CARL
VON
OSSIETZKY
universität OLDENBURG

FAKULTÄT II – INFORMATIK, WIRTSCHAFTS- UND RECHTSWISSENSCHAFTEN
DEPARTMENT FÜR INFORMATIK

A Language-Driven Development Framework for Simulation Components to Generate Simulated Environments

A dissertation accepted by the School of Computing Science, Business Administration, Economics and Law of the Carl von Ossietzky University Oldenburg (Germany) to obtain the degree and title

Doctor of Engineering (Dr.-Ing.)

by Ms. Liqun Wu, M.Sc.
born on 12.06.1987, in Anhui, China

Reviewers:
Prof. Dr.-Ing. Axel Hahn
Prof. Dr. rer. nat. Thomas Brinkhoff

Date of disputation:
15.05.2020

Acknowledgements

Many people supported me through my time at Oldenburg to accomplish this dissertation. First of all, I would like to acknowledge my supervisor **Prof. Dr. Axel Hahn** for his constant encouragement, inspiration and guidance through my doctoral study. I also would like to thank **Prof. Dr. Thomas Brinkhoff** for his various suggestions for my research during this period. Many thanks go to the other of my thesis committee members **Prof. Dr. Jürgen Sauer** and **Dr. Marco Grawunder** as well, for the great discussion and insightful feedback.

Besides, I want to express my gratitude to my group members in the System Analysis and Optimization Group, for inspiring discussions, useful scientific and technical input of my research, helpful suggestions of my life in Oldenburg, as well as all the fun we had together in the last years. I had a great pleasure to work with you。

My special thanks go to all my friends for life, who saved me from a difficult situation caused by various unfortunate events that happened to my family at the beginning of my doctoral study. It is impossible to continue my study without your help.

Last but not least, I want to thank my husband and dearest friend **He Huang**, my mother, my parents-in-law and other family members. This dissertation would not have been possible without your accompaniment and unconditional support by all means.

Abstract

This thesis focuses on facilitating the development process of software programs that produce data representing simulated environments in simulation scenarios. Such a program is a component of a more complex simulation. It provides controlled stimuli that have influences on the system of interest component in the more complex simulation. Its functionalities depend on the overall simulation goals of the more complex simulations and the required input of the system of interest models in the simulations.

Developing such a component often involves multiple roles. The simulated environments produced by this component should exhibit spatio-temporal heterogeneity that matches the simulation scenario, which should be determined by the component users, i.e., the system of interest modelers. The users view and describe the expected simulated environments in their scenarios from a human observer view based on common sense. Such descriptions have fewer details than implementable software models and mix information about different software artifacts. The realization of this component requires the knowledge of spatial data structures and operations in software, while the users may not necessarily have them. Thus, the component may need to be implemented by other experts. Besides, modelers who provide the computational models of environmental phenomena may not be experts in software engineering either. They often express phenomena in mathematic formations. The different perspectives of the various involved roles in development bring huge challenges to the development process. Developing such a component requires much effort, and the solutions remain case-based.

The solution proposed by this thesis to overcome the above-explained challenges is a language-driven framework. The components of this framework are anchored by a domain-specific executable description language named Simulated Environment Description Language (SEDL). First, a metamodel is specified to allow the users of the components under development to specify the simulated environments they require in their simulation scenarios. This metamodel is grounded on common conceptualizations of the spatial information theory. It serves as the abstract syntax of SEDL, which provides intuitive vocabularies to describe the relevant characteristics of environmental phenomena, as well as what types of spatio-temporal changes they may exhibit during simulations at the cognitive level. A description in this language corresponds to a human-oriented Computation Independent Model (CIM) of the simulated environment in a high-level functional simulation scenario.

Then, the SEDL model is mapped to following three system-oriented metamodels: the Configuration Schema Description Profile which expresses the parameters that the component users want to be able to modify to set some specific environmental conditions for an execution; the Simulated Environment Structure Profile which expresses the data structure model that carries information of phenomena to be computed; and the metamodel based on UML behavioral elements, which expresses computation flows that update the states of the instances of the data structure model based on a specific configuration. Mapping rules from the SEDL model to these metamodels serve as the operational semantics of SEDL, defining the output when executing an SEDL description. This output is a set of inter-related models described by these metamodels, which represents a component for simulated environment generation as Platform Independent Models (PIMs). A model set transformed from an SEDL description follows the structure of a light-weighted configuration language. Its implementation can consume a configuration instance to produce a simulated environment during a simulation run.

Language metamodels in this thesis are defined by modeling standards of Model-Driven Architecture (MDA) to remain implementation-independent. Based on it, the proposed framework is designed, which includes the framework architecture that integrates the specified language metamodels and a guide of the development process with this framework. A full implementation of this framework supports semi-automatic transformation from an intuitive requirement description of simulated environment to software skeletons, with only application-specific functions to be filled in or invoked. It enables rapid incremental prototyping development. An EMF-based implementation of the framework is provided to demonstrate the usage of the framework with use cases.

The proposed framework contributes to the development of simulated environment components from the following aspects. First, it enables the participation of component users in the development processes. They can write executable SEDL descriptions of their required environmental conditions in simulation scenarios at the cognitive level. Second, it facilitates communication among different roles involved in development with formally expressed models. Third, it assists developers with automatic generations of software models from cognitive level requirement descriptions. Fourth, it preserves functional requirements in the development process and ensures intuitive user interfaces in products through well-defined metamodels and transformation chains.

Zusammenfassung

Diese Arbeit konzentriert sich auf die Erleichterung des Entwicklungsprozesses von Softwareprogrammen, die Daten produzieren, die simulierte Umgebungen in Simulationsszenarien darstellen. Ein solches Programm ist oft ein Bestandteil komplexerer Simulationen. Es stellt kontrollierte Stimuli zur Verfügung, die Einfluss auf das System haben, das in den komplexeren Simulationen von Interesse ist. Seine Funktionalität hängt von den übergeordneten Simulationszielen der komplexeren Simulationen und dem erforderlichen Input des Zielsystems in den Simulationen ab.

