Extending Concern-Oriented Reuse to Existing Modelling Languages

Yanis Hattab

A thesis submitted to McGill University in partial
fulfilment of the requirements of the degree of

Master of Science

School of Computer Science
McGill University
Montréal, Québec, Canada

April 2020

Abstract

Modern software systems constitute remarkably large and complex entities made up of an intricate web of components and libraries that render their development exclusively with source code ill-advised. Model-driven engineering promises to alleviate such issues by making models fundamental to all development phases. These models are to be specified according to appropriate and relevant modelling languages with the right level of abstraction while emphasizing a strict separation of concerns. MDE also emphasizes reusing existing standardized models and modelled design patterns to simplify the design process and increase productivity. Unfortunately, reuse is far from common in actual modelling practice, too often development teams will prefer using completely new models, at the cost of both time and effort, either because they have specific notation needs or are unaware of the potential benefits of reuse. Existing modelling frameworks and tools do little to facilitate this task, lacking intuitive and efficient reuse mechanisms and providing arcane interfaces to reuse and tune languages from the modelling community. Hence we propose, in the following thesis, our contribution to improve support for modelling reuse and language tailoring by extending the Concern Oriented Reuse (CORE) modelling framework to support multiple external languages and augmenting them with language independent reuse capabilities. Furthermore, with our proposed concept of perspectives, we would allow a language designer to tailor languages for a specific purpose. These perspectives also pave the road for concern-oriented multi-view modelling, as they can be designed to orchestrate the combined use of multiple languages to frame a design process.

By redesigning and improving the existing reuse oriented CORE modelling framework through the addition of the language concept, we allow it to support any external abstract syntax, i.e. language, defined with a metamodel. Furthermore, CORE languages have a novel manner of describing their semantics through language actions, enabling our proposed concept of perspectives to

tailor and combine languages for domain-specific usages. To validate the expressiveness and usability of this unified language design approach, we used it to define UML Class Diagrams and tailor them for specific usages through perspectives. These CORE additions were incorporated in the implementation by redesigning the associated TouchCORE intuitive modelling tool to be language agnostic.

Abrégé

Les systèmes informatiques modernes constituent des entités de taille et de complexité remarquables, composés d'un enchevêtrement de composants et de librairies qui rend malavisée leur conception à travers la programmation exclusivement. L'ingénierie dirigée par les modèles propose de corriger ces problèmes en faisant des modèles le fondement de toutes les phases de développement. De tels modèles seraient spécifiés avec les langages de modélisation les plus pertinents et appropriés ainsi qu'avec le bon niveau d'abstraction tout en mettant l'emphase sur la séparation des préoccupations. L'IDM met aussi en avant la réutilisation de modèles existants standardisés et de design pattern modélisés pour simplifier le processus de design et augmenter la productivité. Malheureusement, la réutilisation au niveau des modèles est loin d'être une pratique commune en pratique, les équipes de développement préférant trop souvent créer de nouveaux modèles, au prix de précieux temps et efforts, car il ont des besoins particuliers ou ne connaissent pas les bénéfices. Les outils et les cadres de modélisation existants ne font pas non plus grand chose pour faciliter cette tâche car ils n'ont pas de mécanismes de réutilisation intuitifs et efficaces se contentant d'interfaces ésotériques pour réutiliser ou adapter les langages existants de la communauté. C'est pourquoi on propose, dans la thèse qui suit, notre contribution pour améliorer le soutien à la réutilisation et l'adaptation de langages en étendant la plateforme de modélisation *Concern Oriented Reuse* (CORE) afin qu'elle supporte de multiples langages externes et leur ajoute ses puissantes capacités de réutilisation. De plus, avec le concept de *perspective* qu'on propose, on permet au concepteur de langages d'adapter des langages existants pour une utilisation précise au lieu de partir de zéro. Ces perspectives tracent aussi le chemin pour la modélisation à vues multiples orientée sur les préoccupations logicielles, étant donné qu'elles peuvent être utilisées pour combiner plusieurs langages ensemble afin de construire un processus de design cohérent.

En repensant et améliorant la plateforme CORE existante, qui est orientée pour la réutilisa-

tion, avec l'ajout du concept de langage à part entière, on lui permet de supporter n'importe quelle syntaxe abstraite (langage) définie par un méta-modèle. Par ailleurs, les langages CORE ont une nouvelle façon de décrire leur sémantique à travers des *actions de langage*, réalisant ainsi le concept de perspectives qu'on propose pour adapter et combiner des langages à des utilisations spécifiques. Afin de valider l'expressivité et l'utilisabilité de notre approche unifiée pour la conception de langage, on l'a utilisée pour définir les diagrammes de classes de UML et les adapter pour une utilisation spécifique avec des perspectives. Ces ajouts à CORE ont été implémentés dans l'outil de modélisation intuitif TouchCORE afin de réaliser notre approche et le rendre indépendant des langages de modélisation qu'il supporte.

Acknowledgements

To Jörg, whose near-infinite wisdom, knowledge, wholesomeness and enduring patience form the main reason I was able to carry this project to fruition.

To my dear parents, sister and brother, who stood by my, supported and helped me throughout all of my studies and projects.

To Mathieu, Márton and Adrien, my graduate school comrades for their invaluable help and support.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problematic

Software systems nowadays are immensely complex and tightly interconnected while rapidly growing to form the foundation of many critical services to society. The development process of such systems at their current scale needs to be, now more than ever, very rigorous and thorough while being able to meet the market needs and schedules.

Traditionally this development relies mainly on programming languages, which are now more powerful and versatile than ever, offering robust established libraries for nearly any need. Unfortunately, their extensive sophistication also leads to extraordinary platform complexity with common middleware and libraries overflowing with countless classes and methods linked in complex dependency hierarchies.

Learning how to use such technology stacks for a new hire on a project requires a lengthy ramp-up that still won't guarantee them the skills to fine-tune them properly and understand potential obscure subtle side effects. Add to that the challenge of maintaining or migrating existing software as it rapidly grows and evolves that often leads to considerable development effort diverted from building new systems. Developers relying on programming languages alone to tackle modern software challenges often leaves them struggling to see the bigger picture and reason correctly about large software systems. By focusing solely on code, only fragmented views of the system can be acquired which can lead to implementing suboptimal solutions [28]. This antiquated approach often leads to duplicated code, violations of key architectural principles and unforeseen interactions that significantly increase the complexity of testing and maintaining the system as it grows and evolves [28].

These issues grow worse as the systems mature, getting compounded by the departure of knowledgeable developers and the arrival of ones less familiar with all the inherent intricacies. A 2015 survey of the ramp-up journey of new developers hired to work on 8 majors projects at Microsoft

found that the most commonly cited challenge is the lack of proper documentation [26]. New software developers felt that the lack of detailed documentation, often scattered in various locations and different formats, if not out-dated, strongly increased the time to first commit in the majority of the projects, and moderately did so in the others [26].

The software industry as a whole is struggling with this complexity ceiling that compels developers to dedicate years to master APIs and usage patterns, often specializing in a narrow subset of the platforms [28]. Such specialized skills the developers acquire, alongside varying backgrounds and seniority, further contribute to their fragmented perceptions of the system they are developing together. They further complicate identifying portions impacted by new code and potential side effects while impacting the new hires' ramp up, the survey showing moderate or strong increased time to first commit in most Microsoft projects [26].

1.2 Motivation

A growing number of experts believe that the most promising approach to address the towering complexity of modern software development lies in the widespread adoption of tool-assisted model-driven approaches. By bringing the focus of development toward models early on rather than code, we can express concepts independently of the underlying implementation details and have a higher-level view of the system being built.

Academic surveys empirically showed that the use of models by engineers properly trained to do so provides significant improvements in design quality and correctness of changes made to existing complex systems [5]. The benefits of model-driven development aren't reflected specifically in one system but emerge holistically in the form of well-articulated designs and architectures with extensive reuse of high-quality components. More specifically, applying a well designed, tailored and intuitive model-driven approach embedded in the development standards could lead to a significantly more reusable code base with robust coherent documentation.

Unfortunately, reuse while being common at the programming level remains rarely used at the modelling level, and even less so in modelling language design. Developers often start from scratch when they are modelling in established modelling languages or when designing their own purpose-specific language, at the cost of extensive efforts and time. The reason lies in the near absence of reusable models openly available, besides token examples, and in the inability of existing modelling tools to allow creating or importing reusable models.

Existing modelling languages were not designed with model reuse as a priority and they offer no support for it missing, among other things, standardized model modularization guidelines or defined reuse interfaces. Reuse, as it is intended for model-driven development techniques, should also be possible no matter the modelling language used and as such the mechanisms for it should lie

outside the language itself. Modelling approaches that incorporate multiple languages, each well-tailored to their specific purpose, are thus the most able to leverage the benefits of model reuse when they are supported with intuitive and powerful modelling tools. There is thus a need for a reuse oriented modelling framework that allows the combination and customization of modelling languages to enrich them with reuse functionalities and form a coherent and efficient development process. The effectiveness of such a framework and the languages it brings together is a measure of the following success metrics that we aim to maximize [17]:

- Reliability
- Usability
- Contribution to Productivity
- Learnability
- Expressiveness
- Reusability

The work presented here aims to contribute toward the edification of a model-driven development ecosystem by developing a framework and its proper tools that enrich any modelling language with powerful reuse functionalities. We accomplish this by improving on our existing Concern Oriented Reuse (CORE) framework to support any modelling language and offer an intuitive way to customize or combine them to foster reuse and coherence. Being able to bring newly defined or existing languages into the CORE framework allows us to leverage its powerful model reuse and customization features regardless of the language used. Furthermore, we introduce the concept of a *Perspective* that offers a way to tailor an existing modelling language without redefining it completely, adapting it to development needs on the fly. All these changes are implemented in our TouchCORE modelling tool that is further extended with a language and two perspectives to showcase the power of this modelling framework.

1.3 Contributions

The detailed contributions we bring forward are stated as follows:

1. **Definition of the *Language* concept within CORE**: A CORE language consists of a metamodel, describing its abstract syntax, as well as a set of *Language Actions* that specify how a *modeller* can create and manipulate models expressed in that language.

2. **Restructuring of the CORE metamodel**: In order to integrate models expressed in different languages, CORE's own inner metamodel had to be significantly restructured to be language agnostic. In particular, the new CORE Scene is the central unit realizing a concern feature instead of a model, the latter is now outside of CORE. The scene aggregates *artefacts* that act as proxies to model elements and allow their reuse by offering them access to CORE's reuse interfaces.

3. **Definition of the *Perspective* concept within CORE**: A CORE perspective allows a *modelling language developer* to tailor a language for a specific use. The perspective can then be used by a modeller instead of a language. The modelled CORE scene holds a reference to this perspective, and the perspective itself references the external language it is built on. As such perspectives are designed in a way that, in the future, they could even be able to combine multiple languages for a specific use.

4. **Validation of languages and perspectives**: The TouchCORE modelling tool was updated to support languages and perspectives creation and edition, adapting it to the new CORE metamodel. To validate the applicability of languages and perspectives, a new language that expresses *UML Class Diagrams* was designed and fully integrated into TouchCORE. To demonstrate the usefulness of perspectives, two perspectives for *Domain* and *Design* modelling were defined, they reuse the Class Diagram language but tailor the allowed modelling operations to expose only the relevant ones.

1.4 Thesis Outline

In the next chapters, we introduce and explain our proposed approach to streamline multiple modelling languages support, reuse and combination in a consolidated framework and the benefits it provides. We illustrate the expressivity gains of the approach by describing how it allows us to define a language and customize it with a perspective straightforwardly and efficiently.

We first state and define in chapter 2 the concepts that serve as the background and foundation of our modelling approach. The contributions 1, 2 and 3 concerning the updates to CORE are introduced and detailed in chapter 3. We define there the relevant concepts of language, language action and perspective while explaining their benefits for customizing and maintaining modelling processes consistent. Then, chapter 4 demonstrates how an existing modelling language can be defined with this updated CORE framework and then tailored for another purpose with a perspective. Chapter 5 discusses the changes done to our modelling tool implementation to support the new CORE concepts and describes the actual modelling process using our newly defined language and perspective through examples. Both these chapters show how we achieve the last contribution of

validating the usability and benefits of our language and perspective concepts. Finally, we discuss related projects from other research teams and how they compare to CORE in Chapter 6 before concluding and laying out future ambitions for our framework in chapter 7.

2
Background

This chapter introduces and defines the necessary preliminary concepts that serve as framing and foundation for building the modelling framework described in later chapters.

2.1 Model-Driven Engineering

Model-driven software engineering is an approach that promotes modelling to tackle the challenge of developing modern systems that are characterized by outstanding levels of complexity, size and scale. MDE combines prescribed architecture and process to structure and facilitate the task of building, maintaining and adding features to software infrastructures.

It addresses our problematic of growing development complexity by recognizing the need to abstract from the implementation details when reasoning about a large system, especially for developers joining an ongoing project. If they start contributing to a code base without first acquiring a complete understanding of it, they often end up triggering unwanted side effects on other sections of the code or being themselves a victim of their colleagues' side effects. The MDE process and its associated technologies aim precisely at tackling such development challenges by refocusing the developers' efforts within a holistic understanding of the system conveyed using models that abstract away unnecessary details.

Moreover, MDE prescribes modelling using multiple languages that allow the designers to capture more information about a system and present it more efficiently and meaningfully to each other and their stakeholders. It coherently combines *domain-specific modelling languages* alongside model transformation rules and generators, synthesizing various artefacts (e.g. source code) used in later steps of the design process. [28]. MDE also advocates and prescribes how creating domain-specific constraints and model validation rules allow detection and avoidance of errors in a software project earlier in its life cycle.

One of the main advantages of this approach is the higher level of abstractions that allows teams to visualize and discuss the system under construction without getting lost in its multiple

implementation details and intricacies. Beyond the improved system understanding, a whole new field of model analysis emerges with MDE, allowing to capture defects and anti-patterns automatically and to generate useful development metrics and documentation. This also fosters developing a more complete view of the system and foreseeing unexpected side effects upon improvements or feature addition [28]. Once a system is fully defined and planned, the traditional implementation process can take over, aided by code and artifacts directly generated from them models besides their great support as detailed documentation and specifications.

