FIDDLR: Streamlining Reuse with Concern-Specific Modelling Languages

Maximilian Schiedermeier
McGill University, Canada
max.schiedermeier@mcgill.ca

Jörg Kienzle
McGill University, Canada
joerg.kienzle@mcgill.ca

Bettina Kemme
McGill University, Canada
kemme@cs.mcgill.ca

Abstract

Model-Driven Engineering (MDE) reduces complexity, improves Separation of Concerns and promotes reuse by structuring software development as a process of model production and refinement. Domain-Specific Modelling Languages and Aspect-Oriented Modelling techniques can reduce complexity and improve modularization of crosscutting concerns in situations where the features of general purpose modelling languages are not well aligned with the subject of study.

In this article we present *FIDDLR*, a novel framework that integrates the ideas of Domain-Specific Modelling Languages, Concern-Oriented Reuse and MDE to modularize concerns that cross-cut multiple levels of abstraction of the software development process and streamline the reuse process. It also prescribes the integration of the different tooling along this process. We demonstrate the effectiveness of our framework and the potential for reduced complexity and leveraged reuse by building a reusable concern that exposes the services a system offers through a REST interface.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Software design engineering**; **Reusability**; *Source code generation.*

Keywords: Model-Driven Engineering, Concern-Oriented Reuse, Concern-Specific Languages

ACM Reference Format:
Maximilian Schiedermeier, Jörg Kienzle, and Bettina Kemme. 2021. FIDDLR: Streamlining Reuse with Concern-Specific Modelling Languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486608.3486913

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '21, October 17–18, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9111-5/21/10…$15.00
https://doi.org/10.1145/3486608.3486913

1 Introduction

Modern software needs to cope with the ever increasing complexity of systems [14], and hence *reducing complexity* is a primary objective of software engineering. In *Model-Driven Engineering* (MDE), the combined use of multiple modelling languages allows the developer to express properties of the system under development at different levels of abstraction and from different points of view, thus promoting *Separation of Concerns* (SoC) and reducing complexity. Model transformations connect models across levels of abstraction, effectively *reusing architectural and design knowledge*, or *platform-specific development expertise* when generating code.

Still, model complexity can be a challenge. For one, a conceptual mismatch of context and language introduces *accidental complexity*. Furthermore, targeted reuse of partial models is not easily possible in conventional modelling languages. *Domain Specific Modelling Languages* (DSMLs) address the former challenge, whereas the latter can be dealt with by compositional approaches, e.g., *Aspect-Oriented Modelling* (AOM) techniques or *Concern-Oriented Reuse* (CORE).

In this article we present *FIDDLR*, a *F*ramework for the *I*ntegration of *D*omain-Specific Mo*D*elling *L*anguages with concern-oriented *R*euse. The contributions of *FIDDLR* are:

- *FIDDLR* provides clear steps on how to *integrate and reuse existing MDE, AOM/CORE and DSML tooling* when creating reusable software artefacts – called *concerns* – that address a specific development issue.
- With *FIDDLR*, a concern can define its own *Concern-Specific Language*, a DSML designed to express the concepts of the concern *as well as streamline its reuse*.

In Section 2 we present MDE, DSMLs and CORE, and then discuss the motivation for combining the building blocks in Section 3. Section 4 presents the main features of *FIDDLR*. In Section 5 we evaluate *FIDDLR* by means of a case study. In Section 6 we compare existing work to our approach and examine commonalities and differences. Section 7 summarizes the advantages of our proposal and how current limitations to our approach could be addressed in the future.

2 Building Blocks

SoC has been identified early on as one of the main mechanisms for tackling complexity during software development [4]. SoC refers to the ability to temporarily focus one's attention solely on one development concern or issue.

Reuse is simply the process of creating software systems from existing software artifacts rather than creating them from scratch [21]. In this paper we are mostly interested in *planned reuse* as opposed to opportunistic reuse.

2.1 MDE, Modelling Languages and Processes

Model-Driven Engineering (MDE) [17, 28] is a unified conceptual framework in which the whole software life cycle is seen as a process of *model production*, *refinement* and *integration*. Models are built representing different views of a software system using different formalisms, i.e. modelling languages. The language is chosen in such a way that the model concisely expresses the properties of the system that are important at the current level of abstraction.

A typical MDE process uses one or several *General Purpose Modelling Languages* (GPML). The left side of Fig. 1 depicts typical software development phases found in object-oriented, model-driven development methods. At a given level of abstraction, consistency constraints ensure that the different models form coherent views of the system. In Fig. 1 these constraints are depicted with black double ended arrows. Ultimately, code generation is used to generate a significant part of the object-oriented implementation from the design models. Guidelines for refinement, and model transformations that implement partial refinement and code generation are depicted with thick grey arrows in Fig. 1.

SoC is at the heart of MDE. Every model that is created is an abstraction of the system under development – unnecessary details are omitted. When establishing a model, the most appropriate modelling language is used, focusing the attention of the modeller on the current properties of interest. Each model describes the system under development from a different point of view, and can therefore focus on a different development concern.

2.2 Domain-Specific Modelling

Accidental complexity arises out of mismatch of modelling language and modelled matter. GPMLs such as UML mostly cover the typical structural and behavioural modelling needs for *software development*. Because of their general purpose nature there is a (sometimes significant) semantic gap between a specific application domain and the concepts offerend by GPMLs. This gap can be bridged with DSMLs [12].

DSMLs target a specific application domain. They are typically developed in house, i.e., within an organization, by experts of the domain, and usually cover a range of abstraction levels. Model transformations or code generators are then used to derive other models or code from the domain-specific models. This is illustrated in the center of Fig. 1. The model transformations that generate partial GPML models at different levels of abstraction are depicted with thick white arrows. Other approaches, reviewed in Section 6, do not generate code for DSMLs, but provide run-time DSML integration, i.e., the domain-specific models are executed / interpreted alongside the other development models.