Die Entwicklung einer solchen Komponente umfasst oft mehrere Rollen. Die von dieser Komponente erzeugten simulierten Umgebungen sollten eine räumlich-zeitliche Heterogenität aufweisen, die dem Simulationsszenario entspricht, das von den Benutzern der Komponente, d.h. dem Modellierer des Zielsystems, bestimmt werden sollte. Die Benutzer betrachten und beschreiben die erwarteten simulierten Umgebungen in ihren Szenarien aus der Sicht eines menschlichen Beobachters auf Grundlage des gesunden Menschenverstands. Solche Beschreibungen haben weniger Details als implementierbare Softwaremodelle und vermischen Informationen über verschiedene Software-Artefakte. Die Realisierung dieser Komponente erfordert die Wissen von Raumdatenstrukturen und -operationen in der Softwareentwicklung über die die Benutzer nicht immer verfügen. Daher muss die Komponente von anderen Experten implementiert werden. Außerdem sind Modellierer, die Berechnungsmodelle für bestimmte Umweltphänomene bereitstellen, möglicherweise auch keine Experten im Software-Engineering. Sie formulieren die Phänomene in mathematischen Formeln. Die unterschiedlichen Perspektiven der verschiedenen beteiligten Rollen in der Entwicklung bringen große Herausforderungen für den Entwicklungsprozess mit sich. Die Entwicklung einer solchen Komponente erfordert viel Aufwand, und die Lösungen bleiben fallbezogen.

Die in dieser Arbeit vorgeschlagene Lösung zur Bewältigung der oben beschriebenen Herausforderungen ist ein sprachgesteuertes Framework. Die Komponenten dieses Frameworks werden durch eine domänenspezifische ausführbare Beschreibungssprache namens Simulated Environment Description Language (SEDL) verankert. Zunächst wird ein Metamodell spezifiziert, das es den Benutzern der zu entwickelnden Komponenten ermöglicht, die simulierten Umgebungen zu spezifizieren, die sie in ihren Simulationsszenarien erwarten. Dieses Metamodell basiert auf gemeinsamen räumlichen Konzeptualisierungen der Theorie der räumlichen Information. Es dient als abstrakte Syntax von SEDL, die das intuitive Vokabulare zur Beschreibung der relevanten Merkmale von Umweltphänomenen sowie der Arten von raum-zeitlichen Veränderungen, die diese während der Simulationen auf kognitiver Ebene aufweisen können, bereitstellt. Eine Beschreibung in dieser Sprache entspricht einem menschorientierten Computation Independent Model (CIM) der simulierten Umgebung in einem high-level funktionalen Simulationsszenario.

Dann wird das SEDL-Modell in drei Aspekten auf systemorientierte Metamodelle abgebildet: erstens ein Beschreibungsprofil für das Konfigurationsschema, das die Parameter ausdrückt, die die Komponentenbenutzer ändern können wollen, um einige spezifische Umgebungsbedingungen für eine Ausführung festzulegen; zweitens ein Profil der simulierten Umgebungsstruktur, das die Datenstruktur ausdrückt, die Informationen über zu berechnende Phänomene enthält; und Metamodelle, die Berechnungsflüsse ausdrücken, die die Zustände der Datenstrukturmodellobjekte auf der Grundlage einer spezifischen Konfiguration aktualisieren. Abbildungsregeln vom SEDL-Sprachmodell auf diese Metamodelle dienen als operative Semantik von SEDL, die die Ausgabe bei der Ausführung einer SEDL-Beschreibung definieren. Diese Ausgabe ist ein Satz von miteinander verbundenen Modellen, die durch diese Metamodelle beschrieben werden, die eine Komponente für die Generierung von simulierten Umgebungen als Platform Independent Models (PIMs) darstellen. Ein aus einer SEDL-Beschreibung transformierter Modellsatz folgt der Struktur einer einfachen Konfigurationssprache. Seine Implementierung kann eine Konfiguration zur Erzeugung einer simulierten Umgebung ausführen.

Die Sprachmetamodelle in dieser Arbeit werden durch Modellierungsstandards der Model-Driven Architecture (MDA) definiert, um eine Unabhängigkeit von der Implementierung zu gewährleisten. Darauf aufbauend wird das vorgeschlagene Framework entworfen, das die Framework-Architektur umfasst, welche die spezifizierten Sprachmetamodelle integriert, und einen Leitfaden für den Entwicklungsprozess enthält. Eine vollständige Implementierung dieses Frameworks unterstützt die automatische Transformation von einer intuitiven Anforderungsbeschreibung der simulierten Umgebung in Software-Skelette, wobei nur anwendungsspezifische Berechnungsfunktionen ausgefüllt oder aufgerufen werden müssen. Sie ermöglicht eine schnelle inkrementelle Prototypentwicklung. Eine EMF (Eclipse Modeling Framework)-basierte Implementierung des Frameworks wird bereitgestellt, um die Verwendung des Frameworks mit Anwendungsfällen zu demonstrieren.

Das vorgestellte Framework trägt zur Entwicklung von simulierten Umweltkomponenten unter folgenden Aspekten bei. 1) Es ermöglicht die Beteiligung der Nutzer der Komponenten an den Entwicklungsprozessen. Sie können ausführbare SEDL-Beschreibungen ihrer erforderlichen Umgebungsbedingungen in Simulationsszenarien auf der kognitiven Ebene schreiben. 2) Es erleichtert die Kommunikation zwischen verschiedenen Rollen mit formal ausgedrückten Modellen. 3) Es unterstützt Entwickler mit der automatischen Generierung von Softwaremodellen aus Anforderungsbeschreibungen auf kognitiver Ebene. 4) Es bewahrt funktionale Anforderungen im Entwicklungsprozess und gewährleistet eine intuitive Benutzerschnittstelle in den Produkten durch wohldefinierte Metamodelle und Transformationsketten.