2.2 Metamodelling

A modelling language can be seen as the set of all models that could possibly be expressed in a way that syntactically respects this language or, in modelling terms, models that conform to the language metamodel [20]. The metamodel is a way to specify a language structurally in modelling notation whereas the language itself is defined as an instance of the metamodel [20]. An actual model is then built according to that modelling language's elements specified in the metamodel, with the purpose of abstracting a concrete software system. Thus, the metamodel serves as a blueprint from which one can get the elements and rules deciding how to create and edit an instance model. We refer to the process of designing modelling languages by defining their metamodel as metamodelling, that is, we are designing a model of the language itself.

An example to illustrate the relationship between a metamodel and its instance models is shown in figure 2.1, where a very small subset of the metamodel of UML Class Diagrams is linked to the model elements it allows to instantiate. The instance model where we see the class *Video Game* sits a the modelling layer where we would find for instance all Class Diagram models created, further down hierarchically and not shown would be instantiations of the classes as objects. On top of that layer, we have the metamodel layer, where class diagrams concepts like Class and Attribute can be found, since the metamodel is also a model they are defined as *metaclasses*, to disambiguate from the Class concept being modelled. A final layer would sit on top of the language metamodel, where a meta-metamodel is used to specify the modelling language used to model metamodels. This is the last layer as the meta-metamodel is reflexive and can be used to describe itself.

The metamodel can also be conceived as the type of its produced model, and this typing is not transitive which is why we can say that the metamodel is the model of a model [20]. It is categorized as a *linguistic instantiation* as it "relates a model with the definition of the language of which it is an expression" [20]. As opposed to an *ontological instantiation* that would relate two models that borrow from the same domain of modelling tokens but are on different logical levels.

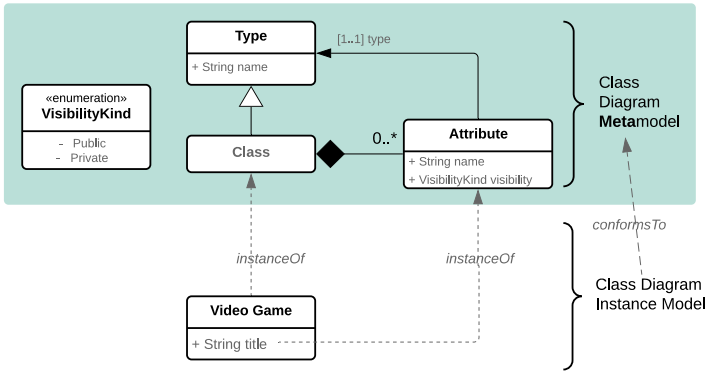

Figure 2.1 – Distinction Between the Metamodel of the Class Diagrams Language and the Instance Models it Defines

2.3 Domain-Specific Languages

Domain-specific languages (DSL) are computer or modelling languages specifically built and tailored to express a particular application domain [24]. The ability to define purpose-specific modelling languages for a domain is particularly useful as it allows us to raise the level of abstraction pertaining to a system under design from implementation details to domain concepts. DSLs offer substantial gains in expressiveness and ease of use in comparison to general-purpose languages [24] as modelling is best able to capture and analyze pertinent features in the domain vocabulary. Representing the components of a system with elements familiar to domain insiders also greatly improves the learning curve when introducing developers and broadens the range of experts that can contribute to the whole development process. Using *metamodels* to define, fine-tune and customize the DSL can further help precisely match the system domain's semantics and syntax when existing general-purpose languages would be inadequate [28].

Unfortunately, designing a DSL for a given domain in a rigorous and thorough way often proves to be difficult, time-consuming and costly, even more so when starting from scratch rather than adapting an existing language [14]. That comes as a consequence of the lack of proper tools and frameworks that would allow rapid DSL definition and prototyping as most of the research and development effort is currently dedicated to building modelling tools. Even once a fitting DSL is defined, it then may further require a specialized modelling tool allowing it to be properly used. The designers thus have to either adapt an existing tool, whose extensibility may be limited and arcane, or build one completely from scratch. In summary, the task of designing DSLs is suffering from a lack of reuse that could enable building languages on top of existing ones in a streamlined way or reusing elements from these existing languages.

Although these issues exist for traditional textual DSLs, they are worse for modelling languages, especially graphical ones, as they cannot be created easily by reusing, extending or translating them into well established general programming languages or textual DSLs [24].

As the scope of the research presented here focuses on modelling languages, we will not extend our discussion beyond these to other DSL types. Further on, we will only refer to the modelling variant, the domain-specific modelling languages, whenever we mention DSLs. Likewise, when talking about languages we refer to modelling languages.

2.4 CORE

Concern-Oriented Reuse (CORE) is a software modelling and development paradigm that aims at tackling the challenges raised by the cross-cutting nature of software development concerns by promoting high-level reuse [29]. Reuse is critical to modern software but it has yet to be leveraged effectively at a higher level of abstraction to allow the modelling of large scale and complex systems by bringing together smaller, reusable models hierarchically. CORE achieves this by separating and packaging software development artefacts, in particular models, within reusable units called concerns. Traditional software reuse is defined as the process of creating new software using existing software artifacts, and to be beneficial, reusing the artifact should be easier than building it from scratch. In order to bring similar levels of reuse to the modelling techniques used to develop software, CORE defines intuitive reuse interfaces, at a higher level of abstraction, that remain concise and formal.

Definition 1: *Concern*. Configurable and customizable unit of reuse encapsulating the modelling artifacts related to a singled out domain of interest. The models within a concern are organized according to features and allow the customization upon reuse to match the context of another concern.

CORE concepts are laid out in a metamodel whose instance models allow to build systems composed of a hierarchy of concerns thus intuitively organizing the realizing models. Thus CORE offers a key advantage from a model building point of view as it allows a modeller to explicitly declare a reuse interface for the models constructed so they can later be integrated into a higher-level model. This model interface achieves beneficial modularity with its three distinct parts [1]:

- **The Variation Interface**: Conveys all the available variations of the concern through a hierarchy of *features* and the impact of each variation on stakeholder goals and non-functional requirements.
- **The Customization Interface**: Informs on how generic elements of a chosen variation of

the concern can be mapped with existing elements of the model that is reusing the concern, allowing the reuse process to produce a customized composed model.

- **The Usage Interface**: Describes how the reusing model can access the defined structural and behavioural elements made available by the reused concern.

In its implementation prior to this paper, a concern in CORE (*COREConcern*) is specified structurally with *class diagrams* while it is specified behaviourally with *sequence diagrams* and *protocol state machines* [29]. These models are *COREModel* instances in the CORE metamodel composed of their respective modelling elements (*COREModelElement*), thus CORE and the defined languages were tightly linked and dependent on each other. Those modelling languages were previously static and specified using metamodels which were built specifically around the CORE architecture, adding languages first required a full understanding of CORE. Each concern is implemented through a *feature model* that structures and aggregates each feature offered by the concern. The individual features are then realized with one or more models from the defined languages through a *COREModel*. A feature represents a configurable characteristic of a concern that is relevant to the developer, they can be mandatory or optional and feature selection will happen when reusing the concern.

2.4.1 Concern Reuse Process

A Concern in CORE serves as the main unit of abstraction and modularization that allows constructing complex systems through the process of reusing other concerns. The reuse process itself is the workflow a developer follows when selecting a specific solution alongside many relevant variations offered by a concern that is of interest. The developer can, in a bottom-up approach, compose a system from intertwined and interacting concerns in a way that maximizes reuse. It is bottom-up because higher-level concerns end up reusing lower level simpler ones as primitives [29]. To achieve this, the developer will use the three interfaces of the concern in the following way:

1. They will choose the desired features they want from a *reused* concern through its Variation Interface, which lists all the valid alternatives offered by the concern.

2. They will then adapt how this solution is realized to their specific usage context (their *reusing* concern) using the *reused* concern's Customization Interface.

3. Then this selected and adapted version of the concern is now accessible with the correct mappings and through its declared Usage Interface from their *reusing* concern.

3

CORE Languages and Perspectives

3.1 CORE Languages

3.1.1 Previous CORE Metamodel

One of the main shortcomings we aimed to improve in CORE is the level of entanglement between modelling languages and CORE as a framework. There was a tight dependency between CORE and its languages and vice-versa as they were developed together, making the addition of new languages tricky and time-consuming. It further hampered the flexibility promised by CORE that it was dependent on in-house languages, preventing its powerful reuse features to be widely adopted by the modelling community. To appreciate the scale of improvements made to the CORE architecture, we first need to look back at how CORE concern features were realized by models before.

Figure 3.1 shows a highlight of the previous CORE metamodel. Features were realized by models directly, and those models were natively recognized by CORE because they could only be from a limited set of predefined languages that were built to use CORE. In other words, there was no separation between languages and CORE. In the previous metamodel, shown in figure 3.1, this translates to features (*COREFeature*) of a concern (*COREConcern*) being realized by a *COREModel*. The predefined modelling languages in CORE had to subclass the *COREModel* class, with their structural modelling elements being subclasses of *COREModelElement*.

We can additionally notice that the feature model language used to represent the features of a concern is itself (*COREFeatureModel*) a model extending *COREModel* with its *COREFeature*s extending *COREModelElement*.

Reusing models was previously allowed by relying on the metamodels of the predefined languages to extend the key classes in the CORE metamodel that described its customization and usage interfaces. Each modelling element that extended *COREModelElement* had to also properly use the *partiality* and *visibility* fields of that class to identify its elements that needed to be

3.1 CORE Languages

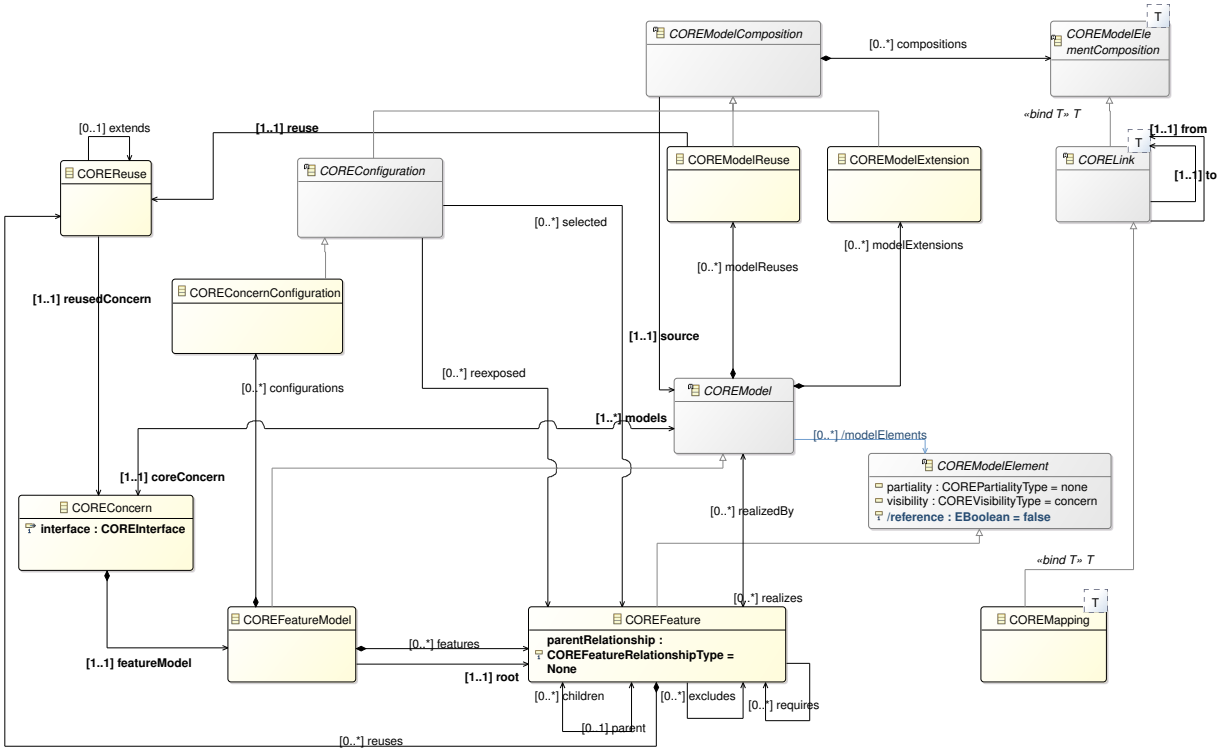

Figure 3.1 – Selected Elements from the old CORE Metamodel Illustrating how Concern Features are Realized and Reused

customized and elements that were re-exposed for usage. The *COREModel* aggregated as *modelReuses* the declared *COREReuse*s, as well as the model extensions, *COREModelExtension*s. The CORE variability interface is specified for a concern through its *COREFeatureModel* that lists all configurations, *COREConcernConfiguration*s, of which one could be referred to at reuse time as the selected configuration.

3.1.2 The New Concept of CORE Language

Scientific literature defines a modelling language as *"the set of all models that conform to a modelling language's abstract syntax, that are represented by some concrete syntax and that satisfy a given semantics"* [27]. An abstract syntax, for a given application domain to model, names all the identifiable concepts and their respective relations and thus is equivalent to a metamodel [27]. Therefore, CORE is informed of the abstract syntax of a language that it integrates through the language metamodel that is provided by the creator of the language.

Concrete syntaxes of languages are their notation, they could be for instance the standard prescribed graphical appearance of a modelling language, how its models are read, written or drawn. CORE does not directly handle how languages' concrete syntaxes are specified as they not relevant to its current structure, therefore the separate task of implementing editors and viewers is left for the language designer.

Finally, the semantics of a language inform us on the meaning of its syntactic elements, they establish a mapping between the notation and what it represents in the semantic domain we aim to model [16]. In order to create and manipulate syntactically correct models that are semantically meaningful, languages offer a set of valid operations that form their structural operational semantics [27]. We decided to express these model construction steps in CORE with the novel concept of *language actions* that we bring forward as our contribution.

