UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Database and Information Systems

Master's Thesis

Submitted to the Database and Information Systems Research Group
in Partial Fullfilment of the Requirements for the Degree of

Master of Science

Enabling run-time state migration between same-purpose applications of different vendors

by
Saman Soltani

Thesis Supervisors:
Prof. Dr. Gregor Engels
Dr. Enes Yigitbas

Thesis Advisor:
Dennis Wolters

Paderborn, April 12, 2021

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Paderborn, April 12, 2021

Ort, Datum

Unterschrift

to my mom, Soghra Amiri.
who always wanted to see my success,
but she passed away middle of my thesis because of COVID-19.

Contents

1

Introduction

1.1 Motivation

In recent years, the popularity of portable devices had significantly grown. These devices can be laptops, tablets, smartphones, and even smartwatches. They have different Operating Systems (OS) and run various applications (Apps). Furthermore, because of the increased computing power of mobile devices, many desktop applications's vendors are providing mobile and tablet versions of their applications (e.g., Adobe Photoshop) [1]. With these applications, people can perform various tasks, and they may use different devices to accomplish them. Therefore, their lives become a multi-device experience. The analysis in [2] shows the market share of desktops, mobile phones , and tablet devices. The use of desktop devices had decreased by 56.75 percent from January 2009 to April 2020. However, the use of mobile devices had increased by 53.94 percent in the same period.

Users have to decide on which device they want to perform a certain task (e.g., sending an e-mail on mobile phone). The same task can be performed on a wider range of devices (e.g., sending the same e-mail on a Windows laptop). Switching the interaction between devices is a routine behavior from a user. This switch to another device can happen because of other devices are more comfortable in a specific situation (e.g., more desirable user experience or user interface, a larger screen, or even better sound quality on another device). Another reason for a switch can be when the switch is forced because the battery is drained or because another device offers a better specific functionality or even the need to have private data in a device that is not shared [3].

Currently, users are doing this migration mostly manually. For example, Firefox and Chrome serve the same purpose and both are web browsers. Consider a user who wants to read an article on a website. If she only uses Chrome on all her devices, she could switch between devices with the Tab Synchronization feature of Chrome. However, if she wants to continue reading the article by switching from desktop PC with Chrome to an Android device which only has Firefox, she has to bring Chrome's current state to Firefox manually (e.g., set the address of the article, and find the exact page or scroll position). All of these steps are done manually because applications are from different vendors, and they do not support Tab Synchronization between each other. Users who want to switch between devices in the middle of their work do not like it if they have to start their tasks again. Instead, they appreciate continuing a task seamlessly and effortlessly with their past data on the other device [4, 5].

At the moment, some solutions are designed for a specific platform, ecosystem, or appli-

cations. For instance, Apple Handoff [1] only works on Apple devices like MacBook (macOS), iPhone (iOS) and, iPad (iPadOS). Also, there are research approaches for enabling state migration for web applications. Their focus is on the ability of run-time state migration by persistence state of JavaScript for the same application and only works on web applications [6]. In addition, some applications already are supporting an approach to migrate persistence state. For instance, IMAP can be used to sync the persistent state. Various existing applications support run-time state migration. For instance, users can continue streaming media on applications like YouTube, Netflix, or Spotify on their devices (e.g., from mobile to desktop), but it is designed only for that particular application on different platforms. Currently, no solution supports run-time state migration for same-purpose applications of different vendors on all ecosystems, devices, or platforms.

1.2 Goal

The goal of the thesis is to enable run-time state migration between applications serving the same purpose which are developed for different platforms and by different vendors. For this, a model-driven approach, as outlined in Figure 1.1, shall be developed. This approach shall allow adapting existing applications so that they support run-time state migration. To do so, relevant types of possible run-time states for an application shall be modeled as in terms of an *Application State Model* (see ❶). An *Application State Modeling Language* (see ❷) is needed to describe *Application State Models*. An *Application State Model* can be common among multiple applications serving the same purpose. For instance, an *Application State Model* for an e-mail application can be valid for multiple e-mail applications, e.g., K-9 Mail for Android and Mailspring for desktop operating systems. If two applications support the same state model, it shall be possible that a *Run-time State* (see ❸), which is an instance of an *Application State Model*, can be extracted from the source application (see ❹) and be injected into the target application (see ❺). The approach focuses on interactive applications for end-users and shall be extensible in the sense that if a new application with serving the same purpose supports the same *Application State Model*, migration from and to this application shall be supported as well. It is out of scope for the thesis to support run-time state migration between applications when no common *Application State Model* can be found.

To enable run-time state migration, developers should develop an *Application State Model* (or use an existing one) and add support for injecting and extracting run-time states to their applications. To simplify this, special libraries shall be developed that provide the basic functionality of run-time state migration, such as providing an API for validation, injection, and extraction. The *Application State Model* defines which types of *Run-time States* can occur at run-time. These libraries shall be able to validate the *Run-time States* based on an *Application State Model*. When being integrated into an application, the supported *Application State Models* need to be coupled to the applications by defining how the different types of *Run-time States* are being injected and extracted. To do so, an interface that is derived based on the *Application State Model* needs to be implemented. However, this coupling of the state model to a respective application is highly application-specific and needs to be implemented for each application by their developers or someone who has access to the source. These libraries have to exist for the different programming languages in which source and target applications are written.

If devices are in the same local network, these libraries establish a point-to-point connection. Otherwise, they use a middleware to communicate over the internet. For the indirect solution, a middleware shall be developed to which applications can introduce themselves to communicate with other same-purpose applications supporting the same *Application State Model*. The library

[1] Apple - Use Handoff to continue a task on your other devices

uses the developed interface at run-time to extract the *Run-time State* of the source application. The library shall pass *Run-time State* to a target application, either directly or with the help of the middleware. Developers are responsible for enhancing target application so that the library can validate and inject *Run-time State* to it. For this, they have to change the state of target application based on the received *Run-time State* (e.g., adjusting or placing data in a proper position on the UI).

To show the feasibility of this approach, as part of the thesis, two open-source e-mail clients shall be adapted to support run-time state migration based on the developed approach. These applications shall be for different platforms, and their source code should be freely accessible to allow the integration of the library. Suggestions for e-mail clients to be adapted are K-9 Mail for Android and Mailspring for desktop perating systems. K9-Mail Android application is developed in Java and Kotlin; therefore, it requires an Android library. Moreover, Mailspring is developed in TypeScript on top of Electron for desktop operating systems. As TypeScript is a superset of JavaScript and transcompiles to it[7], Mailspring needs a JavaScript library.

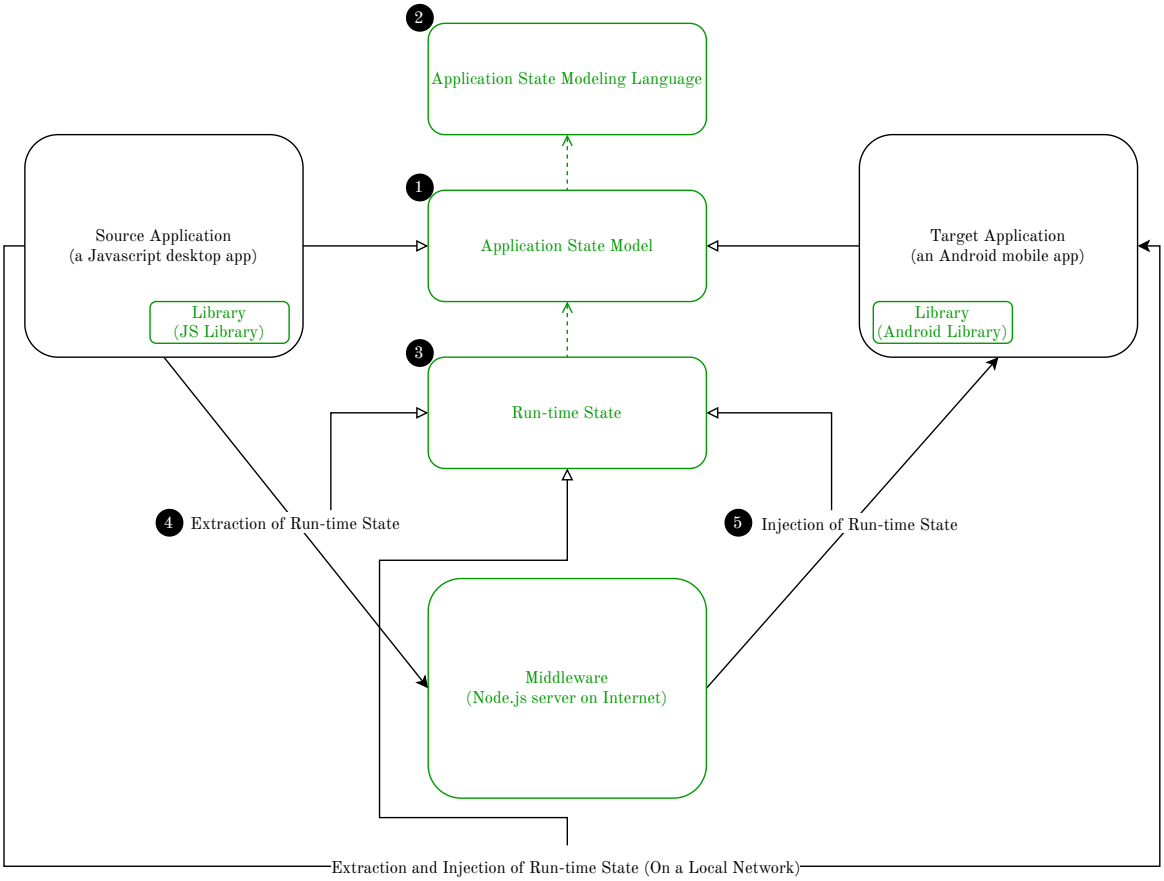

Figure 1.1: Showing the approach of run-time state migration between two applications.

1.3 Structure of This Thesis

The rest of this thesis is structured as follows.

Chapter 2 describes the fundamentals of Model-driven Software Engineering, Domain-specific language, JSON, and an explanation of their terminologies. In Chapter 3, state of the art in run-time state migration as related works is presented. Chapter 4 addresses the requirements and analysis towards the modeling approach and a brief solution overview. In Chapter 5, we explain the Domain-specific language for our approach. Chapter 6 explains ideas of our approach as an architecture overview. In Chapter 7, the details of the implementation part of this thesis are explained. Chapter 8 discusses the adaption of the run-time migration approach as a real-world example. The thesis is concluded in Chapter 9 with a description of the approach, its limitations, and the work that can be done in the future.

2

Fundamentals

In this chapter, the fundamentals of Model-driven Software Engineering are discussed. Moreover, we stress on fundamentals of Domain-specific Language and some DSL examples. Since this thesis focus on the migration of run-time states, we also give a categorization of applications states. Furthermore, as we use JSON in defining our DSL, we discuss about JSON and JSON Schema.

2.1 Model-driven Software Engineering

The Model-driven software engineering (MDSE) methodology takes the premise that models, as crucial elements of understanding and sharing complex software, are a practical way of working and thinking by transforming them into first-class citizens in software engineering. The purpose of a model can be anything from communication between people to the ability to develop a piece of software. The actual needs that they will address will determine how a model is defined and managed. [8].

Definition 2.1 (Model [9])**.** *A model is essentially an abstract representation of a software system's structure, function, or behavior.*

The model indicates that parts of the software system are generated. However, it may also indicate that the model should be interpreted at run-time, making it a central artifact as part of the software development process.

2.1.1 Model-driven Development (MDD)

MDD refers to a development paradigm in which models are their primary artifact. In MDD, the implementation is usually generated from the models (semi)automatically[8]. MDD increases development speed. This is achieved through the automatic code generation from a model with transformation steps. In MDD, when a software architectures are defined and implemented, automated transformations increase software quality, since consistency is emphasized[10]. .

2.1.2 Model-based Engineering (MBE)

MBE or Model-based development is referred to as a softer version of MDE that models are not necessarily first-class citizens. The MBE process concerns the use of models in development, although they are not the key artifacts. As one example, consider a development process in which designers specify the domain models of the system during the analysis phase but then

make the models available to programmers as the blueprints, to write the code without automatic code-generation involved [8].

2.2 Domain-specific Language

A Domain-Specific Language (DSL) is a language that is designed exclusively for a certain application domain. The language deals with the concepts and features specific to that particular domain. [11]. Following are the most vital points in a DSL. The notation by which users can write programs is specified in the *concrete syntax*. This notation can be textual, visual, or a combination of these. A data structure containing semantically related information conveyed by a model is known as *abstract syntax*. It is devoid of any knowledge about the notation. In addition to structurally sound in terms of concrete and abstract syntax, a language's *static semantics* is the collection of constraints and/or type system rules that programs must follow. The interpretation of a program to execute is referred to as *execution semantics* [12].

Definition 2.2 (Metamodel [12])**.** *The metamodel of a model is a model that defines the abstract syntax of a language used to describe a model.*

In metamodel, the meta prefix can be interpreted as *the definition of*. A model represents an *instance of* the metamodel. It should be mentioned that metamodel itself also is a model.

2.2.1 Define a DSL

Designing DSLs accordingly requires careful attention to detail to meet requirements. Essentially, the main concern of DSL design consists of implementing concrete syntax and abstract syntax [13].

There are some steps that they have to be considered for creating a DSL [10].

1. Define DSL metamodel

   The metamodel should defines the abstract syntax and the static semantics of a DSL. The metamodel should be defined in way to prevent unwanted instances. In order to define a metamodel, a metamodeling language is required, which is described by a meta meta model. For example, in this thesis we use UML to define our metamodel which explained in 5.1.1.

2. Define concrete syntax

   Since the DSL is used as the 'user interface' for the metamodel, it is important that developers understand, write and interpret the models properly. Thereby, a suitable concrete syntax should be considered. As this thesis involves a textual language, we chose to use JSON, which is discussed in the next section.

3. Define the mapping between abstract syntax and concrete syntax

   There should be a proper mapping to map elements of the concrete syntax to the metamodel elements. For example, in this thesis we mapped our DSL's metamodel in JSON schema which discussed in 5.1.3.

4. Define DSL semantics

The semantics are typically explained informally in natural language. A DSL's semantics must either be intuitively clear to the modeler or well-documented. One advantage of the DSL is its ability to adapt to concepts from the problem space so that domain experts will recognize its 'domain language'.

5. Define model transformations and code generators

   Essentially, model transformations work like programs that take in models as input. Model transformations differ and are used in different ways, which can be expressed differently through their inputs and outputs. When a model is defined based on its metamodel, model transformation indicates which models can be accepted as input and, if appropriate, what models can be produced as output [14].

   The code generators are metaprograms that use specifications or models as input parameters and generate code as output. Also, it should be defined which part will be generated, so the generated code can be separated from the manually-created code, and the developer must integrate both.

2.2.2 DSL Examples

This section discusses two existing textual DSLs.

Amazon States Language

This language is a JSON-based specification for state machines. These state machines are a collection of states that perform tasks. This specification allows the state machine to determine the transition to other states and stop the execution. This language is used within the Amazon Web Services infrastructure, and it use lambda functions to execute tasks [15]. Listing 2.1 shows an simple example of Amazon States Language, which contains two states and starts from *HelloWorld* state and have transition to *NextState* and finishes there.

```
{
    "Comment": "A minimal example of the Amazon States language",
    "StartAt": "HelloWorld",
    "States": {
        "HelloWorld": {
          "Type": "Task",
          "Resource": "aws:lambda:eu-west:function:HelloWorld",
          "Next": "NextState",
        },
        "NextState": {
          "Type": "Task",
          "Resource": "aws:lambda:eu-west:function:FunctionName",
          "End": true
        }
    }
}
```

Listing 2.1: A simple example of the Amazon States language [16]

PlantUML

PlantUML is an open-source DSL allowing users to define different kinds of UML diagrams from a textual language. For example, All UML diagrams in this thesis are generated from PlantUML. Listing 2.2 shows a simple UML sequence diagram in PlantUML syntax. Also Figure 2.1 shows the generated graphic of Listing 2.2.

```
@startuml

autonumber

A -> B: step

activate B
B -> C: step

activate C
C --> C: action
C -> B: step
deactivate C

B -> A: step
deactivate B

@enduml
```

Listing 2.2: A simple sequence diagram in PlantUML syntax

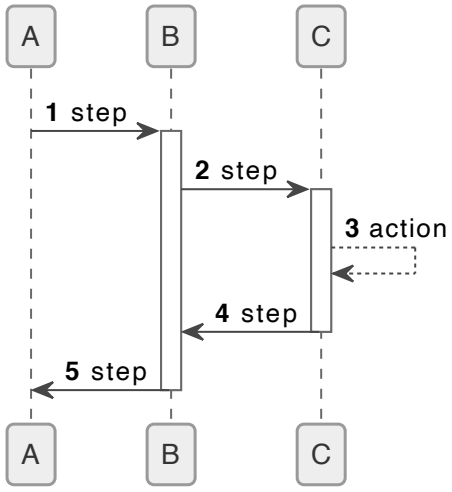

Figure 2.1: A generated simple sequence diagram in PlantUML

2.3 Application State Space

All possible combinations of an application are represented in a state space. Abstractions of state space are helpful when we want to comprehend the behavior of a given system. A state variable represents a particular state of the application; the values of all state variables describe the state of the application. This means each point in the state space represents one state of the application [17]. The application states of existing applications are categorized in this thesis in order to identify those that can be migrated.

Persistent State A state that exist longer than a single execution of the program is considered to be a persistent state. In practice, the state is maintained by storing it as data in the computer's data storage [18]. For example, the settings of a web browser.

Action State A running application will frequently alter the run-time state due to processing, which is called an action state. For example, a web browser is printing a page.

Run-time State In an application, variables represent the storage locations in the computer's memory that the computer uses to hold data stored. The run-time state of an application is the content of these memory locations at any point in its execution [19]. For example, the value of a search input. In this thesis, the focus is on supporting the migration of such states.

