Thesis Proposal

Streamlining Reuse with Concern-Specific Modelling Languages

Student
Maximilian Schiedermeier

Supervisors
Prof. **Bettina Kemme** - Distributed Information Systems Lab
Prof. **Jörg Kienzle** - Software Engineering Lab

Committee
Prof. **Martin Robillard** - Software Technology Lab
Prof. **Muthucumaru Maheswaran** - Advanced Networking Research Lab

Presented for the
Thesis Proposal and Area Examination, Fall 2021

School of Computer Science
November 26, 2021

1 Introduction

In this report I provide detailed information about the current state of my ongoing doctoral thesis in Computer Science. I propose a thesis topic, elaborate on the underlying problem statement and argue how it formulates a relevant, yet unsolved challenge. Throughout this report I present components that in combination address the problem statement and also run a detailed assessment of their individual progress and peer validation. I also outline the required tasks for a successful completion and the envisioned time-schedule.
The main components of my thesis are: A novel Model-Driven Engineering (MDE) framework that targets advanced model reuse, and two case studies that demonstrate this framework's viability. One of the case studies is additionally strengthened through a user study.
The remainder of this document is structured as follows: In Section 1 I provide contextual background for my topic proposal and gradually refine a precise formulation of the scientific problem statement. Sections 2 to 4 deal with the main components of the proposed solution, including individual evaluations of their advancement and designated continuation. In Section 5 I recapitulate the presented components and place them into context, in regard to the original problem statement. The report concludes with an overall evaluation of the program progress, including a critical risk assessment.

Proposal Context

Modern software needs to cope with the ever increasing complexity of systems [4], and hence *reducing complexity* is a primary objective of software engineering. In *Model-Driven Engineering* (MDE) system complexity is reduced with the help of modelling languages, which each represent a given level of abstraction. Multiple modelling languages can also be combined, to allow the developer to express the properties of a system from various points of view, thus promoting *Separation of Concerns* (SoC). On top of SoC, MDE further counters system complexity by model reuse. While opportunistic reuse of entire models is a common practice, reuse of partial (and hence incomplete) models, is more challenging. Still, with aspect-oriented modelling (AOM) techniques it is possible to compose partial models from one context to another. In particular, this practice enables planned reuse, where models are intentionally reduced to partial models, in favour of higher genericness and therefore also higher potential for reuse.
Concern-Oriented Reuse (CORE) [1] is an approach based on AOM that streamlines model reuse by encapsulating common solutions as partial models inside a reusable unit called a *concern*. Concern users can then apply model transformations to connect partial models across levels of abstraction, effectively *reusing architectural and design knowledge*, or *platform-specific development expertise*. From the perspective of a concern user, this reuse process is experienced as three sequential stages [5]:

1. A *variation interface* (VI), which exposes the different variants of the concern with a feature model, and the impact of each variant on high-level system qualities with an impact model.

2. A *customization interface* (CI), where the concern designer exposes the generic entities in the concern that have to be adapted to a specific reuse context.

3. A usage interface (UI), which defines how the functionality encapsulated by a concern may be used, similar to a standard API.

CORE streamlines the reuse process by allowing a *concern user* to a) choose a desired variant (from the VI), b) adapt the chosen models to the specific reuse context (with the CI), and then c) use the structure and behaviour encapsulated by the concern (exposed in the UI).

In the above workflow, the concern user relies on *General Purpose Modelling Languages* (GPMLs) throughout *Customization* and *Usage* stages. GPMLs such as UML mostly cover the typical structural and behavioural modelling needs for software development, therefore for most concerns this provides an adequate level of abstraction. However, the SoC power of MDE is limited when it comes to development concerns that do not align with the levels of abstraction of the MDE process and the GPMLs used. Some development concerns, e.g., Security, need to be considered not only during the requirements phase, but also during architecture, design and implementation. Addressing security properly requires dealing with security-related structure and behaviour at all phases of development, and hence, security-related model elements end up scattered across multiple models. In this case, the use of modelling languages that are not aligned with the development concern in question introduces what is called *accidental complexity*. Accidental complexity arises out of mismatch of modelling language and modelled matter. In the context of CORE this translates to concerns that cannot be easily applied at CI and UI stage, due to their inherent mismatch on GPML concepts.

In general, the semantic gap between a specific application domain and the concepts offerend by GPMLs can be bridged with a *Domain Specific Modelling Language* (DSML) [12]. As such the idea to integrate concern-tailored DSMLs into the CORE reuse process is promising. Specifically, such a DSML would be part of the concern unit of reuse. In the following I therefore refer to this concept as a *Concern Specific Modelling Language* (CSML).

Unfortunately, while CSMLs bear the potential to ease the concern reuse process, they likewise add complexity to the concern integration task. Designing a purposeful concern is already by nature challenging: A concern designer must express thorough domain knowledge into versatile models, for maximized reusability and convenience. Adding the design and placement of a CSML into the concern integration process renders this an even greater challenge.

Therefore, the concern integration task itself should be assisted by a clear, modular plan of action that incrementally guides toward an expressive CSML-enabled concern.

In my proposal I provide a preliminary answer to this challenge. I present *FIDDLR*, a *F*ramework for the *I*ntegration of *D*omain-Specific Mo*D*elling *L*anguages with concern-oriented *R*euse. I discuss the framework in detail and present two case studies that allow for an evaluation of *FIDDLR*'s potential. Afterwards I summarize the current state of research and project how I intend to resolve remaining weaknesses of my approach in the near future. The report concludes with a short evaluation on the general thesis program advancement.

