions between action entities must maintain statefulness, necessitating a stateful workflow system. Stateless synchronous pipelines are not allowed.

3.1.2 Normalized Elements and Expanders

NST emphasizes a fine-grained modular structure to eliminate combinatorial effects. During the transformation of functional requirements into software primitives, distinct software elements are defined. For information systems, the following elements have been identified:

- *Data elements* for data variables and structures,
- *Task elements* for instructions and functions,
- *Flow elements* for control flow and orchestration,
- *Connector elements* for input/output commands,
- *Trigger elements* for periodic, clock-like controls.

In principle, any software program can be implemented using these elements by defining an element for each fundamental primitive – data, instructions, functions, and input/output commands.

NS expanders are tools designed to generate software elements based on requirements and other specifications, such as templates for target technologies. The output is a complete, ready-to-use enterprise application composed of fine-grained, modular software elements. However, not all aspects of the application can be generated automatically; some manual work is required. To accommodate this, the generated code includes anchors where custom code can be injected.

When requirements change, the application can be regenerated. Before regeneration, the expander collects the custom code from the anchor points (a process known as *harvesting*) and reinserts it into the newly generated application code. The rejuvenation process for NS applications is depicted in Figure 1.

3.2 Interaction Flow Modeling Language (IFML)

The Interaction Flow Modeling Language (IFML) (Brambilla & Fraternali, 2015) is a platform-independent modeling language designed to specify the content, user interactions, and behavior of front-end applications. It enables the modeling of front-end applications irrespective of the target technology

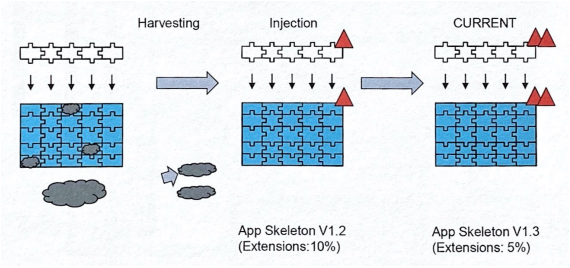

Figure 1: The rejuvenation of NS software (Mannaert et al., 2016).

or platform, addressing multiple aspects of front-end application design.

Composition of the View. IFML defines the visualization units composing the view, their organization, and whether they are displayed simultaneously or mutually exclusively. Each diagram includes one or more top-level *ViewContainers*, which may contain internal containers, forming a hierarchy. An example of container composition is shown in Figure 2.

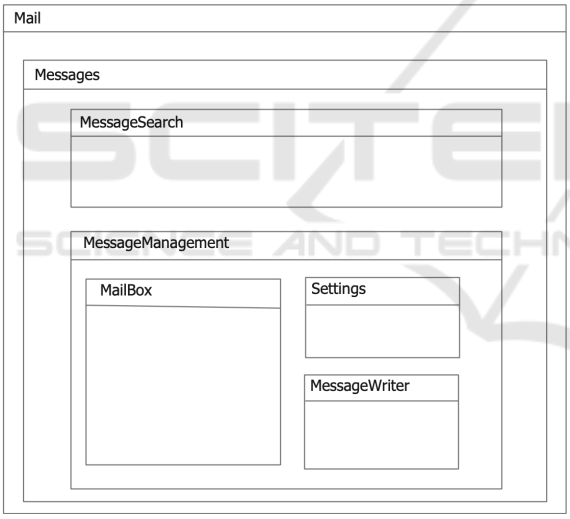

Figure 2: Example of IFML view containers and their composition (Brambilla & Fraternali, 2015).

Content of the View. IFML specifies the content elements, including data displayed to the user, input elements, and user-provided data. *ViewComponents* are used to present content (e.g., a list of elements) or to capture input data (e.g., a form). These components are placed within view containers. An example of various view components is shown in Figure 3.

Events. IFML models also define interaction events and the business components triggered by them. Events can be associated with *ViewContainers* or

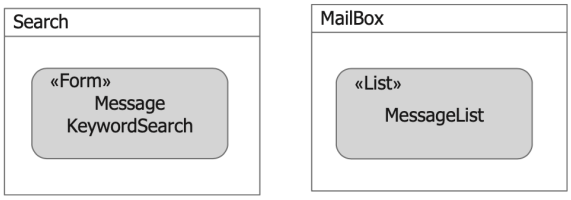

Figure 3: Example of different IFML view components within view containers (Brambilla & Fraternali, 2015).

ViewComponents that support user interaction. The effects of events are represented by the interaction flow, which links the events to the corresponding *ViewContainers* or *ViewComponents*, indicating changes in the user interface state. An event may also trigger an action executed before the user interface state is updated. An example of events related to actions is shown in Figure 4.