2.4 JSON

JSON (JavaScript Object Notation) is a file format. It is a lightweight and human-readable text to store and transmit data based on the data types of the JavaScript programming language [20]. In the last few years, many programming languages support JSON, which gained popularity among developers, and has become the primary data format for exchanging information. JSON documents consist of key-value pairs, in which the value can be again a JSON object. Objects are key-value pairs. Each key is a string and denotes a property of the object. The value can be of a primitive data type (number, string, boolean, etc.), it can be an array of values or again an object, and there is no limit to the nesting level [21].

A simple JSON document is shown in Listing 2.3 which *fullname* and *email* are keys and *"John Doe"* and *"john@mail.upb.de"* are their string values. Moreover, *info* is an object containing another key that is *hobbies* contains an array value.

```
1  {
2      "fullname": "John Doe",
3      "email": "john@mail.upb.de",
4      "info" : {
5          "hobbies" : ["watching movies", "wood carving", "football"]
6      }
7  }
```

Listing 2.3: A simple JSON document.

2.5 JSON Schema

After the huge popularity of JSON, certain scenarios could benefit from a declarative method for identifying a schema for JSON documents. A declarative schema specification would give programming languages and developers a standardized language to specify what types of JSON documents are valid as inputs and outputs [21].

The schema can be another JSON document that defines acceptable key-value pairs. For instance, the simple JSON document in Listing 2.3, can be declare in JSON Schema document in Listing 2.4.

```
{
    "properties": {
        "fullname": {
            "type": "string"
        },
        "email": {
            "type": "string",
            "format": "email"
        },
        "info": {
            "type": "object",
            "properties": {
                "hobbies" : {
                    "type": "array"
                }
            }
        }
    }
}
```

Listing 2.4: A simple JSON Schema document

JSON Schema can define a document must have several , including regular data-types like objects, arrays, strings, booleans, numbers and, null. Each of these types has different keywords that help to specify and restrict the schema. The most important attribute is “type” whose value is defining the schema. With validation keywords included in the schema, constraints can be applied to an instance. [22]. For example in Listing 2.4, `{"type": "string"}` specify a value with string type.

We use YAML syntax to represent the JSON Schema in this thesis for readability reasons. YAML is a human-readable syntax and can be converted to JSON syntax without extra configuration. Listing 2.1 shows a simple example based on Listing 2.4 which is a JSON Schema document.

```
properties:
  fullname:
    type: string
  email:
```

```
    type: string
    format: email
  info:
    type: object
    properties:
      hobbies:
        type: array
```

Listing 2.1: Example of expressing JSON Schema in YAML syntax.

3

Related Works

In this section, we discuss state of the art in run-time state migration.

Various articles have been published regarding state migration, mainly focusing on run-time state migration of one particular platform to another device. For instance, in [23], the researchers present an approach that can migrate part of a web application with its state across devices. Users need to use the developed tool called *Partial Migration* to select elements of a web page. The list of these elements is auto-generated from HTML and CSS tags in a hierarchical tree view. The user can choose a target device from the provided list called *Migration Panel* which contains the device's hostname by an icon highlighting the type of the corresponding device (desktop, PDA, smartphone, large screen). She can start the migration process by clicking on *MIGRATE* button. The target device receives a migration request with the information of the source device. If she accepts the migration request, the target device shows a web page that contains the chosen elements with their state. This method uses a user interface description languages [24] and tools that can automatically reverse engineer existing desktop Web content to build the corresponding logical description in a hierarchical tree. As the primary developed tool is a desktop application, in this approach, the source application should always be a desktop web application which its front-end is implemented using HTML, CSS, and JavaScript.

Another migration approach focusing on web applications is discussed in [25]. The researchers presented Panelrama, a web framework for cross-device UI distribution. Their implementation enables developers to migrate at a relatively low cost through built-in mechanisms for synchronizing UI states, which do not require drastic changes to existing languages. Developers can use divided groups called *panels* in Panelrama to separate the user interface of an application into different groups. Panelrama gives developers the opportunity to score the importance of certain device characteristics to this panel's usability. Applications with multi-device support can distribute panels to best fit the devices for an optimal user experience, such as video to the largest device and remote controls for the closer devices. This framework is only targeting the web and hybrid applications that are based on HTML5.

The paper [26] also suggests how to migrate interactive web applications to users with multiple devices. Regardless of the devices being used (desktops, tablets, smartphones), the environment enables dynamic push and pull of interactive Web applications across desktop and mobile devices while preserving their state. Users may choose to migrate to a specific device using a migration client by accessing it through a browser. This approach only works on web applications, and its needs a browser to operate.

All of these approaches are oriented towards web applications, whereas our approach is also applicable to desktop and mobile applications.

Furthermore, several studies have discussed different aspects of run-time migration in a soft-

ware system. The paper [27] describes ScudOSGi, a framework for handling task migration that enables facility-involved task migration in the OSGi which is a Java framework for component-based development and dependency management. With ScudOSGi they allow to migrate those components. They use a whiteboard model to simplify the state's management and maintenance. This framework has four components which are task migration trigger and task host, server, and local infrastructure. The server manages the state and resources of user tasks. Computation and facilities services in the local space are available to tasks and are managed by local infrastructure. In order to allow local infrastructure to determine whether the task requires certain facilities, the task migration trigger parses both the task and the application description. The task host hosts the applications needed by the task, and it is responsible for the communication channel required to get information from the server about the task's state and resources. An application model with two levels is proposed by them. Compared with other migration frameworks, they focus on maximizing usage of local facilities for user tasks. As examples for applications of ScudOSGi, they investigated two use-cases: smart player and multi-user shopping.

The purpose of [28] is to demonstrate the efficiency of multi-engine distributed process execution in an effort to improve adaptability in response to ad-hoc context changes. They present an abstraction meta-model of migration data that can be used to enhance existing processes with the ability to perform run-time migration. Both process modelers and initiators can include their intentions and privacy needs in the approach, as well as execute processes sequentially or in parallel. In addition to the use of physical process fragmentation, a concept has been proposed for realizing logical fragmentation based on the guidelines of process migration. Process migration is more flexible than physical fragmentation because it provides the flexibility of distributing running process instances at run-time while respecting the process modeler's guidelines. Additionally, the papers address privacy and security issues in an explicit manner.

These researches focus on different aspects of run-time migration, while our approach focuses on supporting actual run-time state migration.

Furthermore, there are some approaches in the field of state migration that are meant to be used across particular ecosystems or for specific purposes. For example, Apple Handoff[1] is an approach that works between not same-purpose applications but between the same applications for different platforms on the same platform (e.g., Mail app on iPhone and Mac). Also, users of the Google Cast[2] can stream video and audio from an Android app or iOS app to a TV or audio system (e.g., streaming Youtube). After migration, source applications can act as a remote control to control the streaming media.

To summarize, none of these solutions support run-time state migration across all ecosystems, devices, and platforms for same-purpose applications of different vendors.

[1] https://support.apple.com/en-us/HT209455
[2] https://developers.google.com/cast

4

Towards Model-based Run-time State Migration

In this chapter, we discuss requirements and a potential solution towards an approach for supporting run-time state migration. Moreover, we analyze some same-purpose applications to identify requirements for our DSL and to know which migratable run-time states are involved in the application's scope..

4.1 Requirements Analysis

This section provides a requirements for a run-time state migration approaches for same-purpose application of different vendors. The rationale behind every requirement is given when discussing the individual requirements.

4.1.1 R1 Applicable on Existing Applications

This approach shall allow enabling run-time state migration on new and existing same-purpose applications. The source code of these applications must be accessible. Developers should be able to use this approach on applications, even if they are not part of the applications' developers team. Thereby, it enables the support of a vast number of existing applications.

4.1.2 R2 Platform-independent State Specification

It shall be possible to model which and when parts of existing applications can be migrated. In addition, as the approach is meant to work with different platforms, it shall be possible to model these states in a platform-independent manner.

4.1.3 R3 Model Repository

Developers wanting to adapt the run-time state migration approach to existing application shall write a state specification or have access to existing state specifications. Thereby, it shall be possible to have a central place to store models that are state specifications. Developers can search for common models or upload models they defined. This repository helps wider possibilities for migration, more reuse and reduce the time of development.

4.1.4 R4 Device Management

It must be possible to find, list and select devices offering the same states and migrate them. Also, the device discovery shall be possible in a way that the approach can express if a device knows another devices has the same run-time state on the same network.

4.2 Same-purpose Applications Analysis

Based on **R1 Applicable on Existing Applications**, the following analysis is to determine what pairs of application states exist and what needs to be modeled. These applications are from different platforms, and they are interactive applications for end-users and have sufficient complexity (e.g., applications not have only one single state).

4.2.1 State Analysis of Applications

In this section, we compared four pairs of existing applications to determine which states are common between applications. Also, the type of the states is categorized into three types which are run-time state (R), persistent state (P), and action (A). Furthermore, the "Migratable" column shows if the state can be a candidate for run-time state migration.

States of E-mail Applications

Table 4.1 shows a list of available states of two open-source e-mail clients: Mailspring[1] on macOS and K-9 Mail[2] on Android. Some states are migratable like "Single E-mail" and "Search View". The "Single E-mail" state contains a view of a single e-mail in reading mode. In "Search View" the user can search and find an e-mail by typing in a query input box. The information of some states like "Compose a new E-email", "Forward an E-mail" and "Replay an E-mail" are almost similar, and they can be considered as one state for sending an e-mail.

[1] https://getmailspring.com/
[2] https://k9mail.app/

State / E-mail client	Mailspring	K-9 Mail	Type	Migratable
Welcome Screen	✓	✓	P	
Account Registration	✓		P	
Account Login	✓		P	
Add an e-mail service account	✓	✓	P	
Import Settings		✓	P	
Export Settings and Accounts		✓	P	
Settings	✓	✓	P	
Sync New E-mails	✓	✓	P	
Compose a new E-email	✓	✓	R	✓
Forward an E-mail	✓		R	✓
Replay an E-mail	✓	✓	R	✓
Sending an E-mail	✓	✓	A	
Trashing an E-mail	✓	✓	A	
E-mails list (Inbox, Sent, Draft, ...)	✓	✓	P	
Single E-mail	✓	✓	R	✓
Show Original Version of E-mail	✓		R	
Search View	✓	✓	R	✓
Searching	✓	✓	A	
Loading E-mails	✓	✓	A	
Loading Activity Data	✓		A	
Activity View	✓		P	
Exporting Activity Data	✓		A	
Sharing Report of Activity Data	✓		A	
Marking Star/Spam/Read/Unread an E-mail	✓	✓	A	

Table 4.1: States of E-mail Applications

States of Browsers

Table 4.2 shows a list of available states of two browser applications: Firefox[3] on macOS and Chrome[4] on Android. The information in the current tab can be found in the "Current Tab" state. Also, searching is possible within the "Find" state. Information of "Developer Console" and "Private/Incognito Mode" states is whether they are opened as a window or not.

Some states information are from official Chrome Page lifecycle API[5] and Mozilla Web API[6]. Also, For browsers there is a standard called W3C Page lifecycle API[7].

[3] https://www.mozilla.org/en-US/firefox/
[4] https://www.google.com/chrome/
[5] https://developers.google.com/web/updates/2018/07/page-lifecycle-api
[6] https://developer.mozilla.org/en-US/docs/Web/API
[7] https://wicg.github.io/page-lifecycle/

State / Browser	Firefox	Chrome	Type	Migratable
Home page	✓	✓	P	
Single Tab (Preferences, Bookmarks, Performance, ...)	✓	✓	P	
Extensions Tab	✓		P	
Syncing	✓	✓	A	
Browsing	✓	✓	A	
Current Tab	✓	✓	R	✓
Developer Console	✓	✓	P	
Signing in	✓	✓	A	
Find (in page)	✓	✓	R	✓
Finding	✓	✓	A	
Printing	✓	✓	A	
Print View	✓	✓	P	
Downloading	✓	✓	A	
Sharing		✓	A	
Private/Incognito Mode	✓	✓	R	✓
Light mode		✓	R	

Table 4.2: States of Browsers applications

States of Video Player Applications

Table 4.3 shows a list of available states of two video player applications which are IINA[8] on macOS and VLC[9] on Android. In these applications, all migratable states' are related to playing a video and can be considered almost identical. These states are "Pause", "Playing Video" and "Streaming".

To support run-time state migration on these video players, we needed to access the persistent storage and the video file. Considering these problems, run-time state migration for video players which have access to a persistent storage, is not the right choice. However, run-time state migration can be supported on video streaming applications with the same media.

[8] https://iina.io/
[9] https://www.videolan.org/vlc/

State / Video Player	IINA	VLC	Type	Migratable
Welcome Screen	✓	✓	P	
Video player tips		✓	P	
Audio player tips		✓	P	
Pause	✓	✓	R	✓
Deleting		✓	A	
Browsing view	✓	✓	R	
Loading Directories		✓	A	
Single Page view (About, Settings, Audio, Video, ...)	✓	✓	P	
Search result		✓	R	
Searching		✓	A	
Playing video	✓	✓	R	✓
Playing audio	✓		R	
Showing picture	✓		R	
Streaming	✓	✓	R	✓
Loading local network devices		✓	A	

Table 4.3: States of Video player applications

States of Note Taking Applications

Table 4.4 shows a list of available states of two note-taking applications: Laverna[10] on macOS and Joplin[11] on Android. Some states are migratable, which are "Search" and "Writing Note". In "Search" the user can search and find a note by typing in a query input box. In "Writing Note" the user enters some text as the note's content.

State / Note taking application	Laverna	Joplin	Type	Migratable
Loading Screen	✓		P	
Welcome Screen	✓		P	
Encryption view	✓		P	
Syncing on Cloud	✓	✓	A	
Importing and Exporting settings	✓		A	
Importing and Exporting data	✓	✓	A	
Single Page view (All notes, Favorites, Settings, ...)	✓	✓	P	
Search	✓	✓	R	✓
Searching	✓	✓	A	
Writing a Note	✓	✓	R	✓
Writing a Todo		✓	R	
Saving a Note	✓	✓	A	
Note Properties		✓	P	
New Notebook	✓	✓	A	
New Tags	✓	✓	A	
Share view		✓	P	
Trashing	✓		A	
Removing	✓		A	

Table 4.4: States of Note taking applications

[10] https://laverna.cc/
[11] https://joplinapp.org/

4.2.2 State Values

This section shows an analysis of the source code of two before mentioned email clients, Mailspring and K-9 Mail. These applications are open-source and their source code are freely accessible. The analysis of the source code and these values helps us to determine what possibly can be part of state specification and how we can define a state specification.

Values of two migratable run-time states listed in tables. Table 4.5 and 4.6 are showing the "Sending an E-mail" state and Table 4.7 and 4.8 are showing the "Search View" state which were mentioned in comparison of these two applications (Table 4.1).

States Values of E-mail Applications

Field	Type	Example/Description
id	String/Id	pYkSAMPpfU9bU1E33219fbaJyVoQ71hS8Vvs7gDZC
aid	String/Id	0a6dbf86
v	Number	3
metadata	Array	Information About Mail/Link tracking
to	Array	[]
cc	Array	[]
bcc	Array	[]
from	Array	[]
replyto	Array	[]
date	Number	1595782508
body	String	This is a test body
files	Array	[]
unread	Boolean	False
events	Array	[]
starred	Boolean	False
threadid	String/Id	""
subject	String	Test Subject
draft	Boolean	True
pristine	Boolean	False
plaintext	Boolean	False
folder	Object	
"file_ids"	Array	[]
object	String	"draft"

Table 4.5: State Values: Compose a new Email in Mailspring

Field	Type	Example/Description
_ID	String	The Id of the draft
SEND_DATE	String	Time and Date
SENDER	Object	Sender Contact
RECEIVER	Object	Recipient Contact
SUBJECT	String	Subject Text
TEXT	String	Body of E-mail
ACCOUNT	Object	Mailbox Information
SENDER_ADDRESS	String	Sender E-mail
RECEIVER_ADDRESS	String	Recipient E-mail

Table 4.6: State Values: Compose a new Email in K-9 Mail

Field	Type	Example/Description
isSearching	Boolean	False
query	String	"master thesis"

Table 4.7: State Values: Search in Mailspring

Field	Type	Example/Description
query	String	"master thesis"

Table 4.8: State Values: Search in K-9 Mail

4.2.3 Modeling States

To find common values in run-time states and define a state specification values of the state has be to analyzed and modeled.

Modeling States of E-mail Applications

By define one state specification for each run-time state, these two applications (Mailspring and K-9 Mail) can support same state specifications. For example in “Sending E-mail” state values in Table 4.5 and Table 4.6 have some common pair values like “from/SENDER”, “to/RECEIVER”, “subject/SUBJECT” and “body/TEXT”. In Table 4.9 we modeled these values and it can be use to define a state specification.

Field	Type	Description
from	String	The sender email
to	Array of String	The reciever email
subject	String	The body text of the email
body	String	The subject text of the email

Table 4.9: Modeling the State: Sending-Email in E-mail Clients

Furthermore, in “Search View” state there is a common pair value: “query”. This state is modeled in Table 4.10.

Field	Type	Description
query	String	The search phrase

Table 4.10: Modeling the State: Search in E-mail Clients

4.3 Requirements for a DSL

Based on **R2 Platform-independent State Specification** and analysis of same-purpose existing applications, we chose to develop a DSL to model the run-time states by specifications. Same-purpose application got analyzed and we identified the most common structure a state has and defined them as requirements for a DSL to specify a state as follows.

4.3.1 D1 State Specification

The DSL shall be able to provide the specification of a state by defining a model. State which are following these specifications can be migrated between applications. Also, state specification shall provide a way to define what part of a run-time state is required and what part is optional.

4.3.2 D2 Finding Same State Specification

Same-purpose applications needs to figure out if they are supporting same state specification. To find common models between applications, the DSL must allow specifying a way for finding the same model. These specifications can be a unique model name, model version and keywords. Also, these specifications help developers to find a common models which resolve part of **R3 Model Repository**.

4.3.3 D3 Validating

The DSL shall be able to provide a way that a state must be valid according to its specification. Also, In case of improving the DSL for long-term support, the DSL must have a version to allow validating the state specification. So, it can be possible to know the state specification using which version of the DSL.