Table of Contents

Abstract i
Zusammenfassung iii
Table of Contents v
Abbreviations viii
List of Figures ix
List of Tables x
1 Introduction 1
1.1 Motivation 1
1.2 Challenges 2
1.3 Research Objectives 4
1.4 Overview of Chapters 5
2 Research Foundations 7
2.1 Model-Driven Development (MDD) 7
2.1.1 Model-Driven Architecture (MDA) 7
2.1.2 Standards for MDD 8
2.1.3 Multilevel Metamodeling 9
2.2 Computer Languages 10
2.2.1 Elements of a Computer Language 10
2.2.2 Domain-Specific Languages 12
2.2.3 Language Workbenches 13
3 Related Works 15
3.1 CIM-PIM Transformations 15
3.1.1 CIM-PIM Transformations in Specific Domains 15
3.1.2 Analytic Approaches for CIM-PIM Transformations 16
3.1.3 Automatable CIM-PIM Transformations 16
3.1.4 Summary and Relation to This Thesis 18
3.1.4.1 Evaluation 18
3.1.4.2 Lessons Learned and Missing Points 21
3.2 Spatial Conceptualization and Data Representation 21
3.2.1 Spatial Conceptualizations 21
3.2.2 Representation of Spatial Data 22
3.2.3 Summary and Relation to This Thesis 23
3.3 Simulated Environments in Software 24
3.3.1 Styles of Simulated Environment Components 24
3.3.2 Summary and Relation to This Thesis 26
4 Language-Driven Development Framework of Simulated Environment Components 28
4.1 Executable Meta Languages 28
4.1.1 Simulated Environment Description Language (SEDL) 29
4.1.2 Configuration Schema Description Profile 30
4.1.3 Simulated Environment Structure Profile 31
4.1.4 Metamodel of Environment Computation 31
4.2 Build Software Applications as Computer Languages 32
4.2.1 Development Framework as IDE of SEDL 32
4.2.1.1 SEDL Core Language Model 33
4.2.1.2 Basic SEDL Tooling 34
4.2.1.3 Platform-Specific Mapping Layer 35
4.2.1.4 SEDL Extension Layer 35
4.2.2 Simulated Environment Specification by Configuration Language 36
4.3 Development Process with the Proposed Framework 37
4.3.1 System Analysis 37
4.3.2 Software Design 38
4.3.3 Implementation 39

4.3.4 Development Activity Flow with Iterations 40
5 Simulated Environment Description Language 42
5.1 SEDL Language Model 42
5.1.1 Conceptual Modeling Principles 42
5.1.1.1 Level of Modeling 42
5.1.1.2 Perspective of Modeling 43
5.1.2 Abstract Syntax 44
5.1.3 Descriptive Semantics 44
5.1.3.1 DescriptionItem and Configurable 44
5.1.3.2 ConfigurableParameter 44
5.1.3.3 Composition of Environment 45
5.1.3.4 Expression of Exhibited Changes 47
5.1.3.5 Conceptual Approximation in the Formulation of Change Expressions 48
5.1.3.6 Chang Types of an Individuality 49
5.1.3.7 Alternative Change Modes 53
5.1.3.8 Characteristic Variation among Instances of an EnvironmentalPhenomenon 53
5.1.3.9 ExecutionRoutine 57
5.2 PIM Layer Metamodels 58
5.2.1 Configuration Schema Description Profile 58
5.2.1.1 Summary 58
5.2.1.2 ConfigSchema 59
5.2.1.3 ConfigItem and Primitive Types 59
5.2.1.4 ConfigComponent Types 60
5.2.1.5 SubComponent and ConfigOption Associations 60
5.2.1.6 Usage Outside the Framework 61
5.2.2 Simulated Environment Structure Profile 61
5.2.2.1 Summary 61
5.2.2.2 Utility Datatypes 62
5.2.2.3 Runtime Simulated Feature Types 62
5.2.2.4 Single-Valued Feature Types 64
5.2.2.5 Collective Feature Types 65
5.2.2.6 Subtypes of CollectiveFeatureType 66
5.2.2.7 SpatialFunction 71
5.2.2.8 Requested Snapshots 72
5.2.3 Metamodel of Environment Computation 74
5.2.3.1 Summary 74
5.2.3.2 Two Views of Behavioral Models 75
5.2.3.3 Life Cycle Control of Simulated Features 76
5.3 Transformations of SEDL Descriptions 77
5.3.1 CIM-PIM Transformation Process 77
5.3.2 Description2Config 78
5.3.3 Description2Structure 78
5.3.4 Description2Computation 79
5.3.5 Map Description of Spatial Heterogeneity to Design Models 81
5.3.6 Generate Execution Routine 82
6 Prototypical Implementation 85
6.1 Eclipse Modeling Framework 85
6.2 EMFText 86
6.3 OCLInEcore 87
6.4 ATL 88
6.5 Acceleo 89
6.6 Other Involved Tools 91
7 Use Cases 92
7.1 Focus of the Use Cases 92
7.2 Use Case 1: Sea Environment for the Path Assessment 92
7.2.1 SEDL Description 93