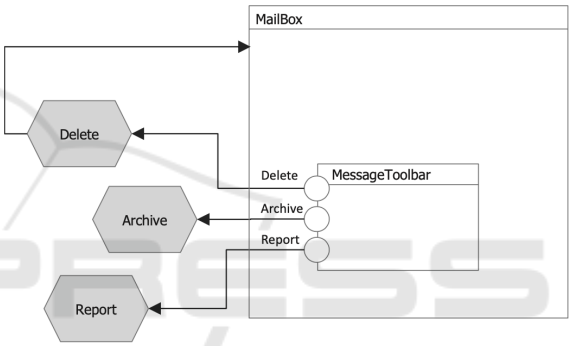

Figure 4: Example of events triggering business actions (Brambilla & Fraternali, 2015).

Parameter Binding. In the previous example, no specification was provided regarding the object on which the action should operate. Dependencies between inputs and outputs of view elements within the interaction flow, or between view elements and actions, can be defined using *parameter bindings*. Examples of parameter bindings are shown in Figure 5.

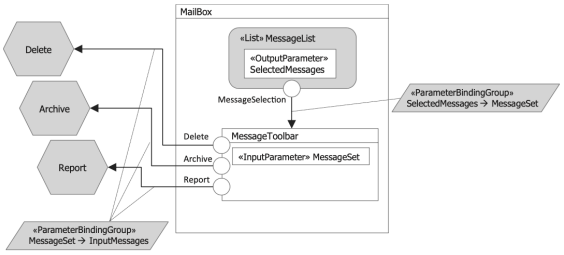

Figure 5: Example of IFML parameter bindings (Brambilla & Fraternali, 2015).

3.2.1 IFML Design Principles

IFML adheres to several design principles to balance the representation of complex front-end applications with usability and clarity:

- **Conciseness**: Minimize the number of diagrams needed to describe the user interface. With IFML's visual syntax, a single diagram suffices for the front-end model.
- **Inference from the Context**: IFML uses inference rules to deduce information from existing model elements, reducing the need for redundant input by modelers.
- **Extensibility**: IFML provides a core set of concepts for capturing interactions, which can be extended to accommodate specific use cases, such as new interface components or event types.
- **Implementability**: The importance of tooling–IFML supports representations such as XMI to enable model transformations and code generation.
- **Not everything in the model**: IFML focuses on semantics, deliberately omitting aspects such as presentation details. Other essential but peripheral aspects, such as those described in UML class diagrams, are delegated to external models.

Our goal is to leverage IFML to describe user interface requirements, providing a structured and detailed specification that can be utilized as input for the expander. By employing IFML, we aim to ensure that the defined requirements are both precise and platform-independent, facilitating the generation of modular and evolvable software elements in alignment with the principles of NST.

3.3 Elm

Elm (Czaplicki, 2020) is a statically typed, pure functional programming language recognized for its strong compile-time error detection. By compiling Elm code into JavaScript for browser execution, its compiler eliminates many common runtime errors typically associated with JavaScript. This approach improves application reliability and facilitates safe refactoring by offering clear, compiler-driven guidance to maintain functionality.

The high-level architecture of Elm, referred to as The Elm Architecture (TEA), is illustrated in Figure 6. Notably, everything in Elm is either a data type or a function. The architecture is composed of these primary components:

- **Model**: Represents the application's core data structure, defined as a data type.
- **View**: A function that maps the model to a virtual HTML structure, which the Elm renders into the actual HTML. The view also handles user interactions by translating them into messages.
- **Update**: A function that updates the model in response to received messages. It takes the current model and a message as input, producing an updated model as output. It can create commands to execute side effects, such as HTTP requests.
- **Subscriptions**: A function for handling events not directly linked to user interactions, such as timers or interactions with external JavaScript.

The Elm runtime invokes the update function in several scenarios: when handling messages from the view, processing the outcomes of commands (e.g., HTTP responses), or responding to subscriptions.

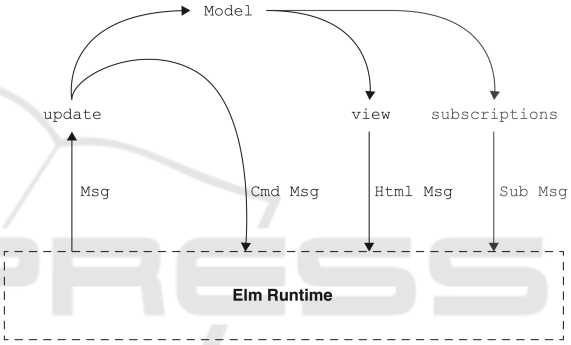

Figure 6: The Elm Architecture (Feldman, 2020).

Although Elm's architecture provides clarity and structure, the language retains flexibility in coding approaches. Numerous Elm patterns (Porto et al., 2024) have been introduced to enhance code quality and maintainability. By leveraging the Elm language and adopting appropriate patterns, we aim to generate Elm code as the output of the NS Expander, ensuring that the resulting code aligns with best practices for robustness and scalability.

3.4 Related Work

NST has been applied in various domains, such as building evolvable software (Oorts et al., 2014), developing design systems (Slifka & Pergl, 2020), and even improving document evolvability (Knaisl & Pergl, 2022). We extend this framework to focus specifically on web front-ends, employing Elm and IFML models as our foundation.