4.4 Solution Overview

This section provides an overview of the approach and explains how it addresses requirements R1 to R3. A visualization of the approach is shown in Fig. 4.1. As mentioned in **R1 Applicable on Existing Applications**, the approach shall apply to existing applications, and they should be able to support run-time state migration. For ease of enabling run-time migration for existing applications, we chose to develop the necessary library. This library shall provide an API for validation, injection, and extraction states. As stated in **Requirement R2**, the approach shall enable the ability of state specification. Thereby, modeling a run-time state as state specification is possible by the DSL, which will be integrated with the library by interfaces.

Besides libraries, our approach needs device management and device discovery, stated in **R4 Device Management**. The middleware serves as device management, allowing devices to introduce themselves and allows devices to find each other and get noticed when a device joins or leaves. These libraries can provide a point-to-point connection if devices are on the same local network. Otherwise, they use the middleware to communicate over the internet to migrate the run-time state. Since the focus of the thesis is not providing different types of communications, only one of these communication styles is implemented.

Moreover, the library need interfaces based on state specification to migrate and validate run-time states. These interfaces are converted version of state specification but provided in the same programming language of the application. They can be written manually by developers or get generated by some tools.

Furthermore, developers should write some glue code to bind the source code of the application to the library and interfaces.

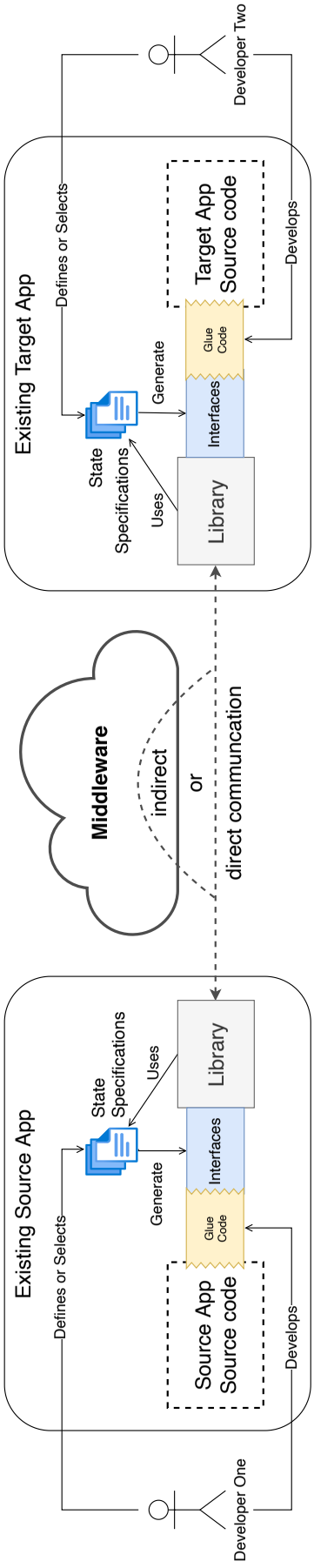

Figure 4.1: General overview of the approach.

5

Language for Run-time State Migration

As stated by **R2 Platform-independent State Specification**, need a platform-independent representation of run-time state. For this, we develop a language to describe run-time states. The requirements for this language are on the requirements D1 to D3 (Chapter 4). In this chapter, the DSL definition and interpretation, specification of run-time state, and terminology used are described. The definition of the language occurs on different layers (Figure 5.1). On the top layer, we define a language for specify a type of state an application. Such a specification of a specific type of state resides on the middle layer. Concrete instances of this type, a specific run-time state of an application, is located on the bottom layer. All three layers are explained subsequently.

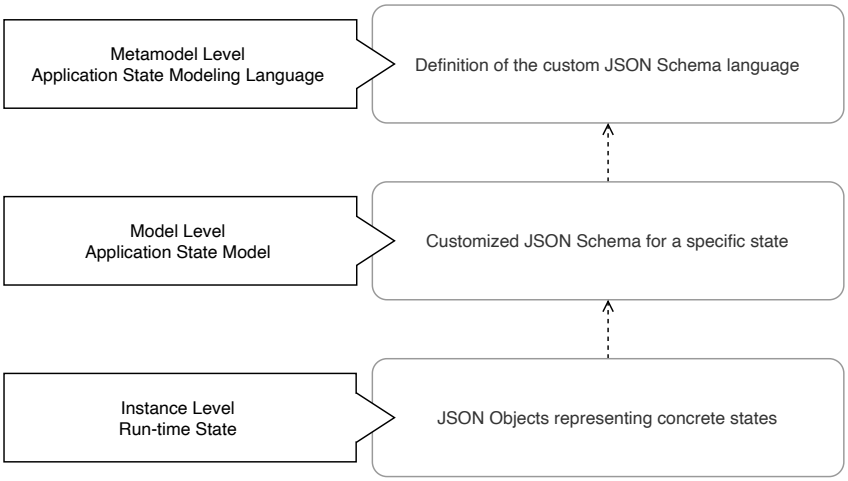

Figure 5.1: The definition of the language on different layers.

5.1 Application State Modeling Language

After a research and looking for suitable existing state modeling languages, it has been decided to create a new light-weight DSL for data representation based on run-time state migration requirements. A DSL can be developed from scratch, or it can be a modification or a modified form of an existing domain-specific modeling language.

The Application State Modeling Language (ASML) is a DSL that defines any type of software application state. The focus of this language is on modeling single types states of the run-time.

As mentioned in requirements **D1 State Specification**, this language supports run-time

states that are migratable. An application can have multiple states, and users can switch between them. These states and their transition can be modeled in different ways (e.g., UML state machine diagram), but as we analyzed some applications in requirements (Chapter 4), we deal with the data perspective of run-time states and have to access to the actual values and migrate them. Therefore, the run-time state's actual data will be migrated not their transition. For example, Amazon States Language considers the state transition as some tasks that need to be done step by step. Moreover, there is no need to include the transition between states in state specification as we assume we know in which state application is.

Each model can cover only an individual state. The advantage of having an individual model for each state is improving interoperability, allowing different domain applications to share a specific part of them. For example, an e-mail application may have a calendar that can migrate data with a mobile phone calendar application. Also, there is no need to model the states which are not migratable. So, it reduces the time of modeling state and the amount models.

Figure 5.2 shows an overview of Application State Modeling Language (see ❶). The specification of a type of state using the ASML is called an Application State Model (see ❷), which is discussed in subchapter 5.2. There are some applications support the run-time state migration and have common Application State Models (see ❸). The actual values of a state on run-time is Run-Time State, which is discussed in subchapter 5.3.

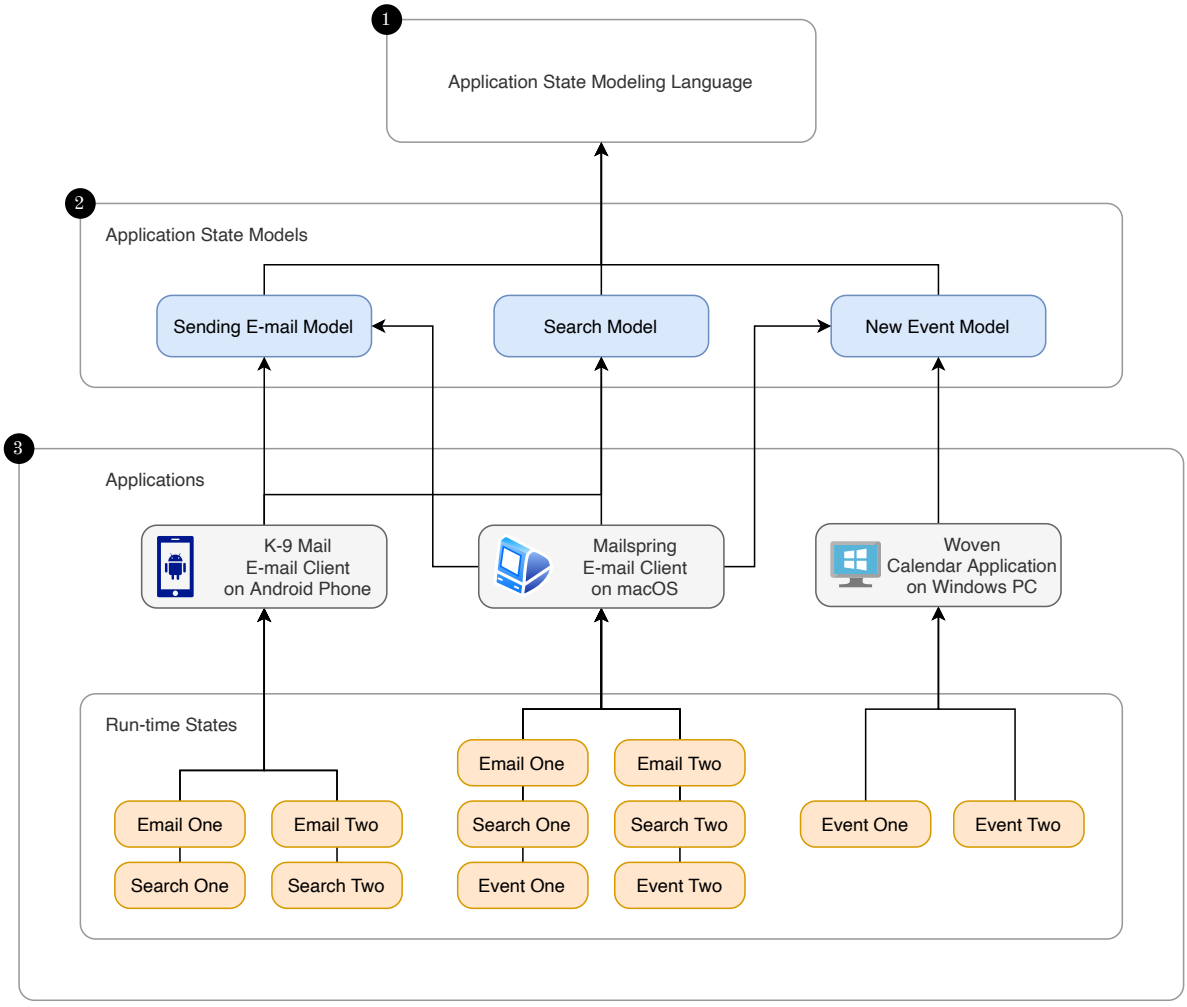

Figure 5.2: Overview of Application State Modeling Language

Considering three states which are modeled as *Sending E-mail Model*, *Search Model* and *New Event Model*. In this example, we have three applications which are K-9 Mail is an e-mail client on an Android phone, Mailspring is another e-mail client which also has calendar management on macOS, and Woven Calendar is a calendar application for Windows. K-9 Mail has two migratable run-time states, which are *Sending E-mail* and *Search*. Mailpsring has three migratable states, which are *New Event*, *Sending E-mail* and *Search*. Waven has two migratable run-time states, which are *New Event* and *Search*. As Mailpsring and Waven have a mutual *New Event* run-time state, they can share a common model: the *New Event Model*. However, Mailspring has two other common run-time states which have common models with K9 Mail: the *Sending E-mail Model* and *Search Model*. So, Mailspring and Woven can migrate actual run-time data of the *New Event* state. Furthermore, Mailspring and K-9 can migrate actual run-time data of the *Sending E-mail* state and the *Search* state. This shows the same model can be used for different applications. The figure will be further explained in detail in the relevant sections.

5.1.1 Metamodel

Figure 5.3 shows the metamodel of our DSL as an UML class diagram. An Application State Model can describe one state. Each state specified in the Application State Model has a unique name, a description and some keywords. The extra information of a state is the version and author that can be defined in Number and String. Each state can have multiple values. Each value also can have multiple values which consists of a unique name and a type which can be a primitive type like Array, Object, Boolean, File, Number, and String. Also, a value can have some constraints like format, regex pattern, required and etc.

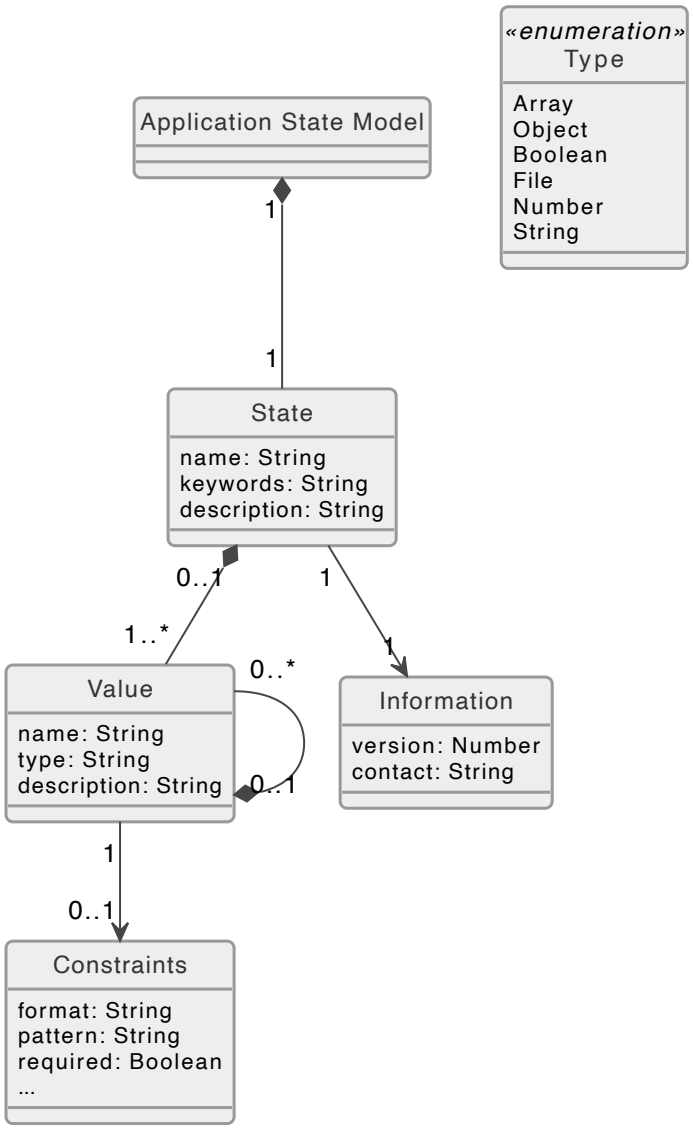

Figure 5.3: UML Class Diagram of our DSL Metamodel

5.1.2 Language Stack

There are many ways to define a DSL; it can be made from scratch (e.g., developing a language base on Xtext framework) or extended or restricted versions of other languages. The ASML is an extended version of JSON Schema.

5.1.3 Language Schema

The ASML itself is a JSON Schema document based on JSON Schema Draft-07 [1] and is following the same logic and syntax. Table 5.1 shows the the schema of ASML with its most important properties in a condensed form. The full schema of ASML is available on its GitHub repository [2].

Mapping Metamodel to JSON Schema

The metamodel of ASML (Fig. 5.3) is mapped in JSON Schema. Therefore, each state should be defined in one JSON Schema document which is Application State Model. Each state should have a name, a description and keywords. Values of a state should be defined in `properties` of JSON Schema, in which each value should be define as an object with a unique name as a key. The type of this value can be defined as a key-value pair inside value object, the key is "type" and its string value can be a primitive data type like number, string, boolean, and etc. Also, each value can have a description. The extra information like version and author's contact are mapped to `info` element. Moreover, each value can have some constrains like format, pattern and required that they can be defined inside value object. These constrains can be part of JSON Schema capabilities.

Property	Description	Value
title	The name of the DSL.	Application State Modeling Language
description	The description of the DSL.	A DSL for enabling run-time state migration between same-purpose applications of different vendors
version	The current version of the DSL, which can be used in for version checking for validating models.	1.0.0
properties	Contains values which an Application State Model should have.	{"asml", "info", "properties", "required"}
required	Defining the required items which a model must have.	["asml", "info", "properties"]
additionalProperties	The DSL is not allowing developers to add additional properties to models.	false
definitions	Contains the definition of the DSL items and the basic JSON Schema specification.	version, asml, info, contact, Schema

Table 5.1: ASML Schema

[1] JSON Schema Draft-07 Release Notes
[2] https://github.com/asml-lang/

5.2 Application State Model

The Application State Model is a custom JSON Schema document that defines what a state must contain at run-time and is validated against Application State Modeling Language. To enable run-time state migration, developers should write a custom Application State Model or use an existing one. In contrast to a regular JSON schema, the Application State Model must have the fields defined by the ASML.

Any application has different states; Each state should have its own model, which is a JSON Schema document. At the time of integration, Application State Models must be coupled with the source and target applications to determine which states are available for migration.

For migration to happen, source and target applications must have a common Application State Model. As each state is modeled individually, different domain application with the same-purpose part can migrate their states. For example, an e-mail client may have a to-do list feature which can be a standard characteristic for a task management application. So, they may have some common states like writing a task or search.

Developers can write their own Application State Model or using the existing ones. Variables of a run-time state that need to be migrated shall be identified and associated to keywords. Developers can look into the Model Repository (explained in 6.5) and search for these keywords to find a suitable model for their application. They may use an existing one or even extending it by making a new model. To write a new model, variables of a run-time state which they are associated to some phrases, should be specified by their name, type or their format. For example, when a search state needs to be migrated, we need to migrate a text and maybe the submission status. These variables can be associate with some phrases like "query" as a string and "submit" as a boolean. These associated phrases and their type must be specified in `properties` section of Application State Model which is described in next section.