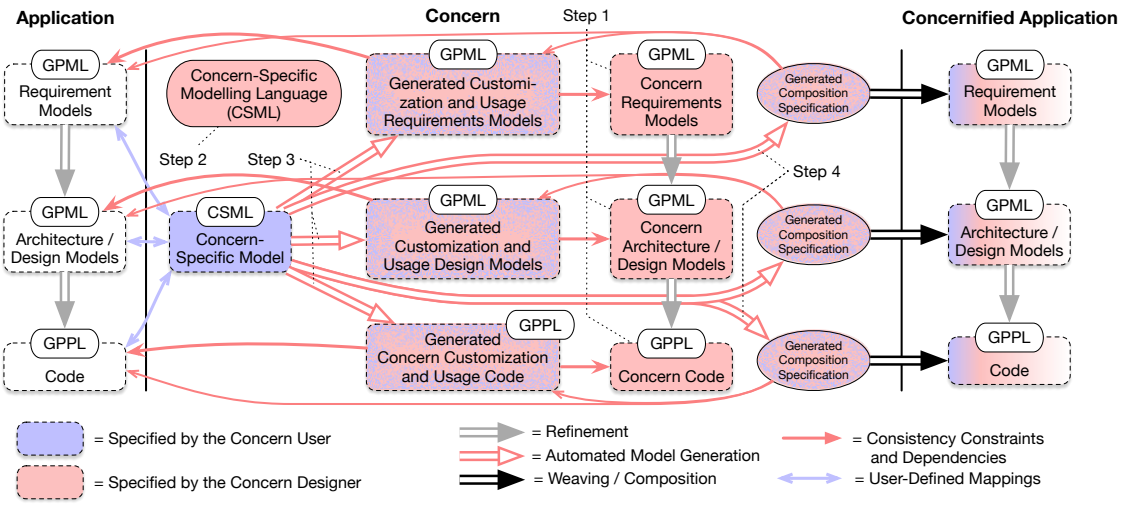

Figure 1: The *FIDDLR* Framework

2 *FIDDLR*

Motivated by the complementarity of MDE, DSMLs and CORE we elaborated *FIDDLR*, a *F*ramework for the *I*ntegration of *D*omain-Specific Mo*D*elling *L*anguages with concern-oriented *R*euse, illustrated in Fig. 1. The *FIDDLR* framework was presented in [15].

FIDDLR puts forward the idea that DSML technology can be exploited effectively for implementing and applying concerns that do not align well with standard GPML concepts. In particular, *FIDDLR* defines an approach for packaging a DSML with a concern, and as a framework provides clear tasks to integrate the concern implementation with MDE tooling, existing GPML models and code. *FIDDLR* therefore is beneficial for both concern designers and concern users.

2.1 Concern Design

In alignment with CORE, the unit of reuse in *FIDDLR* is the *concern*. Designing a concern is by nature a complex task. If a DSML is used within, it becomes even more complex. Even with FIDDLR, the design of a concern is still complicated, since the concern designer must excel in multiple disciplines such as DSML design, model transformations, and of course expertise on the concern's domain. The contribution of FIDDLR is that it splits the task of designing a concern into smaller, independent steps, namely *concern realization*, *CSML design*, *CSML→GPML transformation*, and *CSML→composition specification*. Each step reuses existing technologies whenever possible, thus simplifying concern design and reducing the amount of work required significantly. The *FIDDLR* concern design steps can even be distributed over a team of individuals that are experts in their field.

Step 1) Concern Realization In the spirit of MDE, a *concern designer* realizes a concern using the most appropriate GPML models at the right levels of abstraction. Fig. 1 depicts the artefacts created by the concern designer in red, i.e., concern-related requirements models,

architecture and design models, as well as code. Which models are needed depends on the MDE process being used, and on the nature of the concern. Some concerns are relevant at all levels of abstraction, e.g., *Security*, and therefore such concerns contain many realization models. This step should be performed by a developer with expertise in implementation of the concern, in collaboration with an expert of the GPML modelling languages used in the MDE process.

Step 2) CSML Design Whenever the nature of a concern and its properties do not align or can not easily be expressed with GPMLs, or when a concern covers several MDE abstraction layers, the concern designer can provide a CSML together with the concern realization models (also shown in red in Fig. 1) that exposes the main concepts of the concern and streamlines the concern customization and usage. This step should be performed by a DSML expert collaborating with the concern domain expert, who would define the language metamodel and actions for manipulating models.

Step 3) CSML→GPML Transformation The concern designer must also create model transformations that, given a CSML model as input, can generate the appropriate GPML models/code that customize and make use of the developed GPML realization models/code of the concern for each relevant level of abstraction. This step should involve a model transformation expert, possibly again in collaboration with a concern implementation expert.

Step 4) CSML→Composition Specification Finally, the concern realization expert and the MDE expert need to decide at which level of abstraction the concern-specific model is best composed with the application's realization models. For example, some of a concern's behaviour might best be composed at the code level, while other behaviour can better be composed at the level of state charts or sequence diagrams. A model transformation expert then designs a transformation that, given a CSML model and mappings provided by the user as input, produces composition specifications for the customized GPML models produced in step 3.

2.2 Concern Use

With *FIDDLR*, whenever an application reuses a concern that comes packaged with its own CSML, the concern user has access to language elements tailored specifically for the concern reuse. Thus the standard CORE reuse process [5] is significantly streamlined for the concern user. In standard CORE, the concern user has to manually customize each GPML realization model by mapping the generic model elements (and code) to application-specific elements (and code). Furthermore, for each GPML model of the application, the concern user must specify how the concern is used. Thanks to the CSML, the concern user can simply create a model describing the concern-related properties in the context of the application in which it is reused. This is shown in blue in Fig. 1. Customization and usage then only require linking the appropriate model elements from the created CSML model to model elements in the GPML models of the application as illustrated with the blue arrows.
An illustrative example for this practice is given in Section 3.

2.3 Concern Composition

To combine the application and concern models, *FIDDLR* reuses existing MDE, DSML and CORE tooling as much as possible. From the perspective of a concern user this step therefore runs fully autonomous.

From the concern-specific model provided by the concern user, the model transformations provided by the concern designer automatically generate GPML models that contain the application-specific customization mappings and usage of the concern API (step 3), as well as composition specifications that connect the generated models with the application models at each relevant level of abstraction (step 4). The automatically generated models and composition specifications are highlighted in speckled blue/red in Fig. 1. The composition specifications and models are then provided as input to the CORE model weavers, which generate the concernified application, i.e., the GPML models in which the concern-specific and application-specific structure and behaviour have been combined.

The model weavers then handle the actual fusion of partial concern models with the application's realization models. The outcome of this process is a completed application model that effectively reuses concern provided solutions. This model can be used as is for subsequent code-generation.

3 *RESTify*

In this section I extend the previously provided description of *FIDDLR* by a case study that illustrates the concern integration process guided by the *FIDDLR* approach. Concern integration is followed by an exemplary concern usage. The concern used for this case study is a concern named *RESTify*. *RESTify* is a concern to assist the conversion of legacy applications into RESTful services (see Section 3.1). This concern is a good case study subject, because of it's natural mismatch on GPML concepts. This justifies the integration of a CSML to sidestep accidental complexity and allow for an elegant concern integration and reuse.

The below case-study was likewise part of the general proposal of *FIDDLR* [15], with focus on two primary research questions:

- Can the *FIDDLR* framework be applied with reasonable effort, to design and implement a CSML-enabled concern?
- Does the provision of a CSML facilitated by FIDDLR streamline concern reuse?