The challenge of generating software from models is not new and has been explored in several contexts, including domain-specific modeling (Kelly & Tolvanen, 2008), code scaffolding (Inayatullah et al.,

2019), and the generation of front-end code from UML (Brisolara et al., 2008). There are other approaches, such as Naked Objects (NO) (Pawson & Matthews, 2001), generating UIs directly from the domain model for rapid development. Additionally, IFML has been used to create CRUD applications (Rodriguez-Echeverria et al., 2019) and to generate Android UI components (Fatima et al., 2019).

Our approach stands out due to its integration of IFML and Elm, supported by NST and its expander framework. Unlike traditional code-generation methods, which typically result in a one-way process where modifications to the generated code break the model's connection, our approach employs a harvesting mechanism from NS. This allows developers to safely regenerate code from updated models while preserving custom code modifications, ensuring the system remains flexible and evolvable.

The Generation Gap pattern (Vlissides, 1998) promotes the separation of generated and handwritten code by placing them in different classes using inheritance. While NST adopts the principle of separation, it achieves extensibility through well-defined insertion points within the generated code, rather than relying on inheritance. This aligns well with our approach, as we employ Elm—a purely functional language that does not support object-oriented inheritance.

4 OUR APPROACH

An overview of our approach is illustrated in Figure 7. The process begins with an IFML Model as an input. To enable subsequent components to interact with the model, we implement a **Model Parser**, to be used in the stage, which involves a **Generator**, a component responsible for creating the resulting Elm application from the parsed model.

As detailed in Section 3.1.2, not all aspects of the application can always be generated directly from the model. In such cases, manual customizations may be necessary. To address this, we introduce a **Harvester**, which extracts these customizations before generating the next version of the application. The **Generator** then re-inserts these customizations into their appropriate locations in the newly generated code.

In the following subsections, we provide a detailed exploration of each step and explain the functionality of the individual components in practice. Our implementation is developed using the Python programming language, with the Jinja2 templating library employed for template-based code generation.

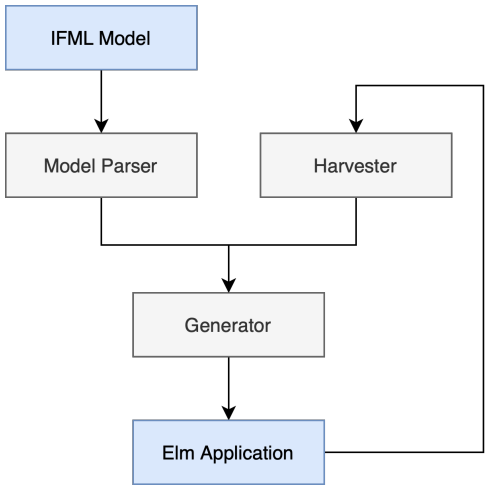

Figure 7: Overview of our approach.

4.1 Parsing and Interpreting the IFML Model

To begin, we require IFML models for the applications we intend to expand. For this purpose, we chose to use the online IFML editor, IFML-Edit.org (Bernaschina, 2020). One key advantage of this tool is that it stores models in a JSON format, which is easy to read and process programmatically.

The JSON representation includes all IFML entities and their relationships. These are parsed into corresponding Python structures, which are subsequently supplied to Jinja2 templates to facilitate the generation of Elm code.

4.2 Mapping IFML Components to Elm Application Structures

In this section, we will examine the IFML components in detail and outline our approach to transforming them into software primitives in Elm.

4.2.1 View Components

View Components in IFML are designed to either present content (e.g., in a list view) or capture user input (e.g., through forms). Since these components are both encapsulated and stateful – for instance, they may load data to display or maintain a selected state in a list – we implement them using the **Nested TEA pattern** (Porto et al., 2024).

The Nested TEA pattern is commonly employed as applications grow larger, enabling the separation of individual parts into dedicated modules. This aligns with the separation of concerns design principle in NST, as outlined in Section 3.1.1. Each module in the Elm architecture (described in Section 3.3) is created

for the component to focus on its specific concerns, such as fetching and displaying list of items from an API. These modules are then integrated into the main application at the corresponding components.

The source code below illustrates the structure of a child module implemented using the Nested TEA pattern for the components. Notably, the `Config` type is utilized to manage global application configurations, such as the API URL and other shared settings. Implementation details and additional module imports are omitted for brevity.

```
module Components.Component exposing (..)

import ...

type alias Model = { ... }

type Msg = ...

init : Config -> ( Model, Cmd Msg )
init = ...

update : Config -> Msg -> Model -> (Model,
↪  Cmd Msg)
update ...

view : Model -> Html Msg
view = ...
```

The following source code demonstrates how component types are integrated into the app module. The specific model and message types of the components are encapsulated within the app module's model and message types and are appropriately managed within the update and view functions.

```
module App exposing (..)

import Components.Component as
↪  Component

type alias Model =
  { config : Config
  , ...
  , componentModel : Component.Model
  }

type Msg
  = ComponentMsg Component.Msg
```

By adopting this modular approach, we can clearly delineate the responsibilities of each component, ensuring that the main application logic remains manageable and does not become overly complex. In the following subsections, we delve into individual view components and their specific implementations.