5.2.1 Top-level Fields

There are some fields that they need to be in an Application State Model to define a state specification.

asml

As stated in requirement **D3 Validating**, a model must have an object field named "asml" whose string value represents the version of ASML. This version follows SemVer pattern [3] and helps to validate the model against the correct version of our DSL. Listing 5.1 shows how the "asml" field has to be defined. The version of ASML in this Application State Model is "1.0.0".

```
1 asml: 1.0.0
```

Listing 5.1: Application State Model "asml" field example.

info

As stated in requirement **D2 Finding Same Model**, a model must have an object field named "info" whose object value represents the basic model's information. This object must have two string values "title" and "version". Also, "keywords" is an array of string which should contains

[3] https://semver.org/

some keywords that other developers can find this model in Model Repository. Common models can be fined and distinguished by values of "title", "version" and "keywords". Other fields are "description" and "contact", which are optional. The "description" is a string value that represents the descriptive text about the purpose of the model, and "contact" is the author's information in case of further contacts. Listing 5.2 shows an example of how the "info" field has to be defined.

```
asml: 1.0.0
info:
  title: sending-email
  description: A schema model for sending an email
  version: 1.0.0
  keywords:
  - email
  - e-mail
  - compose
  contact:
    name: Saman Soltani
    email: saman@mail.upb.de
    url: samansoltani.com
```

Listing 5.2: Application State Model "info" field example.

properties

A model must have an object field named "properties" whose object value represents a state specification. Based on ASML, any element of the "properties" object is considered a variable that can be migrated. Listing 5.3 shows an example that has four fields with different types such as from, to, subject and body. These fields should be in the state.

```
properties:
  from:
    type: string
    format: email
  to:
    type: array
    items:
      type: string
      format: email
  subject:
    type: string
  body:
    type: string
```

Listing 5.3: Application State Model "properties" field example.

required

As stated in requirement **D1 State Specification**, a model can have an object field named "required" which its array of strings value represents compulsory variables in the state. Listing 5.4 shows an example which has three string value in an array. These fields must exist in the state as they are specified in "required" field.

```
required:
- from
- to
- body
```

Listing 5.4: Application State Model "required" field example.

5.2.2 Example Models

Composing New E-mail

Considering an e-mail client application in which the user can write an e-mail. The state of composing a new e-mail can have a minimum of four fields which are "from", "to", "subject" and "body". A combination of Listings 5.1, 5.2, 5.3 and 5.4 shows an example of this model. The complete version of the Application State Model of composing a new e-mail is shown in Listing 1 which is in appendixes.

Writing Note

Considering a note-taking application that has only one state for migration which is writing a note. It can be modeled with a compulsory "text" string field as in Listing 5.5.

```
asml: 1.0.0
info:
  title: writing-note
  version: 1.0.0
properties:
  text:
    description: content of the note
    type: string
required:
- text
```

Listing 5.5: Note Writing example Application State Model as JSON Schema in YAML.

Search

Searching between data is very common between applications. Suppose an application includes a search function. It also provides a result for searches. The search state can be modeled so that the application knows if the search has been submitted by the user or not; it can adjust itself base on a received state. The search model can have a compulsory "query" string field, which

defines the text that the user is looking for, and a "submit" boolean field shows if the search has been submitted.

This Application State Model can be used in different purpose applications which they have the search feature. Listing 5.6 shows a search state modeled defined as Application State Model in a JSON Schema document.

```yaml
asml: 1.0.0
info:
  title: search
  version: 1.0.0
properties:
  query:
    description: the query of search
    type: string
  submit:
    description: shows if the query is already has been submitted
    type: boolean
required:
- query
```

Listing 5.6: Search example Application State Model as JSON Schema in YAML.

5.3 Run-time State

The Run-time State is a JSON document that contains the actual key-value pairs of a state. Run-time State is validated against Application State Model. If the state is valid, it can be migrated from source to target application. Target application should adjust itself with the new state.

5.3.1 Example Run-time States

Composing New E-mail

Based on composing new e-mail Application State Model (Listing 1) valid example values for sending e-mail state is shown in Listing 5.1.

```json
{
  "from": "saman@mail.upb.de",
  "to": [
    "engels@uni-paderborn.de",
    "dennis.wolters@upb.de"
  ],
  "subject": "Master Thesis Documents",
  "body": "Dear Prof. Engels\n, Here is my master thesis."
}
```

Listing 5.1: A Run-time State for sending e-mail as JSON document.

Writing Note

Based on writing note Application State Model (Listing 5.5) valid example values for writing note state is shown in Listing 5.2.

```
{
    "text": "ideas about the thesis"
}
```

Listing 5.2: A Run-time State for writing note as JSON document

Search

Based on search Application State Model (Listing 5.6) valid example values for search state is shown in Listing 5.3.

```
{
    "query": "paderborn university",
    "submit": true
}
```

Listing 5.3: A Run-time State for search as JSON document.

6

Architectural Overview

In this chapter, the architectural overview of the run-time state migration approach is discussed. First, the publish-subscribe pattern is explained as a mean to exchange state information and manage available devices as well as which state models are supported by an application. The middleware is explained next, which uses the publish-subscribe pattern and acts as a message broker. Subsequently, the library as well as the interfaces that need to be implemented by the respective applications are explained. Moreover, we explain the message exchange in the different life cycle phases of the migration approach. The last part explains the Model Repository and suggests an improved concept.

6.1 Publish–subscribe pattern

An infrastructure is needed to exchange run-time states of applications across different devices. Also, devices should know which other devices with common Application State Models are available and notice if they join or leave the network. Moreover, devices should be able to migrate the run-time state and acknowledge other devices when migration is done.

In this approach, we chose to use the publish-subscribe pattern. Thereby, the middleware is the message broker, devices are publishers and subscribers, Application State Models are topics, and run-time states are messages. Also, there are extra topics and messages that allow us to fulfil requirements such as device introduction, device connectivity and finalizing the run-time state migration.

Figure 6.1 illustrates an overview of publish–subscribe pattern in our approach. A Mac device (see ❶) subscribes to *Model-A* and *Model-B* topics, a Windows device (see ❷) subscribes to *Model-A* topic and an Android device (see ❸) subscribes to *Model-B* topic. The Mac device publishes a message contain *"X"* to topic *Model-A*, and the middleware makes sure it gets forwarded to the Windows device. Also, a message containing *"Y"* is published from the Mac device to the topic *Model-B* and forwarded to the Android device by the middleware.

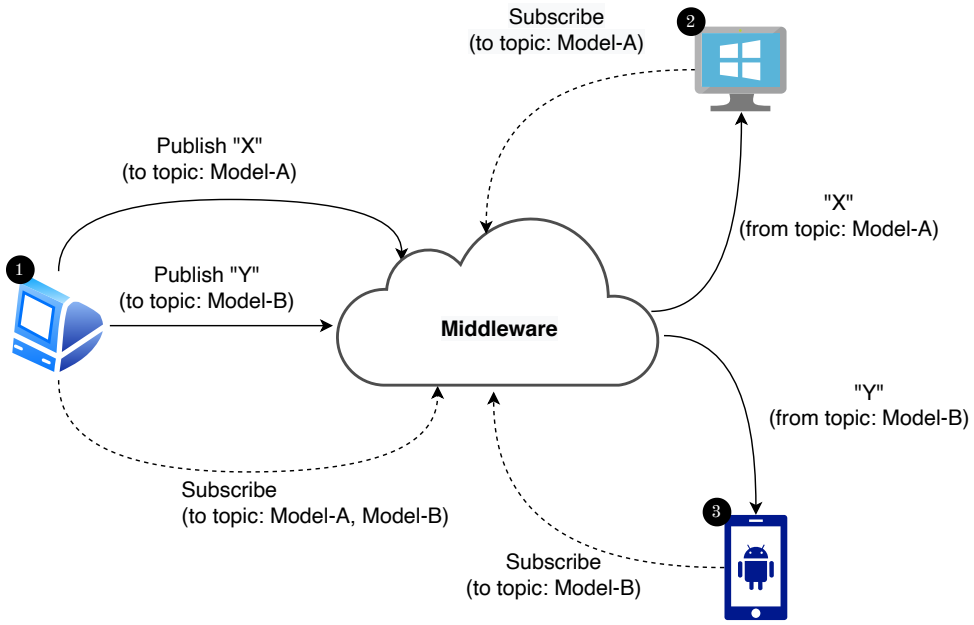

Figure 6.1: Overview of Publish–subscribe pattern in our approach.

6.1.1 Middleware

As the middleware acts as a message broker, if devices are not on the same network, a middleware should exist in which applications can communicate with each other. Figure 4.1 shows the role of middleware. This middleware can handle sending and receiving the run-time state. The goal of having this middleware is applications which coupled with libraries, can introduce themselves, find each other, and migrate the run-time state.

Also, as mentioned in **R4 Device Management**, the middleware is responsible for device discovery and notifying other devices if a device joins or leaves with the same Application State Model. The message exchange for device discovery is explains in subchapter 6.5.

As in this thesis, we considered the publish-subscribe pattern as the architecture, the middleware is the intermediary message broker.

6.1.2 Topics

There are some topics each device should subscribe to them. Devices who subscribed to these topics receive a message if another device publish a message to them.

Topic for Connectivity Status

All devices should subscribe to a topic about their connectivity status. This topic is `online`. If a device joins or leaves the network, other devices get notice by a message.

Topics for Application State Models

Each Application State Model has a topic. Devices that supports these Application State Models should subscribe to these topics. So, if a device joins the same topic, it should notify other devices about joining and supporting the same Application State Model and also other devices notify the new device about their existence as a response. Moreover, for each Application State Model, devices should subscribe to the subtopic of device identification, that they can receive an individual message from other devices and vice versa. As users may use different devices and

applications device identification must be unique like UUID; to improve the readability of the following example, we use a combination of the operating system and application name as the UUID. Of course, the is only done for the examples in the thesis and would lead to name clashes in a broader scenario. In a live setting, the UUID would be randomly generated.

Example Scenario

Considering the scenario of Figure 5.2. Three devices share some Application State Models. All devices should subscribe to `online` topic. Mailspring supports three Application State Models, so it should subscribe to topics:

```
online
search
new-event
sending-email
```

Also, to guarantee that other devices on the same topic can send message to Mailspring, it should subscribe to these subtopics with its identification:

```
search/mac-mailspring
new-event/mac-mailspring
sending-email/mac-mailspring
```

As K-9 Mail supports two Application State Models, it has to subscribe to these topics:

```
online
search
sending-email
search/android-k9mail
sending-email/android-k9mail
```

Also, as Woven supporting one Application State Model, it has to subscribe to these topics:

```
online
new-event
new-event/windows-woven
```

6.1.3 Messages

In this architecture, we need some message to fulfill our requirements like device introduction and transferring a run-time state.

Device Introduction

When a user runs an application, the application should introduce itself to other devices on the network. Moreover, other devices should get notify and response about their existence by an introduction.

Device Leave

After an application got closed or a device went offline, it should notify other devices about its absence. Moreover, other devices should not be able to communicate with that device anymore.

Device Has Run-time State

If a device has a run-time state, it should notify other devices which support common Application State Model about having a run-time state.

Request Run-time State

A device can request a run-time state of other devices supporting the common Application State Model. The source device should respond to this request by sending its run-time state.

Send Run-time State

As the target device can request a run-time state, the source device should respond to this request by sending its run-time state. Also, the target application should be aware of receiving it.

Finalizing Run-time State Migration

After the target device received a run-time state and is adjusted into the application, the target device should notify the source device about finalizing the run-time state migration to react to this message like by resetting its UI and removing its run-time state.

6.1.4 Actions

In this architecture, we need some actions to fulfill our requirements, like store a run-time state and adding the new device that joined recently.

Store Run-time State

When a device has a run-time state, it should be stored in case other devices request a run-time state migration.

Add Device

Whenever a device receives a Device Introduction message, it should add the device to its list so that it can access it in the future.

Remove Device

Whenever a device receives a Device Leave message, it should remove that particular device from its list to prevent further communication.

6.2 Library

As state in **R1 Applicable on Existing Applications**, this approach is meant to support run-time state migration for existing applications on different platforms; special libraries in different programming languages and platforms should be developed. These libraries should provide basic functionality such as communication, validation of run-time states based on an Application State Model, and an API to support extraction and injection of run-time states between same-purpose applications with common Application State Model.

Developers can integrate these libraries in existing applications to enable run-time state migration, and they should implement some glue code in existing source code to adapt the

library into existing applications (Figure 4.1). The library reflect are the generic part of run-time state migration, the glue code and the interfaces derived from the application state model reflect the application specific part.

These libraries should be written in the same programming languages as the source and target applications. They should be able to communicate directly or with the help of the middleware. If devices are on the same local network, these libraries should establish a point-to-point connection. Otherwise, they should be able to communicate with help the middleware over the internet. In this case, they should introduce themselves to the middleware to find each other; then, they can migrate the run-time state. As mentioned before, the focus of the thesis is not providing different types of communications, only indirect communication with help of the middleware should be implemented.

6.3 Interfaces

Interfaces are derived from Application State Model, and they guarantee certain values on a run-time state. They can be auto-generated or manually written by developers with same programming language of the source and target applications. To integrate the libraries into source code, developers should implement some glue code and use the interfaces like Figure 4.1. Developers should add interfaces to the library with glue code. Thereby, the library will know which Application State Models has to be supported.

Considering Mailspring written in TypeScript, Listing 6.1 shows a generated interface based on search Application State Model (Listing 5.6).

```
1  export interface SearchObject {
2      /**
3       * the query of search
4       */
5      query: string;
6      /**
7       * shows if the query is already has been submitted
8       */
9      submit?: boolean;
10 }
```

Listing 6.1: Search example interface in TypeScript.

6.4 Life Cycles

After explanation of entities of the architecture in previous sections, the interaction between these entities in different life cycle phases is explained. These life cycles phases consist of main entities, actions and sequence of messages discussed in the messages section, and they are implemented in chapter 7.

6.4.1 Initializing

After the user starts an application, an initialization step should happen to introduce itself to the other applications. This step uses the Device Introduction message. Figure 6.2 shows a source application that joins the network and introduces itself with a message to library. Library publishes this message to all supported topics of Application State Models (e.g search), and the middleware forwards it to the library of the target application which subscribed to the same topic. Thereby, the library of the target application add the new device to a device list and notifies the target application about a new device. In response, target application introduce itself in the same way to the source application by using the before mentioned device specific subtopic for this specific application (e.g search/<source-device-uuid>).

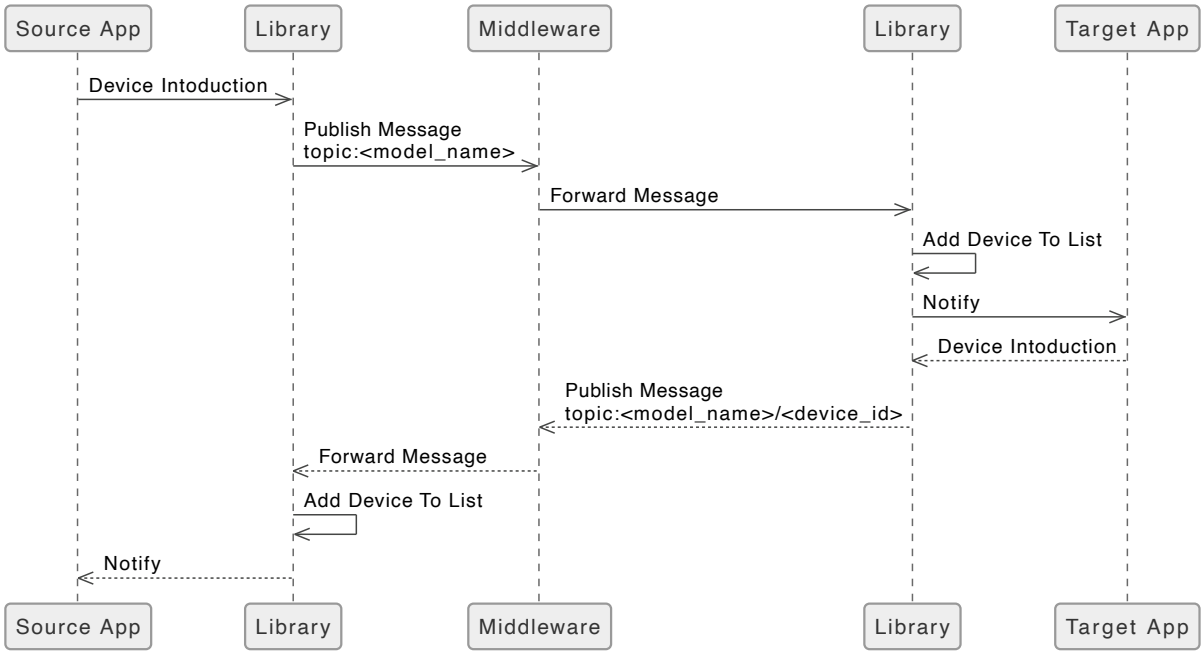

Figure 6.2: Initializing the Source Application

6.4.2 Going Offline

An application might go offline intentionally or unintentionally, e.g. due to faults or network outages. A plan should cover these incidents. This step uses the Device Leave message. After disconnection, other devices are not allowed to send any message to that particular device.

Graceful

A user might close the application. In this case, the application goes offline gracefully, and it should notify other applications about its absence. Figure 6.3 shows the source application going offline gracefully. With the library's help, the source application informs its absence to the middleware on `online` topic, and the middleware forwards it to the library of the target application. Thereby, the library of the target application removes the device from devices list and notifies the target application about the leaving of the source application.

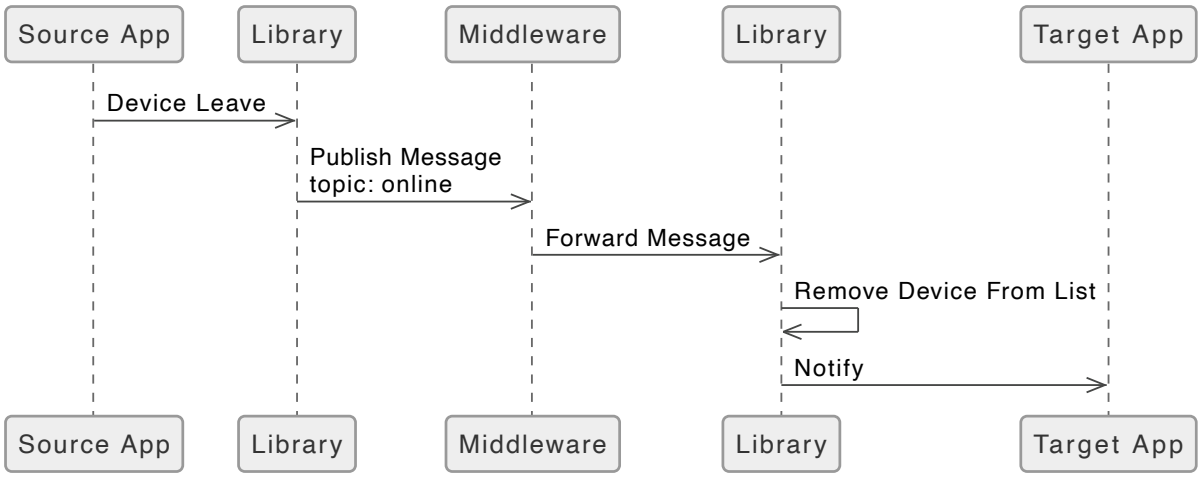

Figure 6.3: Going Offline Gracefully: Source

Ungraceful

An application might go offline by other causes like network problems, device crashing, application failure, etc. In this case, the application goes offline ungracefully, and the middleware should notify other applications about its absence. Figure 6.4 shows the source application going offline ungracefully. The middleware checks if the application is connected; if it does not get any response, request gets a timeout and it publishes a message to `online` topic. Thereby, the library of the target application removes the device from devices list notifies the target application about the leaving of the source application.