7.2.2 Transformed Artifacts 94
7.2.3 Summary 98
7.3 Use Case 2: Storm Avoidance Strategy Evaluation 99
7.3.1 SEDL Description 100
7.3.2 Transformed Artifacts 100
7.3.3 Summary 103
8 Discussions 105
8.1 Contributions and Objective Fulfillment 105
8.2 Limitation of Model Transformations in Development 106
8.3 Visions of SEDL Extension 108
8.3.1 Environmental Phenomenon as Network 108
8.3.2 Influence Between Phenomenon Types 108
8.3.3 Using Spatial Predicates at the System Analysis Phase 109
8.4 Reuse of Implemented Environmental Phenomena 110
References 112
Appendix A: CIM-PIM Transformations 118
Appendix B: SEDL Descriptions for Use Cases 128

Abbreviations

AD	Activity Diagram
ARIS	Architecture of Integrated Information Systems
ATL	Atlas Transformation Language
BPD	Business Process Diagram
BPM	Business Process Model
BPMN	Business Process Modeling Notation
CD	Class Diagram
CIM	Computation Independent Model
CSS	Cascading Style Sheets
DCD	Domain Class Diagram
DCM	Domain Class Model
DFD	Data Flow Diagram
DSL	Domain Specific Language
DW	Data Warehouse
EBNF	Extended Backus–Naur form
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
EPB	Elementary Business Process
GUI	Graphic User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IFML	Interaction Flow Modeling Language
ISO	International Organization for Standardization
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MMT	Model to Model Transformation
MOF	Meta-Object Facility
MPS	Meta Programming System
MTL	Model to Text Language
NetCDF	Network Common Data Form
OCL	Object Constraint Language
OMG	Object Management Group
OGC	Open Geospatial Consortium
PIM	Platform Independent Model
PSM	Platform Specific Model
PSI	Platform Specific Implementation
QVT	Query/View/Transformation
SDSED	System's External Behavior
SDSIB	System's Internal Behavior
SEDL	Simulated Environment Description Language
SoaML	Sercive-Oriented Architecture Modeling Language
SOD-M	Service-Oriented Development Method
SSD	System Sequence Diagram
TIN	Triangular Irregular Networks)
UC	Use Case
UCD	Use Case Diagram
UML	Unified Modeling Language
UWE	UML-based Web Engineering
WebRE	Web Requirements Engineering
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

List of Figures

Figure 1.1: Chapter Overview........ 6
Figure 2.1: Semantic Areas of UML........ 8
Figure 2.2: Relationships between QVT metamodels........ 9
Figure 3.1: Taxonomy of CIM-PIM Transformation Components........ 19
Figure 3.2: Styles to Provide Simulated Environments........ 26
Figure 4.1: DSL Models in the Proposed Framework........ 28
Figure 4.2: Logical Components of the Proposed Framework........ 33
Figure 4.3: Model Related Artifacts Correspondence to MDA and Metamodeling Levels. 34
Figure 4.4: Developed Components as Syntax-Directed Applications. 36
Figure 4.5: Development Process with Processed Framework. 37
Figure 4.6: Main Activities in the Development Process........ 41
Figure 5.1: Forms of Simulated Environments at Different Development Phases........ 42
Figure 5.2: SEDL Abstract Syntax. 43
Figure 5.3: Abstract Syntax of Individuality Change Types........ 52
Figure 5.4: Abstract Syntax of AlternativeMode........ 53
Figure 5.5: The Computation Chain of Characteristic Variations........ 55
Figure 5.6: Configuration Schema Description Profile........ 58
Figure 5.7: Conceptual Links of Entities Related to a Simulation. 63
Figure 5.8: Stereotypes of Runtime Simulated Feature Types........ 67
Figure 5.9: Geometry Illustration of CollectiveFeatureTypes........ 68
Figure 5.10: Snapshot Types........ 72
Figure 5.11: Define the getUnit() Operation for the CollectiveFeatureUnit Stereotype........ 74
Figure 5.12: Elements in Instance Models of Behavioral Metamodels........ 75
Figure 5.13: CIM-PIM Transformation Process. 77
Figure 6.1: EClasses in the SEDL Ecore Model. 85
Figure 6.2: SpatialFunction(Left) and Created EClass (Right) in the UML Editor........ 86
Figure 6.3: Textual Editor for SEDL Description........ 87
Figure 6.4: OCL Constraints for Change Description........ 87
Figure 6.5: Validation Method Example........ 88
Figure 6.6: Warning Message for OCL Constraint Violation. 88
Figure 6.7: ATL Rule Example. 89
Figure 6.8: Acceleo Code Generation Example........ 91
Figure 7.1: PIM-Layer Configuration Model in Use Case 1........ 94
Figure 7.2: PIM-Layer Data Model in Use Case 1........ 94
Figure 7.3: PSM-Layer Java Classes of Data Model in Use Case 1........ 95
Figure 7.4: PSM-Layer Java Classes of Compute Model in Use Case 1........ 95
Figure 7.5: Constructor of the "ComputeShip" Java Class. 96
Figure 7.6: Code for the Initialization of Ships in the Background Traffic........ 96
Figure 7.7: Methods for Computation Units of a Ship in the Background Traffic........ 97
Figure 7.8: Method for Compute the Location of a Ship in the Background Traffic. 97
Figure 7.9: Code for Computing New States of a Ship in the Background Traffic........ 98
Figure 7.10: Transformations of Use Case 1........ 98
Figure 7.11: PIM-Layer Configuration Model and Data Model in Use Case 2........ 101
Figure 7.12: PSM-Layer Java Classes of Configuration and Data Model in Use Case 2........ 101
Figure 7.13: PIM-Layer Java Classes of Computation Model in Use Case 2. 102
Figure 7.14: Transformations of Use Case 2........ 104
Figure 8.1: An influencedBy Link in an SEDL Description. 109
Figure 8.2: Conceptual Architecture for Implementation Reuse........ 110