In CORE, we propose to capture the structural operational semantics of a language by having it declare a set of operations that can be executed in a meaningful sequence to build a model while preserving syntactic validity throughout the process. These operations, that we call *language actions*, are defined by the language designer and have to be provided alongside the language metamodel whenever a language is to be used in CORE. Together with the metamodel that specifies the structure of language, these language actions specify its behaviour and the model construction semantics, i.e. how one would go about creating and modifying model elements.

Definition 2: *Language Actions*. Set of actions a modelling language exposes to allow the creation and editing of models that enforce a consistent use of the metamodel elements.

On top of these and to offer the reuse functionality of CORE, a *weaver* is also provided for a language, prescribing the composition mechanisms that allow linking its elements with other

languages'. To fully reap the reuse benefits offered by CORE, models need to be composable, hence any modelling language within CORE must define a weaver specifying the composition rules of its modelling elements. The provided weaver is critical for allowing CORE to weave the language metamodel elements together, i.e., algorithmically decide which elements should be replaced, combined or deleted and how it should be done. Although CORE by itself allows defining mappings between modelling elements that need to be weaved, weavers are still needed to specify the right combining procedure. Weavers are fully supported in the CORE metamodel but will not be explored in more detail in this thesis as they are the responsibility of language designers and fall outside of our research motivations.

3.1.3 Redesigned CORE Metamodel

We redesigned CORE structurally at the metamodel level to introduce the abstract *artefact* element that will allow languages to access the three reuse interfaces. As such, a language is brought into CORE first by referencing its independent metamodel that defines the language abstract syntax. The language actions provided by the language designer are then loaded into CORE and serve as proxies to expose the semantics of the language. Lastly, the language weaver is referenced by CORE and called whenever reused and reusing model elements need to be weaved together.

The new CORE metamodel, thus, fully achieves a complete disentanglement from languages as it was restructured to allow support for loading externally defined languages and language actions. We can see this separation in the new CORE in figure 3.2, where elements supporting the three discussed reuse interfaces now relate to a language-agnostic element, the artefact. Languages are now represented by *CORELanguage* which extends *COREExternalArtefact* that holds a reference to the root metamodel element of the external language. Language actions are directly accessible from it, represented as *CORELanguageAction*s.

Definition 3: *Artefact*. A modelling artefact represents a model created for a specific purpose in a concern, using an externally defined modelling language. It stands as a language-agnostic placeholder for models made in languages defined outside of CORE, offering them access to its customization and usage interfaces without explicit dependencies between CORE and the language. Artefacts can then directly be used in CORE compositions and referenced by scenes to realize features, regardless of how the external language is defined exactly.

We also see the introduction of the concept of a scene (*COREScene*) that is now what realizes a feature of a concern, it serves as the central organizing structure of all related modelling artefacts. It aggregates all artefacts relating to the realization of its feature except those that pertain to the concern's reuse, like the feature model, *COREFeatureModel* which is the variability interface of

3.1 CORE Languages

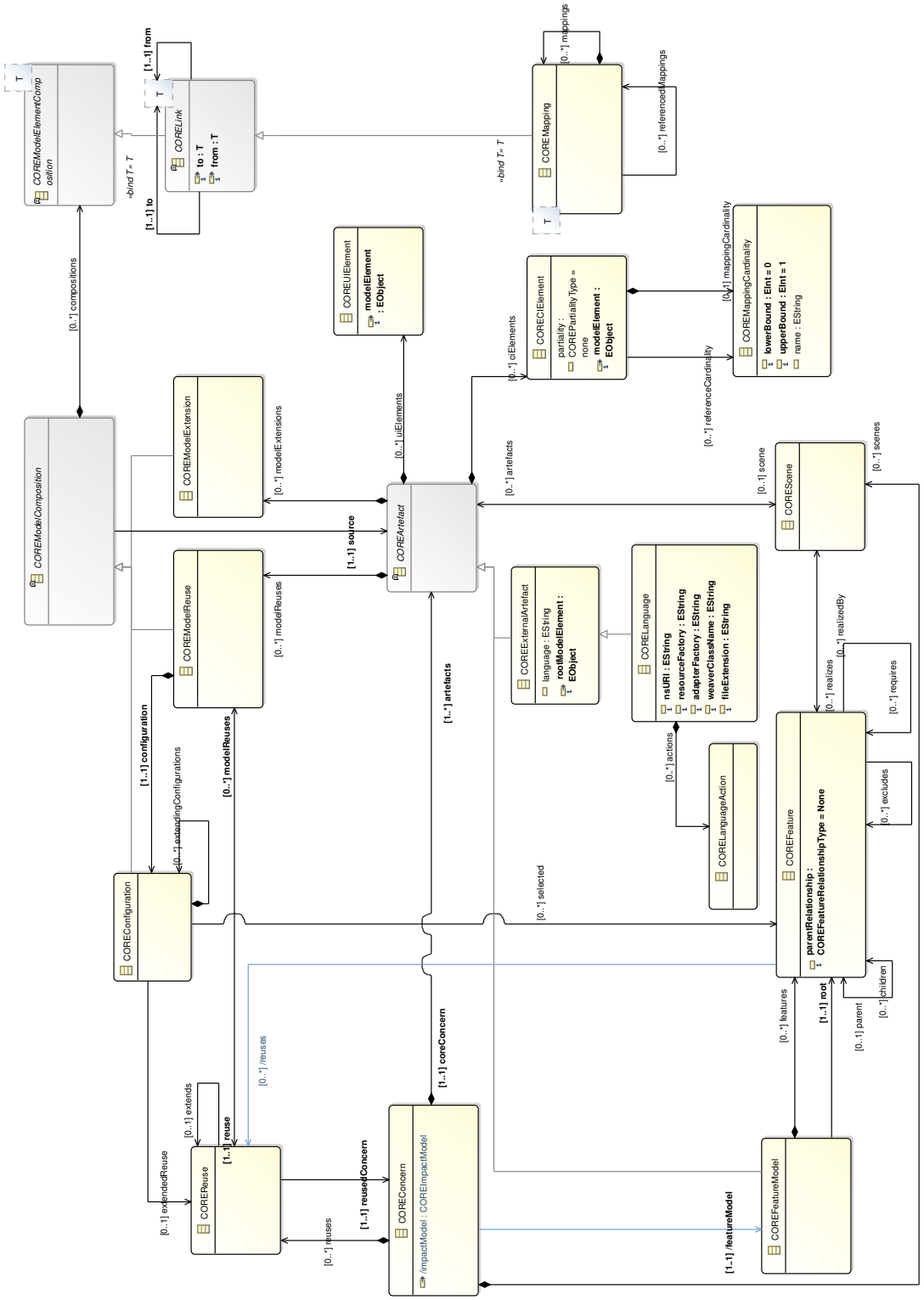

Figure 3.2 – Selected New CORE Metamodel Elements Defining Languages and Perspectives

the concern. The *COREArtefact*s have references to the model they reuse, *COREModelReuse*, and the ones they extend, *COREModelExtension*. They also specify their usage interface elements, *COREUIElement*, and customization interface elements, *CORECIElement*, each of which have references to model elements in the external language metamodels. These references fully separate CORE reuse from the intricacies of languages, as there is no more inheritance involved and even CORE CI partiality is controlled in CORE.

Finally, the aforementioned weaver of a language can be specified as an attribute of this *CORELanguage* (*weaverClassName*).

3.2 Perspectives

Notwithstanding how critical standardized general-purpose modelling languages are to MDE, they are by themselves not enough to cover all modelling domains and purposes efficiently. To ensure adequate modelling expressiveness for a specific system at a specific stage, custom-built DSLs are often necessary due to the system's peculiarities or the specificities of its development. In practice though, the development of a DSL remains to this day a tedious process that takes considerable efforts and time to do properly and that can benefit immensely from more reuse.

Thus, in order to alleviate this process, we introduce the concept of *perspective* to allow the quick creation of domain-specific modelling languages in an intuitive and flexible reuse-oriented way. A perspective is, in our vision, an intermediary conceptual layer applied on top of a language to tailor-make it for a specific purpose. It would encourage and facilitate the reuse of existing general modelling languages by allowing developers to tweak them to fit more closely to their development process. These developers could then spend less time defining new languages and instead quickly tailor existing languages in-house to best match their needs.

Definition 4: *Perspective*. A redefinition of existing modelling languages through selective re-exposing and combining to tailor them for a specific purpose or domain. This is accomplished by choosing which subsets of the languages' elements and interactions to re-expose, combine and hide. If it is applied to a single language, we refer to it as a *single-language perspective*, as opposed to a *multi-language* one when applied to multiple languages.

In the future, perspectives can evolve to become full-fledged Multi-View Modelling views but our main purpose here was to first allow convenient reuse of existing languages rather than offer viewpoints to separately display the different aspects of a system. Further discussion on Multi-View Modelling and comparisons between the existing approaches and our perspectives is provided in the related work chapter.

3.2.1 Single-Language Perspective as DSL

Perspectives, in their simplest form, serve as a way to restrict a language for a more specific usage. Nowadays, many DSLs are built by starting with a general-purpose language due to the extensive variety and quality of existing standard modelling notations. The general language's structure and behaviour are then limited with the intention of enforcing a targeted modelling process. Specifically, the DSL designer decides which elements of the language should be kept as-is and which should be modified or restricted to best capture the target modelling domain. Then this designer often also has the task of coming up with a modified concrete grammar, or notation, than the one of the general language.

For example, in class diagrams, we could restrict the creation of *operations* in a *class* to emphasize domain concepts elicitation in a perspective. The original language is thus said to be *typed* by the created perspective that reuses and re-purposes it for a specific usage or purpose that is pertinent to the system being developed. A language could cumulate multiple different perspectives it may have many, allowing a quick language prototyping process with reduced efforts.

In this thesis, our focus was on single-perspective as a means of quickly building a DSL by reusing an existing CORE language, this constitutes the first step in building more complex perspective types. Our contribution consists of defining a structure and the associated process to allow *typing* CORE languages with CORE Perspectives to tailor them for a specific usage.

Currently, before being able to use it in a perspective, a language must already be loaded in CORE, giving us access to its metamodel and its language actions. Then the perspective designer, the DSL designer role in our approach, decides which elements of the metamodel should be re-exposed and which language actions are allowed whereas others are restricted. Defining new perspectives does not necessarily make existing models of the original languages obsolete, they can be reloaded in the new perspective and reused to show them from a different angle. This is possible because instance models created according to a perspective still conform to the modelling language the perspective is based on and comply with their shared metamodel. Thus, perspectives offer a way to adjust the structure and behaviour of the language to the precise abstraction level and expressiveness required by the current development phase while maintaining consistency with the existing body of models.

3.2.2 Integrating Perspectives into the CORE Metamodel

The first step to introduce the concept of perspectives was to create *COREPerspective* in the CORE metamodel, which holds a reference to the root of the external metamodel of the language it is based on. This led to the refined metamodel shown in 3.3 where now a *COREScene* can be typed by a perspective, specifying with which tailored language it is realized. We also decided that to ease

3.2 Perspectives

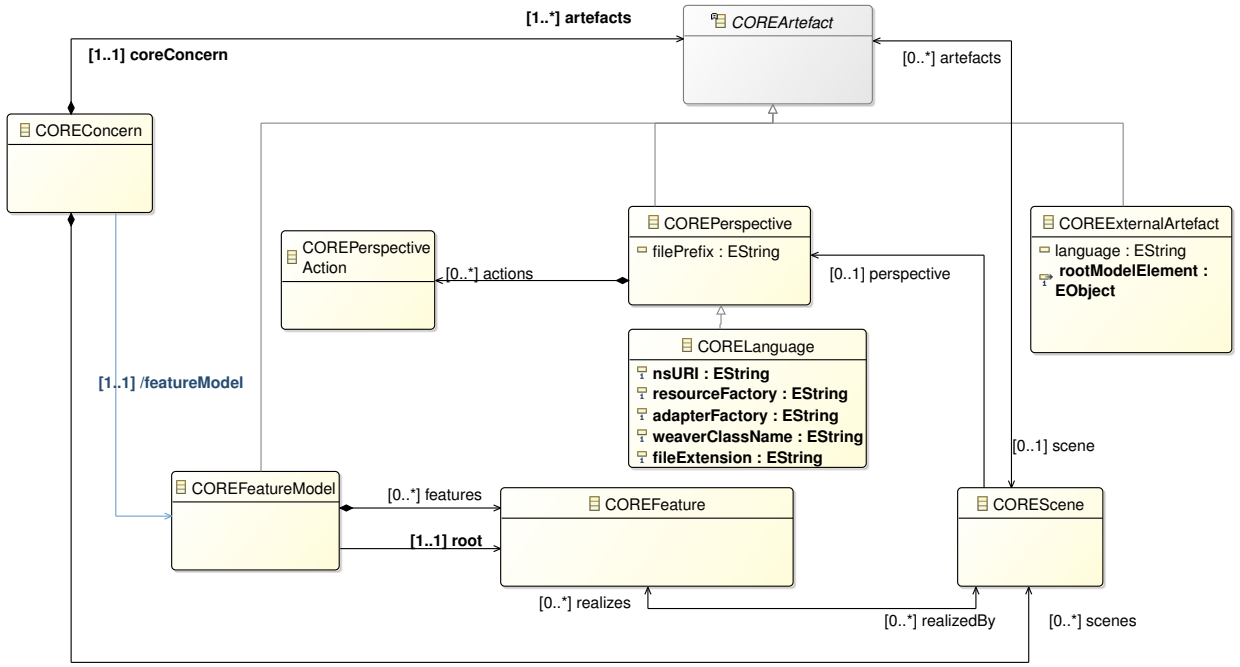

Figure 3.3 – Highlight of the New CORE Metamodel Defining Languages and Perspectives

the implementation of the first version of perspectives, they would serve both for their intended purpose explained above but also as an adaptor for external languages. As such, CORELanguage now extends COREPerspective allowing the perspective itself to serve as the interface between the external language metamodel and CORE. We now only have *COREPerspectiveAction*s accessible from CORE, as they also act as adaptors for external language actions.