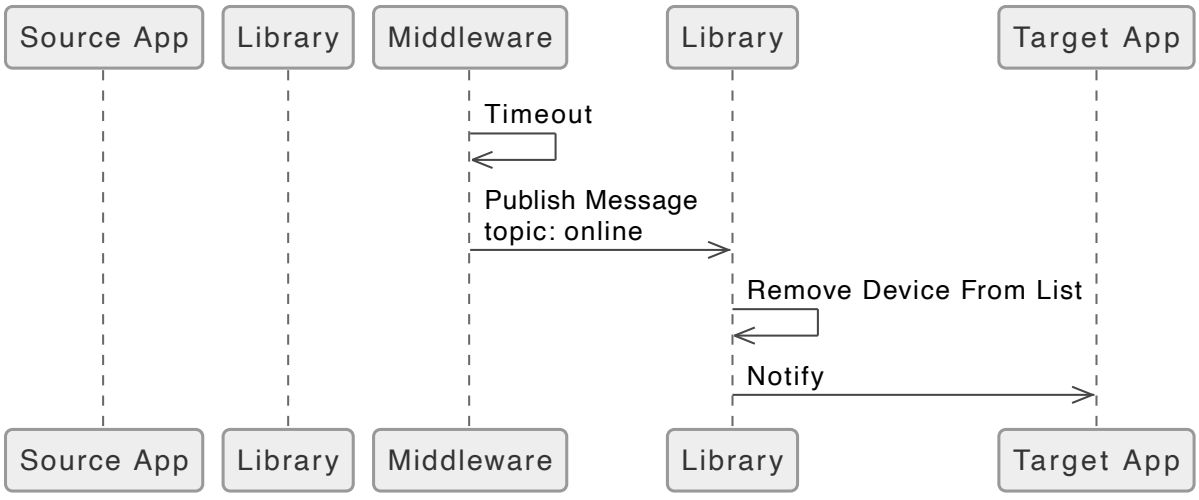

Figure 6.4: Going Offline Ungracefully: Source

Has Run-Time State

Any application with a run-time state should notify other applications about having a run-time state, so that they aware that the run-time state exists and can be migrated. This step uses the Device Has State message. Figure 6.5 shows the source application has a run-time state and informs the middleware by publishing a message to topics of Application State Models and the middleware forward it to the library of the target application. Thereby, the library notifies

the target application about the source application, which has a run-time state. The target application may react by enabling a button that would allow pulling the run-time state from the source to the target application. We explain these reactions later in Migration Patterns section.

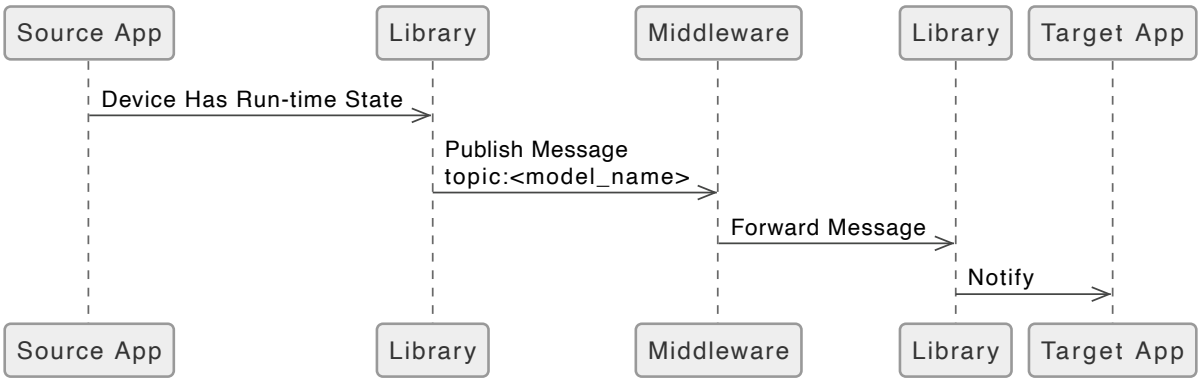

Figure 6.5: Source App inform other devices that has a state

6.4.3 Store Run-time State

As different applications have different events for observing the changes on their current state information, run-time state should be temporarily stored and ready for migration if other applications request it. Figures 6.6 shows that the source application can temporarily store its run-time state in the library.

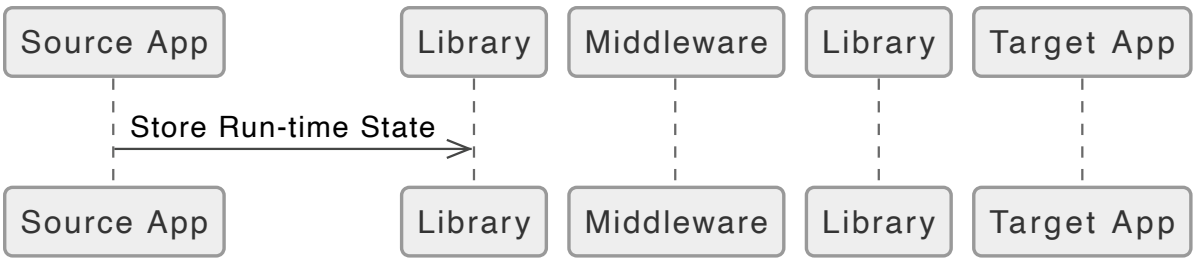

Figure 6.6: Source Application stores a run-time state in library.

6.4.4 Migration Patterns

In this section, we describe two patterns for run-time state migration.

Pull Method

In the pull method, the target application request a run-time state from the source application. Figure 6.7 shows the target application gets the list of devices with a common Application State Model, then selects the source application and requests its run-time state by sending a message from the library to the middleware. The middleware forwards the message, and the source application gets a request for migrating its run-time state. After processing the request, the source application sends its run-time state by the library to the middleware. The middleware forwards the message, and the target application receives the run-time state. The target application adjusts the new run-time state and remove its run-time state and notifies the source application about finalizing the run-time state migration by sending a message throw the library and the middleware.

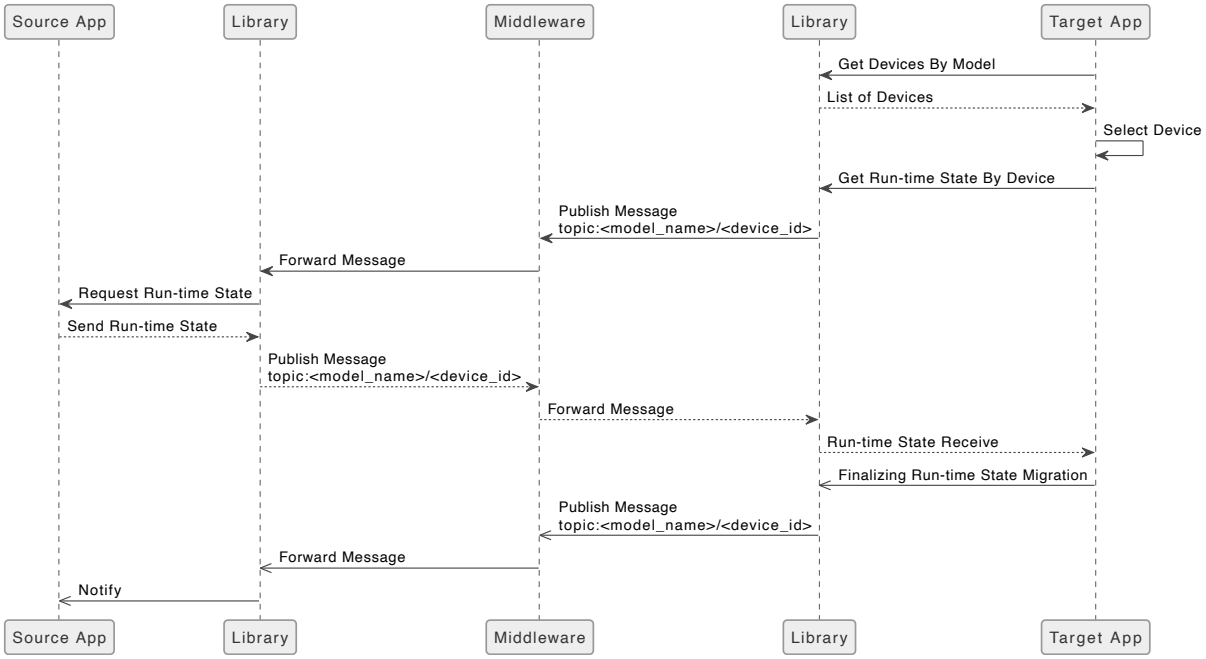

Figure 6.7: Pull Method: Migration Source to Target

Push Method

In the push method, the source application send its run-time state to the target application without a request by force. Figure 6.8 shows the source application gets the list of devices with a common Application State Model, then selects the target application and requests its run-time state by sending a message from the library to the middleware. The middleware forwards the message, and the target application receives the run-time state. Target application adjusts the new run-time state and remove its run-time state and notifies the source application about finalizing the run-time state migration by sending a message throw the library and the middleware.

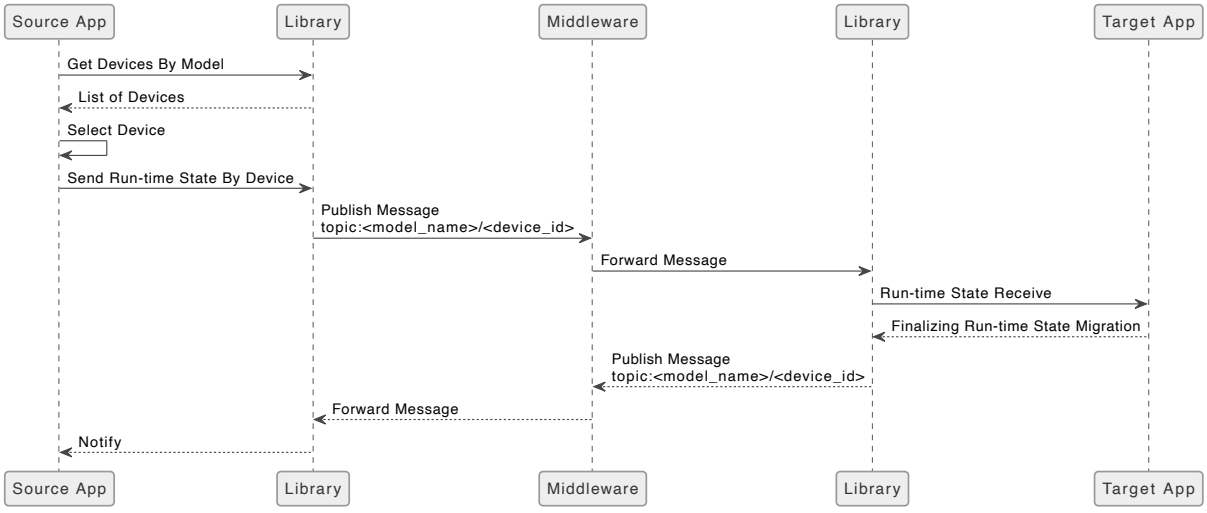

Figure 6.8: Push Method: Migration Source to Target

6.5 Model Repository

As stated in requirement **R3 Model Repository**, we made a repository manager call Model Repository located on GitHub[1]. Some models are available on this repository. Developers may use the search ability of GitHub to find their suitable Application State Models. Also, after making a new model, developers can add it the repository manager by making a pull request. So, other developers may use it for their applications.

Currently, a GitHub repository is sufficient for this thesis. Although conceptually, we may be able to improve the Model Repository so that developers may easily search the model they need by name, description, or keywords and download or customize them. In addition, if the developer does not find the model they need, they could create a custom model using the Model Repository editor. Additionally, if they have already developed a model, they may upload it to the Model Repository. The concept behind the solution is valid; however, implementation is a future work objective. Figure 6.9 is an improved Model Repository that shown as a wireframe concept.

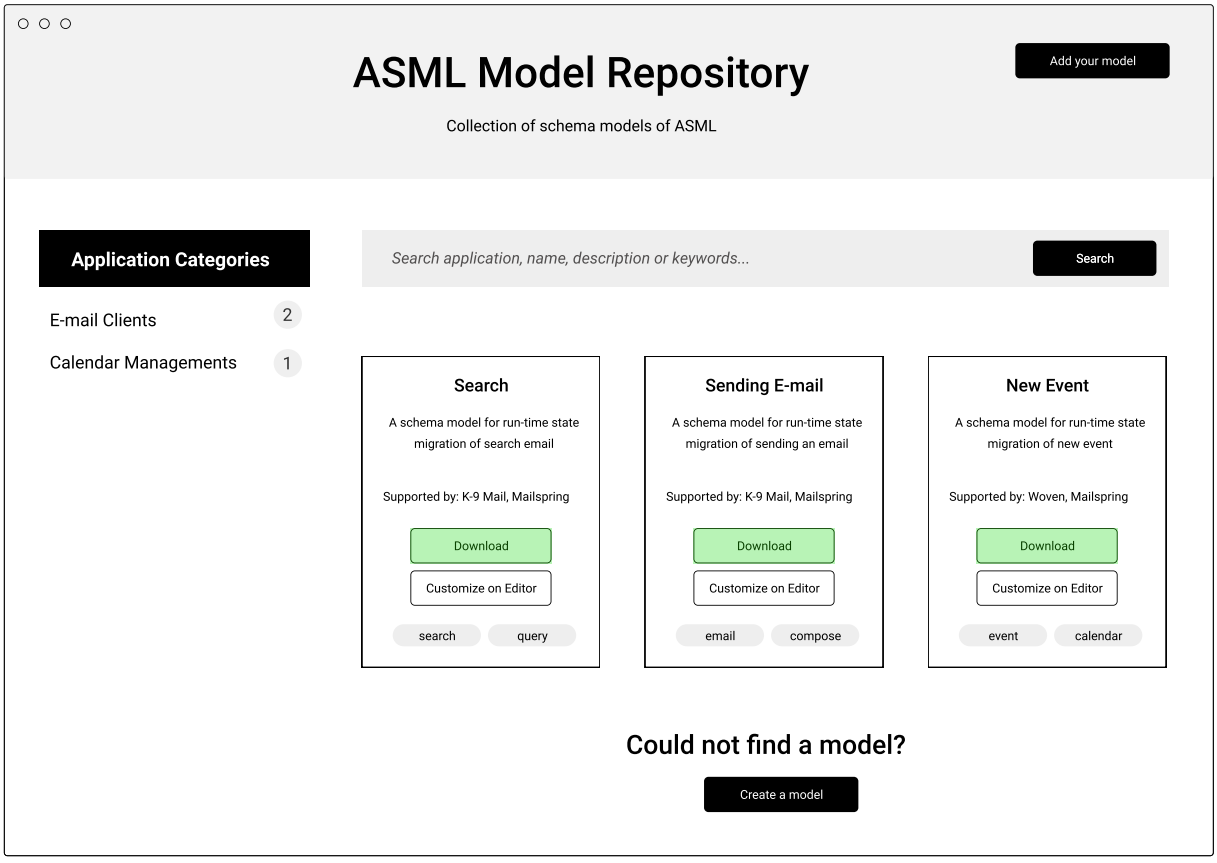

Figure 6.9: A wireframe concept for Model Repository

[1] https://github.com/asml-lang/model-repository

7

Implementation

This chapter discuss the implementation of the different architectural parts of the approach, which are outlined in the chapter before.

7.1 Middleware

As stated in the architectural overview, a middleware shall be provided that devices can use to communicate with each other and exchange run-time states. In this prototypical implementation, we used MQTT Protocol and Eclipse Mosquitto™[1] which is a MQTT-based message broker.

7.1.1 MQTT

MQTT is a lightweight messaging protocol for the Internet of Things (IoT). As it allows connecting a large number of devices and bi-directional communications, it is ideal for being a message broker in a publish/subscribe infrastructure. The protocol supports a variety of popular programming languages and platforms[29]. MQTT is lightweight and optimizes network bandwidth because of small message headers. As many devices rely on unreliable cellular networks to communicate, the support for persistent sessions in MQTT reduces the time it takes for the client to reconnect with the message broker.

We use an open-source MQTT broker, Eclipse Mosquitto™, which offers a MQTT server and client implementations that are compliant with relevant standards. [30]

7.1.2 Server Specifications

We installed Mosquitto on a VPS with Ubuntu 18.02 operating system to have a running MQTT broker. This VPS has a public IP address so that devices can connect over the internet as clients. The hardware specification of this VPS is 1 CPU, 1GB RAM, and 20GB SDD.