Reuse is central to DSMLs and MDE in general, the main unit of reuse being the modelling language. A modeller using a modelling language is reusing knowledge of the language engineer when building models by instantiating language concepts. Modelling languages typically come with a tool that ensures consistency between views of the system at the same level of abstraction. Reusable model transformations, partially or completely automate the refinement of models when moving between levels of abstraction, thus *reusing architectural and design knowledge* or *platform-specific development expertise*.

2.3 Aspect-Oriented and Concern-Oriented Reuse

In Aspect-Oriented Modelling (AOM), a modelling language is augmented with advanced language features that enable the modularization and composition of model fragments. Model fragments are models that are not necessarily viable in isolation. A model weaver is a special model transformation that takes as an input two models and a composition specification, and produces a new *composed* output model in which the two input models have been merged (see right illustration in Fig. 1).

Concern-Oriented Reuse (CORE) [1] is an approach based on AOM that streamlines model reuse by encapsulating model fragments inside a reusable unit called a *concern*. A *concern designer* must provide three interfaces for a concern [18]: The *variation interface* (VI) exposes the different variants of the reusable entity with a feature model, and the impact of each variant on high-level system qualities with an impact model. With the *customization interface* (CI) the concern designer exposes the generic entities in the concern that have to be adapted to a specific reuse context, while the usage interface (UI) defines how the functionality encapsulated by a concern may be used. 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).

Behind the scenes, based on the information provided by the concern user (i.e., the selected features, the customization mappings and the usage dependencies), a weaver combines the model/code fragments of the reused concern corresponding to the selected features with the application models/code.

3 Critical Assessment

3.1 Separation of Concerns

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 used GPMLs. Some development concerns, e.g., *Security*, need to be considered not only during

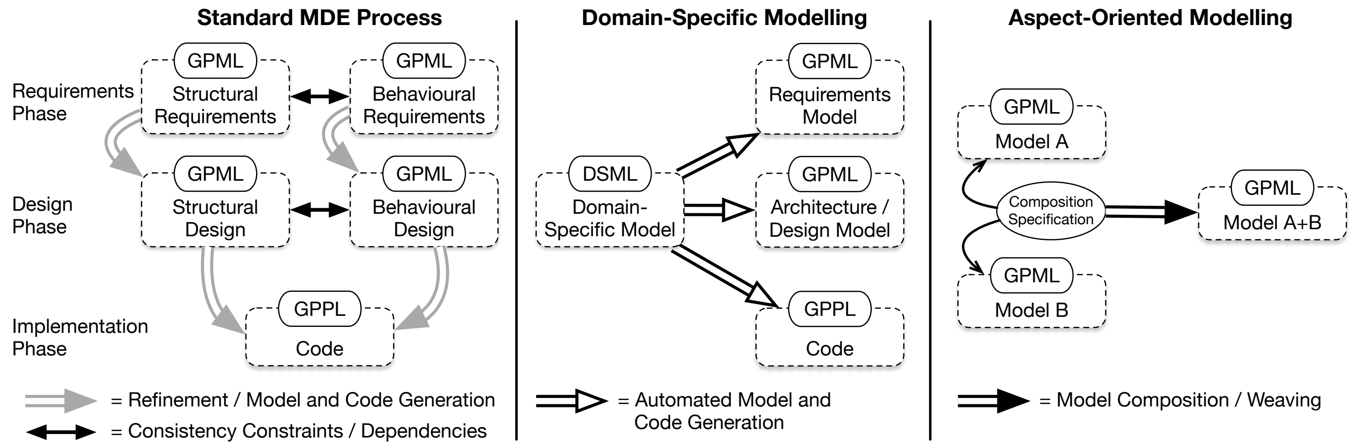

Figure 1. MDE, DSML and AOM

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*.

DSMLs have the potential for addressing this drawback, because they can define language features that allow a modeller to express properties relating to any level of abstraction of software development for the targeted domain.

Furthermore, MDE also fails to properly address SoC within a model. Consider a design class diagram of a system under development that contains functionality for user *Authentication*. The structural properties needed to deal with *Authentication* (e.g., the users, their credentials, sessions) are tangled with the rest of the application design structure. Hence, in the overall design class diagram, there is no clear separation between classes related to *Authentication* and classes related to other business functionality of the application.

AOM and CORE address this drawback, as they augment modelling languages with advanced constructs that make it possible to decompose a model into fragments, and later on re-compose them with a model weaver. Provided the AOM/CORE decomposition mechanisms align with the boundaries of a concern, proper SoC is possible.

3.2 Reuse

As previously discussed, the modelling language is the main unit of reuse in MDE and DSMLs. It allows for reuse of domain knowledge, as well as model transformations and other tool support provided by the language engineers.

Within a language, however, reuse in MDE and DSMLs is relatively limited. Because most modelling languages lack language features that enable proper modularization and packaging, it is not easy to encapsulate a set of model elements that, e.g., represent a recurring modelling pattern as a reusable model. This is exactly what CORE is good at, because it extends the modularization technology offered by AOM with language constructs for grouping model fragments behind well-defined reuse interfaces, and provides means for customizing generic model elements with application-specific structure and behaviour.

3.3 Accidental Complexity

CORE integrates well with MDE. The relevant properties of a concern can be expressed at the appropriate level of abstraction using the most appropriate GPML. Customization and usage also work well for concerns whose nature is aligned with the concepts of the GPMLs used in the MDE process. For example, the *Observer* design pattern, when modularized with CORE, can be reused easily by mapping the `Subject` and `Observer` classes and their operations to the corresponding application design classes and operations.

However, for concerns that do not align well with GPML concepts, the standard way of customization and usage offered by CORE introduces significant accidental complexity. For example, imagine a situation where the design of an application is modelled using class-, state- and sequence diagrams. Imagine now a *Workflow* concern that can be used to define and execute workflows that are constituted of interdependent and potentially concurrent activities. Neither state nor sequence diagrams are well suited to model workflows. While those models can be used to design a workflow execution engine, the customization of this workflow engine design would be very difficult for a concern user, who would have to understand the internal design details of the engine.