Each CORE Scene can then be optionally typed by a perspective that serves as an adaptor for an existing modelling language, a tailored version of such a language or even a combination of multiple languages. This adaptor mechanism is how currently we realize a feature with an existing modelling language, although it is referenced by the *COREExternalArtefact*, in order to be usable in CORE the language is cast, without changes, as a perspective.

For now, this CORE perspective allows the language to be used in CORE and exposes all of its language actions as *COREPerspectiveAction*s. The modelling language then becomes fully usable in CORE as a *CORELanguage* and can now be used to realize a feature when used as the perspective of its realizing scene.

An external language can still be brought as is in the form of a *COREExternalArtefact*, it will

hold a reference to the root of the external metamodel.

3.2.3 Towards Multi-Language Perspectives

Perspectives may also be used to group two or more different modelling languages and prescribe the relationships between them to achieve a specific purpose, or help with validation and consistency. Such multi-language perspectives could, for instance, one day, provide a way to link one element of one language to another in a different language to keep them consistent. They would also create novel perspective actions not defined in the reused languages' actions could consist of combining existing language actions in some order. Our long-term vision for such perspectives is that they allow building on top of their language's actions more complex *perspective actions* offering higher-level composite interactions. Examples of such higher-order perspective actions could be:

- **Enforcing** that the lifelines in a *sequence diagram* correspond to a class in a joint *class diagram*.
- **Linking** the creation of an operation in a *class* to the automatic mapping to a corresponding *sequence diagram* model.
- **Disallow** creating a model element in one language (e.g. *class*) that doesn't have an equivalent in another language (e.g. *use case actor*)
- **Infer** automatically the states and transitions of a *statechart diagram* model from a corresponding *sequence diagram* model.

Currently, such inter-language perspectives aren't fully supported by our framework and as such will not be covered in further details for the remainder of this paper. Nonetheless, the metamodel changes were made while keeping multi-language perspectives in mind to allow their future full realization. For instance, the CORE Scene was introduced specifically to support the realization of a feature by combining multiple models in different languages together in one referenced perspective.

3.3 Using Languages and Perspectives

First, it is appropriate to identify the three main roles that developers can play when using CORE as their approach to model-driven engineering, although someone may stack up more than one of those roles.

The first one is the familiar one of a *modeller*, in charge of using the modelling tool to realize a certain feature from a concern of interest, it is assumed they are familiar with the language offered by the perspective they use to model their solution [4].

Another role that was mentioned previously is the *language designer* who is in charge of designing a completely new modelling language when existing languages, like those under the UML family, can't capture the system being built appropriately [4]. The task of defining a new modelling language with a metamodel can be complex and time-consuming but remains sometimes necessary if the domain is very peculiar and unexplored, CORE now allows these new languages to be plugged in as external artifacts seamlessly.

Finally, the third and new role we introduce here is the one of a *perspective designer* that is in charge of customizing and combining existing languages through tailored perspectives that allow the most effective modelling of the targeted section of the system.

The modelling approach we propose involves these roles collaborating in an ecosystem that begins with language designers defining a language and publishing it for others to use. Then, perspective designers could use the published languages to design relevant perspectives they will also make available for others to use and build with. Modellers thus get access to a vast and heterogeneous library of languages and perspectives to suit the needs of their specific project. Linguistic needs will differ across projects and their different development phases but a mature ecosystem would consistently offer an appropriate language or perspective.

CORE thus offers a complete language definition and modelling workflow following MDE principles to satisfy any design requirement. This workflow and the different roles involved as well as their respective tasks can be seen detailed in figure 3.4 or summarized as follows.

1. Create a CORE Concern that encapsulates the module being built and can be itself reused.

2. Add relevant CORE Features to the concern in the form of an extensive CORE Feature Model.

3. Each feature can then be realized by a CORE Scene which organizes the models used into their respective CORE Perspectives.

   A perspective can be just an adaptor for an external language, the model is then a CORE external artefact.

   A perspective could also be tailor-made by a perspective designer for a specific purpose, reusing one or more external languages.

4. The modeller then gets access, through some editor, to the perspective modelling elements and actions to construct a syntactically correct model that accomplishes the feature's intent.

3.3 Using Languages and Perspectives

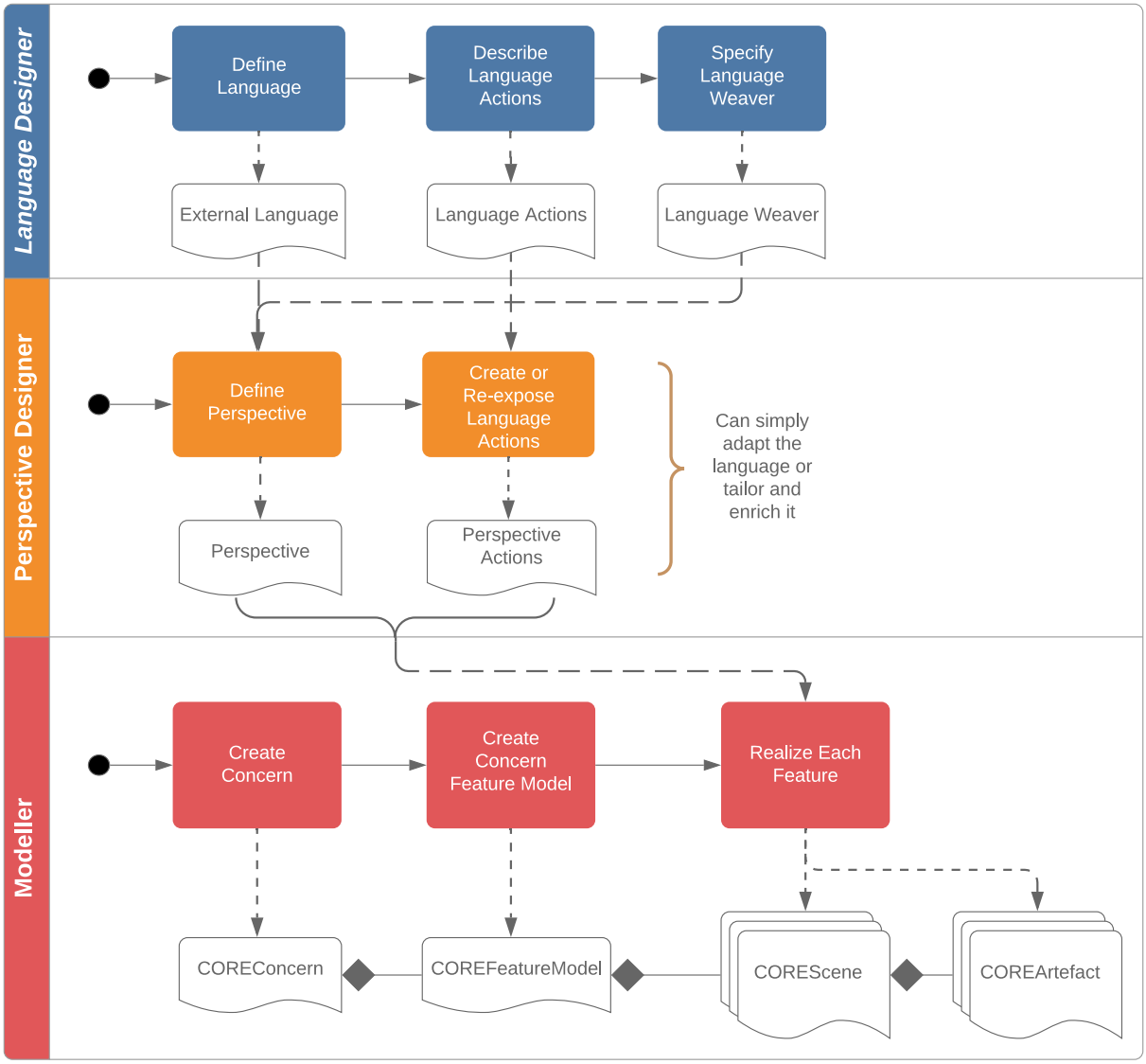

Figure 3.4 – Modelling Workflow in CORE Using Languages and Perspectives

4
Perspectives in Action

Although up to now we've discussed the addition of language support and perspectives conceptually, they really take shape in practice and to show so we will now detail the creation of a language and two perspectives on top of it. Namely, the UML Class Diagram modelling language that was restricted with two perspectives to be used for Domain Modelling and Design Modelling, respectively.

4.1 Class Diagram Language

A Class Diagram is a type of static model capturing the structure of an object-oriented system. It is part of the Object Management Group (OMG) standardized modelling notation called Unified Modelling Language (UML) [18]. The Unified Modelling Language defines Class Diagrams specifically as a way to describe the structural properties at the level of classes, their attributes and relationships between each other [13]. Structural elements of a system are the concepts in its domain alongside the additional internal ones created during development. It is said to be a static model because it does not capture the time-dependent behaviour of the system but rather non-dynamic architectural features.

The main modelling elements in a class diagram are classes, their attributes, their operations, their relationships with other classes which are either associations, of various kinds, or generalizations. The complete definitions of these concepts are found in the official published UML specifications[25] but we will provide a summary of each below [18].

A class describes a concept from the application domain or its solution, it is identified by a name and can contain structural attributes as well as methods to define its behaviour. Classes used to represent more implementation-dependent concepts can be abstract, in which case they cannot be instantiated either because some of their methods aren't implemented or because they are intended solely for specialization. Visibility is also provided for a class, which controls if this class can only be seen by other classes of the model (private), of inheriting models (protected), of

its parent model organizing unit (package) or by any other class (public).

Associations link two classes together in some direction or bidirectionally, making one of the association ends or both navigable. Each end of an association points to a class and can have a role name, a multiplicity defined by a lower and an upper bound and a reference type making it either a regular, a composition or an aggregation association.

An association of upper bound multiplicity bigger than one can be decorated with two boolean properties, its ordering and its uniqueness. Ordering is the property of an aggregation or composition stating if the values contained are ordered or not, by default aggregations are unordered. Uniqueness is a property of the elements contained in an aggregation or composition that specifies if duplicate elements are allowed or not and whether it is a bag or set. By default, elements of an aggregation are unordered and unique.

Associations are said to be n-ary when they link together three or more classes, in opposition to simpler more common binary associations [18]. Multiplicities of n-ary associations are less straightforward to use than multiplicities for binary associations. The multiplicity range at one specific association end represents the minimum and the maximum number of potentially associated instances when holding all the other *n-1* ends' fixed [18]. Navigability is also challenging to read in the case n-ary associations, multi-directional navigation being harder to reason about in practical terms. Aggregation and composition are meaningful only for binary associations and shouldn't be allowed for n-ary ones at all.

Association classes make it possible to enrich an association occurrence with all the properties of a class. They are represented by a class linked to the geometrical centre of an association.

The end of an association can also be decorated with a qualifier, a typed attribute that serves as a key or an index to accessing a unique element in an aggregation or composition. A qualifier is used to distinguish and select na object from a set on the other end of an association whose elements are uniquely identified by some attribute [18]. It is typed by a primitive type or a class that exists in the model and that can be used to retrieve a specific element.

A relationship could also be of the generalization type, commonly known as an inheritance in programming. It is always directed from the more specific subclass (child) towards the more general superclass (parent).

An attribute of a class is a structural feature that holds some relevant identified data and that has a visibility. Attributes are usually typed by primitive types like *String*s or *Integer*s, object attributes typed by another class defined in the model should be shown with associations.

Classes can also have operations to specify a behaviour, a transformation or a query that this class or an object that is an instance of this may be called upon to execute. The operations executed by the class are said to be static. An operation is composed of its name, a visibility, a returned

type and a list of typed and identified parameters. The operations that are meant to be executed by the class and not its instances are identified as static operations. They can also be abstract, in which case they are meant to be specified by another class inheriting from the one they are defined in. There are three categories or types an operation can belong to: it can be a regular operation, a constructor or a destructor.

Finally, a class diagram can contain a note, a textual description that can be linked to any model element to further inform about it [18]. A note is a modelling element itself and can be used to present any kind of information about an element, such as a comment, a constraint or some relevant statistic.

4.2 Class Diagram Implementation

4.2.1 Metamodel

As described in chapter 3, the first step to designing a modelling language for CORE is to come up with a complete metamodel specifying its modelling elements. The complete metamodel for class diagrams is shown in figure 4.1, one can find most of the concepts discussed in the previous section as classes here.

For instance, associations can be n-ary when they have more than two *AssociationEnd* elements in their *ends* aggregation. Specific constraints are applied to the n-ary association ends to make sure they are all navigable and that the reference type can't be aggregation or composition.

Association classes are represented by associations that point to a class with their *associationClass* reference.

Qualified associations, on their side, are associations that point to a *Type* with their *qualifier* reference.

Notes are full-fledged modelling elements and can annotate any identifiable modelling element *NamedElement*.

Finally, a useful special case of a class is shown in *ImplementationClass* which refers to an existing class in a programming language that can be brought into the model for convenience. For spacing and formatting reasons, primitive types such as strings, int, float and double number, characters, arrays and enumerations have been omitted from the depicted metamodel.