List of Tables

Table 3.1: Summary of CIM-PIM Transformation Researches. 20
Table 5.1: Available Options of ParameterType. 45
Table 5.2: Possible Number of Dimensions of a SimulatedEnvironment. 45
Table 5.3: Types of Variable and Variant. 48
Table 5.4: Individuality Level Change Types. 50
Table 5.5: No-locational Characteristic Level Change Types. 51
Table 5.6: Express Required Characteristic Variation and Alterable Conditions in SEDL. 56
Table 5.7: Available Options of ExecutionMode. 57
Table 5.8: Available Options of OutputRange. 57
Table 5.9: Stereotypes for the Simulated Feature Life Cycle Control 77
Table 5.10: Apply a Stereotype to the Data Structure Class. 79
Table 5.11: Overall Steps of Description2Computation. 80
Table 5.12: Generation of the Dependency Graph for a SpatialIndividuality. 81
Table 5.13: Generation of a SpatialFunction. 82
Table 7.1: Initial Requirements of Environmental Phenomena in Use Case 1 93
Table 7.2: Initial Requirements of Environmental Phenomena in Use Case 2 100

1 Introduction

1.1 Motivation

Computer-aided simulations are widely used to explore behaviors of real-world phenomena, to validate designs of new artificial systems, and to verify hypothetical processes. Research targets (i.e., systems of interest) in simulations are reduced to models representing their essential characteristics and functions based on mathematical abstractions. They may be natural phenomena, technical systems or social entities. An execution process of a simulation imitates behaviors of the modeled real-world system along the timeline.[1], [2]

The system of interest model in a computer simulation often does not work alone. While its real-world counterpart is influenced by other phenomena in its situated environments, behaviors of the real-world system in reaction to such influences should be captured by the system of interest model. For computing the modeled reactive behaviors during simulation executions, the system of interest model needs digital input which are abstractions of the situated environment of the real-world system of interest, i.e., the simulated environment[3].

Thus, a simulated environment component should generate necessary inputs for the system of interest model it serves. This component itself is also a simulation application whose model should capture the real-world phenomena that constrain and alter the behaviors of the system of interest. It produces data representing simulated environments that act as controlled stimuli for running the system of interest model[3]. Different from the system of interest model whose behaviors are to be explored, simulated environments should be generated in a desired way. The composition of these simulated environments depends on the captured environmental influences in the system of interest model. The more kinds of environmental influences a system of interest model considers, the more complicated simulated environment it requires. Behaviors of the environmental phenomena should match the expected conditions of simulation scenarios and often should be alterable to some extent, so the behaviors of the system of interest under different environmental conditions can be computed, compared and analyzed. Even for simulations about the same system, simulated environments may be diverse due to different investigation purposes, and some of them could be very complex. Sources and forms of models that produce these phenomena in simulations inherit this diversity and complexity.

The simulations of maritime systems such as vessel simulations offer a good example of the diversity among simulated environments. Maritime systems are usually costly, and their activities are often safety-critical. Simulations are an appropriate way of testing and analyzing these systems before real operations are performed[4]. The maritime environment is a physical world that holds complex phenomena varying over space and time. Crucial influences from the environments cannot be simply ignored by a vessel model in safety-critical simulations. In many cases, the ship model should be tested with simulated environments that resemble real-world situations.

Practical vessel models often need to consider the influence of the tidal, sea current and wind.[5] For a simulation that identifies parameters of a mathematic model for ship dynamics, the simulation often starts with setting the vessel model in idealized environmental situations. In this case, the vessel model is assumed to be set in open, still water. It is fed with some input based on simplified deterministic or statistic patterns, such as no tide, constant force from the sea current and random values representing turbulence of wind. The simulated environment component needs to enclose functions to compute this value and fed them to the ship model.

Before the identified vessel model is used for further purposes, its quality should be evaluated. A widely used strategy is to execute a set of simulation runs using this model with simulated environments from available historical records of environmental information. Outcomes of these executions are compared with the recorded vessel behaviors, e.g., the trajectories, to check if the model can successfully reproduce the behaviors of the real vessel. In this case, the component that provides the simulated environments may need to have the functionalities to access the observation data source, acquire the necessary context and send the data to the ship model in a suitable form.

The evaluated model can be used for simulations that help analyze its modeled vessel. For such simulations, the vessel model needs to be set in simulated environments that imitate desired real-world situations. For instance, to assess the suitability of a planned path in different seasons, typical spatio-temporal patterns of tidal, sea current and wind in corresponding seasons within the area that the voyage should take place may need to be provided. To test the robustness of autonomous ship controllers, simulated environments should imitate evolution patterns of influential phenomena during various extreme conditions, e.g., the evolution patterns of wind force during a storm. To analyse vessel behaviors near a port, the simulated environment should also include infrastructures of the port and its dynamics, as well as other nearby moving systems. Simulated environment components in these cases need their own models of computation that are well-studied and controllable to produce desired patterns of environmental phenomena required by corresponding scenarios.

With more complex environmental situations being considered, the complexity of components that produce simulated environments increase. Models for computing realistic patterns of spatio-temporal varied phenomena such as wind could require comprehensive knowledge to cope with and are different from each other. For a specific simulation goal, the system of interest model may need environmental phenomena being simulated at different levels of complexity. This leads to huge challenges in developing simulated environment components, especially when the simulations are spatial-aware. These challenges gained the author's attention and motivated the research of this thesis. They are summarized in the next section based on which the research objectives of this thesis are identified.

1.2 Challenges

Various challenges emerge throughout the whole development process of a simulated environment component as identified in this section. The research objectives of this thesis listed in the next section are derived with the aim of overcoming these challenges.

Challenge 1: a huge amount of communication efforts is needed among the involved roles with different expertise during the development of a computer simulation with multiple components.