This is where DSMLs can help. Accidental complexity can be avoided by defining a *Concern-Specific Modelling Language* (CSML) targeted at exposing the concepts of a concern, just like a DSML would expose the concepts of a domain.

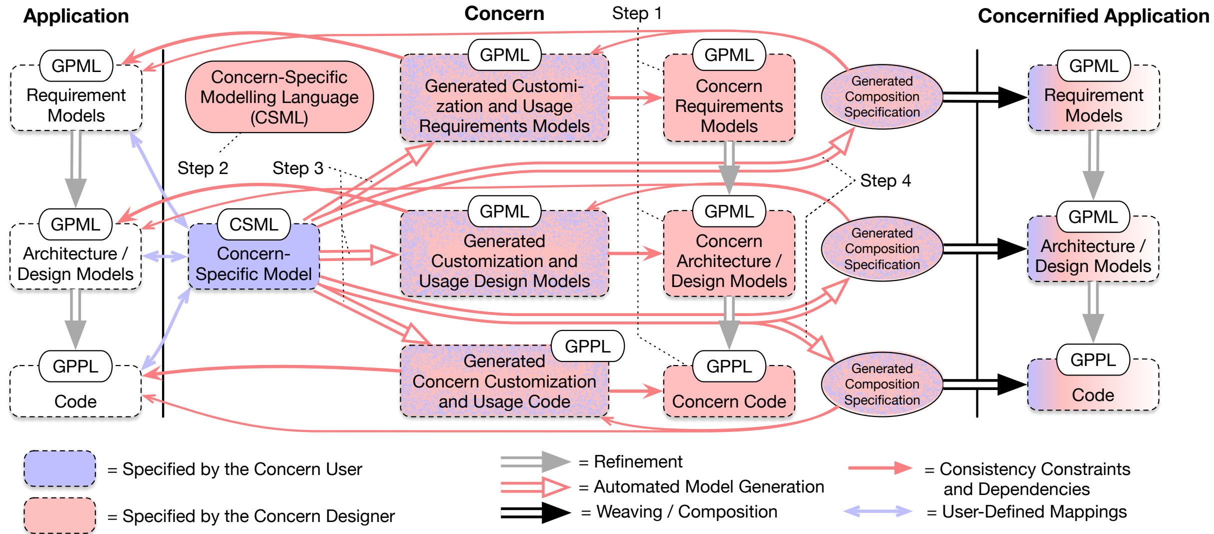

Figure 2. The *FIDDLR* Framework

Additionally, a CSML is also designed to streamline the reuse of a concern, i.e., facilitate its customization and use.

4 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. 2.

FIDDLR puts forward the idea that DSML technology can be exploited effectively for implementing and applying concerns that do not align well on 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.

4.1 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. 2 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. 2) 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.

4.2 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 [18] 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. 2. 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.

4.3 Concern Composition

To combine the application and concern models, *FIDDLR* reuses existing MDE, DSML and CORE tooling as much as possible.

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. 2. 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.

5 Evaluation / Case Study

To evaluate the potential of *FIDDLR* we conduct an in-depth case study, motivated by 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?

We pursued these questions by implementing and reusing a sample concern, related to the *Representational State Transfer* (REST) architectural style. REST allows the invocation of remote services through a resource-oriented interface. The re-exposure of existing functionality through REST commonly requires thorough domain expertise. In this section we explain how *FIDDLR* supports the design of *RESTify*, a concern that can be used to re-expose any application functionality through a REST-adherent (RESTful) interface.

In a first subsection, we provide an overview of REST, describe the common stages required when manually adding a REST interface to an application, and discuss the associated challenges. Subsequently we present how *FIDDLR* supports the design of *RESTify*, a concern to streamline the REST refactoring process. In a third subsection, we exemplify how a concern user applies the concern to an application using *RESTify*'s key stages. Finally, the last subsection presents a qualitative and quantitative comparison of the *RESTify* concern with manually adding a REST interface to an application.

5.1 REST and Service Refactoring

RESTful service interfaces consist of hierarchically structured resources with selected CRUD operations [5] (Create, Read, Update, Delete) enabled. Those operations are commonly invoked over HTTP as *Put, Get, Post* and *Delete* requests. Clients interact with a service that is adherent to the REST style (i.e., a *RESTful service*) uniquely over those selectively enabled CRUD operations. Having a RESTful service therefore constitutes a layer of abstraction, as it strictly conceals service implementation details[1]. Over the last decade, RESTful service interfaces gained widespread acceptance for modern web architectures and component-based systems, notably in a Micro-Service context [31]. This is mainly due to the efficient abstraction from implementation details, but also due to the versatility of HTTP, which provides free choice of implementation language for communicating software components. Yet the design of a proper REST interface for a given functionality remains a challenging task. For one, because correct interface engineering is subject to a variety of design rules. Secondly, because the underlying web technology that enables the execution of a RESTful service imposes a complex technological stack. A side effect of this complexity is that real-world services often showcase misuse or even anti-patterns to the REST style [6].

In the following we show how a simple Java desktop application of a *Bookstore* can be refactored to a RESTful service,

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

illustrating the common challenges when re-exposing service functionality through REST. The Bookstore database holds sample book metadata, reviewer comments and inventory of individual stores. In its original state, the Bookstore offers a set of public methods that allow for local querying and manipulation of the Bookstore data. Adding a REST interface is beneficial because it allows clients to consult or modify the database remotely.

Fig. 3 provides an overview of the main steps that we expect an experienced developer to perform to manually add a REST interface to an existing application. It starts with a technical choice. Enabling REST for the Bookstore requires the integration of a framework or library that implements the runtime communication infrastructure and protocols required for REST. We assume the developer prefers *Spring Boot* over various implementations of the *Jakarta RESTful Webservices* (JAX-RS) [8, 10, 11, 15] specification, which constitute other viable alternatives.[2] Spring is a JDK-external artefact and therefore can only be invoked if referenced by the system's classpath. Thus, our developer modifies the Bookstore's existing build system configuration, declaring a dependency to Spring Boot, and exchanging the default build settings by a Spring-specific plugin. Next, the developer replaces the original Java launcher class with one that initializes the Spring framework during startup. These preliminary steps are summarized in the *Prepare* stage of Fig. 3.