4.2.2 Language Actions

The second step for integrating a language with the new CORE is to specify its language actions. These language actions encode how models and model elements expressed using the language are created and edited. As such, language actions can be seen through the prism of the four basic functions commonly used in persistent storage: Create, Read, Update, Delete or CRUD. They are

4.2 Class Diagram Implementation

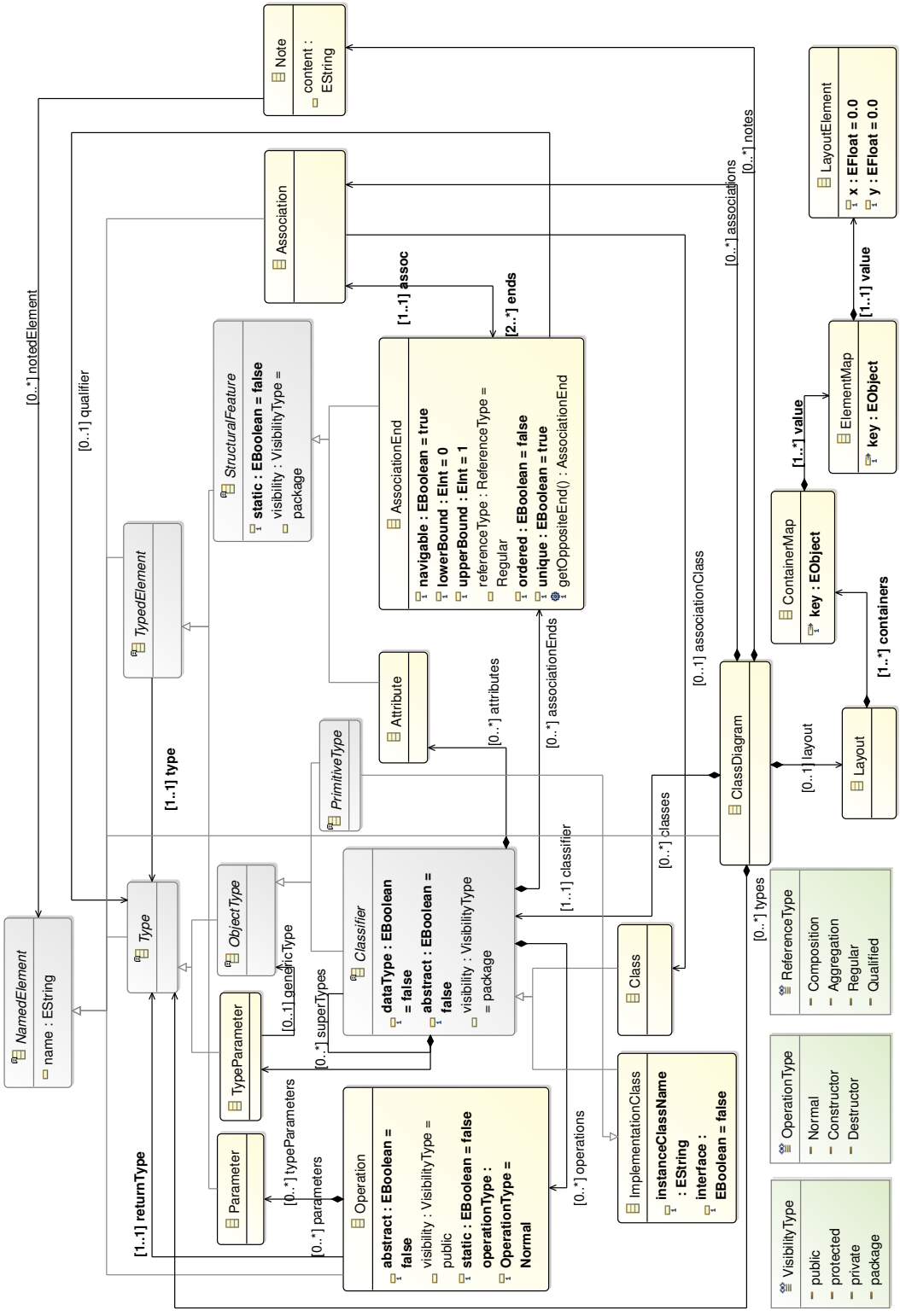

Figure 4.1 – Class Diagram Language Metamodel

however higher level CRUD functions that combine more primitive model edits into a coherent relevant modelling transformation the language designer exposes. Because, as described earlier, we are interested in language actions as operators allowing to modify a model while maintaining syntactic correctness, we do not need *read* operations.

The language actions we defined for the class diagrams modelling language are listed in tables 4.1, 4.2 and 4.3 respectively for addition, editing and deletion actions. The second column of the tables lists the metamodel elements affected by each language action to highlight the complementary nature of metamodel elements and language actions. Parameters are specified for the various actions whenever they are needed. All these actions apply to a loaded Class Diagram model instance and will either add elements to it, modify some elements or delete them, making the current class diagram an implicit parameter to all actions. For instance, creating a class takes as parameter the name of the new class and will instantiate a class model element with that name before adding it to the class diagram. Therefore, typically several lower-level model transformations are with a language action. For example, the action to add a class combines creating the class, setting its name attribute and adding it to the class diagram, which are respectively one *create* and two *update* primitives. The action allowing the creation of an association, on the other hand, takes as parameters the classes that the association should link together and a direction if it is not an n-ary association. The action to create an implementation class can also have a parameter listing the required generic types.

4.3 Domain Modelling Perspective

During early software development phases, it is a common activity to analyze the structure of a problem domain by specifying relevant domain concepts alongside their properties and relationships. While in theory any structural modelling language can be used for domain modelling, practically most software development processes that incorporate domain modelling use a variant of class diagrams to express domain models. Most of the time, only a subset of the class diagram language is used. For example, since in domain models the focus isn't on behaviour, operations are typically not used.

Mainly, domain models are composed of classes with attributes, alongside relationships such as associations with multiplicities, aggregation or composition and inheritance. It can thus be represented easily with a tailored version of the class diagram language. The adjustments needed would consist of restricting the use of any behavioural elements of the language as they are irrelevant to describing the system at the domain modelling phase. A perspective would then be the quickest and most efficient way to repurpose our class diagram language for the domain modelling use case. That is why domain models are a great choice in order to showcase the capabilities of perspectives.

Table 4.1 – Language Actions of Addition Type

Language Action	*Affected Metamodel Element*	Action Parameters
Create Class	*Class*	Name, Visibility
Create Association Class	*Class, Association*	Target Association, Name, Visibility
Create Note	*Note*	Note Text, Annotated Element
Create Inheritance	*Class*	Parent Class, Child Class
Create Association	*Association*	Linked Classes, Direction
Create N-ary Association	*Association*	Linked Classes (3 or more)
Create Implementation Class	*ImplementationClass*	Name, Generic Types
Create Enumeration	*Enum*	Name, Visibility
Add Enumeration Literal	*Enum, Enum Literal*	Name
Add Association Qualifier	*Association, Type*	Target Association, Qualifying Type
Add Attribute	*Attribute, Class*	Target Class, Name, Type, Visibility
Add Operation	*Operation, Class*	Name, Target Class, Visibility, Return Type, Parameters
Add Operation Parameter	*Parameter, Operation*	Target Operation, Name, Type

Table 4.2 – Language Actions of Editing Type

Language Action	*Affected Metamodel Element*	Action Parameters
Move Element	Any Movable Element	New Position
Edit Element Name	Any Named Element	New Name
Edit Note	*Note*	New Note Text
Edit Association Multiplicity	*Association, AssociationEnd*	New Multiplicity
Change Association Type	*Association, AssociationEnd*	New Reference Type
Change Association Direction	*Association, AssociationEnd*	New Direction
Change Attribute Type	*Attribute, Type*	New Type
Change Operation Return Type	*Operation, Type*	New Return Type
Change Operation Parameter Type	*Parameter, Operation, Classifier*	New Parameter Type
Change Visibility	*Operation, Class* or *Attribute*	New Visibility
Toggle Association Ordering	*Association, AssociationEnd*	Target Association End
Toggle Association Uniqueness	*Association, AssociationEnd*	Target Association End
Toggle Class Abstract	*Class*	Target Class
Toggle Operation Abstract	*Operation*	Target Operation
Toggle Association End Static	*Association, AssociationEnd*	Target Association End
Toggle Attribute Static	*Attribute, Structural Feature*	Target Attribute
Toggle Operation Static	*Operation*	Target Operation

Table 4.3 – Language Actions of Deleting Type

Language Action	*Affected Metamodel Element*	Action Parameters
Delete Class	*Class*	Target Class
Delete Note	*Note*	Target Note
Delete Association	*Association*	Target Association
Delete Enumeration	*Enum*	Target Enumeration
Delete Implementation Class	*ImplementationClass*	Target Implementation Class
Remove Enumeration Literal	*Enum, Enum Literal*	Target Enumeration Literal, Owner Enum
Remove Association Qualifier	*Association*	Target Association
Remove Inheritance	*Class*	Target Classes
Remove Attribute	*Attribute, Class*	Target Attribute, Owner Class
Remove Operation	*Operation, Class*	Target Operation, Owner Class
Remove Operation Parameter	*Parameter, Operation*	Target Parameter, Owner Operation

A domain modelling perspective would disable the class diagram language actions that create and manipulate modelling elements irrelevant for domain models. Namely, it would disallow adding operations to classes and consequently their modification or edition, the same would apply to operation parameters. Implementation classes, which are referenced classes of popular programming languages, would also be disallowed completely. Making attributes static is also implementation-related and unneeded in a domain modelling use case. Visually, we would also alter how classes are displayed to always hide the area displaying operations and only show the attributes container if a class has attributes. Visibility for classes and attributes as well as arrows indicating the direction of associations also need to be hidden as it is not needed in domain models. Furthermore, since navigability has no meaning in domain models, associations should be shown without direction arrows and their navigability should not be editable.

4.3.1 Illustrating Example

To allow us to illustrate how the Class Diagrams language and the Domain Modelling perspective can be useful in developing a system, we will devise an example to show what the results could look like. We will first describe a simple system we would like to implement before showing how one could model the domain of the system using the perspective we just built.

In recent years there was a shift in the urban food delivery market toward the use of independent online services to provide riders that will deliver the food, like Uber Eats™. These food delivery services have excelled at harnessing the wide adoption of smartphones and recent technological advancements to provide convenient and cost-effective delivery for any restaurant. They offer an unprecedented variety and fast delivery times to users, thus contributing to a recent 20% annual growth in delivery sales which are expected to reach $365 billion worldwide in 2030 [7].

To implement the backbone system for such a delivery company and quickly bring a reliable product to market its developers could greatly benefit from employing an approach like CORE. The first step would be to define requirements from the company stakeholders and map out a vocabulary for the problem domain using our domain modelling perspective. Namely, the system being developed needs to manage and coordinate drivers subscribed to the service and dispatch them to the closest restaurant a user has ordered from to allow a quick delivery. After the delivery is complete the user payment needs to be processed and split between the rider, the restaurant and the service. An example of a domain model capturing those requirements and built with our domain modelling perspective can be seen in figure 4.2.

4.3 Domain Modelling Perspective

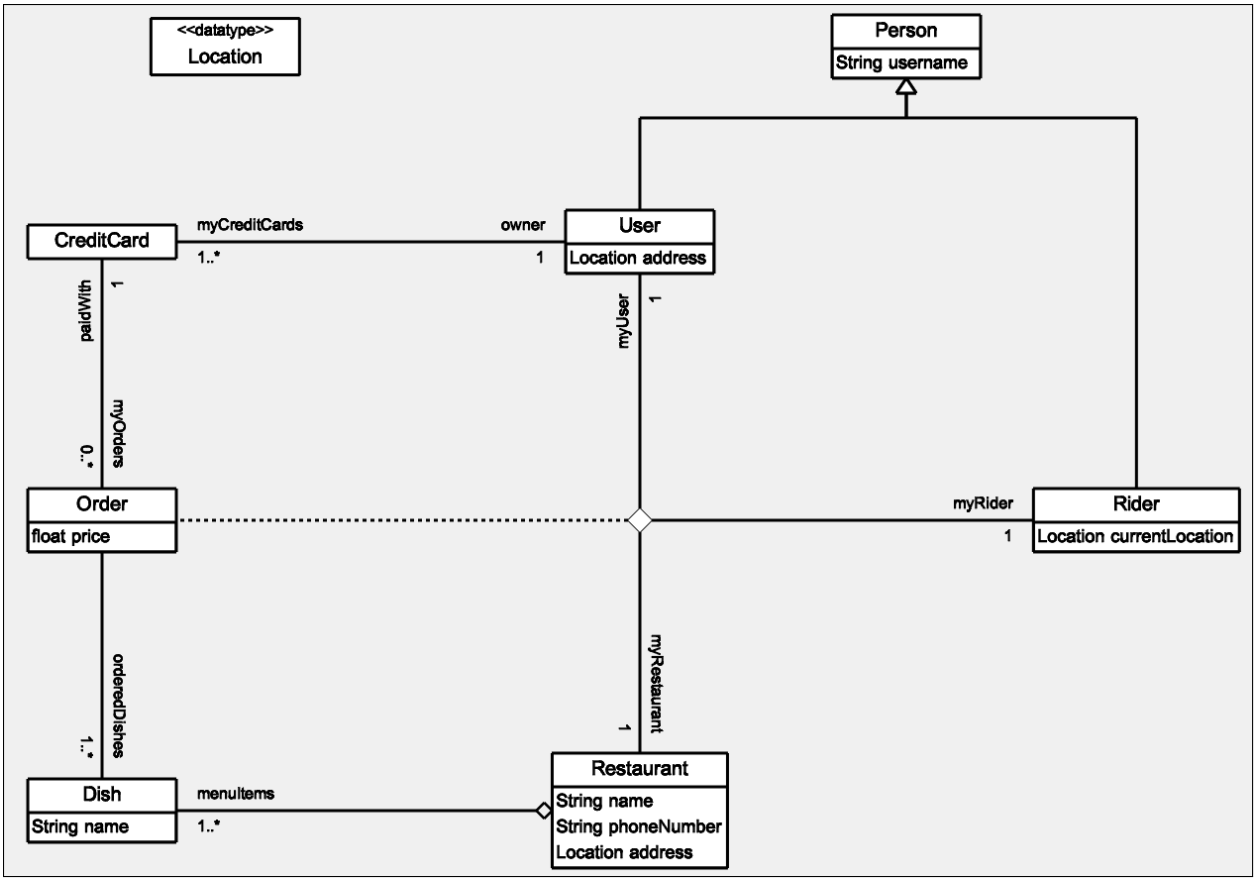

Figure 4.2 – Domain Model of an Online Food Delivery System Designed with a TouchCORE Perspective

4.4 Design Modelling Perspective

Further down the development process of a system, class diagrams are commonly used to prescribe or document the implementation solutions of a system, this is typically called design modelling. Design models should be comprehensive abstractions of a system's implementation. The modeller is at this phase refining previous domain models by clarifying how certain relationships will be implemented and by assigning the system's behaviour to classes.