7.2 Deriving Interfaces

Developers should derive interfaces from Application State Models. These interfaces must be written in the programming language of source and target applications. Developers should write a glue code to connect the source code of the existing application and use Application State Models' interfaces. These interfaces should be used as a type for a run-time state, and they

[1] https://mosquitto.org/

guarantee these values. Also, some methods in these interfaces should be exist for parsing run-time state values. These interfaces can be written by developers or get generated with ASML CLI, a helper tool explained in 7.5.3. Listing 7.1 shows an example generated interface as data type for search Application State Model in Java. Also, Listing 7.2 shows an example usage of deserializing method (Converter) generated by ASML CLI.

```java
public class SearchClass {
    private String query;
    private Boolean submit;

    /**
     * the query of search
     */
    @JsonProperty("query")
    public String getQuery() { return query; }
    @JsonProperty("query")
    public void setQuery(String value) { this.query = value; }

    /**
     * shows if the query is already has been submitted
     */
    @JsonProperty("submit")
    public Boolean getSubmit() { return submit; }
    @JsonProperty("submit")
    public void setSubmit(Boolean value) { this.submit = value; }
}
```

Listing 7.1: Example code generated for search Application State Model in Java.

```
private onStateReceive(modelName: String, device: Device, state: String) {
    if (modelName == "search") {
        Search searchState = Converter.fromJsonString(state);
        searchView.setQuery(searchState.query, searchState.isSubmit)
    }
}
```

Listing 7.2: ASML CLI example code usage.

7.3 Libraries

Two libraries are developed in two different popular programming languages, which are JavaScript and Java. These libraries are using the native API of their platform (Electron and Android) to help developers implement run-time state migration. Each library is using a client library

for MQTT Protocol to manage the communication with other devices. Developers must write some glue code to integrate libraries into existing applications. Figure 7.1 shows the component diagram of the library. The *ASML Schema* is used by *Validator* to validate a run-time state. The *Library API* is the implementation of API Reference which can communicate with MQTT Library. The *MQTT Library* can publish messages via *Middleware*. Moreover, developers can write glue code and call methods of the library via *Library API*.

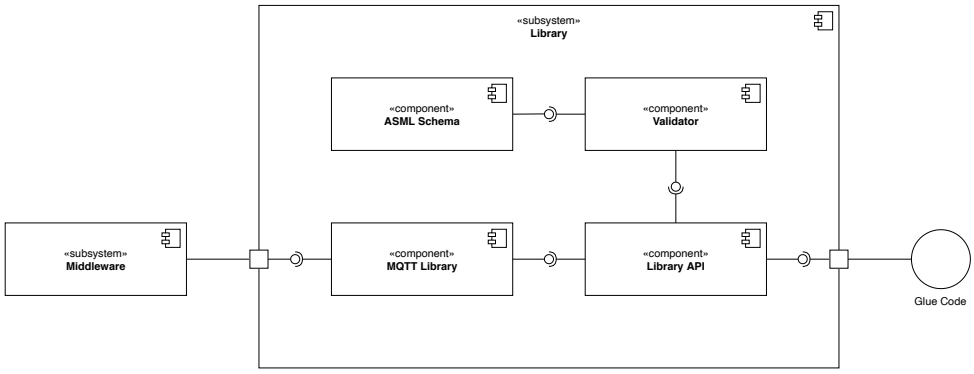

Figure 7.1: The component diagram of the library

7.3.1 API Reference

In the following, the API reference for the library is provided. This API reference is support by the two existing implementation for JavaScript and Java. Also, if other developers want to support other programming languages, they can follow this API reference.

addModel

Each Application State Model must be added to the library. This method allows developers to add a valid Application State Model's support by adding its interface as an input parameter.

getModel

This method allows retrieving the added Application State Model by its name.

getModels

This method allows retrieving all Application State Models that the device is supporting.

introduce

An application should introduce itself to other applications on the network via the middleware. In this method, configuration parameters must be provided. These parameters are device identification, device and application name, middleware IP address, and a value that shows if it is a new device. The library should generate a unique device identification like UUID. This method subscribes the device to the online topic, and all topics of supported Application State Models, then publishes configuration parameters as a message in JSON format via MQTT Protocol to corresponding topics. The middleware forwards the message to all devices that support the same Application State Model. A callback *onDeviceJoin* which explained in 7.3.2, will be called on all recipients of an introduction message and they should add the new device to their devices list. Also, they responds with an introduction message but only to the new device. Thereby,

the new device adds other devices to the devices list. For instance, Application State Models which Mailspring supports, are *search* and *sending-email*. Also, K-9 Mail supports *search* and *sending-email* as well. Listing 7.3 shows the introduction of Mailspring published to all devices which support *search* and *sending-email*.

As each application and platform has different life cycle and entry points, calling this method is part of the glue code and needs to be coupled with the application. So, when an application starts, this method should be called.

```
// topics:
// search
// sending-email
```

```
{
   "action":"device",
   "data":{
      "device":{
         "id":"35fe23c2-bf7c-4988-b934-46f7ba12a807",
         "name":"Mailspring - macOS"
      },
      "new":true
   }
}
```

Listing 7.3: The device introduction message.

Also, Listing 7.4 shows K-9 Mail introduction respond which published to Mailspring's *search* and *sending-email* topics.

```
// topics:
// search/35fe23c2-bf7c-4988-b934-46f7ba12a807
// sending-email/35fe23c2-bf7c-4988-b934-46f7ba12a807
```

```
{
   "action":"device",
   "data":{
      "device":{
         "id":"46f7ba12a807-bf7c-4988-b934-35fe23c2",
         "name":"K-9 Mail - Android"
      }
   }
}
```

Listing 7.4: The device introduction respond message.

getDevice

This method allows fetching the configuration parameters of the current device. These parameters should set by the introduction method.

setState

Developers needs to extract the run-time state and adjusted it to corresponding interface. To keep track of run-time time state and extract it, different events might occurs on changing a run-time state on different programming languages (e.g onClick or onChange). This method allows developers store the current run-time state after adjustment in the library. If a device requests a run-time state, it is ready to get migrated.

setHasState

A device must have a run-time state to migrate. This method notifies other devices about having a run-time state for a particular Application State Model. This method sends a message in JSON format via MQTT Procotol to all supported Application State Models topics and inform them whether this device has a state or not. Other devices should add or remove this device in the list of their devices based on the value in the message. For instance, as mentioned before Mailspring supports *search* and *sending-email*, Listing 7.5 shows Mailpsring publish a message to corresponding topics.

```
// topics:
// search
// sending-email
```

```
{
   "action":"has-state",
   "data":{
      "device":{
         "_id":"35fe23c2-bf7c-4988-b934-46f7ba12a807",
         "name":"Mailspring - macOS"
      },
      "value":true
   }
}
```

Listing 7.5: Mailspring informs other devices that has a run-time state.

getDevices

A list of available devices that support common Application State Models should be stored in the library. This method allows fetching all available devices for a particular Application State Model. Moreover, it should be distinguishable if a device has a run-time state or not.

getStateDevice

After the target device got chosen from *getDevices* method, the source device can request its run-time state. This method allows the get a run-time state of a particular Application State Model from a particular device. This method should send a message in JSON format via MQTT Protocol to a device from which a state shall be retrieved and request its run-time state. Listing 7.6 shows an example in which Mailspring requests a run-time state of *search* Application State Model from K-9 Mail by publishing a message to the corresponding topic.

```
// topic:
// search/46f7ba12a807-bf7c-4988-b934-35fe23c2
```

```
{
   "action":"request-state",
   "data":{
      "device":{
         "_id":"35fe23c2-bf7c-4988-b934-46f7ba12a807",
         "name":"Mailspring - macOS"
      }
   }
}
```

Listing 7.6: Mailspring request a run-time state from K-9 Mail.

sendState

This method allows devices to send a particular run-time state to a particular device. This migration process can be the push method, which sends the run-time state directly to a target device, or the pull method, which is a response to the request of *getStateDevice* method. Listing 7.7 shows K-9 Mail sends the run-time state of *search* Application State Model to Mailspring by publishing a message to the corresponding topic.

```
// topic:
// search/35fe23c2-bf7c-4988-b934-46f7ba12a807
```

```
{
   "action":"response-state",
   "data":{
      "device":{
         "_id":"46f7ba12a807-bf7c-4988-b934-35fe23c2",
         "name":"K-9 Mail - Android"
      },
      "state":{
         "query":"this is a test",
         "submit":true
      }
   }
}
```

Listing 7.7: K-9 Mail sends run-time state of *search* to Mailspring.

setMigration

This method allows the target device to notify the source device about finalizing the run-time state migration process. This method should be called when the target application adjusted the new run-time time state to its UI. This method should send a message in JSON format via MQTT Protocol to the source device and inform the end of migration. Also, the source application should react upon receiving this message by removing its run-time state. For instance, if an e-mail draft is migrated, this draft can be closed on the source device. Listing 7.7 shows Mailspring sends the migration message of *search* Application State Model to K-9 Mail by publishing a message to the corresponding topic.

```
// topic:
// search/46f7ba12a807-bf7c-4988-b934-35fe23c2
```

```
{
   "action":"migration",
   "data":{
      "device":{
         "_id":"35fe23c2-bf7c-4988-b934-46f7ba12a807",
         "name":"Mailspring - macOS"
      }
   }
}
```

Listing 7.8: Mailspring sends migration message to K-9 Mail.

onMessage

This method should be called on applications which receive a message from the middleware. This method is responsible of message process and filtering and calling a corresponding callback. For instance, when a device receive a message with `{"action": "response-state"}` (Figure 7.7), this method calls the *onStateReceive* callback, which explained in 7.3.2.

onOnline

This method should be called on applications which receive a message about connectivity status of other devices from the middleware. This method process the online status. If the a device goes offline, this method calls *onDeviceLeave* callback, which explained in 7.3.2.

7.3.2 Callback Events

Devices should get a notice when they receive a message via the middleware. The library is responsible for the process of selecting messages for reception and processing them by *onMessage* method. There should be callback events that allow developers to implement a proper behavior. As different applications have different life cycles, this implementation should be part of the glue code.

onStateRequest

This event should be called on the source application when the target application requests a run-time state via *getStateDevice* method.

onStateReceive

This event should be called on the target application when the source application sends a run-time state via *sendState* method.

onStateMigration

This event should be called on the source application when the target application is finalized the run-time state migration and sends a corresponding message via *setMigration* method.

onDeviceJoin

This event should be called on any device when a device that supports the same Application State Models joins the network via *introduction* method.

onDeviceLeave

This event should be called on any device when a device that supports the same Application State Models leaves the network.

7.3.3 JavaScript Library

In recent years JavaScript is one of the most popular programming languages and supports various platforms. Also, TypeScript is a programming language that is a superset of JavaScript and transcompiles to JavaScript [7]. In our approach, the API Reference methods are implemented in a JavaScript library. This library is written in TypeScript, which supports static typing.

The library source code is available on GitHub[2] and published on Node Repository Manager (NPM)[3].

The JavaScript library is using `MQTT.js` and `async-mqtt` libraries to bring MQTT Protocol in our JavaScript library.

7.3.4 Android Library

As the run-time state migration approach is meant to be supported on different platforms, we implemented the API Reference methods in another library for Android to support mobile devices. This library is written in Java, and the source code is available on GitHub[4]. Also, it is published on Bintray[5].

The Android library is using `paho-mqtt` library to bring MQTT Protocol in our Android library.

7.4 Demo Applications (MVP)

Two demo applications are developed as minimum viable products (MVP). The purpose of developing these applications is to test the state exchange and communication part of the approach without mixing it with the application logic of an existing application. Figure 7.2 shows the screenshot of these applications. A Desktop application[6] which is develop by Electron and written in JavaScript and an Android application[7] written in Java. Both applications are using the corresponding library.

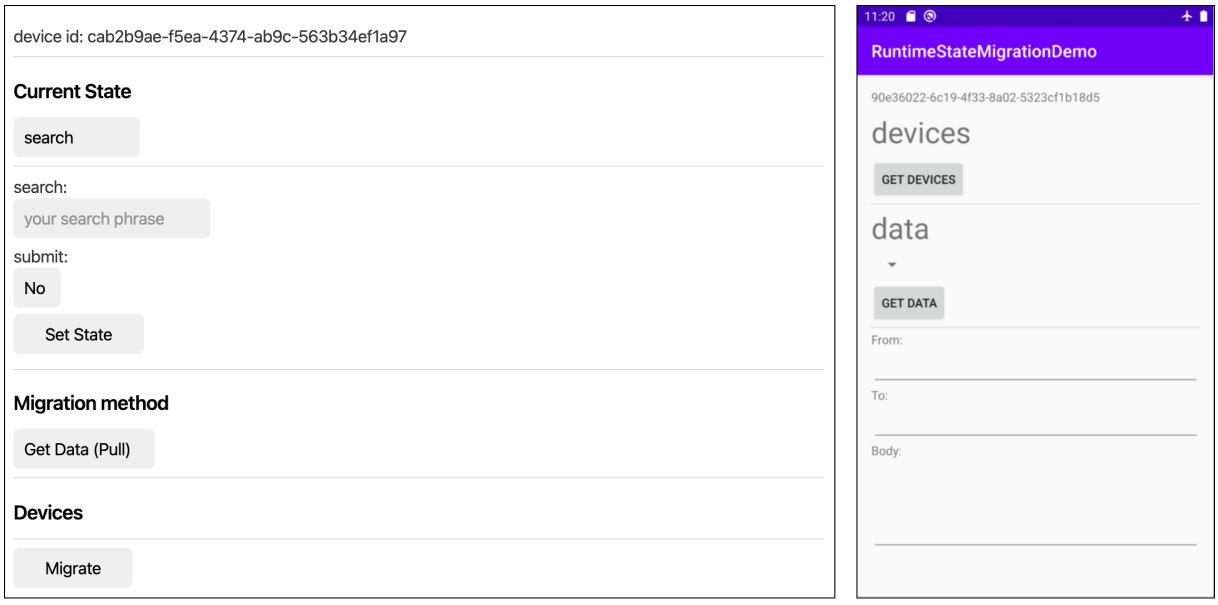

Figure 7.2: Electron (Left) and Android (Right) MVP Applications

[2] https://github.com/asml-lang/rsm-node
[3] https://www.npmjs.com/package/rsm-node
[4] https://github.com/asml-lang/rsm-android
[5] https://bintray.com/saman/maven/rsm-android
[6] https://github.com/asml-lang/rsm-demo
[7] https://github.com/asml-lang/rsm-demo-android

7.5 Helper Tools

In the section, we explain some tools which are implemented to help developers. These tools are designed to increase code quality, reuse existing code, and reduce development time.

7.5.1 ASML Schema Library

A repository hosted on GitHub[8] contains the latest version of Application State Modeling Language abstract syntax, which is JSON a Schema document. Also, this repository contains some examples and the DSL manual. Moreover, for ease of implementation, this repository is published on NPM[9]. Furthermore, other JavaScript packages in this thesis are using it as the main schema.

7.5.2 ASML Editor

ASML Editor is a playground web application[10] developed in HTML,CSS, and JavaScript. ASML Editor's purpose is to assist developers in writing Application State Models and validate a Run-Time State against it. In case of a validation or syntax error in any section, an error box shows up below that section and displays the error description like in Figure 7.3.

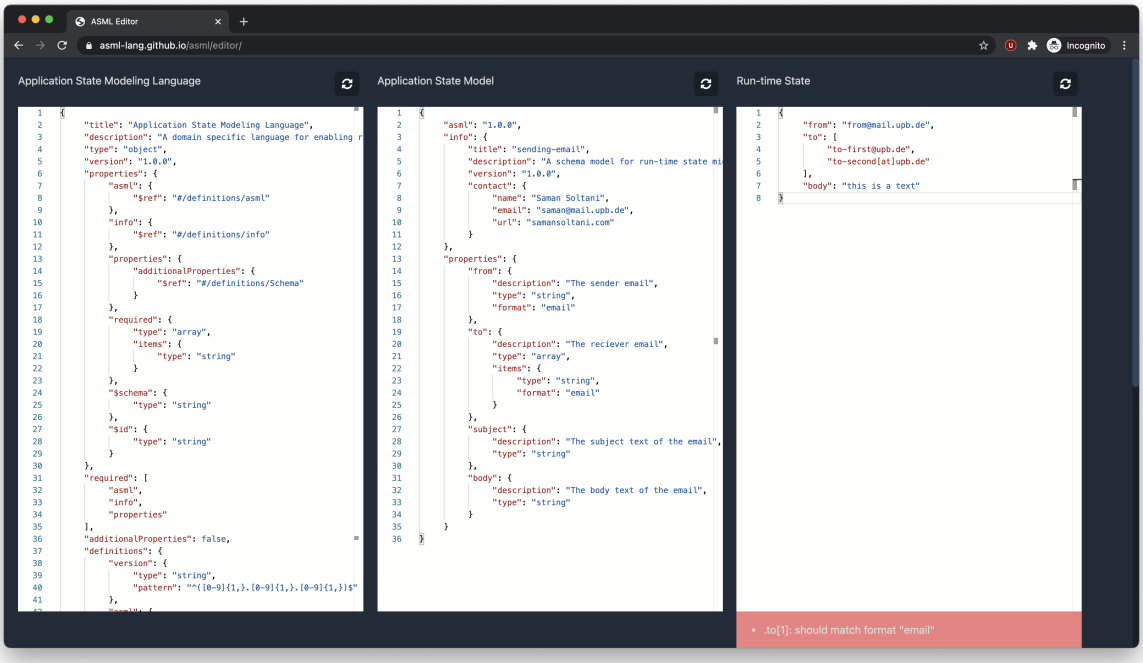

Figure 7.3: ASML Editor live playground

7.5.3 ASML CLI

ASML CLI is a command-line interface that helps developers to validate Application State Model and generate interfaces based on them for different programming languages. This tool is

[8] https://github.com/asml-lang/asml
[9] https://www.npmjs.com/package/asml
[10] https://asml-lang.github.io/asml/editor/

published on NPM[11].

This tool generates data types and some helpers methods for serializing and deserializing the run-time state in JSON. ASML CLI supports code generating in these programming languages: Ruby, JavaScript, Flow, Rust, Kotlin, Dart, Python, C#, Go, C++, Java, TypeScript, Swift, Objective-C, Elm.

7.5.4 ASML Validator Library

ASML Validator is a JavaScript library that allows a JavaScript application to validate the Application State Model and Run-time State against Application State Modeling Language. This library using ASML Schema Library as the main schema for validating. The source code of this library is available on GitHub[12]. Also, this library is published on NPM[13], and it is used in Run-time State Migration JavaScript Library, ASML Editor, and ASML CLI.