To answer the second question, we needed two components: A functional version of the *RESTify* concern, integrated in accordance to the *FIDDLR* approach, as well as a sample reuse context, to which the concern can be applied to. For the latter we coded a minimal yet representative desktop application that provides access to a fictitious bookstore database [13].

3.1 Concern Description

RESTify is a concern closely related to the *Representational State Transfer* (REST) architectural style. REST allows the invocation of remote services through a resource-oriented interface. A manual refactoring process toward re-exposure of existing functionality through REST commonly requires thorough domain and technical expertise. The purpose of *RESTify* is the provision of assistance for the conversion of legacy applications into RESTful services. The concern simplifies the process by focusing the user's attention on design questions, while concealing implementation details as much as possible.

As mentioned initially, the essence of *RESTify* can not be easily grasped with standard GPML concepts. RESTful service interfaces consist of hierarchically structured resources with selected CRUD operations [2] (Create, Read, Update, Delete) enabled. Those operations are commonly invoked over HTTP as *Put, Get, Post* and *Delete* requests. Having a RESTful service therefore constitutes a layer of abstraction, as it strictly abstracts from service implementation details[1].

State-of the art libraries implementing REST provide means to define a resource tree definition by placing annotations that encode CRUD operations on individual URL-branches, each marking a location in the overall resource tree. The exact syntax of these annotations varies with the technology chosen. Listing 1 contains annotations specific to the Spring-Framework [16]. By decorating a Java method, these annotations map existing functionality to an HTTP method and URL location.

Listing 1: Spring Annotated Bookstore Method. Accessible by HTTP GET Request at e.g. "`/bookstore/stocklocations/minastirith`."

```
1 @GetMapping(value = "/bookstore/stocklocations/{stocklocation}", produces =
      "application/json; charset=utf-8")
2 public Map<Long, Integer> getEntireStoreStock(@PathVariable("stocklocation")
      ↪ String city) {
3     return stocksPerCity.get(city).getEntireStock(); }
```

In most cases these annotations are scattered over the code base. The entirety of placed annotations then implicitly defines a REST interface. However, at no point in time is the developer confronted with a visualisation or even textual summary of the overall designed interface. This conceptual mismatch imposes a high mental load on the developer. A side effect of this complexity is that real-world services often showcase misuse or even anti-patterns to the REST style [3]. A meaningful concern should draw the user's attention to explicit design choices, standing in contrast to implicit modelling via implementation details. This is why a guided process should conceal code-level annotations and their placement. Instead it should assist the explicit design of a REST interface, including mapping of CRUD operations on target or legacy functionality.

We argue that existing GPMLs cannot accurately capture the essence of these design choices, i.e., the selection of a REST framework, and the design of a resource layout and mapping of CRUD methods and parameters on existing functionality. Using a CSML however bridges the aforementioned semantic gap, by turning implicit into explicit modelling

[1] This notably distinguishes REST from simple *Remote Procedure Calls*.

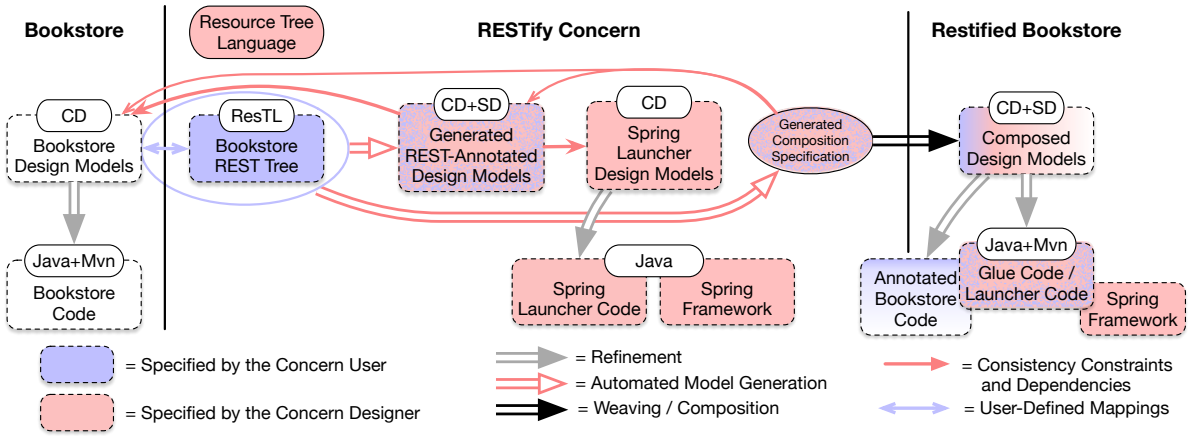

Figure 2: *FIDDLR* applied to the *RESTify* Concern

decisions.

We therefore elaborated a novel language for the specification of hierarchically arranged resources and basic CRUD operations. Note that in contrast to existing REST interface description languages, e.g. the *Web Resource Modeling Language* (WRML) [8] or *RESTful API Modeling Language* (RAML) [9], our language purposely omits the description of a complete REST interface specification. It is meant to integrate into a concern, therefore it is reasonable for language models to remain incomplete unless their partiality is combined with a mapping on application-specific *realization models*. In [12] we originally presented this idea of partial languages in the context of an integration into a refactoring workflow. This idea was presented even before we had elaborated a formal specification of *FIDDLR*. Next I summarize the steps performed by a concern designer, who integrates the aforementioned novel language as CSML into a *RESTify* concern. This is followed by a description of how the concern reuse is experienced from the perspective of a concern user, with special emphasis on the streamlined nature of a concern-assisted refactoring process.

3.2 Concern Integration

Fig. 2 illustrates the design process of the *RESTify* concern, if based on *FIDDLR*. As before, components and transformations provided by the concern designer are red, artefacts created by the concern user are blue, and speckled blue/red components depict generated components.

In a first step, the concern designer has to decide on the REST technologies the concern will support and express them in a feature model. At the time of writing the concern reference implementation only covers Spring. Support for alternative REST frameworks is in the making.

As we have seen before, the manual conversion required an implicit definition of REST resources via annotations. A more direct approach is an explicit design of the desired resource layout. However, existing GPMLs are not made for modelling resource trees, hence the concern designer should define a CSML for this specific purpose (step 2). We therefore elaborated the *Resource Tree Language* (*ResTL*), a CSML designed for the specification of hierarchically arranged resources and basic CRUD operations. Fig. 4 shows a possible model

that a concern user could design using *ResTL* to define a resource layout for the Bookstore.