As a next step, the application API needs to be re-exposed in REST format. Using Spring, Java methods can be mapped on CRUD operations of REST resources using Spring-specific Java annotations. An annotation parameter then specifies the resource location. An example for the syntax used is shown in Listing 1. Spring annotations are highlighted in green.

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

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

Similarly, individual parameters can be annotated where needed, to resolve method arguments to details of the mapped resource query. This can be either a dynamic fragment of the resource path, a HTTP query parameter or the parsed HTTP body. Regarding the Bookstore, our developer therefore identifies the existing Java methods that must be exposed and decorates their signatures with the required Spring annotations. This step is represented by the *Expose* stage in Fig. 3. The modified code only becomes of practical use for remote clients, if built and deployed on a server. Building is uncomplicated, as the configured build system compiles the modified Bookstore into a self-contained JAR file that can be executed as-is on any system with a compatible Java Runtime Environment. Self-contained means that the JAR includes Spring and its transitive dependencies. If executed, the launcher class invokes Spring, which in turn powers up an embedded web server. Using reflection, Spring detects the added annotations and ensures that inbound HTTP queries are delegated to the decorated methods and that parameters are correctly resolved. The Bookstore has hereby effectively become a RESTful service. This final step is illustrated by the last stage, *Deploy* in Fig. 3.

Table 1. LOC Modified during Bookstore Conversion

File(s)	SLOC Added	SLOC Modified	SLOC Removed
Legacy Launcher	-	-	31 / 31
New REST-Service Launcher	10	-	-
Annotations in other Classes	15	10 / 296	0 / 296
Build System Configuration	25	2 / 109	27 / 109

Table 1 shows the code modifications performed by our developer. Only a relatively low fraction of the code has been touched. Yet these modifications require significant expertise. For example, during the *Prepare* stage of Fig. 3, the developer first has to make a choice of which REST framework to use, and then update several files, i.e., the build files and launcher classes. This is not straightforward and requires detailed framework knowledge (curved arrow in Fig. 3).

In the *Expose* stage, the placed annotations *implicitly* encode an entire REST interface, that represents the *design of an entire resource tree* with selectively enabled CRUD operations and parameter mappings. The developer has to build this tree by placing annotations that encode individual branches of the tree in URLs. While in case of the Bookstore the REST interface was expressed with only 28 annotations, those are scattered over the code base. This conceptual mismatch – building a tree by writing URLs sprinkled over several source files – imposes a high mental load on the developer. Furthermore, the developer must have a thorough knowledge of the Spring annotation syntax. Hence the *Expose* stage is not at all straightforward, and is therefore illustrated as a twirly arrow in Fig. 3. Only the build and deployment of the refactored code base are straightforward and illustrated as a straight arrow in the *Deploy* stage.

On top of an implicit and scattered REST interface design process, the described procedure is prone to errors. As illustrated in Table 2, Spring's annotations establish resource mappings through arguments of type *String*, representing the resources' absolute locations. As a result, path elements closer to the root are replicated as Strings in annotation parameters of lower level resources mappings. Additionally,

[2] Spring was assumed due to it's high industrial relevance, however any alternative could have been used for the purpose of this case study.

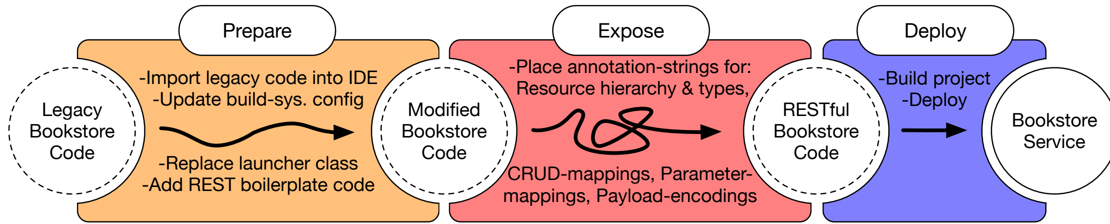

Figure 3. Manual Steps Required for Adding a REST Interface to Existing Code

Table 2. Annotations Added to Bookstore

Annotation	Amount
Parameter-Mapping	17
Resource CRUD Mapping	11
Boilerplate	4

Table 3. Resources String Replications across Annotations

Resource	String replications
isbn (isbns subresource)	13
bookstore	12
isbns	8
stocklocation	6
comments	5
commentid	4
stocklocations	4
isbn (stocklocation subresource)	4

parameter mappings may on top refer to resource path elements and therefore further increase String replication.

Table 3 shows the string replication counters for the manually converted Bookstore. Since these mappings are string-encoded they are exempt from verification at compile time. As a result, typographic mismatches will not be detected unless the service is deployed and tested.

To summarize, even the conversion of simple applications is subject to a tedious introduction of boilerplate code and requires sophisticated knowledge of the applied technologies. The refactoring process involves implicit design choices, scattered over the code-base. We argue that existing GPMLs cannot accurately capture the essence of above 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. In the next subsection, we demonstrate how the above challenges can be addressed with a concern built according to the *FIDDLR* approach.

5.2 Designing the *RESTify* Concern

The purpose of the *RESTify* concern is to maximally streamline the process of adding a REST interface to expose application functionality. When applied, the concern must guide the user through essential design choices, hide implementation details and automate any repetitive development tasks.

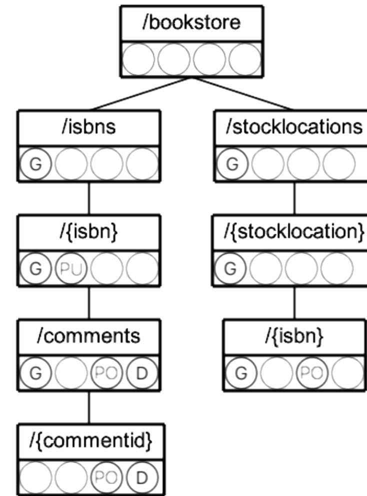

Figure 4. Bookstore Resource Layout Designed with the *ResTL* Editor. Circled Letters Below a Resource Represent Enabled CRUD (**G**et, **Pu**t, **Po**st, **D**elete) Operations.