[11] https://www.npmjs.com/package/asml-cli
[12] https://github.com/asml-lang/asml-validator
[13] https://www.npmjs.com/package/asml-validator

8

Adaption of Example Applications

In this chapter, to show the feasibility of the run-time state migration approach, two open-source e-mail clients (Mailspring and K-9 Mail) adapted to support run-time state migration based on the developed approach as part of the thesis.

As a developer who wants to implement the enabling run-time state migration, we used the developed approach, middleware, libraries, deriving interfaces, helper tools, and adaptability to existing same-purpose applications.

8.1 Application State Models

After analyzing states of Mailspring and K-9 Mail applications and their source code (Table 4.1), we decided to model *search* and *sending-email* states. We have developed two common Application State Models in Chapter 5 for these two applications which are Listing 5.6 and 1. In this chapter, as a developer we add the support of these two Application State Models to Mailspring and K-9 Mail.

8.2 Example Applications

Suggestions for e-mail clients to be adapted are K-9 Mail for Android and Mailspring for desktop operating systems. These applications are for different platforms, and their source code is freely accessible to allow the integration of libraries. They are interactive applications for end-users and have sufficient complexity (e.g., applications do not have only one single state). K-9 Mail Android application is developed in Java and Kotlin; therefore, it requires the Android library, which we developed for run-time state migration. Moreover, Mailspring is developed using Electron and TypeScript for desktop operating systems, and it needs the JavaScript library for run-time state migration.

8.2.1 Mailspring

Mailspring is an open-source e-mail client application[1]. The source code of this application is available on GitHub[2]. This application can be installed on desktop operating systems like macOS, Linux, and Windows.

[1] https://getmailspring.com/

[2] https://github.com/Foundry376/Mailspring

Architecture

Because Mailspring is written in TypeScript (which is a superset of JavaScript) and has been built on top of Electron, its main script which specified in `package.json`, is referred to as the main process. The main process's script can display a UI by generating web pages. Only one main process is involved in an Electron application. Also, Electron has another type of process, which is the renderer process, and each web page runs in its own renderer process. Electron provides a special API for communication of main process and renderer processes [31]. Furthermore, Mailspring's UI is developed in ReactJS, which is a JavaScript UI library. Each part of the UI is a ReactJS component.

Adaptions

The run-time state migration JavaScript library and Application State Models interfaces are integrated into the main process's script. On the other hand, all UI adaptions are in ReactJS components. Adaptions are added to the fork of this project, hosted on GitHub[3].

Following is a brief description of the main implementation parts of in Mailspring.

Library Usage The main process's script of Mailspring is implemented in `application.ts` which is in `app/src/browser/` directory. We have imported the JavaScript library of run-time state migration, *search* and *sending-email* Application State Models and their interfaces in this script. Listing 8.1 shows how this import is done.

```
import RuntimeStateMigration from 'rsm-node';
import sendingEmail from '../models/sending-email.json';
import searchEmail from '../models/search.json';
import {
    SearchObject,
    Convert as SearchObjectConvert
} from '../models/Search';
import {
    SendingEmailObject,
    Convert as SendingEmailObjectConvert
} from '../models/SendingEmail';
```

Listing 8.1: Mailspring Adaption: Import of JavaScript library of run-time state migration, Application State Models and their interfaces

Application Initialization The main process's script has a entry point which is `start` method. An instance of JavaScript library has been made in this method with some configuration parameters. Also, callback methods which we have implemented in main process's script are bound to the instance of the library as parameters. Furthermore, Application State Models are added in this method with *addModel* method. Moreover, *DeviceIntroduction* method is being call here. Listing 8.2 shows a snippet that we use to implement the instruction above.

[3] https://github.com/asml-lang/mailspring

```
this.rsm = new RuntimeStateMigration(
  {
    name: 'Mailspring MAC',
    server: {
        url: 'mqtt://130.185.123.111',
        port: 1883
    }
  },
  this.rsmOnStateRequest.bind(this),
  this.rsmOnStateReceive.bind(this),
  this.rsmOnStateMigration.bind(this),
  this.rsmOnDeviceJoin.bind(this),
  this.rsmOnDeviceLeave.bind(this),
);

this.rsm.addModel(sendingEmail);
this.rsm.addModel(searchEmail);
this.rsm.introduce();
```

Listing 8.2: Mailspring Adaption: Application initialization necessary codes

Transferring Run-time State In case of any changing in inputs of *search* and *compose* views in Mailspring, the current run-time state of them is being stored with *setState* in library, in case of an application request for them. Also, other devices with the same Application State Model should be noticed whether this device has a state or not with *setHasState* method. Listing 8.3 shows how we used this method for *search* state.

```
const model_name = 'search';

this.rsm.setHasState(model_name, true);

const search: SearchObject = state;
this.rsm.setState(model_name, search);
```

Listing 8.3: Mailspring Adaption: Using the setState and setHasState methods

As we discussed in Chapter 7, when any device request for run-time state of the source application, the *onStateRequest* callback will be called. Listing 8.4 shows how we have implemented this method in Mailspring.

```
rsmOnStateRequest(data) {
   this.rsm.sendState(data.model_name, data.device.id);
```

```
}
```

Listing 8.4: Mailspring Adaption: Using the onStateRequest method

When a device receives a state the *onStateReceive* callback will be called. In Mailspring we wrote a code in *rsmOnStateReceive* method to find the corresponding window by and send the run-time state to UI with Electron `ipcRenderer` by *transferToWindow* method. In UI components we implemented a method to adjust the new run-time state. Listing 8.5 shows a method that adjust the new run-time state of *search* state.

```
rsmOnStateReceive(data) {
    this.transferToWindow(data.model_name, data.state);
}

// UI Code Example
_onState = (data, wndwKey) => {
    const search: SearchObject = SearchObjectConvert.toSearch(data);

    // Mailspring native variable and methods that we have changed
    this._searchQuery = search.query || "";
    this._submit = search.submit || false;
    this.trigger();
};
```

Listing 8.5: Mailspring Adaption: Using the onStateReceive method to adjust new run-time state

After finishing the migration, the target device should notify the source device about finalizing the run-time state migration with *setMigration* method. In Mailspring we changed the UI to call *setMigration* method when the run-time state adjusted. Listing 8.6 shows this implementation for *search* state. Also, when the source device receives the migration message and *onStateMigration* callback is being called, its the run-time state should be reset. Listing 8.7 shows how we implemented this callback.

```
const model_name = 'search';
this.rsm.setMigration(model_name, source_device_id);
```

Listing 8.6: Mailspring Adaption: Notifying source device the migration is complete

```
rsmOnStateMigration(data) {
    const search: SearchObject = {};
```

```
        this.rsm.setState(data.model_name, SearchObject);
        this.rsm.setHasState(data.model_name, false);
    }
```

Listing 8.7: Mailspring Adaption: Resetting the run-time state when receives a migration message

8.2.2 K-9 Mail

K-9 Mail is an open-source e-mail client application[4]. The source code of this application is available on GitHub[5]. This application can be installed on Android devices.

Architecture

K-9 Mail is written in Java 8. However, the new code base is partially migrated to Kotlin. The K-9 Mail project consists of different modules. The *k9mail* is the main module that includes code for database interaction, notification, and activities. Another module is the *k9mail-library* which is the back-end code for decoding e-mails and contacting mail providers.

Adaptions

As we worked only on the application, all changes are implemented in *k9mail* module. The run-time state migration Android library has been added to this module as a dependency. The *k9mail* consist of different packages. The integration of run-time state migration is implemented in *ui* package. Adaptions are added to the fork of this project, hosted on GitHub[6].

Following is a brief description of the main implementation parts in K-9 Mail.

Library Usage We have imported the Android library of run-time state migration, *search* and *sending-email* Application State Models and their interfaces in each corresponding class which are *MessageList.kt* for *search* state and *MessageCompose.java* for *sending-email* state. Listing 8.8 shows how this import is done for *search* state.

```
1  import com.github.asml.rsm.android.RuntimeStateMigration
2  import com.github.asml.rsm.android.interfaces.OnDeviceJoinListener
3  import com.github.asml.rsm.android.interfaces.OnDeviceLeaveListener
4  import com.github.asml.rsm.android.interfaces.OnStateMigrationListener
5  import com.github.asml.rsm.android.interfaces.OnStateReceiveListener
6  import com.github.asml.rsm.android.interfaces.OnStateRequestListener
7  import com.fsck.k9.models.SearchState
```

Listing 8.8: K-9 Mail Adaption: Import of Android library of run-time state migration, Application State Models and their interfaces

[4] https://k9mail.app/
[5] https://github.com/k9mail/k-9
[6] https://github.com/asml-lang/k-9

Application Initialization An instance of Android library has been made in *messageListUiModule* with some configuration. Listing 8.9 shows initializing the library.

```
RuntimeStateMigration.init(
        androidApplication(),
        Config(
            Server("tcp://130.185.123.111", 1883),
            "K9-Mail Android"
        )
)
```

Listing 8.9: K-9 Mail Adaption: Application initialization necessary codes

Listing 8.10 shows *MessageList* class make an instance of the library on its constructor. Also, callback methods which we have implemented are bound to the instance of the library with their setter methods.

```
private val RSM_MODEL = "search"
private RuntimeStateMigration rsm = RuntimeStateMigration.getInstance();

rsm.setOnDeviceJoinListener(this::onDeviceJoin)
rsm.setOnDeviceLeaveListener(this::onDeviceLeave)
rsm.setOnStateRequestListener(this::onStateRequest)
rsm.setOnStateReceiveListener(this::onStateReceive)
rsm.setOnStateMigrationListener(this::onStateMigration)
```

Listing 8.10: K-9 Mail Adaption: Instance of Android Library

Each class has a entry point which is `onCreate` method, which *DeviceIntroduction* method is being called there and Application State Models are added in this method with *addModel* method. Listing 8.11 shows a snippet that we use to implement the instruction above.

```
rsm.addModel(SearchState)
rsm.introduce()
```

Listing 8.11: K-9 Mail Adaption: Using *DeviceIntdocution* in Android Library

Transferring Run-time State In case of any changing in inputs of *MessageList* and *MessageCompose* views other devices with the same Application State Model should be noticed whether this device has a state or not with *setHasState* method. Listing 8.12 shows how we used this method for *search* state.

```
override fun onQueryTextChange(newText: String?): Boolean {
    rsm.setHasState(RSM_MODEL, newText?.isNotEmpty() == true)
    return false
}
```

Listing 8.12: K-9 Mail Adaption: Using the setHasState method

As we discussed in Chapter 7, when any device request for run-time state of the source application, the *onStateRequest* callback will be called. Listing 8.13 shows how we have implemented this method in K-9 Mail for *search* state.

```
fun onStateRequest(modelName: String?, device: Device?) {
    if (modelName == RSM_MODEL) {
        rsm.setState(
            RSM_MODEL,
            SearchState(
                searchView.query.toString(),
                submit_value)
            .toString()
        )
        rsm.sendState(RSM_MODEL, device?.id)
    }
}
```

Listing 8.13: K-9 Mail Adaption: Using the onStateRequest and sendState methods

As we mentioned before, when a device receives a state the *onStateReceive* callback will be called. In K-9 Mail we developed a method to distinguish between the received run-time state, assign it to corresponding view and adjust the new run-time state. Listing 8.14 shows a method that adjust the new run-time state of *search* state.

```
val searchState = SearchState.fromJsonString(state)
searchView.setQuery(searchState.query, searchState.isSubmit)
```

Listing 8.14: K-9 Mail Adaption: Using the onStateReceive method to adjust new run-time state

Like Mailspring, in K-9 Mail we use *setMigration* method to notify other devices about finalizing the migration. Listing 8.15 shows this implementation for *search* state. Also, the implementation of *onStateMigration* callback which resets *search* state in K-9 Mail shown in Listing 8.16.

```
rsm.setMigration(RSM_MODEL, device?.id)
```

Listing 8.15: K-9 Mail Adaption: Notifying source device the migration is complete

```
if (modelName == RSM_MODEL && searchView.isShown) {
    searchView.setQuery("", false)
}
```

Listing 8.16: K-9 Mail Adaption: Resetting the run-time state when receives a migration message

8.2.3 UI Adaptions

In this section, we explain all implemented UI adaptions in both applications to achieve a proper behavior of run-time state migration.

Notifications

When a device joins or leaves, other devices which are subscribed to the same Application State Model's topic will be notified by *onDeviceJoin* and *onDeviceLeave* callbacks. Devices that gets this message show a native notification on their application and inform the user about the connectivity status of the device. Figures 8.1 shows a notification on Mailspring and K-9 Mail about joining a new device to *search* topic.

Run-time State Migration Button

For each view of a state, we developed a floating action button as the main button of run-time state migration. When the user clicks on this button, applications display two other buttons as *Set State* for push method and *Get State* for the pull method. The user can start the migration process by clicking one of these buttons on the view of current run-time state. Figures 8.1 and 8.2 show run-time state migration button on different views of Mailspring and K-9 Mail.

Device List Modal Box

When the user clicks on the run-time state migration button and clicks on one of the migration methods' buttons, applications show a modal box. This modal box displays the name of the current run-time state, migration method, and the list of the devices available for migration by *getDevices*. The user should select a device from the list and click on the *migrate* button. By clicking on the *migrate* button, If the chosen method is push, the run-time state can be migrated by *sendState* method explained in 7.3.1. Otherwise, for the pull method, the run-time state can be requested by *getStateDevice* method explained in 7.3.1. In the end, the modal box gets closed. Figure 8.3 shows device list modal box on Mailspring and K-9 Mail.

Run-time State Adjustments

We adjust both applications for both *search* and *sending-email* states. If the target device receives a new run-time state by *onStateReceive* callback explained in 7.3.2, application react accordingly. For *search* state, the input of the query text gets updated with the source application's query text. Moreover, if the search is already submitted on the source application, the target device also tries to submit the query text and shows a result. Moreover, for *sending-email* state, if the target device gets a new run-time state by push method, it displays the new draft on a new compose window. Also, the user can click on the run-time state migration button on the compose window and pull and push the run-time state of the current window. After adjustment, the target device announces the end of migration to the source device with *setMigration* method explained in 7.3.1.

Removing Run-time State

When *onStateMigration* callback explained in 7.3.2 gets called, application resets all input of the corresponding run-time state. For *search* state, in query text input will be empty, and the list of results will be reset. For *sending-email*, the current compose window will be closed and the draft will be deleted.

UI Screenshots

In this section, we demonstrate UI adjustments in example applications.

Figure 8.1 shows a run-time state migration button which belongs to *search* state. Also, this figure displays notifications for joining K-9 Mail and Mailspring by sharing the *search* state.

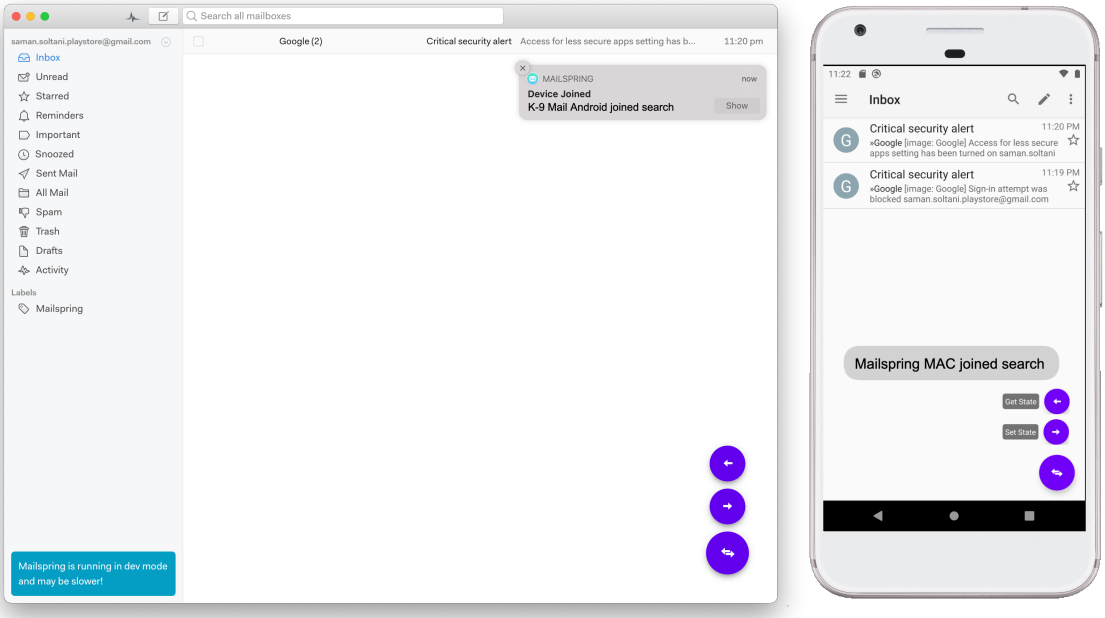

Figure 8.1: Screenshot of Native Notifications and Run-time State Migration Button

Figure 8.2 shows the compose window of Mailspring and K-9 Mail which have a run-time state migration button.