Note that the concern designer does not need to define a language for establishing the mappings between the *ResTL* model and the legacy Bookstore application. CORE already provides a generic artefact for model element mappings, which allows the concern user to map CRUD operations to elements of the base application, i.e., the methods that the Bookstore offers.

One goal of *FIDDLR* is a maximized reuse of existing MDE and CORE concepts and tooling. The concern designer therefore has to decide at which levels of abstraction the REST concern is best integrated with the GPML models and code of the base application. For *RESTify* we decided to perform the integration at the design level only, e.g., using class diagrams and sequence diagrams, and rely on standard MDE code generation to produce the running application.

First, the concern designer creates design models (step 1). In case of Spring, this would be behavioural models to invoke REST launcher code required by Spring.

Also the concern designer would provide a CSML to bridge the semantic gap between the concern's level of abstraction and GPML expressiveness (step 2). In case of *RESTify* this corresponds to the provision of *ResTL*.

The next step is to provide a model transformation (step 3) that transforms the CSML model into GPML design models, i.e., that converts the mapped *ResTL* models into class diagrams and sequence diagrams that include the required annotations and trigger the Spring launcher behaviour during startup of the application.

Finally, in order to integrate the generated models with the functionality of the original application using CORE technology, the concern designer must provide a second transformation that, given the mappings between CSML and GPML models, produces *composition specifications* (step 4). These composition specifications, when given to the CORE weaver, compose the GPMLs of the base application with the generated GPML models containing the REST-specific information.

No further work is necessary. Notably for program code, it is not required to implement an adapted weaver or code generator, as the standard CORE weaver is used to compose the design models, and the standard MDE code generator is used to generate the executable.[2] In our case, this tooling is provided by the CORE reference implementation, TouchCORE [7].

In summary, from the perspective of a concern designer, *FIDDLR* requires only the definition of the *ResTL* CSML, the design models for launching Spring, and the two *model transformations* generating the GPML models and the composition specification. The technologies that are reused are the mapping-concept and the weaver provided by CORE, the code generator provided by MDE, and the Spring framework itself.

3.3 Concern Usage

This subsection illustrates how easy it is for a concern user to add a REST interface to an application following *FIDDLR*'s structured reuse process. By means of the Bookstore example we showcase how *RESTify* maximally focuses the concern user on REST-specific

[2]One insight of implementing *RESTify* was that an additional weaver for build-system configurations could be beneficial for the support of various REST technologies. This option is further discussed in Section 5.

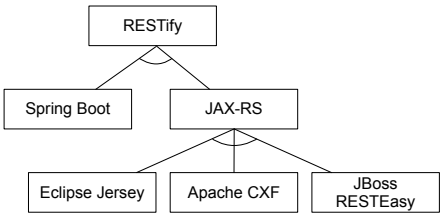

Figure 3: Variation Interface of the *RESTify* Concern

decision making and efficiently automates all technology-specific integration tasks.

From a concern user point of view *RESTify* is perceived as a sequence of three graphical model editors. Each editor allows explicit, but assisted decision making for an essential design question.

Equivalent to the manual approach, the process starts with selecting the desired REST technology. Where in the classic conversion a developer needs expert knowledge on viable alternatives and actions needed for their integration, *RESTify* offers this choice with a CORE-based variation interface (VI) that captures the technologies considered by the concern designer as shown in Fig. 3. The VI can also contain information on the impact of user made choices on resulting software qualities of the outcome, such as *performance*, *security*, *etc.* This information stems from an optional goal model provided by the concern developer.

Once the desired technology selected, the user is brought to the *ResTL* model editor. For illustration purposes we assume the concern user here decided for the *Spring Boot* variant. The concern user then models a possible resource layout as previously shown in Fig. 4, assisted by the editor that enforces a coherent layout. Thanks to the *ResTL* CSML provided by the concern designer, the concern user is maximally focused on this REST-specific design task. In case of *RESTify* the spotlight is on the definition and organization of resources and exposing of CRUD operations. Detailed REST interface information, e.g., input- and return parameters, is purposely omitted at this stage. As mentioned in 3.1 the *ResTL* as a CSML intentionally limits expressiveness to the requirements at a given reuse stage. More REST interface details are not needed here, as they can be semantically derived from subsequent mappings to the application realization models, in our case mappings to the design models of the Bookstore. This illustrates the key differences between a standard DSML and a CSML.

To connect the newly created resource layout with the Bookstore application logic, the concern user must now establish mappings between the CRUD operations of the resource tree and the methods of the Bookstore application. State of the art MDE tooling can perform an automatic signature extraction from existing artefacts, e.g. JAR files into design models. Depending on target signatures the user may also have to define additional mappings for method parameters.

The mappings are defined in a third editor that uses a split view: one side of the screen displays the class diagram, showing the Bookstore's classes and methods and the other side the *ResTL* model of the Bookstore's resource layout. The concern user then proceeds to establish links between individual CRUD operations and existing Bookstore methods. If needed, the user also provides mappings between signature parameters and intermediate dynamic resources. The latter are represented as dynamic path fragments (denoted as a placeholder enclosed by curly brackets). Afterwards, remaining un-mapped parameters are

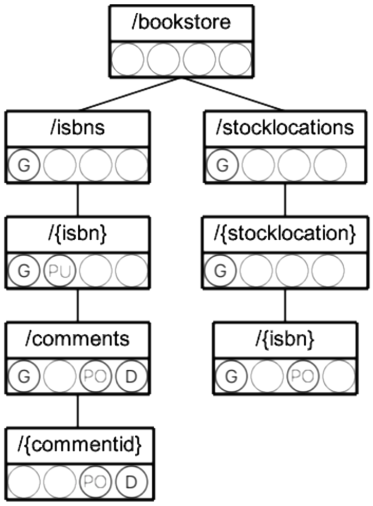

Figure 4: Bookstore Resource Layout Designed with the *ResTL* Editor. Circled Letters Below a Resource Represent Enabled CRUD (**G**et, **Pu**t, **Po**st, **D**elete) Operations.

assumed to be either HTTP query parameters or encoded as body payloads. Consequently it is not necessary that the concern user covers all legacy signature parameters with mappings. Fig. 5 depicts this split view and illustrates user-defined mappings between *ResTL* and legacy application Design Models.