Designing and implementing the *RESTify* concern itself, however, is not straightforward. This is where the guidance of *FIDDLR* helps. Fig. 5 illustrates the *FIDDLR* based design process of the *RESTify* concern. 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. So far, the implemented concern only supports Spring, but other REST frameworks could be integrated in the future.

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 model using *ResTL* to define a resource layout for the Bookstore.

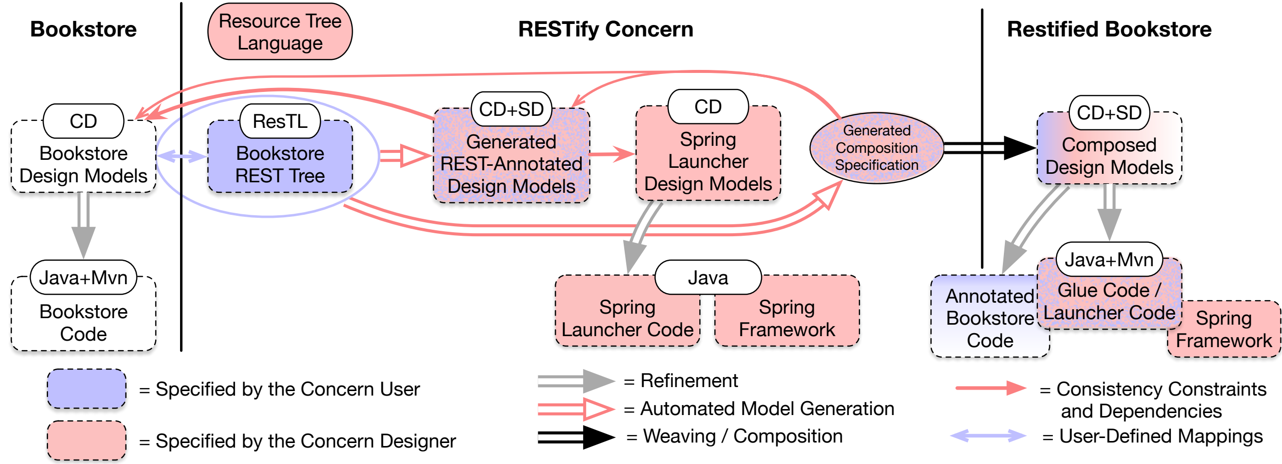

Figure 5. *FIDDLR* applied to the *RESTify* Concern

Note that the concern designer does not need to define a language for establishing the mappings between the *ResTL* model and the rest of the Bookstore application. CORE already provides a generic artefact for 1:1 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. The Bookstore *ResTL* model and the mapping (illustrated as a blue arrow in the figure) are the only artefacts the concern user needs to provide, once a REST technology is selected. We provide an example of how to create the *ResTL* model and mappings in section 5.3.

One goal of *FIDDLR* is a maximized reuse of existing MDE and CORE concepts. 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) that invoke the REST launcher code required by Spring. Fig. 5 shows these models as *Spring Launcher Design Models*.

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. This transformation is depicted in Fig. 5 as a red horizontal double-lined arrow.

Furthermore, 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, produces a *composition specification* (step 4). This composition specification, when given to the CORE weaver, composes the GPMLs of the base application with the generated GPML models containing the REST-specific information. This second transformation is also shown as a red double-lined arrow in Fig. 5.

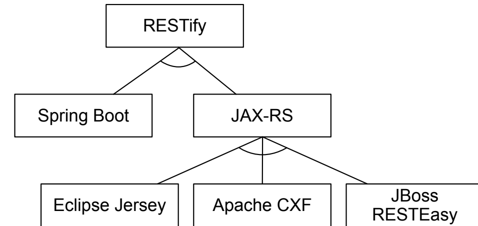

Figure 6. Variation Interface of the *RESTify* Concern

No further work is necessary. Notably, 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. In our case, this tooling is provided by the CORE reference implementation, TouchCORE [22].

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 mappings and the weaver provided by CORE, the code generator provided by MDE, and the Spring framework itself.

5.3 Applying the *RESTify* Concern

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 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. 6. 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. 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 a 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.

This illustrates one of the key differences between a standard DSML and a CSML. While a REST-DSL would have to specify detailed parameter information, the *ResTL* language does not. It integrates perfectly into the concern reuse workflow. The parameter names and types are specified at the right level of abstraction in the application models, in our case in the design models of the Bookstore.

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 CORE implementations allow an automatic signature extraction from existing artefacts, e.g. JAR files. Depending on target signatures the user may also have to define additional mappings for method parameters. Note that these mappings are generic and CORE-provided. That is to say no extra CSML is required to instantiate these mappings.

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. These represent dynamic path fragments (denoted as a placeholder enclosed by curly brackets). Afterwards, remaining un-mapped parameters are assumed to be either HTTP query parameters or encoded as body payloads. Fig. 7 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 model transformations, model weaving and code generation. The latter also produces a build system configuration that ensures automatic integration of Spring at compile time.

5.4 Qualitative and Quantitative Comparison

With *RESTify*, adding a REST interface to an application is done in three steps – selecting a technology, designing the resource layout, and establishing the mappings to the application. Each step is as simple as possible, performed at the right level of abstraction supported by the right modelling notations. No expert REST knowledge is required by the concern user. Likewise there is no need to deal with any technical details, e.g., framework-specific boilerplate code, annotation syntax or intricate configuration file modifications. In the manual approach, the developer spent a significant fraction of the overall efforts on preliminary tasks and implicit decision making, whereas with *RESTify* the process is guided and straightforward with explicit decision making and minimal overhead as illustrated in Fig. 8.