4.2.2 List

The List component in IFML is designed to display a collection of items, with its notation illustrated in Figure 8. This component has several specific properties. First, it requires defining the data binding – specifying the data collection from which the items are sourced and the fields to be displayed in the list. Additionally, the List component includes a selection event that can be linked to other components, such as a Details component, to display the details of the selected item.

Figure 8: IFML View Component List.

In the generated Elm source, in addition to the common component structure introduced earlier, the List component requires additional elements. Specifically, two properties are added to the model: the list of items and the selected item ID. The list of items is defined as `ActionResult (List a)`, where `a` represents the specific data entity defined by its fields (discussed further in a this section).

Since the items are fetched from an API, `ActionResult` is a custom type designed to represent the data retrieved from the API and its possible states. This type is defined using the following constructors:

```
type ActionResult a
  = Unset
  | Loading
  | Success a
  | Error String
```

Each state represents a different phase of the API call, allowing the UI to respond accordingly:

- **Unset**: The request has not been initiated.
- **Loading**: The request has been sent, and the application is awaiting a response.
- **Success**: The request was completed successfully, and the data has been retrieved from the API.
- **Error**: The request failed, and an error message is available.

The selected item uses the `Maybe` type to allow for the possibility of no selected item.

Other parts of the List component module are updated according to these model additions. The `init`

function initializes the new fields, setting the items to the `Loading` state, and generates a command for an API request based on the collection defined in the IFML model. Additionally, the model includes new messages to handle the completion of the item retrieval and to manage item selection within the list.

The `update` function needs additional complexity to handle interactions defined in the IFML model. For example, a list selection event can be connected to other components, such as displaying details when an item is selected. Consequently, when an item is selected, the component must also transmit this information if a selection event is connected.

To achieve this, we utilize the **Translator pattern** (Porto et al., 2024), which makes the child module generic and capable of producing parent messages. As a result, the `update` function is designed to accept an additional argument – a custom type called `UpdateConfig`. The implementation of this approach is outlined in the following source code:

```
type alias UpdateConfig msg =
  { wrapMsg : Msg -> msg
  , onSelect : Maybe (String -> Cmd msg)
  }

update : UpdateConfig msg -> Msg -> Model
 ↪  -> ( Model, Cmd msg )
update cfg msg model = ...
```

The `UpdateConfig` contains two key components. First, it includes a function to wrap module-specific messages (`Msg` type) into a generic `msg` type. Second, it provides an optional function that accepts the item ID and produces a generic command.

Within the `update` function, when an item is selected and this optional function is defined, the function generates a command to transmit the selection information to the corresponding details module. This mechanism ensures communication between the List component and other connected components.

The `view` function dynamically renders content based on the current state of the `ActionResult`. It displays a loader while data is being fetched, and subsequently shows either an error message if the request fails or the retrieved data presented in a table.

In addition to the component module, we generate a new data type representing the data displayed in the list. This data type is created based on the field configuration of the IFML component and is placed in its own Elm module, along with a corresponding `decoder`. In Elm, JSON data from an API cannot be used directly; it must first be decoded into the expected type, providing an additional layer of type safety. Below is an example of such a generated data type and its associated decoder:

```
type alias MailList =
  { id : String
  , subject : String
  }

decoder : Decoder MailList
decoder =
  D.succeed MailList
    |> D.required "id" D.string
    |> D.required "subject" D.string
```

Finally, the newly generated component is integrated into the main application. In addition to the common structures required to incorporate it into the main application's model, view, and update functions, additional elements are generated to support the component's update configuration. If a selection event is defined, a corresponding message and a handler function are generated. The handler ensures the selection event is processed by updating the relevant component in the application.

For instance, if a selection event is defined, an additional constructor for the application's `Msg` type is generated. The name of this constructor is derived from the event name specified in the IFML model:

```
type Msg
  ...
  | SelectedMsg String
```

Next, a handler function is generated within the application module to connect the selection event with the corresponding details component. This function ensures that when an item is selected, the relevant details component is updated accordingly. Below is an example of such a handler:

```
handleSelectedMsg : String -> Model
  -> ( Model, Cmd Msg )
handleSelectedMsg id model =
  let
    ( newMailModel, cmd ) =
      Mail.update model.config
                  (Mail.selectItemMsg id)
                  model.mailModel
  in
  ( { model | mailModel = newMailModel }
  , Cmd.map MailMsg cmd
  )
```

In this example, the handler updates the details component named `Mail` by using its `selectedItemMsg` message. The details component then responds by fetching the relevant data and displaying it. This process will be examined in greater detail in the next section.