This is all the concern user needs to do to add a REST interface to the Bookstore. From there, as described in the previous subsection, *RESTify* is able to internally perform the required model transformations, model weaving and code generation. The latter also produces a build system configuration that ensures automatic integration of Spring at compile time.

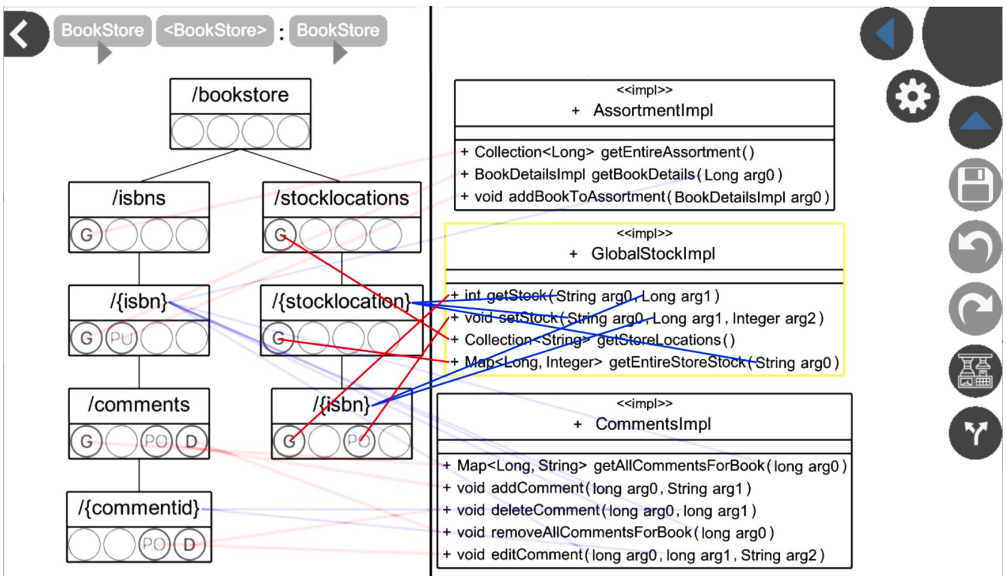

Figure 5: TouchCORE Screenshot showing Split View in Action. Mappings are here highlighted selectively to improve visibility.

3.4 Concern Contribution

We see the described concern usage workflow as convincing evidence for a maximally simple procedure, performed at the right level of abstraction and supported by the right modelling notations. Using *RESTify* no expert REST knowledge is required by the concern user. Likewise, compared to a maniual conversion there is no longer a need to deal with any technical details, e.g., framework-specific boilerplate code, annotation syntax or intricate configuration file modifications.

Another advantage of *RESTify* is the elimination of unnecessary redundancy. In [15] we run a quantitative comparison of both approaches. The manual Bookstore conversion showcased severe replication of resource strings, since URL paths closer to the resource tree root are replicated over scattered annotations. For instance the string of the root resource "`/bookstore`" was replicated 12 times. In *RESTify* models define every resource name exactly once, hence eliminating a source of potential errors. Furthermore we ran a quantitative comparison of atomic user operations required to accomplish the refactoring task. Although it is not simple to apply a fair metric to compare a textual with a tool-guided refactoring approach, we were able to demonstrate the streamlined nature of *RESTify* over the traditional refactoring process.

Finally, *RESTify* also facilitates evolution, as changes can be made efficiently at the right level of abstraction. This is an indirect consequence of *FIDDLR*'s strict separation of concerns, which enforces the concern user to diligently deal with one task at a time using the right modelling notation. For example, restructuring the REST interface's URL tree can be done easily by rearranging the resource tree layout in the *ResTL* editor. A second example is a bleated switching of the selected REST technology, which in a manual approach is an tedious process as it potentially touches vast parts of the code base.

Integration Status

Our current reference implementation of *RESTify*, which was entirely integrated following the *FIDDLR* approach is mainly functional and subsequent code-generation produces deployable REST service code that behaves as expected. Over the past years our main efforts were spent on a refinement of the existing TouchCORE code generator, notably to support the generation of build system configurations and software dependency management. Since REST technologies are JDK external artefacts, this was an inevitable requirement. Also we had to enrich the existing design models by a notion of annotations and annotation parameters, and likewise enable their support on the existing Java code generator. While the conceptual design of the *ResTL* CSML was relatively straightforward, the development of CSML→GPML and CSML mapping→GPML mapping transformations was a complex technical task that required more time and effort than expected.

The *RESTify* case study has already exceeded the initially envisioned proof of concept, and in terms of a thesis component can be considered mostly completed. Along its implementation the concern provided us with valuable information on *FIDDLR*'s general viability and continuously contributed to a gradual framework refinement. The remainder of this report shift the spotlight from what has already been done to open tasks. Open limitations of the*RESTify* concern, and how to solve them in the near future are mentioned in Section 5.

4 Case Study: Concern 2

The previously presented case study on *RESTify* provides evidence on *FIDDLR*'s viability and demonstrated the practical benefit of CSMLs in action. *RESTify* constitutes an essential component of my contribution, to underline the versatility of the suggested framework.

I intend to elaborate a second CSML-enabled concern case study as important component of my thesis. However, at the time of writing I can not present a fully developed second example. In parallel to the development of *RESTify* we did however consider two potential candidates, one of which will very likely be subject to further elaboration and implementation over the next year.
In the remainder of this section I provide the conceptual backgrounds of these two concern candidates and argue why they would be appropriate choices. For both concerns I summarize their potential for reuse on the example of recurring solution patterns. I briefly discuss the benefits of a CSML enabled concern integration and discuss the principal compatibility to main stages on the path of a concern integration with *FIDDLR*. Finally I summarize to which extent existing technologies can be reused.

4.1 Resource Access Delegation

Deployed services often involve some form of access protection. Especially in a Micro-Service context, where applications consist of fine grained RESTful services, resource-oriented access control is inevitable. Since a single service may contribute to various applications, access protocols often define sets of privileges that are associated with scoped resource access. The OAuth2 protocol is the de-facto standard for inter-service access delegation. It assumes resource ownership by *Resource Owner*s who can grant access to foreign services, based on predefined privileges. Access is realized using revocable tokens, which sidesteps any need for credential sharing. A common analogy is the idea of a *Valet car key*, that grants access to a resource (a car), but places access conditions, e.g. a speed limit, or restricted access like blocked *trunk*, *fuel tank* [11]. Note that in OAuth2 the association between privileges and resources is static, whereas association of foreign services and privileges happens at runtime. Defining the desired privilege model is not straightforward with existing GPMLs concepts. Privileges may inherently form a hierarchy and are associated to selected fractions of a given resource access structure. In its simplest form a CSML designed for specifying resource access would allow the definition of descriptive permissions, which can subsequently be mapped on an API. A preliminary visualization of such a CSML is depicted in Figure 6.