In our case, we would like a design modelling perspective to restrict our class diagrams language by removing any modelling notation that is hard to correlate with implementation solutions, such as n-ary associations and association classes. It would also encourage the modeller to specify completely the structure of the system, and do so in a way that better captures the lower level implementation imperatives. A design modelling perspective would, on the other hand, enable class operations, static attributes, visibility and importing existing implementation classes which were masked for domain modelling. Visually, everything not shown by the domain modelling perspective should be shown for design models, attributes and operations areas, visibilities and editable association end arrows.

Table 4.4, lists which language actions are allowed or restricted for the design modelling perspective compared to those of the domain modelling perspective.

Table 4.4 – Comparison the Design and Domain Perspectives Respective Language Actions

Language Actions	Design Perspective	Domain Perspective
Move Element	allowed	allowed
Create/Delete Class	allowed	allowed
Create/Edit/Delete Note	allowed	allowed
Create/Remove Inheritance	allowed	allowed
Create/Delete Association	allowed	allowed
Create Association Class	allowed	allowed
Create N-ary Association	disallowed	allowed
Create/Edit/Delete Enumeration	allowed	allowed
Create/Delete Implementation Class	allowed	disallowed
Add/Remove Association Class	disallowed	allowed
Add/Edit/Remove Attribute	allowed	allowed
Add/Edit/Remove Operation	allowed	disallowed
Add/Edit/Remove Operation Parameter	allowed	disallowed
Edit Name	allowed	allowed
Edit Abstract	allowed	disallowed
Edit Static	allowed	disallowed
Edit Visibility	allowed	disallowed
Edit Association Multiplicity	allowed	allowed
Change Association Type	allowed	allowed
Change Association Direction	allowed	disallowed
Change Association Ordering/Uniqueness	disallowed	allowed

4.4.1 Design Example

Going back to our food delivery example, we can now employ our design modelling perspective to derive from our domain model a more concrete model suitable for implementation. As developers move to the design phase of the system, it now becomes relevant to specify more precise structural and behavioural features in the models and thus a more appropriate perspective is required. Because the design modelling perspective is reusing the same language as the domain modelling perspective, the developer can open the domain model as a design model to jump start the process, with the elements that don't exist in the design model replaced by their equivalent or removed if none exists.

The ability to transfer models from one perspective to another, if they both share the same source language, is one novel type of reuse that is offered by perspectives and can serve, for example, to simplify a model from one complex perspective into a cleaner more abstract and intuitive one that can be presented to a less technically fluent stakeholder.

Furthermore, reusable models developed in one perspective can be applied in another perspective. For instance, the generic observer design pattern from the CORE concern library can be applied to the *Order* making it the subject and reusing the existing behavior for informing observers like the *Restaurant* or the *User*.

Figure 4.3 shows one potential model for our system designed using our design modelling perspective. We can see for instance the disappearance of the n-ary association that *Order* was an association class of, the latter becoming a stand-alone class with all required associations. Multiple attributes are added to classes to capture data that needs to be stored for the implementation, and corresponding getter and setter operations appear as they are allowed in design models. The aggregation *myDishes* of the *Menu* is also now decorated with an association ordering, the implementation of this ordered list can then reuse an existing concern in CORE for a linked list for instance. Finally, more concrete enumerations are added to track the status of the delivery and the rider.

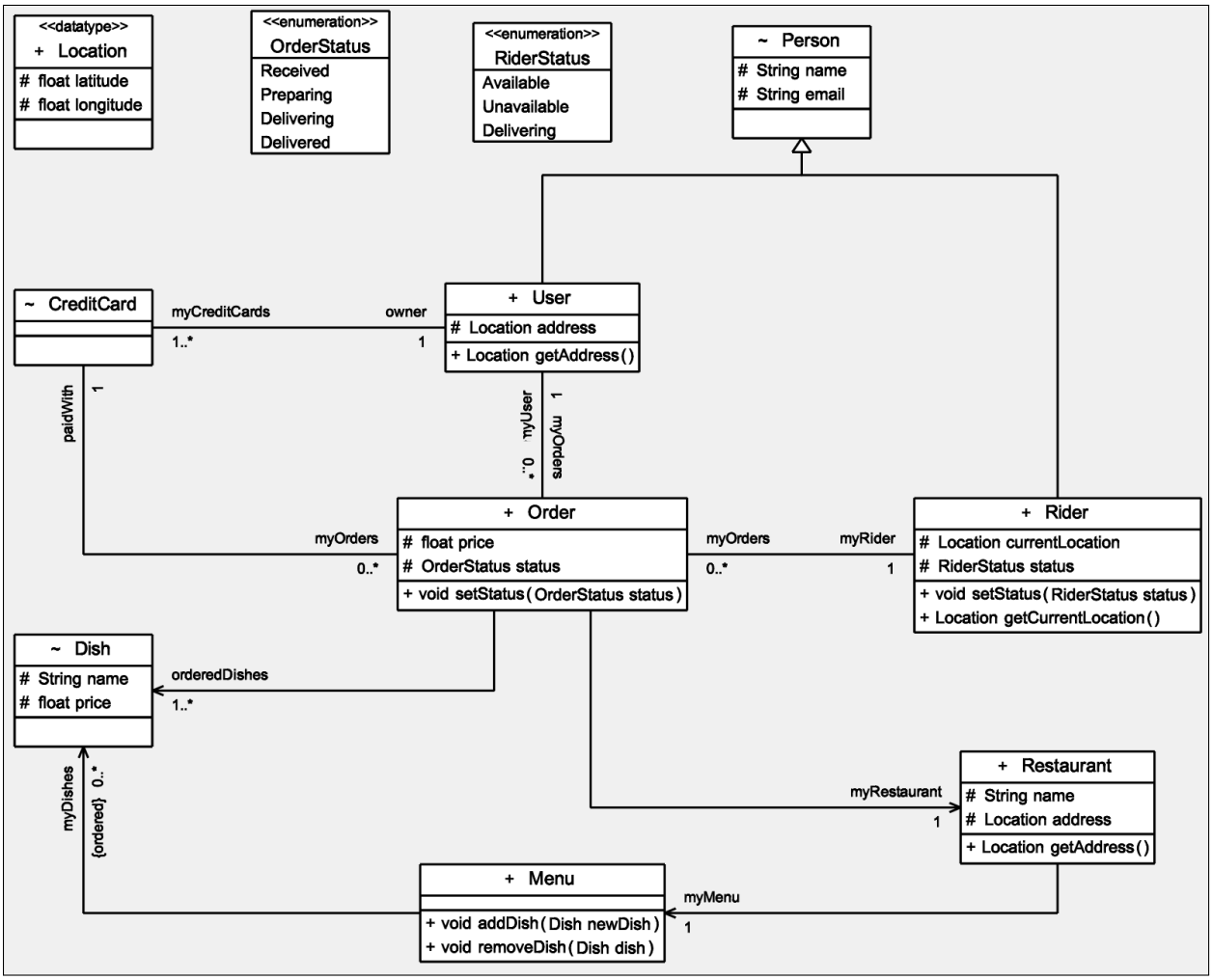

Figure 4.3 – Design Model of an Online Food Delivery System Designed with a TouchCORE Perspective

5 Modelling Tool Implementation

We believe that to encourage broad adoption of MDE as a mainstream software design process, we need powerful and easy to use modelling tools that can be tailored to the specific needs of a development team. Implementing these modelling tools is the crucible that transposes MDE into practice, or in our case that realizes CORE with our TouchCORE modelling application. Thus, TouchCORE was upgraded to support the foundational changes to the CORE modelling framework described in previous chapters and reap their language correctness, specialization and efficiency gains. This section presents an overview of the most important of these changes and how they now allow TouchCORE to offer support for handling existing external modelling languages, as well as creating and using perspectives.

5.1 Technology Stack

TouchCORE is built on top of various technologies, the ones needed to understand we improved it are briefly introduced in the following subsections.

5.1.1 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) [15] is composed of a set of Eclipse plug-ins intended for modelling and generating various kinds of artefacts from models. It offers a complete toolset for building extensive models and generating code to manipulate them in order to integrate them into applications. EMF also distinguishes between the metamodel and the actual model, allowing us to define languages by specifying a metamodel encoding the abstract syntax of a language. In order to design the metamodel, EMF comes with its own modelling language, *ECore*, as a meta-metalanguage that is a custom version of UML Class Diagrams [21]. *ECore* is a subset of the *Meta-Object Facility* (MOF) specified by the Object Management Group (OMG), behind UML, and is similar to the Essential MOF (EMOF). *ECore* allows instantiating *EClass*es with appropriate *EAttribute*s and *EReference*s alongside more primitive types like `int` or `float` as *EDataType*s.

Once a metamodel is defined in *ECore*, EMF automatically generates the model code and associated *edit* code to control it as per the Model-View-Controller paradigm. Creating models using the controllers from the *edit* code ensures that the created models are valid and represent a concrete instance of the metamodel. Furthermore, EMF provides convenience through adapter classes for building views and command base editing to facilitate undoing or redoing changes for instance. The serialization of models is also built-in in the *XML Metadata Interchange format* (XMI) which is the OMG standard for serializing models as XML.

5.1.2 Multitouch for Java

Multitouch for Java (MT4j) [12] is an open-source Java framework that offers tools for building cross-platform applications with an emphasis on multiple-touch enabled interactions. It allows easy and rapid development of 2D or 3D OpenGL hardware-accelerated graphics for Java-based applications with a flexible and modular architecture coupled with good performance. The input system accepts a very broad range of standards on multiple platforms with support for precise complex multi-point gesture customization as well as mouse and keyboard combinations. There is support for the *Tangible User Interface Objects* (TUIO) protocol in addition to the different operating systems in-house touch and gesture libraries. It runs seamlessly on Windows, Linux (Ubuntu) and Mac OS machines and defines interfaces that allow easy integration with existing multi-platform Java applications.

5.2 Language Registry

In order to use external languages in our TouchCORE application, we first had to offer a technical solution for loading them. We decided to implement a Language Registry that serves as the central listing of all the currently loaded and supported languages in TouchCORE. Upon starting the application, a predefined language folder is scanned and any languages found are loaded, after which the tool proceeds to register them as EMF extensions to grant access to their metamodels.

Practically, the language registry is a singleton class called *CORELanguageRegistry* that holds a hash map linking the language name to its *CORELanguage*. The registering process upon application launch is needed as the languages' metamodels are stored as serialized XMI *eCore* models that need to be decoded and loaded. Each language is uniquely identified by a namespace Uniform Resource Identifier (URI) that ensures its modelling elements are uniquely identified and accessible. Once the language metamodel is loaded, we get access to the root object of the metamodel which allows us to start creating instance models.

Perspectives do not need to be loaded like external languages as they are created and stored within CORE whose metamodel is loaded upon starting TouchCORE.

5.3 TouchCORE Architecture Changes

After registering the language in TouchCORE, we had to implement how models created in that language are modified correctly based on the language actions. Now, when selecting a feature of a CORE concern to realize, a selection menu appears showing all loaded perspectives. The user then makes a selection of which perspective he wants to use and that creates a COREScene instance typed with this perspective. Finally, the appropriate modelling view is opened and the user can start creating the model.

TouchCORE follows a strict Model-View-Controller (MVC) architecture where the metamodel of either CORE or the modelling language currently used is the *model* that is edited using controllers. Figure 5.1 depicts a high-level overview of TouchCORE's MVC architecture where the view is the TouchCORE Graphical User Interface (GUI) component.

5.3 TouchCORE Architecture Changes

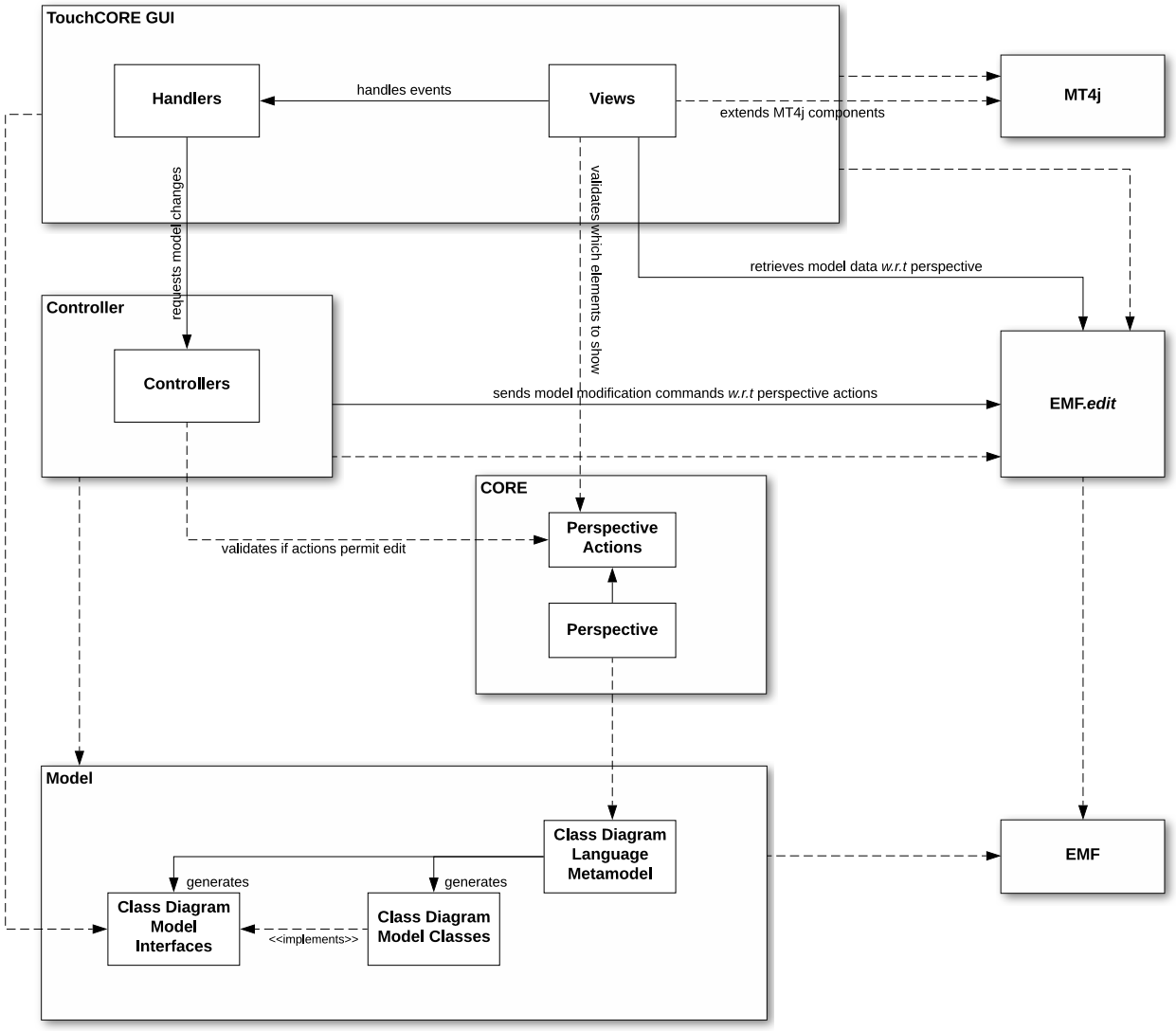

Figure 5.1 – TouchCORE Application Architecture

The GUI is composed of multiple views that correspond to the modelling elements of the language to display, for example, a *ClassView* specifies how to draw a class on the screen. These views extend the MT4j visual components and use their interface to properly display model elements, they act as the language concrete syntax and decide the visual appearance of models created in TouchCORE. Each view has a handler which is in charge of handling user inputs and application events before parsing them to decode which language action they correspond to.