4.2.3 Details

The Details component in IFML is used to display the details of a specific entity. Its key properties include the data collection from which the displayed entity is sourced and the fields of that entity to be presented. Typically, the Details component is linked to the selection event of a List component to determine which specific entity should be displayed. The notation for the Details component is shown in Figure 9.

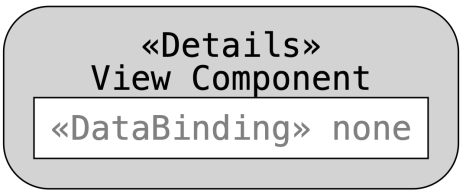

Figure 9: IFML View Component Details.

The generated Elm source code for the Details component includes additional elements beyond the standard component structure. Specifically, an additional field, `item`, is added to the model. This field is of type `ActionResult a`, where `ActionResult` (introduced earlier) represents the state of the data, and `a` is the specific type of the entity to be displayed.

The item is fetched from the API, but the request is not initiated until the component receives a selection message. As a result, the initial state of `item` in the model is set to `Unset`.

To handle selection, a custom `Msg` constructor is defined, along with a wrapper function exposed from the module. This allows the parent module to send selection messages to the Details component efficiently:

```
type Msg
  = SelectItem String
  ...

selectItemMsg : String -> Msg
selectItemMsg id =
    SelectItem id
```

To maintain encapsulation and adhere to the separation of concerns design principle, we do not expose the constructors of the `Msg` type outside of the module. Instead, we provide an additional function, `selectItemMsg`, which acts as a wrapper for the specific constructor we need to use externally. This approach ensures that only the intended functionality is accessible from outside the component module.

The `update` function's signature remains unchanged for this component, as it does not need to produce messages for the parent module. The function handles two scenarios: first, when an item is selected, it initiates an API request to fetch the data of the selected item; and second, it processes the result of the API request, updating the component accordingly.

The `view` function dynamically renders content based on the `ActionResult` in the component's model. It displays an empty state when the status is `Unset`, a loader while awaiting the request's result, and either the fetched data or an error message once the request is completed.

When the Details component is connected to a selection event, its `selectedItemMsg` is invoked with information about the selected item in the parent module. This triggers the Details component's `update` function, which retrieves the corresponding data from the API and displays it.

Additionally, a new data type module is generated, along with its decoder, to represent the data displayed in the Details component. This is implemented in the same manner as for the List component.

4.2.4 Form

The Form component is designed to capture user input, with its notation illustrated in Figure 10. Its configuration specifies the fields to be included in the form for data collection.

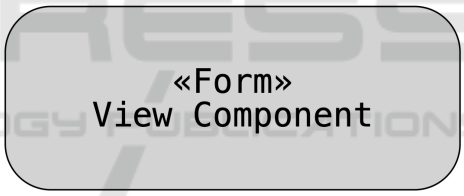

Figure 10: IFML View Component Form.

The generated Elm source code for the Form component includes additional fields in its model, derived from the configuration specified in the IFML model. It also introduces new messages for handling form inputs and displays the form within the view function. However, due to space constraints, we do not explore the Form component in greater detail in this paper.

4.2.5 View Container

View containers organize multiple components into cohesive application screens. These containers can be nested to create a hierarchical structure of views, as shown in Figure 11.

One of the top-level containers can be designated as the *Default* view, which serves as the initial application screen. Top-level containers can also be labeled as *Landmark*, indicating that they should be accessible from anywhere within the application, such as through a navigation menu. Nested containers may

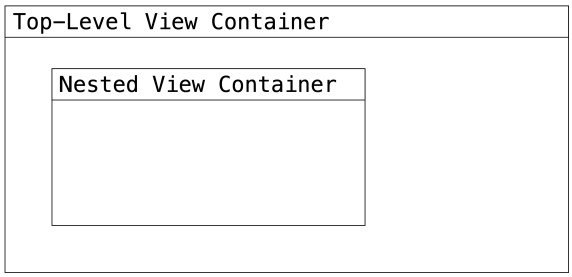

Figure 11: IFML View Container.

be marked as *XOR*, signifying that only one of these containers will be displayed at a time.

In the generated Elm code, view containers are implemented as functions that are invoked from the main application's view function. Each container function renders an HTML `div` element to house its components, and those components are mapped to their corresponding application messages. Below is an example of such a container:

```
viewMails : Model -> Html Msg
viewMails model =
  Html.div []
    [ Html.map MailListMsg <|
            MailList.view model.mailListModel
    , Html.map MailMsg <|
            Mail.view model.mailModel
    ]
```

The view function takes the application model as input, enabling it to pass the relevant component models to the nested component view functions.

4.3 Establishing Backend Integration

This study focuses on the frontend application, leaving backend implementation out of scope. To support the frontend, we use json-server (Typicode, 2023), a tool that enables the creation of a fully functional fake REST API from a JSON file. By defining a JSON file with the necessary collections derived from our IFML model, json-server generates a complete API that can be utilized by the frontend application.