With the required privileges and scopes defined, it would be possible to customize concern provided realization models, to effectively transform CSML and mapping information into GPML models. Alike *RESTify*, a direct definition of these models by a concern user would not be per-se impossible, but would result in severe accidental complexity due to the semantic gap between GPML expressiveness and desired concepts.
Technologies like Spring Boot allow the encoding of OAuth2 configurations, the subsequent generation of project code based on transformed input models is a viable procedure option. So far we have explored the desired target code of manually configured OAuth2-enabled services. This is a first step to allow for a manual inspection of the custom code which we ultimately hope to produce through an integrated concern.

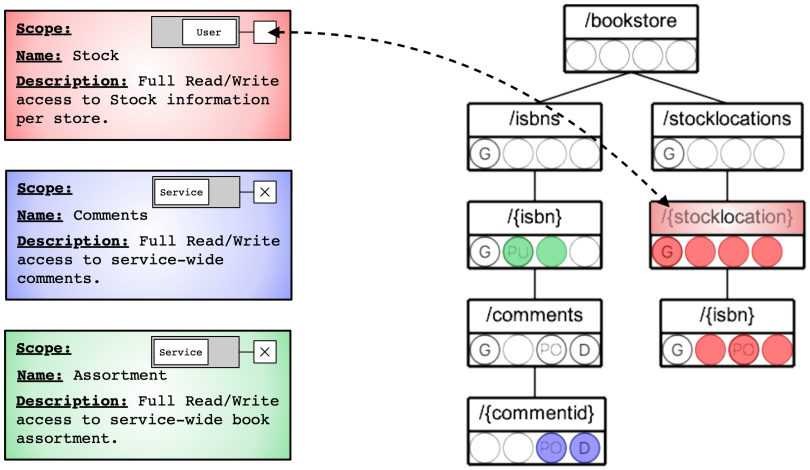

Figure 6: Preliminary concept for an OAuth2 privilege definition and mapping on REST resources. The dashed mapping line indicates an account specific privilege whereas absence of a mapping indicates a service wide permission scope.

4.2 Service Load Balancing

In the context of distributed systems, a recurring interest of modular application design is the option to replicate bottleneck components, which allows a distribution of load over service replicas for improved system performance. This strategy is referred to as *Load Balancing*. However there are different load balancing variants. Service replication could be concealed behind a proxy service, routing the traffic. This form is called *Server-Sided Load Balancing*. Alternatively the list of available service instances can be communicated to the clients, which then in turn directly contact a specific service entity. The choice may then occur based on different criteria, e.g. opportunistic selection of the service with *fastest response time* or altruistic distribution of load through a *round robin* communication. Depending on the service nature, there can be further constraints. For instance if the target service is stateful load balancing requires either service synchronization strategies, such as delegation to a shared database or static association between clients and service instances.

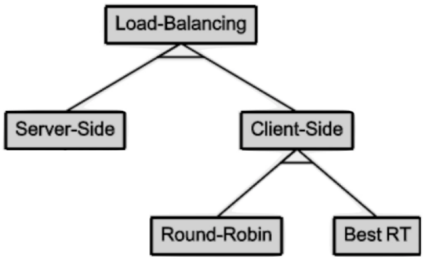

Figure 7: Preliminary variation interface for a Load Balancing concern

These variants could be captured with a concern variation interface, as shown in Figure 7. However with flexible amount of desired replicas and available resources, it is inevitable for the concern user to create a deployment model to specify the desired resource allocation.

Once more this association is not easily described with GPMLs. A promising language for expressing architectures, components and deployment is *Palladio*'s [10] *Palladio Component Model* (PCM). PCM models not only support allocation scenarios, they can in principle also be used as input for performance predictions, as Palladio offers simulations based on queuing models. The option for preliminary performance estimations throughout concern reuse can be seen as a beneficial feature for a future Load-Balancing concern. We explored the integration of PCM concepts into TouchCORE and validated the general feasibility for such deployment models within *FIDDLR*'s default development suite. However, since the integration of PCM with support for performance predictions introduces a complex technological stack, we also considered the option of developing our own, minimal CSML that reduces PCM to a minimal subset of required concepts.

Both of the considered concerns are found in the context of micro-service architectures, which use RESTful services as components. Partially this is motivated by my personal background - the design of meaningful concerns requires sophisticated domain knowledge, therefore my practical experience with RESTful services and Spring Boot is beneficial. However, it is also interesting to select a second concern that inherently reuses *RESTify*. This provides an implicit validation of *RESTify* and, thanks to detailed knowledge on implementation details, also eases reuse of existing functionality. Since the CSMLs still showcase a highly concern-specific nature, I do not see an issue in the concerns' shared application domain.
With the initial explorations achieved I have a slight preference towards selecting OAuth2 as second case study, the main reason being a technical restriction of TouchCORE. So far TouchCORE projects are considered strictly monolithic applications. The introduction of deployment configurations breaks with this tradition and most likely also requires the development of an additional code generator for orchestration-software configurations, e.g. Kubernetes [6]. While OAuth2 is seemingly the more complex concern, it can therefore be assumed easier to integrate. Especially if we set on code-generation toward Spring Boot, a significant part of the *RESTify*'s toolchain can be considered reusable. More considerations on the expected challenges and anticipated solution strategies are listed in the next section.

5 Planned Research

In this section I recapitulate the state of all designated thesis components. Where applicable I summarize the missing evidence and provide a strategy for completion. The section concludes with a summary of upcoming tasks, arranged as a preliminary schedule that provides an estimated time to thesis completion.

5.1 *FIDDLR*

FIDDLR is the key component of my contribution and has already reached a coherent and mostly stable layout. The general design has been peer reviewed and validated by means of a detailed case study that demonstrated the feasibility of a real-world concern integration [15]. While the successful concern integration and application provides convincing evidence for the viability of *FIDDLR*, the conducted case study can by nature not quantify the advantages

compared to an unassisted concern integration. For this we would have to re-integrate the same concern without the help of *FIDDLR*, and compare the required efforts. This empiric comparison would unfortunately exceed the available resources. However, we already ran a detailed theoretical analysis of integration tasks for general CSML-enabled concern integration and argued how *FIDDLR*'s frame is beneficial throughout the entire process [15].