Another advantage of *RESTify* is the elimination of unnecessary redundancy. Where the manual Bookstore conversion showcased severe replication of resource strings, scattered over annotations in multiple files (see Table 2), the *RESTify* models define every resource name exactly once, hence eliminating a source of potential errors.

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. But even more complex evolution scenarios are considerably simplified. For example, switching from the Spring-Boot based implementation to a JAX-RS based implementation such as Jersey would be as simple as selecting a different feature from the variation interface of the *RESTify* concern. A manual migration from one technology to the other would constitute a considerable effort, because here the annotation syntax differs between REST frameworks. Listing 2 shows the JAX-RS semantical equivalent of Listing 1 (Spring-Boot syntax).

Listing 2. JAX-RS Annotated Bookstore Method. Accessible by HTTP GET Request at e.g. `"/bookstore/stocklocations/minastirith."`

```
@Path("stocklocations")
public class GlobalStockImpl {
  [...]
  @GET
  @Path("{stocklocation}")
  @Produces("application/json")
```

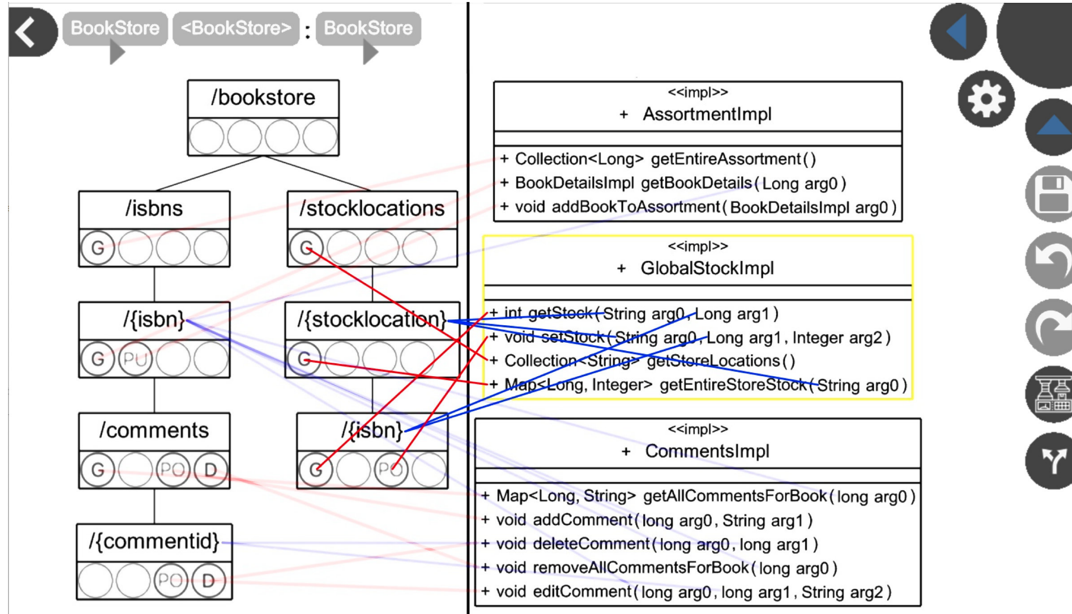

Figure 7. TouchCORE Screenshot showing Split View in Action. Mappings can be highlighted selectively to improve visibility.

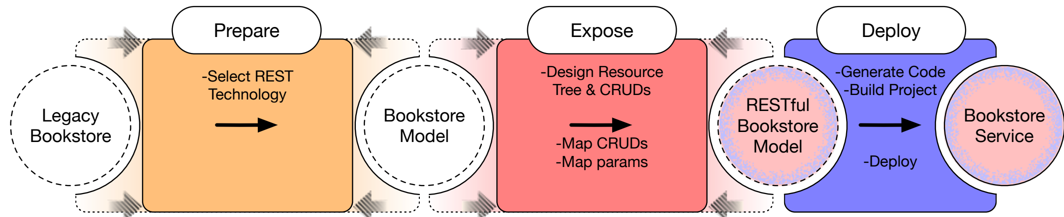

Figure 8. Application of the *RESTify* Concern. Preliminary Tasks are Reduced to a Minimum. Decision-Making is Explicit.

Table 4. Atomic Actions to Convert Bookstore with *RESTify*

Element	Occurrences in Bookstore	Actions
Technology select	1	1
Resources	8	24
CRUD Operations	12	12
CRUD Mappings	12	12
Parameter Mappings	13	13

```
public Response getEntireStoreStock(
    ↪ @PathParam("stocklocation") String city){
  return Response
    .status(Response.Status.OK)
    .entity(stocksPerCity.get(city).
        ↪ getEntireStock())
    .build(); } }
```

By nature it is hard to define a metric for comparing a modeling approach to a code-based approach. To provide a quantitative comparison with the manual approach, we applied an action based metric to compare the efforts required.

The atomic modeling actions that need to be performed in *RESTify* are as follows: The desired technology has to be *selected* with a single click. Creation of the tree model requires three interactions for each modeled resource: one *create* instruction, one interaction to *specify* the resource type, and a third interaction to *enter* its name. *Exposure* of a CRUD operation is achieved with a single click. Finally, every resource or parameter *mapping* requires an additional action. For the Bookstore we end up with a total of 62 modelling actions as shown in Table 4. This represents a reduced effort when compared with the over 100 lines of source code that have to be written, modified or removed in the manual approach as shown in Table 1. We believe the comparison and conclusion drawn is fair. There are more textual code changes required than graphical interactions, and textual modifications constitute a greater effort than atomic clicks.

5.5 Lessons Learned

In regard to the research questions initially stated in 5, the *RESTify* case study validated the feasibility of concern implementation with integrated CSML, following the *FIDDLR*

framework. Among the required steps, the model transformation to convert CSML-application mappings to GPML-application mappings required the most effort. The reusable toolchain provided by CORE (model weaver and code generator) worked as expected.

Furthermore, the case study validated the purposefulness of the *RESTify* concern. While direct modelling of annotations would have been feasible, it would have been highly inconvenient for the concern user. In contrast, the integration of *ResTL* allowed for a focused development of a meaningful resource layout, abstracting away obfuscating implementation details.