This problem appears since the system analysis starts when functional simulation scenarios are determined. These scenarios reflect the high-level functional requirements of the simulation application under development, in which behaviors of environmental phenomena that are influential to the system of interest form an important part. The component producing simulated environments for this simulation should have functionalities to simulate these phenomena. The produced context of this component is not freely decided by environmental modelers but depends on the needs of the system of interest component in this simulation.

In a complex simulation involving multiple models of real-world phenomena, the system of interest modelers and the modelers of phenomena in the simulated environment are often different experts. The system of interest modelers have the knowledge about which environmental phenomena and which characteristics of these phenomena should be provided for running their models, as well as the evolvement patterns that these phenomena should undergo in the simulation to fulfill their simulation goals. However, they do not necessarily have the expertise to produce these contexts digitally. Thus, the simulated environments expected by the system of interest modelers have to be communicated through functional scenarios to the modelers of the simulated environment. However, the involved roles hold different expertise and use different notations in communications that may lead to a difficult, lengthy and error-prone analysis phase.

Challenge 2: a gap exists between the description of the required simulated environments in high-level functional scenarios and application models of a component that produced the digital representation of such environments.

The component design based on the simulated environment in functional scenarios from the analysis phase involves a view switch. The required simulated environments in these scenarios are expressed from a human observer view in terms of their composition and desired variations over space and time at the cognitive level. The underlying concepts of these descriptions are about phenomena in the spatial world.

In contrast, the models of the simulated environment component focus on artifacts of the computer application such as data structures, operations and computation flows.

Further, the functional scenarios have fewer details compared to an implementable component model that should be able to provide digital imitations of the described environments. Figuring out missing details could be hard work. Moreover, a piece of the analysis-phase description of simulated environments mixes information that goes to different software artifacts. No explicit, one-to-one mapping exists from concepts about environments to software artifacts. Some of them may be mapped to data structures and some others are only reflected in execution processes. For instance, an functional simulation scenario may state that a storm should appear in the simulated environment whose influence area moves related to the ground. Data structures to hold the area of influence will be found in the structural model of the component under development (e.g., as classes in an object-oriented language). However, the "move" only exhibits during execution, which requires some operation in the component to manage the execution process.

Challenge 3: computational models of environmental phenomena often do not align with application models simulated of environment components.

The component producing simulated environments in a bigger simulation is a simulation application by itself. It needs to enclose its own abstractions of real-world phenomena. These abstractions may range from connections and queries to recorded data to complex mathematic formulas from which digital values of presented phenomena can be computed. They play the role of computational models of this component. The purpose of executing such simulations is to reproduce environmental phenomena with desired patterns that match simulation scenarios.

However, computational models needed by such components may be developed from other standalone research with a different scientific or industrial purpose than the bigger simulation they serve. Even though such a model may have modeled all relevant aspects of a phenomenon needed by the simulation scenarios, it is likely formalized and encoded differently than the form needed by the simulated environment component. e.g., in some mathematic formations or as a standalone program whose outputs are some statistic descriptors about its simulated phenomenon instances.

To adapt such a computational model to a simulated environment component, it needs to be turned to a functional form that can make sample draws from this model. A sample draw should include aspects relevant to the simulation scenario, e.g., a time series of wind strength in the simulated area. A computed sample is fed to the system of interest component over the execution. This form should also provide component users with simple access to some of the model parameters so that they can adjust the characteristics of the drawn samples from various executions.

A mature computational model of a real-world phenomenon type may cover much richer aspects than a simulated environment needs and have a huge set of parameters. Without expertise about this model, adapting it to a simulated environment component is very difficult, if not impossible. On the other side, experts of this model also do not necessarily master all knowledge about the system of interest component that uses this component. It brings the risk that mismatch may appear when they integrate the computational models into a simulated environment component. The resulting component may not correctly preserve the requirements of simulation scenarios, e.g., output values may not be the expected input to the system of interest component, and the exposed parameters may not control the characteristic of the computed phenomenon as expected.

Challenge 4: modelers of environmental phenomena may not be familiar with the platform that is used to implement the simulation under development, which causes difficulties in implementation.

When the computational models of a simulated environment component are developed from standalone researches, they are often encoded, implemented and tested in a different platform than the one being used in the current development. Their modelers may not be familiar with the implementation platform on which the component should be developed and be integrated into the bigger simulation it serves. Implementing the adapted models of required environmental phenomena with the chosen platform may lead to a long learning curve for them to master the underlying technical architecture and tools.

A platform for implementing simulations with multiple components is often more complex than an experimental tool for a single model. In addition to implementing the logic of the computation functions in such a complex platform, much extra work has to be spent on implementing architectural code and communication functions among model units and among components. The extra work distracts modelers from implementing the essential computation functions and further increases the difficulties in the implementation phase.

1.3 Research Objectives

Similar challenges also exist when developing computer applications in other context domains. Universal solutions can partially overcome these challenges, among which Model-Driven Development (MDD)[6] is a widely-used strategy. It provides modeling languages as communication tools and as metamodels that enable model transformations and code generations, which saves manual work. However, since context domains are potentially infinite, these universal solutions can hardly cover the domain context-related aspect.

This thesis aims at providing a domain-specific solution dedicated to overcoming the identified challenges in the development of components that produce simulated environments in spatial-aware computer simulations, with higher efficiency than domain-independent solutions based on MDD while inheriting merits of them. The scientific focus lays on bridging high-level functional requirements of simulated environment and implementable computer application models, which involves the domain context and thus cannot be addressed at the domain-independent scale. The overall research goal is formed as follows:

Build a domain-specific development framework, which integrates concepts, methods and tools to facilitate the development of simulation components that produce simulated environments for the system of interest component in spatial-aware simulations.