We still expect minor adjustments on *FIDDLR*'s exact layout. The *RESTify* case study already provided feedback that could lead to further refinement. An example is an apparent need for advanced weaving. Notably the integration of a concept for build-configuration weaving seems to constitute an extension that increases genericity of our approach.[3]

5.2 RESTify

The *RESTify* concern is far enough integrated and usable to provide proof of concept. Although not free of minor bugs it allows convenient refactoring of sample legacy application to RESTful services. However, it is not yet advanced enough to cover all envisioned claims. Specific return parameters are incorrectly handled by the implemented CSML transformers and the variation interface currently only supports *Spring Boot*. Over the next months we want to gradually apply it to more complex input models, until all situations expected in a user study are reliably handled by the integrated concern. In parallel we want to finalize the support for at least one JAX-RS variant. The latter will most likely not require a lot of effort since huge parts of the implemented transformers were coded with genericness in mind and provide well modularized functionality that can be conveniently reused for further transformer implementations.

5.2.1 RESTify User Study

From a concern integration point of view we have fulfilled most prerequisites that make it possible to conduct a user study. In the intended layout, the participants are divided into two groups that each convert a provided sample application into a RESTful service, based on a provided target interface description. Half of the participants will attempt this conversion using TouchCORE, *FIDDLR*'s reference implementation, which includes the integrated *RESTify* concern. The control group aims for the same outcome, but must achieve the conversion with traditional code refactoring techniques. Participants are provided with supportive material that provides clear instructions and offsets a possible participant skills level. For further fairness, main and control group are swapped for a second experiment, where the task is the conversion of a second sample application of equal complexity.
At the time of writing, the study layout and intended recruitment have already been approved by the university's *Research Ethics Board*, leaving us sufficient time to conduct the study. A remaining challenge is the compilation of supportive material and task formulation. It is for instance important to provide the target REST interface in a form that provides equal conditions for either workflow. Also, since participants might not know how to perform

[3] In the current reference implementation, the build configuration generator has domain knowledge about specific REST technologies. This constitutes an undesired technological coupling, which could be avoided by concern provided build configuration fragments that are dynamically woven throughout concern reuse.

each workflow, we must provide educative material to offset missing skills. A key aspect of the study is a reliable measurement of required time for task completion and quality of outcome. We received approval to capture participant activity with screen-recordings. Software quality will partially be measured through unit tests. Since the target is pre-defined, we can run systematic `curl`-scripts to determine the functional correctness ratio. Generic tools for this kind of unit testing have already been developed and tested throughout the past months [14]. We hope to observe a significant speed-up of required time for task completion, and a higher success rate of unit tests. This would support our claim that *RESTify* simplifies service conversion.

5.2.2 Tool Paper

We are also planning to submit a paper on *RESTify* (without a *FIDDLR* context) to a distributed system conference. Specifically we want to provide a more technical, in depth description of *RESTify*, and forward our claims regarding model-driven REST-service design.[4] In this article we also want to better highlight the reuse taking place within the integrated concern, with respect to reused technologies. A contribution uniquely on *RESTify* would also allow the selection of a more complex base application than in [15].

5.3 Concern II

Before we can integrate a further concern to obtain a second case study, we first need to identify a good candidate. Concern design requires sophisticated domain knowledge and familiarity with standard solution patterns. Therefore the path to a second case study begins with a thorough analysis of concern candidates for CSML eligibility and feasible recurring solutions.
Currently we investigate two promising concern candidates, preliminarily named *AUTHify* and *Load Balance*. Both concerns seem to bear the inherent justification of a CSML. *AUTHify* envisions the API securing of a RESTful service with the OAuth2 protocol, whereas *Load Balancing* targets increased performance through service replication. So far our investigations suggest the existence of recurring solution patterns, and non-alignment on standard GPML concepts for both concerns. In the following I elaborate further on the individual opportunities and challenges associated with each concern.

5.3.1 AUTHify

The OAuth2 protocol is used to delegate access on resources, between services. That is to say the API of a *Service A* is mapped on fine grained privileges that filter access. Foreign services can then request a specific set of privileges to fulfil a useful task. The required segregation of *Service A*'s resource tree to privileges is a concept that can be well visualized with a tailored CSML. Privileges can also be scoped to individual user accounts, represented by dynamic resources.
The CSML concepts, proposed in Figure 6 are an exciting draft, that could be used to

[4] A first submission of *FIDDLR* to the MODELS21 converence was rejected, partially because the reviewers deemed the *RESTify* component provided too many technical details.

build a first authorization-specific language. Also, AUTHify seems to bear room for variants. Granted access is for instance realized using various forms of cryptographic tokens. Depending on the token format used, the protocol's characteristics change. If e.g. the *JSON Web Token* (JWT) format is used, tokens themselves contain the information required for a cryptographic verification of access, which renders granted privileges irrevocable. Alternatively tokens can be random strings without any information encoded - this would then support dynamic association of privileges over a database, and therefore optional privilege revocation.
The next step toward a case study based on OAuth2 is the implementation of a minimal target code sample: A simple access protected service that exemplifies a possible outcome of successful service reuse. This sample code would allow us to validate the semantic expressiveness of our preliminary CSML, formulate the required concern realization models and also identify required CSML transformations. The previsioned time estimated for this case study is listed in 5.4.

5.3.2 Load Balancing

An alternative second concern serves the assisted integration of load balancing solutions for deployable services. Distribution models that specify the allocation of existing hardware for service replicas can be considered a legit candidate for an integrated CSML, and PCM seems to offer the required concepts. Yet it is not clear how these concepts can be transformed into a GPML equivalent. Also, modern service deployment is often subject to container-system configurations, such as Docker-compose. It would therefore make sense to explore the generation of deployment configurations. TouchCOREs existing code-generator is not able to produce these configurations. Again a first step would be the manual formulation of desired outcomes, validating if the reusable concepts can be accurately expressed with a CSML. A promising candidate for the underlying deployment models seems to be PCM. However the integration of PCM into TouchCORE has already been attempted and can be considered technically challenging. It is unclear how much time the integration of PCM would comsume. Similarly the possibility for Palladio based performance prediction imposes a complex technological stack. Likewise, tests with simple test cases have shown Palladio a poorly documented and hard to use tool - it is unclear how much time an effective integration would require. Therefore it might be faster to disregard the existing tool and write a new, simple CSML for deployment models from scratch. Similar to AUTHify, the subsequent steps would be the extraction of required CSML transformations and concern realization models, based on a minimal sample concern outcome. Unlike AUTHify, Palladio's nature makes it harder to reliably estimate the time required (see 5.4).