The case study did not provide insight on the question whether a concern implementation based on *FIDDLR* actually accelerated concern development. While we strongly believe that the clear task order and reusable toolchain has a positive impact on concern development, we did not have the resources to also develop an alternative implementation of *RESTify* without *FIDDLR* in parallel and compare the required efforts.

6 Related Work and Limitations

This section discusses relevant related work in two subsections. In the first one we present related software engineering and model-driven engineering frameworks and other domain-specific modelling approaches that reduce complexity. The second subsection then elaborates on related modelling approaches for REST interfaces. In a third sub-section we finally point out limitations of *FIDDLR*.

6.1 Modelling Frameworks

FIDDLR is a framework for integrating *MDE*, *DSMLs*, *Separation of Concerns* and *Reuse* both conceptually and technically. *FIDDLR* is built on CORE [1], where the main unit of modularization and reuse is the *concern*. To integrate concerns with each other, standard and aspect-oriented composition techniques are used.

As such, *FIDDLR* is a realization of the *multidimensional separation of concerns* approach proposed by Tarr et al. in [29] applied to models and code. *FIDDLR* also aligns with the *ModelSoC* approach proposed by Johannes et al. in [16]. Compared to ModelSoC, *FIDDLR* also supports software product lines by allowing concerns internally to be modularized according to features, and it allows concerns to be packaged with a CSML. Furthermore, *FIDDLR* does not rely on a central concern management system as does ModelSoC.

One of the main goals of *FIDDLR* is to allow existing tooling infrastructure to be reused by the *concern designer* (as illustrated in Fig. 2 and demonstrated by the RESTify case study). In particular, the CORE tooling is reused to combine the generated GPML models with the application models. Other weavers could be used as well, e.g., the generic composer GeKo [20], or potentially even other composition-based approaches, e.g., facet-oriented modelling [23]. Standard MDE tooling is reused as well, e.g., the Acceleo code generator [9] or the Epsilon family of languages [19] for writing the model transformation from the CSML to the GPML models. Furthermore, since CORE has no special requirements on the languages that are being composed, any DSML tooling and SLE workbenches can be reused to define a CSML for a concern, provided that the generated language is metamodel-based. For example, Degeule et al. provide a set of reusable engine related tools, i.e., a language called *Melange*, for reusing languages, their metamodels [3] and associated transformations. We believe that their techniques can be used by a concern designer to facilitate the creation of CSMLs and the transformations towards GPMLs.

Other modelling frameworks and tools have been proposed in the past that make it easy to reuse DSMLs and integrate them with each other or with GPMLs.

[30] is a language workbench that facilitates the creation and reuse of DSMLs. It also defines an approach for integrating different DSMLs with each other, e.g., by defining relationships and constraints between concepts from different languages.

In [13], *Hardebolle et al.* describe *ModHel'X*, a component-oriented approach for the design and integration of modelling languages to build a multi-formalism modelling environment. The approach focuses on the simulation of behavioural models of a system. SoC is achieved by choosing modelling languages that align with the concern boundaries. Although they do not explicitly focus on reuse, their component-oriented approach suggests language modules as potential unit of reuse.

In a similar spirit, *Bousse et al.* describe *Gemoc*, a workbench intended to ease the development of DSMLs and the required tooling infrastructure (execution/simulation, debugging, trace management) [2].

Our approach differs from *ModHel'X* and *Gemoc*, as they focus on coordination of models, while *FIDDLR* is based on composition and generation. Furthermore, we not only offer SoC and reuse through DSMLs, but also enable SoC and reuse within and across models expressed in the same language.

6.2 Modelling REST

There exist already several DSMLs that allow the specification of REST interfaces. The *Web Application Description Language* (WADL)[3] was designed to describe HTTP resource behaviour in a machine readable manner. It has been shown that WADL can be used to express the contractual interfaces of REST service implementations [7]. However, since WADL does not require the use of the base concepts of the REST

[3]WADL is not to be confused with the *Web Service Description Language* (WSDL), designed for *Simple Object Access Protocol* and *Remote Procedure Call* specifications.

style, it does by itself not assist or guarantee REST compliant interface design [27].

The *Web Resource Modeling Language* (WRML) [25] proposed by *Masse* in [24] is a REST-specific modelling approach based around resources. In contrast to WADL it therefore partially enforces the REST style. To the best of our knowledge, WRML is also the only other REST DSML proposal that considers a graphical editor that implicitly organizes resources into a tree structure. In contrast to WRML, the *ResTL* language purposely does not allow the description of a complete REST interface specification.

The *RESTful API Modeling Language* (RAML) is a textual modeling language that adheres closely to the REST principles [26]. Interface specifications provided in RAML can be converted into OAS/Swagger specifications, and from there into a variety of server and client sided stub implementations. A fundamental difference to the *ResTL* provided by our *RESTify* concern is that the inherent tree structure of a REST interface is not prominent in RAML specifications. Furthermore, with OAS/Swagger-based code generation it is not possible to generate fully working services.

In summary, there are already several DSMLs that allow the specification of REST interfaces. However, these languages target an accurate description of finalized interfaces, followed potentially by the generation of code skeleton implementations. The CSML we defined for *RESTify* was designed to facilitate the process of adding a REST interface to an already existing application structure and logic.

6.3 Limitations

FIDDLR streamlines reuse for the concern user by reducing the work required to the essential steps (see Fig. 8):

1. Choose a concern variant from the concern's VI
2. Model the concern-specific properties of the application using the CSML
3. Specify mappings that connect the concern-specific properties with the application-specific models

As a result, the concern user is shielded from solution-specific design choices and technical intricacies. Furthermore, the complex transformation pipeline that *FIDDLR* is based on – CSML to GPML model generation, weaving, and code generation (see Fig. 2) – is also hidden from the concern user.