These handlers then call the appropriate model controller to validate and execute the requested model transformation. Controllers are built on top of the Model *Edit* code generated by EMF from the language metamodel.

Both views, with their handlers, and controllers depend on the language model classes and interfaces generated by EMF from the language metamodel. The classes and interfaces implement the structure specified in the metamodel, or in other words the language's abstract syntax. The classes' instances at runtime represent the *model* in our MVC architecture, once a model is saved they are committed to persistent storage as XMI files.

This architecture did change when adapting it to the new CORE metamodel described in chapter 3, that is, it now accesses perspectives and their actions for validation purposes. In order to implement our vision of perspective actions as described in chapter 4, TouchCORE must only modify models as specified by the exposed actions. To do so, we introduced validation checks at the controllers, which query the CORE perspective for valid actions and only execute the model transformation requested of them if it is allowed.

Similarly, views will query perspective actions to know which modelling elements to show in the GUI, dynamically adjusting what the user sees. For instance, the class controller has a method that can be called to verify if adding operations is allowed in the current perspective. Thus, the view representing a class can call this method when it is being initialized and use its returned boolean value to either display or not the operations area for a class. When a user interacts with this view, it will also hide the button to add an operation if the controller method informs it that the current perspective disallows creating operations.

5.4 Building a Generic Modelling User Interface

The most critical part in the adoption of a modelling tool has to do with its usability both in terms of what you can do and how you can do it. Now that we support dynamically changing modelling languages, we needed to rethink the Graphical User Interface (GUI) of TouchCORE to take this into account. This is a challenge in itself as it goes beyond just defining what keywords to highlight and how to do indentation in a text editor for a textual modelling language.

Our goal was to implement one single GUI for a language that is designed to automatically adapt to a chosen perspective. The GUI should show only the views that are relevant to the perspective and only offer the user interactions allowed by the re-exposed perspective actions.

The first type of interactions we had to make adaptive relate to constructing models, i.e. the gestures, buttons and menus that allow a user to create, edit or delete elements. A great example of such an interaction is the ability to add operations to a class, which is allowed in the design modelling perspective but disallowed in domain modelling perspective. We implemented at the

model controllers level a verification interface that allows the views from the GUI to check if certain actions are allowed or not in the current perspective. Because each perspective declares a set of exposed language actions, we can easily verify if a specific action is allowed or not by comparing this set to the default one from the external language. If the GUI already has all the input elements for the language the perspective is built on, then it will now hide the ones that correspond to non re-exposed language actions. In order to be compliant with our existing code base, we implemented this verification as optional methods that can be called for most controller methods that return a boolean informing of its enabled status. The controllers' action verification methods are then called by the different view handlers in our application to decide whether to show specific buttons or react to specific inputs or gestures.

How a Perspective chooses to re-expose the language actions to foster or restrict specific use cases will define the functionality provided and the resulting tailored GUI. However, some differences between perspectives are more subtle than what their respective actions can capture. For example, in domain modelling, the directionality of associations is irrelevant and thus the direction parameter used by the create association action needs to be set by default to bidirectional. Consequently, in domain modelling the user cannot edit the direction of the association but they should still be able to edit the role names and multiplicities of the association. This level of fine granularity of the actions represents a challenge in the GUI as all these actions are by interacting with an association end and a custom solution had to be implemented for this scenario.

The other area where the UI has to be adaptive is in how it displays models, which elements or text fields should be drawn on the screen and which shouldn't be for a given perspective. Good examples of such elements in our case are the Class and Attribute visibility fields, which are relevant as class diagram elements used for design modelling but irrelevant and hidden for domain modelling. The hiding of visual representations is also implemented using verification methods mapped to language actions at the model controllers that offer the graphical views the option to query whether to show or hide an element. This visually achieves one of the important goals of perspectives, showing the right information on a system at the right development phase.

Referring back to our online food delivery system example, we first devised a domain model using the TouchCORE domain modelling perspective, to capture only the important concepts needed. The resulting domain model, shown in figure 4.2, displays only classes for these concepts and their important attributes while operations, visibilities or other irrelevant elements are hidden by TouchCORE. Afterwards, a design model is built from that domain model and shown in figure 4.3, which displays operations, appropriate visibilities, relevant enumerations and detailed directed associations.

6
Related Work

To the best of our knowledge, no prior research has focused specifically on augmenting existing modelling languages with reuse features such as the ones offered by CORE's reuse interfaces. In the broader realm of MDE research, though, many other approaches have been proposed to either provide modelling support with multiple languages or allow language designers to build languages by using and adapting existing languages. In this chapter, we present the related approaches along these two dimensions and compare them with our CORE based approach to highlight the differences.

6.1 Approaches to Multi-View Modelling

6.1.1 Single Underlying Model

Integrating together models in multiple languages to act as different views of a system like our perspectives do can also be done by relying on one large comprehensive root model as a foundation. A prominent example of this approach is the Orthographic Software Modelling (OSM) paradigm proposed by Atkinson *et al* [6]. By creating a Single Underlying Model (SUM), expressed in a custom metamodel, they aggregate all modelling elements together and can generate dynamic views that act like our perspectives by selectively showing relevant elements. This underlying model also allows embedding consistencies and invariant conditions that span across modelling languages, one of the goals of our concept of perspective.

This direct approach, unfortunately, has issues when it comes to the evolution of the languages involved which are all integrated into the single underlying metamodel, making changes difficult. In comparison, our approach leaves the used languages separate by having the perspective only refer to their metamodels and tailor allowed elements and language actions. OSM also does not reuse existing languages and their established editors as everything must be built for the single metamodel, inhibiting the modularity of the approach.

To address these shortcomings, Meier and Winter propose to reuse existing models and their metamodels and group them under their *MoConseMI* approach [23]. They form a SUM and its metamodel by applying operators and transformations to both the initial models and metamodels until they arrive at one model and its metamodel that contain everything from all models [22]. This keeps original models and metamodels intact and maintains consistency by back-propagating changes as the operators and transformations are bidirectional, i.e., they can be executed in both directions.

Our approach keeps existing languages and models unchanged and achieves consistency not by propagating changes but instead exposing the appropriate language actions to the modeller using a perspective. The models are typed by the referenced languages' metamodels that remain intact similarly to MoCenseMI, but since modelling is done through the perspective's interface, consistency is ensured.

6.1.2 Vitruvius

Building on OSM to improve it, Kramer *et al.* proposed the Vitruvius approach [19] to support flexible views built on top of multiple models conforming to different metamodels. These existing metamodels are brought together but they are not merged by creating a SUM from scratch, instead, they are grouped together as a *virtual SUM* built in a pragmatic bottom-up fashion through reuse. The metamodels are bound together through Consistency Preservation Rules (CPR) which allow the enforcement of consistencies and invariant conditions by explicitly stating inter-language dependencies.

These CPRs are enforced by three methods, *correspondences, consistency rules* and *response actions* which are independent of the editors used for the original models and metamodels [22]. Correspondences specify one-to-one mappings between two model elements from the same or differing metamodels. Consistency rules, on the other hand, offer a more advanced way to ensure coherence through OCL statements. Finally, response actions group the updates to other models that are triggered by changes in one model to restore consistency between mapped elements.

These methods to keep models consistent require time-consuming specification of mappings and OCL statements by modellers with the required expertise to do so. In comparison, perspectives allow maintaining consistency by simply exposing, hiding or combining straightforward language actions, without writing lengthy and complex OCL expressions. Perspectives thus proactively prevent modifications that break consistency by not allowing them altogether in the editor whereas Vitruvius reuses the existing permissive editors.

6.1.3 Facet-Oriented Modelling

Another approach exists that is even less intrusive and that intervenes at the modelling level rather than the metamodelling one, through the use of *modelling facets*. Facet-Oriented modelling [9] proposes a lightweight extension to metamodels that makes model objects open, allowing them to acquire or drop *facets*. These facets would enrich the object by adding a type, slots and attributes as well as constraints and they would be used dynamically through declared interfaces. Interactions between facets are regulated by *facet laws*, which govern how facets are to be assigned to instances of a metamodel.

The goal of facets is to be modular and non-intrusive allowing their addition to existing models or metamodels as a purely additive layer without any dependency from the existing models. Facets could be automatically assigned to instances of a language metamodel to tailor how they behave, a facet can even be assigned to multiple objects through a pattern matching rule, to re-purpose an existing model for instance. The facets enrich the object with slots and attributes that become transparently accessible from the model object as if they were always part of it. The focus of the paper is on the usage of typed facets at the model level although they could be used untyped at the metamodel level.

The main advantage over other approaches is that facets leave original metamodels untouched, allowing straightforward model evolution. This makes them similar to our perspectives and offers comparable benefits in the long term maintenance and evolution of models and their metamodels. Facets, on the other hand, don't offer similarly strong consistency checks and enforcement in comparison. Since they are applied at the model level, they cannot span different modelling languages like perspectives can to enforce coherence between them, nor can they dynamically tailor language actions to proactively do so.

6.2 Approaches for Reuse in Language Design

6.2.1 COLD

The other contribution achieved by the new CORE lies in how the new perspectives make it simple to reuse an existing language by adapting it for a given purpose. Concern-oriented language development (COLD) [8] was proposed with a similar objective, addressing DSL development challenges by fostering reuse techniques at the language development phase. It introduces the concept of a language concern as a reusable component of a language that offers the three CORE reuse interfaces we mentioned earlier, namely the variability, customization and usage interfaces. COLD first introduced the idea of lifting the concern software reuse techniques up to the meta-level to allow their use for facilitating DSL development by offering a library of reusable language concerns.

It addressed a gap in the language research corpora where they were no reuse mechanisms at the language level that adequately capture the variability of language modules.

Use cases for COLD would include the creation of new languages, the extension and evolution of existing ones or the contribution to other languages through building modular language concerns. A language designer might need to add a specific feature to an existing language without breaking legacy support for their models, they could do this with the addition of a well-customized language concern. Another designer might be interested in extracting a specific interesting feature from an existing language and repackage it as a reusable concern for the benefit of other designers.

One of the goals that would enable the wide adoption of COLD is the creation of a vast library of language concerns. Such a library would be a collection of reusable language concerns a language designer would have access to, allowing him to compose more complex languages suited to specific needs from simpler language modules.

Although COLD was developed as a theoretical framework, the authors were already foreseeing challenges on its path to adoption. It mainly relies on the process of language *concernification* or defining pieces of a language as concerns. Unfortunately, the proper way of breaking down a language is still an open issue and current strategies either break backwards model compatibility or require tremendous effort and fundamental changes to the language. Our approach offers a stepping stone toward the realization of COLD's goal by making it possible to take existing, fully functional languages and quickly tailor them for a specific purpose. It doesn't aim to solve the issue of breaking down a language into reusable pieces as it assumes a suitable language to reuse and adapt already exists.

6.2.2 Melange

Another approach focusing on improving the DSL design process by reusing and customizing existing DSL artefacts is presented in Melange, a meta-language for modular and reusable development of DSLs [10]. It uses typing relations to build a layer that allows reasoning about DSL artefact manipulation to build more complex languages. Operators that allow extensions, restriction and assembling different DSL artefacts are then defined on top of that layer. Extending existing language artefacts boils down to inheriting them and adding new features to re-purpose legacy elements for contemporary usage. Restriction, also referred to as slicing, is an operator that focuses on targeting a specific feature to extract from a legacy language while ignoring the rest. Assembling or merging entails bringing together two independent languages in order to form a new one, this combines their respective metamodels into a unified one.

The abstract syntax of the languages is represented by metamodels in Melange, similar to our approach. The operational semantics, comparable to our language actions, are represented in Melange as methods added to the *metaclasses* of the language metamodel. Merging two languages

thus happens by weaving together their metamodels into a single composite one, including their respective operational semantics.

Because Melange ends up with one single metamodel, challenges appear regarding the future evolution of such languages just like with SUM approaches. Removing a woven language partially or completely from the unified metamodel becomes a complex task that requires extensive knowledge of the initial components and their metamodels. It is nonetheless needed as languages, over time, evolve while others become deprecated, requiring constant updates to the single composite metamodel without extensive loss of compatibility with legacy models. Furthermore, using the new composed metamodel first requires substantial efforts dedicated to adapting the corresponding modelling tools and applications which all rely on its internal structure.

As of right now, perspectives can be used to allow a developer to tailor, specifically to restrict, an existing modelling language in order for it to be used in isolation for a specific purpose. In comparison, Melange offers the ability to create a single new language that brings together the strengths of multiple different languages. Although more powerful, this approach is meant for language designers conceiving large-scale languages meant for broad usage, perspectives offer instead a faster more lightweight approach to fine-tune languages for a project. Furthermore, in the long run, perspectives will grow to be able to combine multiple languages with an emphasis on reuse, making them powerful language and design process creation tools.