The above arguments speak in favour of my initially stated preference toward AUTHify as second case study. Whatever the candidate chosen, the second case study will be reduced to concern integration, without an additional user study. While verification of the practical benefit for concern users is desirable, the main motivation for a second concern integration remains to demonstrate the usefulness of *FIDDLR* for the integration task.

5.4 Preliminary Schedule

Based on the envisioned research presented in the section, a provision of the remaining components within a duration of one year can be assumed feasible. Below schedule lists a detailed breakdown of the main required tasks and their individual assumed duration:

Time Estimation			
Component	Current State	Actions Required	Monotask Time
FIDDLR	Peer reviewed	(See Below)	—
Pom Weaving	Concept Discussed	Demo Implementation	1 Month
RESTify	Proof of Concept	(See Below)	—
Support JAX-RS	Target Code Implemented	Implement Transformers	2 Months
RESTify Peer-Review Distri.Sys.	Planned	Elicit Conference, Write Tooling paper	2 Months
User Study	REB Approved	Fix minor bugs in *RESTify* transformers, Test Material, Run Study	3 Months
Concern 2 (Option A)	Exploration	(See Below)	—
OAuth2 Sample Application	Planned	Implementation	1 Month
OAuth2 Token Variant Exploration	Planned	Reading the Docs	1 Month
OAuth2 CSML Metamodel	Planned	EMF implementation	1 Month
OAuth2 Realization Model Extraction	Planned	TouchCORE integration	1 Month
OAuth2 Transformer Implementation	Planned	TouchCORE integration	1 Month
Concern 2 (Option B)	Exploration	(See Below)	—
PCM Integration / Reduction to CSML	Planned	Integration into TouchCORE	*Unclear*
PCM to GPML transformations	Planned	Implementation	*Unclear*
Kubernetes configuration generator	Planned	Implementation	1 Month
PCM to GPML transformations	Planned	Implementation	*Unclear*
Palladio perf. prediction integration	Planned	Find a way to use Palladio in UI-less mode	*Unclear*

6 Conclusions

The results obtained during these first three years are encouraging and allow me to identify the main components of the envisioned contribution. Still I acknowledge the existence of multiple challenges along the way toward a successful program conclusion, some of which being of scientific nature, some bureaucratic and finally some personal, too.
The greatest scientific challenge I currently see is making a meaningful choice for a second CSML /concern case study. Whatever the outcome, the selection should be taken carefully, as the feasibility of a subsequent implementation rises and falls with a fitting choice. Yet, setting grounds for this choice itself requires a severe time investment, as I need to become as much of a domain-expert as required to foreshadow a potential concern integration and CSML design tasks. This investment is unfortunately required for every concern candidate, regardless of whether it will eventually become a thesis component or not. Also challenging is the goal to attend maximized coverage of thesis components through peer-reviewed conferences. I see the validation of *FIDDLR* as a key component, residing at the heart of my envisioned contribution. Now the next step is to reach peer-validation for *RESTify*, ideally taking into account the perspective of a distributed system's audience.
On behalf of bureaucratic challenges, I see the greatest threat in a straightforward conduction of the *RESTify* user study. As much as I personally endorse high ethical standards for study conduction, participant selection and confidentiality of collected data, the so-far path toward a study design approved by the universities *Research Ethics Board* was a time consuming and tedious process that greatly exceeded my considerations. Due to the many external factors and dependencies I see this component as the most risky remaining part of my dissertation, which is why I strive for its timely advancement and conclusion.
Finally, without surprise the ongoing COVID-19 pandemic introduced a massive change in all of our lifestyles. A change that set new challenges for daily life, none of which I had anticipated in my pre-departure considerations for a thesis program outside of Europe. The last months have reminded me of the importance of a sustainable work-life balance. Still with the progress made so far, with the massive feedback I receive from my supervisors, and with the extremely supportive atmosphere among co-workers and friends I envision the remaining thesis time as an exciting phase that I am greatly looking forward to.

References

[1] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. 2013. Concern-Oriented Software Design. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013 (Lecture Notes in Computer Science, Vol. 8107)*. Springer, Berlin, Heidelberg, 604–621.

[2] Roy T Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine.

[3] Roy T Fielding, Richard N Taylor, Justin R Erenkrantz, Michael M Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. 2017. Reflections on the REST architectural style and principled design of the modern web architecture (impact paper award). In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 4–14.

[4] M. Jamshidi. 2008. *System of systems engineering? New challenges for the 21st century*. Wiley, Hoboken, NJ. 616 pages.

[5] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. 2016. VCU: the three dimensions of reuse. In *International Conference on Software Reuse*. Springer, Berlin, Heidelberg, 122–137.

[6] Kubernetes. 2021. *Kubernetes Documentation*. `https://kubernetes.io/docs/home/`

[7] SCORE Labs. 2021. TouchCORE User Guide. http://touchcore.cs.mcgill.ca/. Accessed: 2021-09-24.

[8] Mark Masse. 2021. Web Resource Modeling Language Definition. `https://github.com/wrml/wrml`. Accessed: 2021-04-28.

[9] MuleSoft. 2021. RESTful API Modeling Language Definition. `https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/`. Accessed: 2021-04-28.

[10] Ralf Reussner, Steffen Becker, Jens Happe, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, Max Kramer, and Robert Heinrich. 2016. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press.

[11] Justin Richer and Antonio Sanso. 2017. *OAuth2 in Action*. Manning.

[12] Maximilian Schiedermeier. 2020. A concern-oriented software engineering methodology for micro-service architectures. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 1–5.

[13] Maximilian Schiedermeier. 2021. Book-Store Internals Sources. `https://github.com/kartoffelquadrat/BookStoreInternals/`.

[14] Maximilian Schiedermeier. 2021. Unit Test Tools for REST APIs. `https://github.com/kartoffelquadrat/LobbyService/blob/master/unit-tests/rest-tools.sh`.

[15] Maximilian Schiedermeier, Jörg Kienzle, and Bettina Kemme. 2021. Internation Conference on Software Language Engineering. In *FIDDLR: Streamlining Reuse with Concern-Specific Modelling Languages*. ACM, New York, NY, USA, 81–88.

[16] Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons, Vedran Pavić, Jay Bryant, Madhura Bhave, Eddú Meléndez, and Scott Frederick. 2021. *Spring Boot reference documentation*. `https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/`