This fundamental operating mode of *FIDDLR* is inherently linked to a general limitation – low traceability between what a concern user models and the generated code. This hinders debugging seriously and can make it very difficult for the concern user to apply corrective actions at the CSML level in situations where the generated outcome does not produce the expected behaviour. While traceability and debugging is a well-known issue with generative approaches as well as with compositional approaches, the problem is even more pronounced in *FIDDLR* because it uses generation *and* composition technologies.

A further limitation of our implementation of *FIDDLR* is that it was built on top of the CORE tooling infrastructure and therefore inherits all its technical limitations, i.e., CSMLs have to be EMF-based, and code generation targets Java and Maven only.

7 Contribution and Future Work

In this article we presented *FIDDLR*, a framework to streamline reuse and promote separation of concerns that integrates MDE, DSMLs and CORE. *FIDDLR* augments the unit of reuse of CORE, the *concern*, with the possibility of including a modelling language that is specifically designed to express the concern's properties and integration in the most appropriate way. In other words, the concern designer can now define a *Concern-Specific Modelling Language* to maximally focus the concern user on the relevant concepts of the concern and facilitate the concern's customization and usage. Just like DSMLs, doing this can significantly reduce accidental complexity as well as integration complexity, in particular for concerns that are not easily expressed with a GPML or that crosscut several abstraction levels or phases of software development.

To validate our proposed framework, we developed an example concern called *RESTify* that streamlines a state-of-the-art development activity: exposing application functionality as RESTful services. We discussed how thanks to *FIDDLR*, the concern designer implementing *RESTify* can reuse the existing MDE, DSML and AOM technology at various levels in the development process. We showed how *RESTify* greatly facilitates the task of exposing application functionality as RESTful services for the concern user, compared to a manual refactoring activity. We concluded that this provides convincing evidence that *FIDDLR* bares great potential and merits further investigation.

In a near future, we plan to run an empirical user study to determine the practical benefits of *RESTify*: we are planning to ask inexperienced developers to expose the functionality of some sample applications with REST, either manually, or using the *RESTify* concern. By comparison with a control group we hope to demonstrate a significant speedup in development time and gain in software quality.

Furthermore, we are planning to evaluate *FIDDLR* further by building another reusable concern, this time one that clearly crosscuts multiple views and levels of abstraction of the system under development. Specifically we are going to modularize, encapsulate and package in a concern everything needed for an application to define and execute *workflows*. We envision that a DSML such as, for example *Use-Case Maps*, is needed to make it easy for the concern user to define workflows and link the activities of the workflow with the functional models of the system.

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] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, New York, NY, USA, 84–89.

[3] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-Language for Modular and Reusable Development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2814251.2814252

[4] Edsger Wybe Dijkstra. 1976. *A discipline of programming*. Vol. 1. Prentice-Hall, Hoboken, NJ.

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

[6] 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.

[7] Marios Fokaefs and Eleni Stroulia. 2015. Using WADL specifications to develop and maintain REST client applications. In *2015 IEEE International Conference on Web Services*. IEEE, Piscataway, NJ, USA, 81–88.

[8] Apache Software Foundation. 2021. Apache CXF Documentation. http://cxf.apache.org/docs/jax-rs.html.

[9] Eclipse Foundation. 2019. Acceleo. https://www.eclipse.org/acceleo/.

[10] Eclipse Foundation. 2021. Eclipse Jersey User Guide. https://eclipse-ee4j.github.io/jersey/.

[11] Eclipse Foundation. 2021. Jakarta RESTful WebServices Online 3.0 Specification. https://jakarta.ee/specifications/restful-ws/3.0/jakarta-restful-ws-spec-3.0.html.

[12] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle. 2007. Domain-Specific Modeling. In *Handbook of Dynamic System Modeling*. CRC Press, Boca Raton.

[13] Cécile Hardebolle and Frédéric Boulanger. 2007. Modhel'x: A component-oriented approach to multi-formalism modeling. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, Berlin, Heidelberg, 247–258.

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

[15] Red Hat / JBoss. 2021. JBoss RESTEasy JAX-RS Community DocBook and Javadoc Documentation. https://resteasy.github.io/docs/.

[16] Jendrik Johannes and Uwe Aßmann. 2010. Concern-based (de) composition of model-driven software development processes. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, Berlin, Heidelberg, 47–62.

[17] Stuart Kent. 2002. Model Driven Engineering. In *International Conference on Integrated Formal Methods – IFM*. Springer-Verlag, London, UK, 286–298.

[18] 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.

[19] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2008. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations*, Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio (Eds.). Springer, Berlin, Heidelberg, 46–60.

[20] Max E. Kramer, Jacques Klein, Jim R. H. Steel, Brice Morin, Jörg Kienzle, Olivier Barais, and Jean-Marc Jézéquel. 2013. Achieving Practical Genericity in Model Weaving through Extensibility. In *Proceedings of the 6th International Conference on Model Transformation - ICMT 2013 (Lecture Notes in Computer Science)*, Keith Duddy and Gerti Kappel (Eds.), Vol. 7909. Springer, Berlin, Heidelberg, 108–124. https://doi.org/10.1007/978-3-642-38883-5_12

[21] Krueger. 1992. Software Reuse. *CSURV: Computing Surveys* 24 (1992), 131–183.

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

[23] Juan De Lara, Esther Guerra, and Jörg Kienzle. 2021. Facet-Oriented Modelling. *ACM Transactions on Software Engingeering and Methodology* 30, 3, Article 27 (Feb. 2021), 59 pages. https://doi.org/10.1145/3428076

[24] Mark Masse. 2011. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. "O'Reilly Media, Inc.", Sebastopol, CA.

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

[26] 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.

[27] Leonard Richardson and Sam Ruby. 2008. *RESTful web services*. "O'Reilly Media, Inc.", Sebastopol, CA.

[28] Douglas C. Schmidt. 2006. Model-Driven Engineering. *IEEE Computer* 39 (2006), 41–47.

[29] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns.. In *ICSE'1999*. IEEE CS, Piscataway, NJ, USA, 107 – 119.

[30] Markus Voelter. 2011. Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, Berlin, Heidelberg, 383–430.

[31] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Piscataway, NJ, USA, 223–236.