On the client side, we configure the API URL for json-server and make requests from the application as needed, such as fetching items for the list view.

4.4 Harvesting and Reinjecting Custom Code for System Evolution

We recognize that not all code can be generated, and some manual customizations may be required. To address this, we first define specific locations within the generated application where developers can insert their custom code. These insertion points are marked by special comments generated by the code generator. Below is an example of such comments within a function for rendering a container view:

```
viewMails : Model -> Html Msg
viewMails model =
  Html.div
    [-- <customization Mails-attributes>
     -- </customization>
    ]
    [ Html.nothing
    -- <customization Mails-before>
    -- </customization>
    , Html.map MailListMsg <|
        MailList.view model.mailListModel
    , Html.map MailMsg <|
        Mail.view model.mailModel
    -- <customization Mails-after>
    -- </customization>
    ]
```

We use `Html.nothing` as the initial element for syntax purposes. This approach allows each subsequent line of customization to begin with a leading comma, ensuring the code functions correctly regardless of whether customizations are included.

The view container provides three locations for inserting custom code. First, we can add custom attributes, such as a CSS class for styling. Second, we can include elements before the components defined by the IFML model. Third, we can add additional components after the existing ones.

When the application is regenerated in the future with an updated IFML model, the harvester first extracts all customizations placed between the special comments. After the new version of the application is generated, these customizations are reinserted into their original locations, ensuring that previously written custom code is not lost. However, some manual updates may be necessary if changes in the model lead to modifications in the generated code. In some cases, customizations may become obsolete – for example, if a component that was customized is completely removed.

This same approach – using special comments to mark customizations, followed by harvesting and reinserting the code snippets – is applied to all other generated components and modules. If additional customization points are identified in the future, the templates can be extended to include these special comments in new locations, further enhancing the flexibility and functionality of our solution.

5 DEMONSTRATION

In this section, we evaluate our solution by building an application model in the IFML editor, expanding it, and introducing customizations to demonstrate its practical application. The focus is on an email-reading application. To begin, we set up a json-server, returning items with the following structure:

```
{
  "id": 1,
  "subject": "Meeting Reminder",
  "body": "Don't forget about the meeting at
  ↪  3 PM.",
  "from": "alice@example.com",
}
```

We begin with an IFML model consisting of a default container that wraps the entire application and a list component for emails. This list is configured to reference the *mails* collection, displaying the *from* and *subject* fields. The initial model is depicted in Figure 12, while the generated application's initial version is shown in Figure 13.

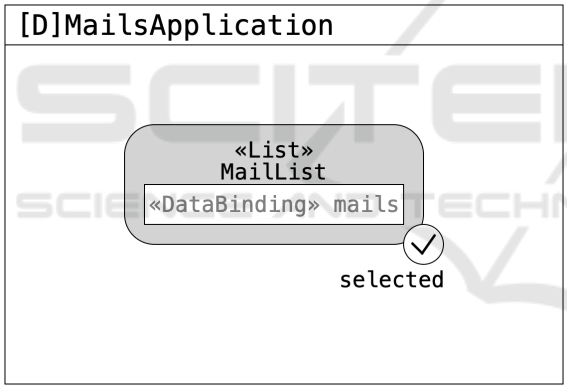

Figure 12: Default IFML model for our application.

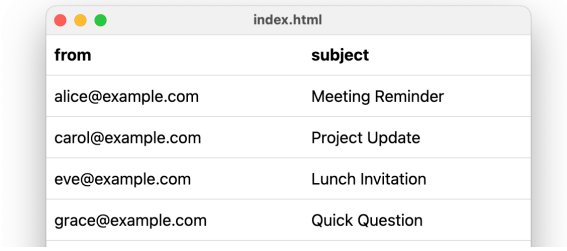

Figure 13: Generated application showing a list of emails.

In the next step, we introduce customizations. First, we add padding by specifying a CSS class in the customization block for attributes. Next, we add a heading to the application. The resulting code with these customizations is shown below:

```
viewMailsApplication : Model -> Html Msg
viewMailsApplication model =
  Html.div
    [-- <customization
    ↪  MailsApplication-attributes>
       Attributes.class "p-3"
     -- </customization>
    ]
    [ Html.nothing
    -- <customization
    ↪  MailsApplication-before>
    , Html.h1 []
        [ Html.text "Mails Application" ]
    -- </customization>
    , Html.map MailListMsg <|
    ↪  MailList.view model.mailListModel
    -- <customization
    ↪  MailsApplication-after>
    -- </customization>
    ]
```

The updated application, reflecting these changes, is shown in Figure 14.

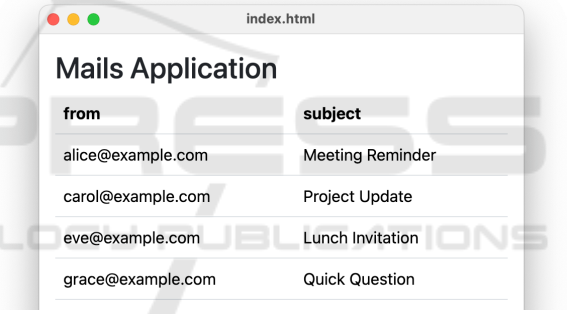

Figure 14: Generated application with custom code.