This goal can be achieved by answering the following three research questions (RQs in the following text) dealing with different addressed challenges in Section 1.2. Several objectives are identified for each question to break the research topic into actionable units.

RQ 1: what are the common concepts underlying simulated environments in analysis-phase functional scenarios of spatial-aware simulations, and what is the meta structure behind descriptions of these environments?

The investigation of this question leads to a domain-specific description language model that can be used to capture required simulated environments in functional scenarios in a structured way. It provides a communication tool to document and exchange requirements about the expected simulated environments at the system analysis phase, which helps to overcome Challenge 1. The following two objectives need to be achieved to build this language.

Objective 1.1: capture a description structure of simulated environments at the cognitive level. A small number of concepts used when human roles view and express simulated environments at the analysis phase in natural languages should be identified and organized in this structure. This structure should be close to the way that human observers organize their perceptions of different aspects of an environment and phenomena in this environment. Thus, it should be understandable and be easily used without particular expertise in modeling and development though little learning effort.

Objective 1.2: capture and formalize concepts to describe changes of environmental phenomena in space and time. A simulation is a dynamic process in which its simulated environment evolves in time, while phenomena in the environment may also be heterogeneous over space. Expected changes of phenomena in this simulated environment have to be described in the functional scenarios. The types of changes humans may perceive in the environment and which concepts of phenomena they are associated with, have to be captured in the description language model.

RQ 2: how the simulated environments in analysis-phase functional scenarios are captured by structures and operations of computer simulation applications?

The answer to this question contributes to overcoming Challenges 2 and 3. It essentially enables transformations from the description of simulated environments in the human observer view to crucial artifacts in the simulation component in the system design view. The following two objectives should be achieved to answer this question.

Objective 2.1: capture a computer application metamodel for simulation components that produces simulated environments for systems of interest. This metamodel should identify crucial units and artifacts that a component should have to be able to produce the possible environments described in the language from RQ1. It should be specified in an implementation-independent manner at the detail level of implementable software design models. This metamodel is also a communication tool that describes and exchanges application models of simulated environment components among different developers, which helps to overcome Challenge 1.

Objective 2.2: establish the mapping between the description model of simulated environments and the application metamodel of simulated environment components. This mapping identifies what necessary artifacts of computer applications should be added and where to locate them within the application model, when an instance of a certain concept is presented in a description of the simulated environment. Thus, developers can be assisted by the established mappings when they design and implement component models based on functional scenarios expressed at the cognitive level.

RQ 3: how to integrate the identified concepts and models into a development framework and use them to facilitate the development of a simulated environment component?

A domain-specific development framework for the overall research goals is provided by answering this question. It integrates all research outcomes from this thesis and utilizes the general MDD solutions when necessary.

Objective 3.1: establish a development framework for simulated environment components. This framework should integrate all theoretical outcomes from previous RQs. It should identify necessary components of the framework, the theoretical outcomes realized by each component, as well as functionalities that each component provides in the development. Further, it should also clarify dependencies and interactions between these components so that they can be used together. This framework should be specified independently of implementation tools but be feasible to be implemented. A realization of this framework on a specific implementation platform contributes to overcoming Challenge 4 by automating the implementation of architectural code and communication functions that can be derived at the platform-specific level.

Objective 3.2: specify a guide of the development process with the framework. Together with the established framework architecture, a development guide should be provided to clarify the usage of this framework. It should go through the software development process starting from the system analysis to the implementation of production. For each development phase, this guide denotes the involved roles and framework components, the way that the framework being used, as well as the input and the output of this phase.

1.4 Overview of Chapters

The main content of this thesis is organized into three parts as shown in Figure 1.1, starting from the next chapter and followed by a summary chapter. Each part includes two chapters with a specific focus as introduced below.

In the first part, the related works of the thesis are introduced. Chapter 2 presents the fundamental research based on which this thesis is constructed. This includes the relevant knowledge of MDD, the fundamental concepts of computer languages in general, as well as the knowledge about DSL. Chapter 3 is dedicated to reviewing the research that shall contribute to the research objectives of this thesis, to figure out their relevance to these objectives, how this thesis can benefit from them, and the remaining problems that need to be solved. The review covers the methodological aspect of transformations from Computation Independent Models (CIMs) to Platform Independent Models (PIMs), which bridges the

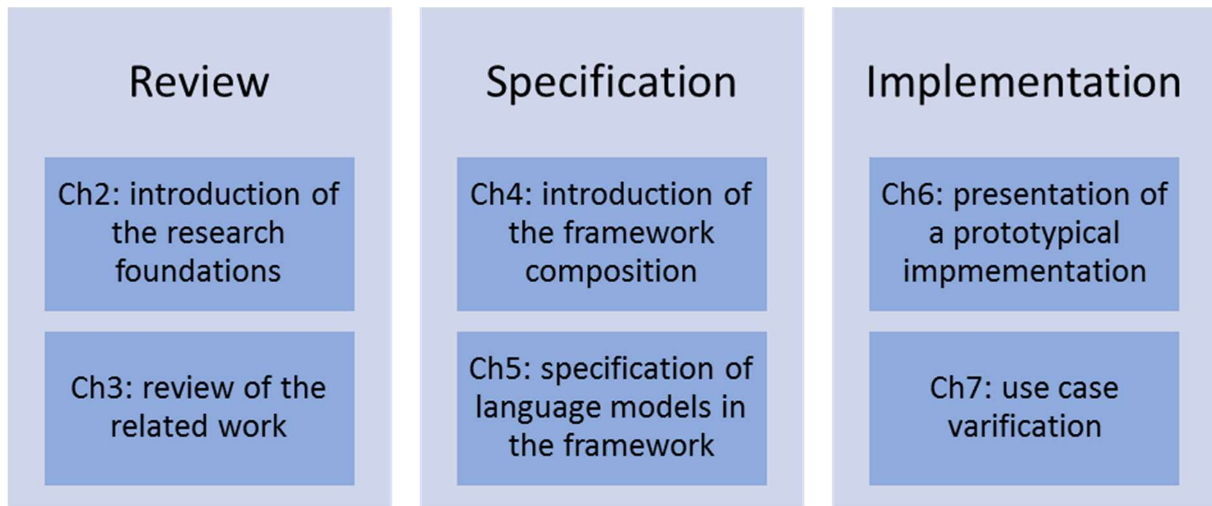

Figure 1.1: Chapter Overview.

human view and the system view in the development. It also covers the modeling work in the context domain that this thesis deals with, i.e., the spatial conceptualization and data representation, as well as the common ways to provide simulated environments in computer simulations.