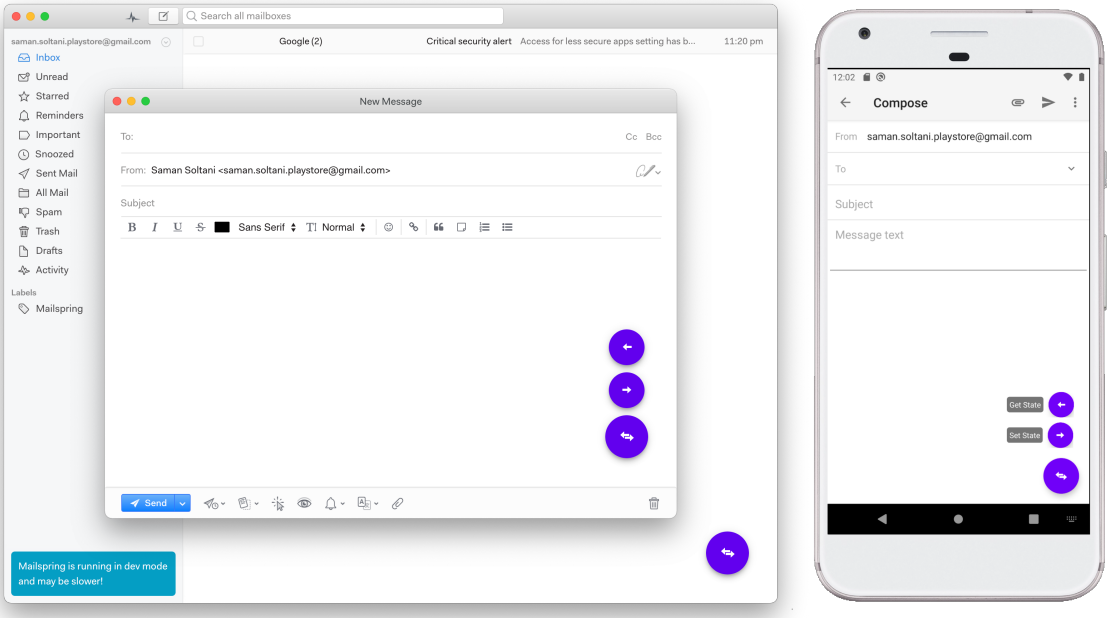

Figure 8.2: Screenshot of Compose Window

Figure 8.3 shows the device modal of Mailspring and K-9 Mail which have a migrate button.

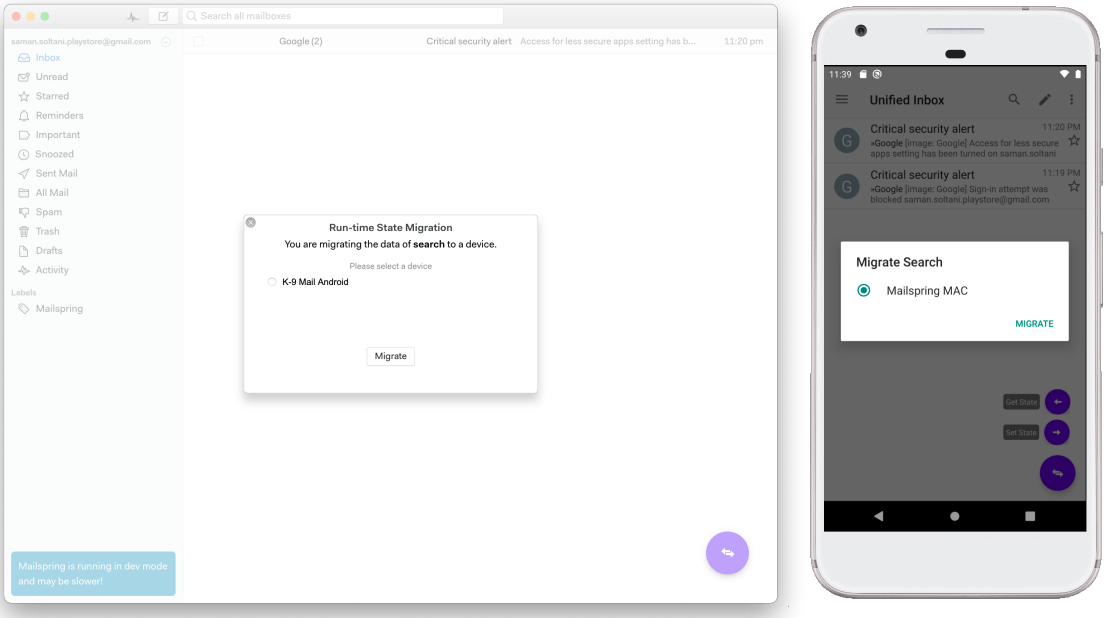

Figure 8.3: Screenshot of Device List Modal Box

8.2.4 Storyboard

Figure 8.4 shows a storyboard of implemented run-time state migration approach on two same-purpose applications which are Mailspring and K-9 Mail.

Shot 1 Shows only K-9 Mail on an Android device is in the network. The user clicks on compose e-mail button on K-9 Mail to write an e-mail

Shot 2 A compose form shows up on K-9 Mail and users writes the e-mail. She wants to switch to another device, so she runs Mailspring on a Mac device.

Shot 3 Mailspring which has the common *sending-email* Application State Model with K-9 Mail, joins the network. When Mailspring joins, K-9 Mail shows a notification to the user. She clicks on run-time state migration button.

Shot 4 K-9 Mail display migration options which *Set State* is for push method and *Get State* is for pull method. The user clicks on *Set State* as she wants to migrate from K-9 Mail to Mailspring.

Shot 5 The *Device List Modal* shows up on K-9 Mail and user choose the target application which is *MailSpring Mac* and clicks on *MIGRATE* button.

Shot 6 The compose form on K-9 Mail gets closed and a compose window appears on Mailspring with the same e-mail that was in the K-9 Mail. The user again wants to migrate the run-time state. She clicks on compose e-mail button on K-9 Mail.

Shot 7 User changes the e-mail on Mailspring and this time as she wants to migrate from Mailspring to K-9 Mail, clicks on *Get State* .

Shot 8 The *Device List Modal* shows up on K-9 Mail and user choose the source application which is *MailSpring Mac* and clicks on *MIGRATE* button.

Shot 9 The compose window on Mailspring gets closed and a compose form appears on K-9 Mail with the same e-mail that user have changed in Mailspring.

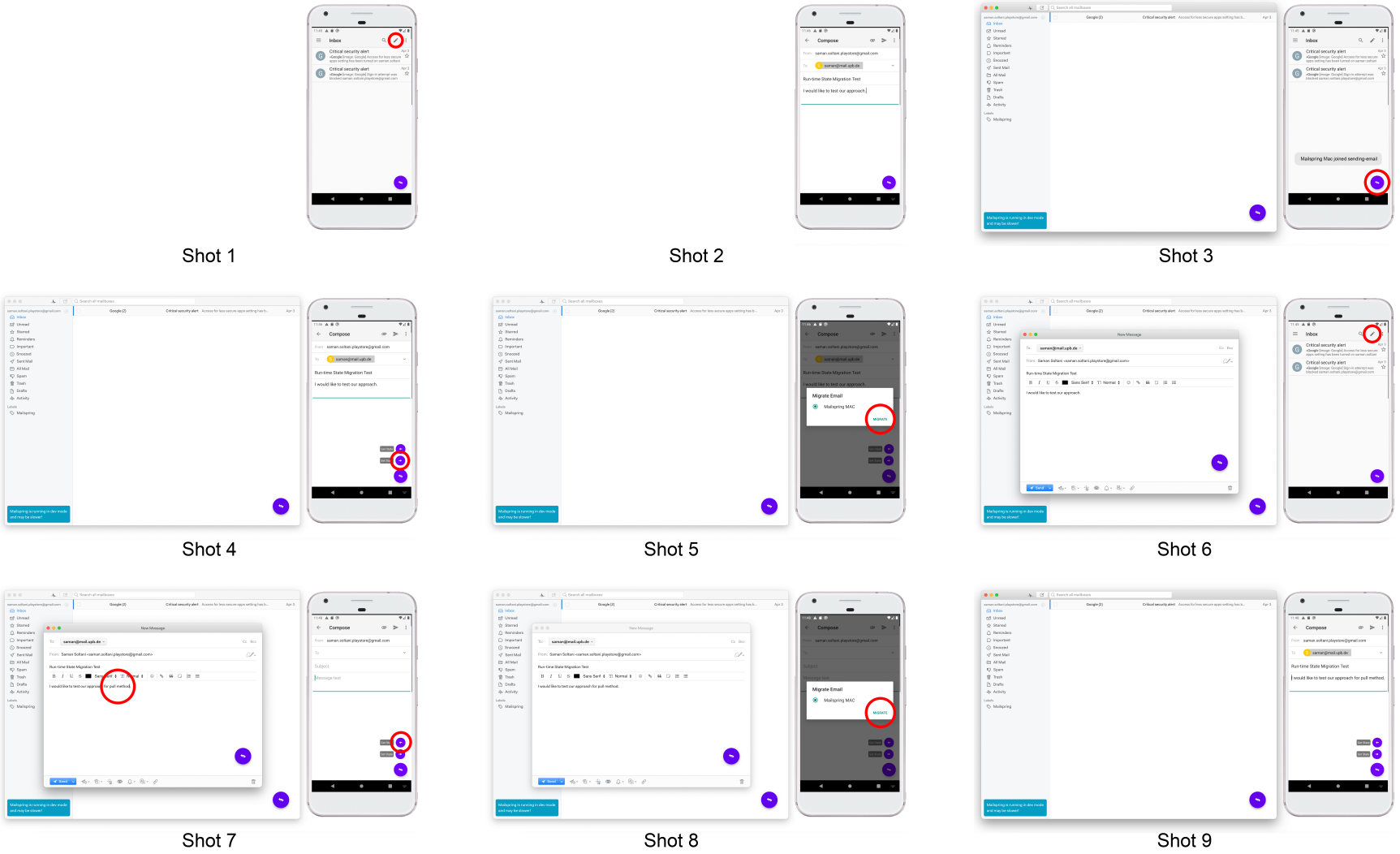

Figure 8.4: Storyboard of run-time state migration

9

Conclusion and Future Works

In this chapter, we summarize the thesis and discuss the future work.

9.1 Conclusion

This thesis presents an approach for enabling run-time state migration between same-purpose applications of different vendors. We investigate the problem by analyzing same-purpose applications and draft some requirements. We suggest a Domain-specific language for modeling run-time states based on JSON Schema. The name of this DSL is Application State Modeling Language (ASML) and allows developers to write a platform-independent state specification for each state. This state specification is the Application State Model. The ASML allows us to validate the Application State Model against its schema. Also, a run-time state can be validated against Application State Model. Moreover, we present a repository manager for Application State Models, called Model Repository. This repository allows developers to find and select an existing Application State Model. Also, they can write their model and contribute by adding it to the repository. We use the publish-subscribe pattern as the main architecture and discuss the role of the middleware as a message broker and its messages, actions, and topics. We decided to simplify the work by making libraries available to other developers. These libraries eliminate the need for developers to re-implement basic functionality, and the integration effort is limited to glue code that uses the interfaces and implements run-time state migration's life cycles. The middleware is based on MQTT Protocol with Mosquitto message broker. Two libraries in JavaScript and Java have been developed based on our API Reference. For testing purposes, we integrate these libraries in two demo applications. Additionally, Developers can rely on our helper tools to simplify their job. Finally, we evaluate the impact of our approach on existing real-world applications. We implement libraries in Mailspring and K-9 Mail and modify their user interfaces to accommodate our approach.

As a result, with our approach, it is now possible to adapt applications that do not support run-time state migration to be altered after their initial development by someone other than the original developer. The adaption can be accomplished by writing a bit of glue code, reusing our libraries, and implementing corresponding life cycles.

9.2 Future Works

In the future, we want to tackle the following points.

File Transfer Support Firstly, we want to add the support of file transfer. Currently, our state specification and implementation only support primitive data types.

Security Secondly, we want to add security to the commutation between the library and the middleware. At the moment, the run-time states are not encrypted. Thereby, it can be end-to-end encryption between devices.

Underlying Persistent State Thirdly, we want to consider the underlying persistent state in our implementation. Currently, we are only interested in the run-time state and are not concerned with the persistent data. For example, at present, two different e-mail accounts (University and Job) have the ability to migrate their run-time states. We would like to add the support for run-time states coupled to unaligned persistency; Meaning the run-time state only makes sense if it is coupled to the same persistent state. Because persistent data is used in migration, it has access to the same network, data storage, etc.

Improving Model Repository Fourthly, we are planning to make enhancements to the Model Repository. Model Repository is currently hosted on GitHub. However, we left the foundation for it by proposing a concept. It is our intention to implement the Model Repository as a web application so that every developer can rely on existing models and contribute their own models.

Refine the Modeling Approach Finally, we intended to support state migration between same-purpose applications based on Application State Models that are partially common but not exactly the same. We want to refine the modeling approach to cope with differences in an Application State Model, e.g., by using model transformations. For instance, in the source application, *firstName* and *lastName* values are contained in the run-time state; however, the target application only has *fullName* value within its run-time state. Thereby, By mapping the *fullName* to the combination of *firstName* and *lastName*, the approach supports the migration of run-time state from the source to the target application.

Bibliography

[1] C. P. Rahul Raj and Seshu Babu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference (INDICON)*, pages 625–629, 2012.

[2] Statcounter global stats - desktop vs mobile vs tablet market share worldwide - jan 2009 - apr 2020. https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#monthly-200901-202004 (visited on 2021-04-10), 2020.

[3] Fabio Paterno. *Migratory Interactive Applications for Ubiquitous Environments*. Springer-Verlag London Limited, 2011.

[4] A. Gallidabino, C. Pautasso, V. Ilvonen, T. Mikkonen, K. Systä, J. Voutilainen, and A. Taivalsaari. On the architecture of liquid software: Technology alternatives and design space. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 122–127, 2016.

[5] Tero Jokela, Jarno Ojala, and Thomas Olsson. A diary study on combining multiple information devices in everyday activities and tasks. In *ACM Conference on Human Factors in Computing Systems*, CHI '15, page 3903–3912, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, page 105–110, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *European Conference on Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*, volume 3. Morgan & Claypool Publishers LLC, Mar 2017.

[9] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., USA, 2002.

[10] Markus Volter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: Technology, engineering, management*. John Wiley & Sons, 2006.

[11] Tomaž Kosar, Pablo E. Martı´nez López, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[12] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[13] Benoit Combemale, Robert France, Jean-Marc Jezequel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering modeling languages: Turning domain knowledge into tools*. CRC Press, 2020.

[14] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *Ibm Systems Journal*, 45:621–645, 07 2006.

[15] Amazon states language. `https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html` (visited on 2021-04-10).

[16] Amazon states language specification. `https://states-language.net/spec.html` (visited on 2021-04-10).

[17] D. Q. Nykamp. State space definition. `https://mathinsight.org/definition/state_space` (visited on 2021-04-10).

[18] Stephanie Balzer. Contracted persistent object programming. In *PhD Workshop, ECOOP*, volume 12. Citeseer, 2005.

[19] Philip A Laplante, editor. *Dictionary of computer science, engineering and technology*. CRC Press, Boca Raton, FL, 2000.

[20] G. Wang. Improving data transmission in web applications via the translation between xml and json. In *International Conference on Communications and Mobile Computing*, pages 182–185, 2011.

[21] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *25th International Conference on World Wide Web*, WWW '16, page 263–273, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee.

[22] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. Json: data model, query languages and schema specification, 2017.

[23] Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. On-demand cross-device interface components migration. In *International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '10, page 299–308, New York, NY, USA, 2010. Association for Computing Machinery.

[24] Fabio Paterno', Carmen Santoro, and Lucio Davide Spano. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16(4), November 2009.

[25] Sonja Zaplata, Hamann Kristof, Kottke Kristian, and Winfried Lamersdorf. Flexible execution of distributed business processes based on process instance migration. *Journal of Systems Integration*, 1, 09 2010.

[26] Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Push and pull of web user interfaces in multi-device environments. In *International Working Conference on Advanced Visual Interfaces*, AVI '12, page 10–17, New York, NY, USA, 2012. Association for Computing Machinery.

[27] Y. Xu, S. Li, and G. Pan. Scudosgi: Enabling facility-involved task migration in osgi framework. In *International Conference on Frontier of Computer Science and Technology*, pages 125–131, 2009.

[28] Jishuo Yang and Daniel Wigdor. Panelrama: Enabling easy specification of cross-device web applications. In *SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, page 2783–2792, New York, NY, USA, 2014. Association for Computing Machinery.

[29] Konglong Tang, Yong Wang, Hao Liu, Yanxiu Sheng, Xi Wang, and Zhiqiang Wei. Design and implementation of push notification system based on the mqtt protocol. *2013 International Conference on Information Science and Computer Applications (ISCA 2013)*, 2013.

[30] Roger Light. Mosquitto: server and client implementation of the mqtt protocol. *The Journal of Open Source Software*, 2, 05 2017.

[31] Maria Parshina. Javascript beyond the browser. 2018.

Implementations

All links related to implementations can be found below.

- ASML Schema Library
  - Source code: https://github.com/asml-lang/asml
  - Package: https://www.npmjs.com/package/asml
- ASML Editor
  - Web Application: https://asml-lang.github.io/asml/editor/
  - Source code: https://github.com/asml-lang/asml/editor
- ASML CLI
  - Source code: https://github.com/asml-lang/asml-cli
  - Package: https://www.npmjs.com/package/asml-cli
- ASML Validator Library
  - Source code: https://github.com/asml-lang/asml-validator
  - Package: https://www.npmjs.com/package/asml-validator
- JavaScript Library
  - Source code: https://github.com/asml-lang/rsm-node
  - Package: https://www.npmjs.com/package/rsm-node
- Android Library
  - Source code: https://github.com/asml-lang/rsm-android
  - Package: https://bintray.com/saman/maven/rsm-android
- Mailspring
  - Original version: https://github.com/Foundry376/Mailspring
  - Adapted version: https://github.com/asml-lang/Mailspring
- K-9 Mail
  - Original Version: https://github.com/k9mail/k-9
  - Adapted Version: https://github.com/asml-lang/k-9

- Demo Applications
  - Android: https://github.com/asml-lang/rsm-demo-android
  - Electron: https://github.com/asml-lang/rsm-demo

Snippets

.1 Composing E-mail Example Model

```yaml
asml: 1.0.0
info:
  title: sending-email
  description: A schema model for run-time state migration of
     sending an email
  version: 1.0.0
  contact:
    name: Saman Soltani
    email: saman@mail.upb.de
    url: samansoltani.com
  keywords:
  - email
  - e-mail
  - compose
properties:
  from:
    description: The sender email
    type: string
    format: email
  to:
    description: The receiver email
    type: array
    items:
      type: string
      format: email
  subject:
    description: The subject text of the email
    type: string
  body:
    description: The body text of the email
    type: string
```

Listing 1: Composing E-mail model as JSON Schema in YAML.

List of Listings

List of Figures

List of Tables