The next step is updating our IFML model by adding a Details view component to display the selected email's details. This component is configured to use the same collection as the list (*mails*) and is set to display the *from* and *subject* fields, as well as the email's *body*. The previously created list's selection event is then connected to this new Details component to ensure the selected list item appears in the detail view. This updated model is illustrated in Figure 15.

Next, we regenerate the application, a process in which the harvester first collects the previously added customizations. It then generates the new application from the updated model and reinserts the customizations. The resulting application is shown in Figure 16. The title and customized padding remain intact, and now we can select an email from the list to view its details, including the body content.

In this section, we illustrated the end-to-end process of creating an IFML model, using our expander to generate the corresponding application source

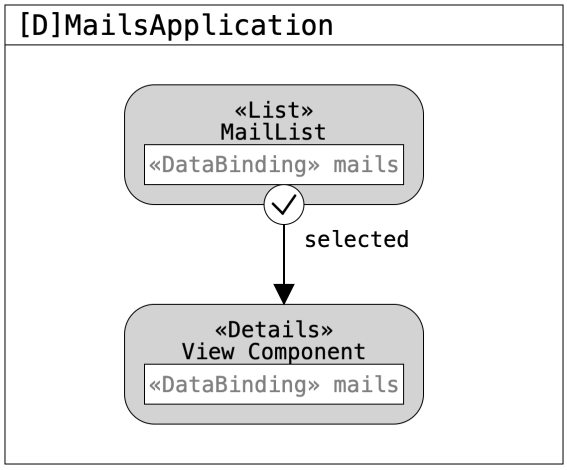

Figure 15: Default IFML model for our application.

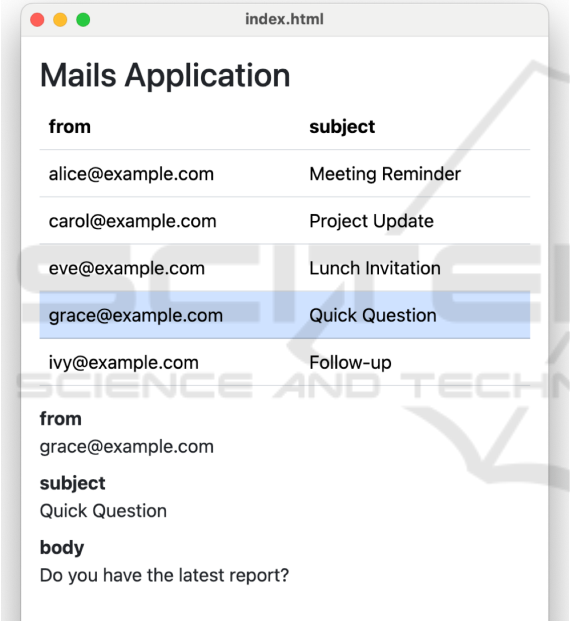

Figure 16: Generated application with custom code.

code, and then customizing the application as needed. We demonstrated how our approach allows developers to enhance and refine the application, introducing customizations to tailor it to specific requirements. Crucially, we showed that even when the underlying IFML model evolves, a new version of the application can be generated without losing these customizations. The harvester's ability to identify, extract, and reintegrate the developer's custom code ensures that the application remains consistent, flexible, and maintainable across iterative development cycles.

6 CONCLUSIONS

In this paper, we focused on integrating IFML and Elm, leveraging the principles of NS to ensure long-term evolvability for the resulting applications. We developed a model parser capable of interpreting the IFML model and preparing it for further processing. Additionally, we designed and implemented a code generator that produces Elm code consistent with the stable software design theorems outlined in (Mannaert et al., 2016). Beyond generating code, our approach also establishes clearly defined locations for custom modifications. These customizations are harvested and reinjected when regenerating the application from an updated version of the model, preserving the adaptability and maintainability of the system.

The primary advantages of our approach compared to traditional code generators include:

- Code generation that adheres to the stability design theorems of NS, ensuring long-term maintainability and evolvability.
- Integration of a harvesting process that supports custom code injections without breaking the model's integrity, enabling tailored solutions while preserving the model's connection to the generated code.

In Section 5, we illustrated the practical application of our solution by designing an application using IFML models, introducing custom code, and regenerating the application from an updated model while preserving those customizations.

Although our current approach uses the Elm programming language, it can be adapted to other languages as well. This would involve defining a new mapping from IFML components to the structures of the target language and framework, along with implementing a corresponding expander. The overall procedure, however, would remain largely unchanged.

6.1 Limitations

Our expanders currently do not cover all IFML features. We have implemented core functionality, but certain elements remain for future development.