6.2.3 Reuse in Collaborative Modelling

Modelling reuse is also considered critical in collaborative modelling environments, where multiple designers model simultaneously one large system and must be able to build on top of their colleagues' components across multiple languages. Academic investigations of the challenges in collaborative modelling highlight the need for an environment flexible enough for designers from diverse backgrounds to use different notations for their respective specializations while still being able to collaborate and reuse each other's work [2, 11].

In collaborative modeling, having a library of reusable models shared by all designers would also help standardize the system's components and avoid the lengthy process of starting from scratch. As such, modelling languages and their environments need to have comprehensive reuse mechanisms to facilitate their adoption in a collaborative setting [2].

An extension to the CORE framework has already been proposed to allow support for collaborative modelling of reusable concerns by relying version control systems [3]. Nonetheless, the extension would not solve the initial limitation of CORE regarding it's fixed set of predetermined languages.

We believe that our approach can be applied to tackle some of the aforementioned lingering challenges in the context of collaborative modelling. By opening CORE to any language, we make

available its generic reuse capabilities to any potential mix of notations that designers would use in a collaborative environment. And with perspectives, we offer a tool that can tailor languages to suit diverse needs and expertises. Although currently TouchCORE only allows collaboration over one large touchscreen, a web interface is being developed to allow remote collaboration of different designers working together in their respective languages or perspectives on the same project.

7
Conclusion & Future Work

In order for the Model-Driven Engineering methodology to efficiently tackle the costly complexity issues many software companies reach it requires powerful tools that best harness its benefits. One of the most important of those benefits is methodical reuse at the modelling level to avoid laboriously creating from scratch common and recurring components and patterns. Fostering intuitive reuse habits that build on top of existing modelling languages through proper tools and frameworks is thus critical to alleviating substantial superfluous effort and cost expenditures. Such reuse practices must be seamlessly embedded in the languages themselves in order to be consistently adopted and used. Our contribution to achieving this was the definition and implementation of a modelling framework that uses existing modelling language improving them with the powerful three-pronged reusability interface of CORE.

Furthermore, to provide appropriate expressiveness at the right abstraction level while maintaining critical separation of concerns, MDE greatly benefits from tailored and efficient language development. Specifically, the ability to create easy to learn, reliable, reusable and expressive DSLs fine-tuned for a project or system with minimal efforts can greatly improve development. We address this need with the introduction of Perspectives as a lightweight approach for quickly tailoring languages for a specific purpose.

These contributions were materialized in the CORE metamodel allowing us to implement them in TouchCORE, the modelling tool that showcases CORE's capabilities.

The specific research contributions we bring to the edification of an improved modelling ecosystem can be summarized as follows. CORE now has a full-fledged language support structure that offers a foundation layer for a future language ecosystem. Whereas it had a limited set of built-in languages previously, the CORE metamodel has been restructured to be separate of languages, which are modules it can load. The languages supported by CORE are specified using metamodels for their abstract syntax as well as with our new concept of language actions for their semantics. The new notion of perspective offers a whole new tool to tailor or combine languages without

altering the original metamodels and still supporting existing models.

All of these concepts have been validated through a proof-of-concept implementation in the TouchCORE application, where concrete models can be built using our language actions. To the best of your knowledge, TouchCORE is the first modelling tool with a user interface that can adapt dynamically to support newly defined languages using their actions. It was also designed to make challenging modelling editing actions as easy and intuitive as possible, and as such, it excels as a tool for teaching modelling to undergraduate students.

Thus CORE, with its new multi-language support and perspective enabled specialization abilities, stands as a powerful MDE framework supported by a robust modelling application. It represents a promising first step towards the wide adoption of reuse as languages can now take full advantage of CORE's variability, customization and usage interfaces.

7.1 Future Work

Designing new Languages for CORE. Going ahead, we aim for CORE to become an even more robust multi-language framework that offers a vast library of languages and perspectives. Standalone languages, such as use cases, state diagrams, and use case maps, are currently being designed that can integrate seamlessly into CORE to express new modelling concepts.

Harnessing Perspectives for Model Navigation. The next goal for Perspectives, on the other hand, is to harness the access they have to the language inner structure in order to provide an adaptable language-agnostic way of navigating models. A dynamic navigation bar could use the perspective actions to filter specific model elements or offer an option to toggle them in and out of view in a model editor like TouchCORE. Interactions between perspectives are also a matter of interest, they could further be combined and have complex dependencies and interactions to allow navigating to related elements in other languages or perspectives.

Extending into Multi-Language Perspectives. Future research will also be devoted to making perspectives able to combine multiple languages and more specifically allow the creation of complex perspective actions that combine multiple actions from different languages. Perspectives could then become powerful language combination tools capable of consolidating a whole design methodology into one reusable coherent unit that can be used just like a single language. The resulting CORE ecosystem would then contain various software modelling concerns, language concerns and perspective concerns all compatible and coherent together. It would offer developers access to a vast library containing exactly what fits the needs of their development process and would foster widespread reuse at all levels.

Realizing Concern-Oriented Language Design. Ultimately, to realize goals set by theoretical approaches like COLD, CORE would need to load languages and then decompose and *concernify* them in modular units. A language designer could then seamlessly open languages like any concern software model and use its feature diagram to customize them or create a new one reusing the best features of existing languages. The whole language design process would happen in CORE and the resulting language metamodel would be created directly alongside the language actions.

Bibliography

[1] Omar Alam. Concern driven software development. *CEUR Workshop Proceedings*, 1115:106–111, 2013. 2.4

[2] Omar Alam, Jonathan Corley, Constantin Masson, and Eugene Syriani. Challenges for reuse in collaborative modeling environments. *CEUR Workshop Proceedings*, 2245:277–283, 2018. 6.2.3

[3] Omar Alam, Vasco Sousa, and Eugene Syriani. Towards collaborative modeling using a concern-driven version control system. *CEUR Workshop Proceedings*, 2019:155–163, 2017. 6.2.3

[4] Hyacinth Ali, Gunter Mussbacher, and Jorg Kienzle. Towards Modular Combination and Reuse of Languages with Perspectives. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 387–394. IEEE, sep 2019. 3.3

[5] Erik Arisholm, Lionel C. Briand, Siw Elisabeth Hove, and Yvan Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006. 1.2

[6] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. Orthographic Software Modeling: A Practical Approach to View-Based Development. In *Communications in Computer and Information Science*, volume 69 CCIS, pages 206–219. 2010. 6.1.1

[7] Andria Cheng. Millennials Are Ordering More Food Delivery, But Are They Killing The Kitchen, Too?, 2018. 4.3.1

[8] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-oriented language development (COLD): Fostering reuse in language engineering. *Computer Languages, Systems & Structures*, 54:139–155, dec 2018. 6.2.1

[9] Juan de Lara, Esther Guerra, Jörg Kienzle, and Yanis Hattab. Facet-oriented modelling open objects for model-driven engineering. In *SLE 2018 - Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2018*, 2018. 6.1.3

[10] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean Marc Jézéquel. Melange: A meta-language for modular and reusable development of DSLs. *SLE 2015 - Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 25–36, 2015. 6.2.2

[11] Davide Di Ruscio, Mirco Franzago, Ivano Malavolta, and Henry Muccini. Envisioning the Future of Collaborative Model-Driven Software Engineering. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 219–221. IEEE, may 2017. 6.2.3

[12] Fraunhofer-Institute for Industrial Engineering. MT4j - an open framework to create visually rich 2D/3D multi-touch applications, dec 2019. 5.1.2

[13] Kirill Fakhroutdinov. UML Class and Object Diagrams Overview - common types of UML structure diagrams. 4.1

[14] Miguel Gamboa and Eugene Syriani. Improving user productivity in modeling tools by explicitly modeling workflows. *Software & Systems Modeling*, 18(4):2441–2463, aug 2019. 2.3

[15] Richard Gronback. Eclipse Modeling Project | The Eclipse Foundation, 2019. 5.1.1

[16] B. Harel, D., and Rumpe. Meaningful Modeling : What ' s the Semantics Much confusion surrounds the proper definition of complex modeling. *Computer Journal*, 37(10):64–72, 2004. 3.1.2

[17] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-Specific Languages in Practice: A User Study on the Success Factors. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5795 LNCS, pages 423–437. Springer Publishing, 2009. 1.2

[18] Grady Booch James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language Reference Manual Second Edition*. Addison-Wesley, 2004. 4.1

[19] Max E. Kramer, Erik Burger, and Michael Langhammer. View-centric engineering with synchronized heterogeneous models. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '13*, pages 1–6, New York, New York, USA, 2013. ACM Press. 6.1.2

[20] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 2006. 2.2, 2.2

[21] 2019 vogella GmbH Lars Vogel (c) 2007. Eclipse Modeling Framework (EMF) - Tutorial, dec 2019. 5.1.1

[22] Johannes Meier, Heiko Klare, Christian Tunjic, Colin Atkinson, Erik Burger, Ralf Reussner, and Andreas Winter. Single Underlying Models for Projectional, Multi-View Environments. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, pages 119–130. SCITEPRESS - Science and Technology Publications, 2019. 6.1.1, 6.1.2

[23] Johannes Meier and Andreas Winter. Model consistency ensured by metamodel integration. In *CEUR Workshop Proceedings*, volume 2245, pages 408–415, 2018. 6.1.1

[24] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, dec 2005. 2.3

[25] Omg. UML 2.4.1 Specification. *October*, 2010. 4.1

[26] Ayushi Rastogi, Suresh Thummalapenta, Thomas Zimmermann, Nachiappan Nagappan, and Jacek Czerwonka. Ramp-Up Journey of New Hires: Tug of War of AIDS and Impediments. In *International Symposium on Empirical Software Engineering and Measurement*, 2015. 1.1

[27] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, oct 2015. 3.1.2

[28] D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, feb 2006. 1.1, 2.1, 2.3

[29] Matthias Schöttle, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. On the modularization provided by concern-oriented reuse. In *Companion Proceedings of the 15th International Conference on Modularity - MODULARITY Companion 2016*, pages 184–189, New York, New York, USA, 2016. ACM Press. 2.4, 2.4.1

A

CORE Metamodel

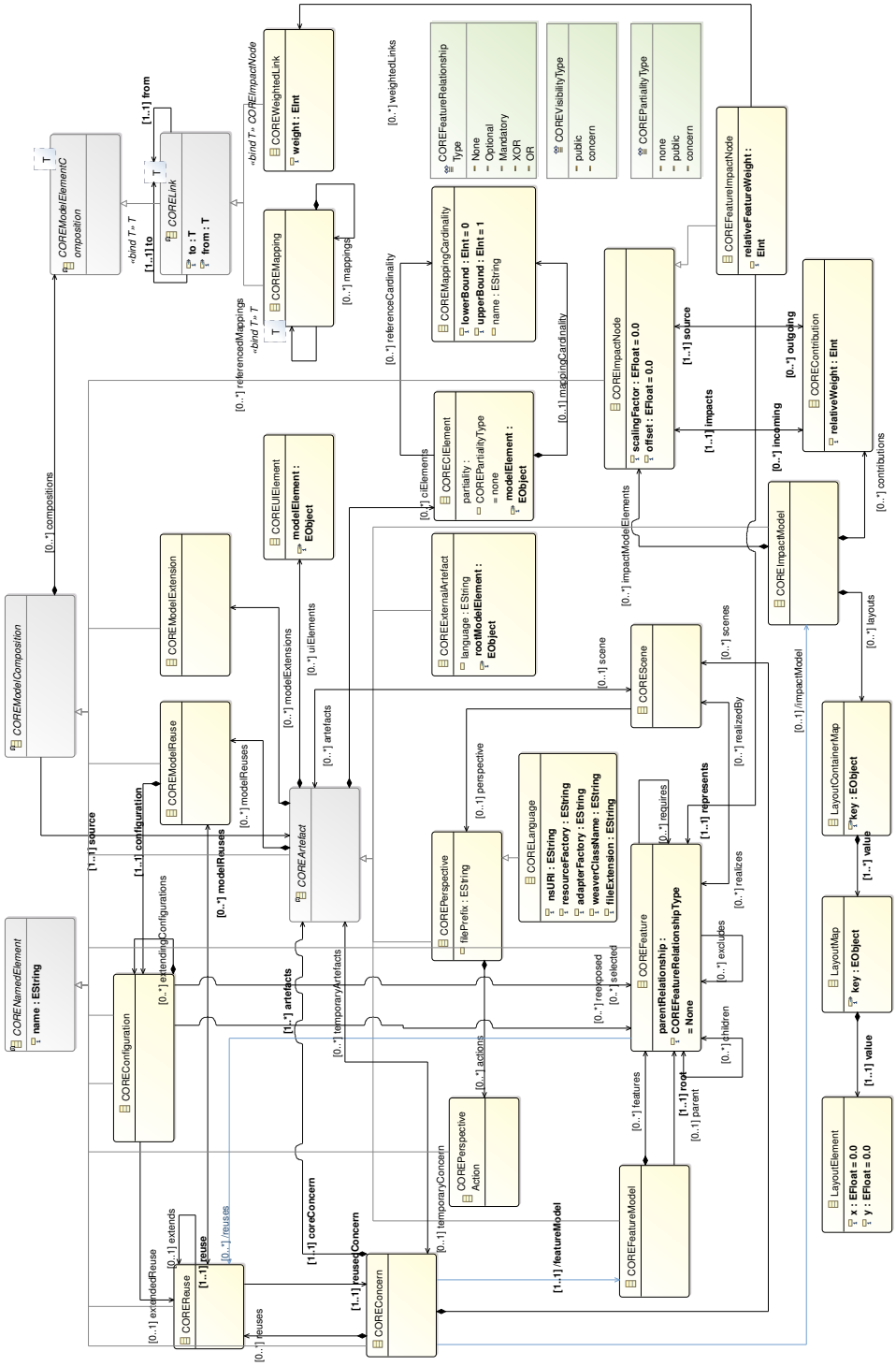

Figure A.1 – CORE Complete Metamodel