Our approach is not a universal solution. While it offers a compelling method for generating web frontend applications from IFML models, it may not be appropriate for all scenarios. For instance, situations requiring highly specialized or unconventional user interfaces, or those that depend heavily on non-standard workflows, might not benefit as much from this model-driven approach.

IFML comprises a core and extensible extensions, enabling the addition of custom components

as needed. The same applies to expanders—once a transformation method and implementation are defined, they can be used as well. Thus, these limitations are, to some extent, addressable.

6.2 Future Work

The foundation established in this paper paves the way for several promising future improvements. One compelling direction would be integrating this solution with established design systems, ideally guided by the same principles of evolvability. For instance, approaches like those explored in (Slifka & Pergl, 2020) could be adapted to ensure that both the application's behavior and appearance evolve seamlessly. By aligning the visual elements with the principles of NS, it becomes possible to achieve a more cohesive and maintainable interface that evolves in tandem with underlying functionality.

An area for further enhancement is the support for cross-cutting concerns common in production systems, such as authentication. Given the absence of specialized IFML constructs for these functionalities, they must be implemented at the Elm level, potentially through standardized patterns or dedicated modules to ensure proper integration.

ACKNOWLEDGEMENTS

This research was supported by grant No. LM2023055 of the Ministry of Education, Youth and Sports of Czech Republic. It was conducted as part of our work at NSLab, CTU in Prague.

REFERENCES

Bernaschina, C. (2020). IFMLEdit.org. http://editor.ifmledit.org. [Accessed: 2019-08-22].

Brambilla, M. & Fraternali, P. (2015). *Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML*. Amsterdam: Morgan Kaufmann/Elsevier.

Brisolara, L. B., Oliveira, M. F. S., Redin, R., Lamb, L. C., Carro, L., & Wagner, F. (2008). Using UML as front-end for heterogeneous software code generation strategies. In *Proceedings of the Conference on Design, Automation and Test in Europe* (pp. 504–509). Munich Germany: ACM.

Czaplicki, E. (2020). The ELM Architecture. https://guide.elm-lang.org/architecture/. [Accessed: 2020-08-03].

Dvořák, O. & Pergl, R. (2022). Tackling rapid technology changes by applying enterprise engineering theories. *Science of Computer Programming*, 215, 102747.

Fatima, I., Anwar, M. W., Azam, F., Maqbool, B., & Tufail, H. (2019). Extending Interaction Flow Modeling Language (IFML) for Android User Interface Components. In R. Damaševičius & G. Vasiljevienė (Eds.), *Information and Software Technologies*, volume 1078 (pp. 76–89). Cham: Springer International Publishing.

Feldman, R. (2020). *Elm in Action*. Shelter Island, NY: Manning Publications Co.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, 28(1), 75–105.

Inayatullah, M., Azam, F., & Anwar, M. W. (2019). Model-Based Scaffolding Code Generation for Cross-Platform Applications. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)* (pp. 1006–1012). Vancouver, BC, Canada: IEEE.

Kelly, S. & Tolvanen, J.-P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Hoboken, N.J: Wiley-Interscience.

Knaisl, V. & Pergl, R. (2022). Improving Document Evolvability Based on Normalized Systems Theory. In A. Rocha, H. Adeli, G. Dzemyda, & F. Moreira (Eds.), *Information Systems and Technologies*, volume 469 (pp. 131–140). Cham: Springer International Publishing.

Mannaert, H., Verelst, J., & Bruyn, P. D. (2016). *Normalized Systems Theory from Foundations for Evolvable Software toward a General Theory for Evolvable Design*. Kermt: NSI-Press.

Oorts, G., Huysmans, P., De Bruyn, P., Mannaert, H., Verelst, J., & Oost, A. (2014). Building Evolvable Software Using Normalized Systems Theory: A Case Study. In *2014 47th Hawaii International Conference on System Sciences* (pp. 4760–4769). Waikoloa, HI: IEEE.

Pawson, R. & Matthews, R. (2001). Naked objects: A technique for designing more expressive systems. *ACM SIGPLAN Notices*, 36(12), 61–67.

Porto, S., Engels, J., Janiczek, M., Callea, A., & Torun, M. (2016/2024). Elm Patterns. https://sporto.github.io/elm-patterns/. [Accessed: 2024-04-10].

Rodriguez-Echeverria, R., Preciado, J. C., Rubio-Largo, Á., Conejero, J. M., & Prieto, Á. E. (2019). A Pattern-Based Development Approach for Interaction Flow Modeling Language. *Scientific Programming*, 2019, 1–15.

Slifka, J. & Pergl, R. (2020). Laying the Foundation for Design System Ontology. In *Trends and Innovations in Information Systems and Technologies*, volume 1159 (pp. 778–787). Cham: Springer International Publishing.

Typicode (2023). Json-server. https://github.com/typicode/json-server. [Accessed: 2025-01-16].

Vlissides, J. (1998). *Pattern Hatching: Design Patterns Applied*. The Software Patterns Series. Reading, Mass: Addison-Wesley.