The specification of the proposed framework is presented in the second part, which provides the theoretic answers to the research questions of this thesis. Chapter 4 specifies the composition of the proposed framework and the development process using the framework, which answers RQ3. The framework specification includes a set of domain-specific languages at both the CIM layer and the PIM layer, an architecture of components for a realization of the proposed framework, and a guide of development processes within the framework. It is followed by Chapter 5 that provides the detailed specifications of the language models in the framework as they form the backbone of the framework. The specification covers the CIM-layer language Simulated Environment Description Language (SEDL) that is the research outcome to achieve the objectives of RQ2, and transformation rules from descriptions in SEDL to PIM-layer component models, which are the research outcome to achieve Objective 3.2. Three PIM-layer metamodels used to describe the PIM-layer outputs are also specified in this chapter, which provide the solution to achieve Objective 3.1. They are used to express user interfaces, data structure and behaviors of the simulation components producing simulated environments, respectively.

The third part then presents a prototypical implementation to demonstrate the feasibility of realizing and using the proposed framework. Chapter 6 introduces the used tools and strategies to implement the prototype. In Chapter 7, this prototype is verified with use cases to demonstrate the framework functionalities against its specification.

Finally, Chapter 8 summarizes the contribution of this thesis, discusses reasons that cause limitations of the framework, as well as lists out open issues with suggested conceptual ideas that shall solve these issues in future work.

2 Research Foundations

This chapter introduces the research on which this thesis is built. First, the solution provided by this thesis contributes to domain-specific system developments, which involve several domain-specific modeling languages. These languages are defined by models and are used to express models. Involved models in the solution are coordinated based on the Model-Driven Development (MDD) paradigm[6], [7]. To support readers' understanding, Section 2.1 briefly introduces relevant knowledge of MDD. Second, this thesis frequently refers to components of executable computer languages when specifying language models and the framework architecture in its solution. Fundamental concepts of computer languages that are necessary to understand this thesis are introduced in Section 2.2.

2.1 Model-Driven Development (MDD)

Model-Driven Development is a paradigm that develops systems based on a set of models. This section introduces the knowledge of MDD that is relevant to this thesis. It gives an overview of the layers of Model-Driven Architecture (MDA)[8], the modeling standards from Object Management Group (OMG)[1], as well as the concepts of the multilevel metamodeling.

2.1.1 Model-Driven Architecture (MDA)

MDA is a general-purpose architecture defined by OMG for model-driven development. A key strategy of MDA is the model transformation that produces models from other models through a transformation pattern. Transformations can generate models from one presentation to another, or from one abstraction layer to another. By automating the transformation paths from high-level models in the view of stakeholders to functional systems, time and cost for developing a system are reduced while the consistency of expectations among different involved roles in the development increase. MDA identifies several architectural layers to locate models with different levels of abstraction[8]:

- **Computation Independent Model (CIM):** a CIM is also referred to as a domain model. It is described with vocabularies that are familiar to experts of the subject which the system deals with. This model describes "real things" in the world. It is a functional description of what a system is expected to do or to resemble from the user perspective, without mentioning the technical aspects of a system. In software development, CIMs are often modeled at the system analysis phase to gather the requirements of systems under development.
- **Platform Independent Model (PIM):** a PIM is also referred to as a logical system model. It is a design model that expresses the structures and behaviors of the system independently from implementation platforms. In software development, a platform is a set of resources that are used to realize and execute the system application in specific programming languages or regulations. Thus, a PIM has a sufficient level of independence to be realized on multiple platforms.
- **Platform Specific Model (PSM):** a PSM refines a PIM with technical details required to realize the system on a specific platform. Since a platform can exist at many layers, the PIM and PSM are relative. To a related PIM, a PSM is any model that is more technology-specific than it. For instance, a software design model described by XML can be mapped to different implementation languages such as Java or C. To models that are specific to these languages, this XML model is platform-independent. The key distinction from the CIM is that both PIMs and PSMs are system-oriented. Transformations between PIMs and PSMs do not add conceptual content but rather technical details.

Final productions of executable systems are implemented based on PSMs. In software development, they are executable code programs. This thesis refers to such a system as a **Platform Specific Implementation (PSI)** for a consistent naming structure.

[1] https://www.omg.org/

2.1.2 Standards for MDD

MDA is an architectural approach built on a set of OMG standard specifications. These specifications provide languages that support expressing models which are cross-domain or specific to a domain, as well as specifying transformations from models to other models. Modeling frameworks and toolkits have been developed based on these specifications. Relevant tools used in this thesis are introduced in the implementation chapter (Chapter 6).

- **Meta-Object Facility (MOF)**[9] provides core principles in MDA. It specifies a platform-independent metadata management framework. MOF is the foundation of metamodel definition in MDA. MOF reuses structural symbols from the Unified Modeling Language (UML) to describe metamodels. A MOF 2.x metamodel is a valid UML 2.x model since UML 2.4.1**.** MOF is closely related to the concept of multilevel metamodeling which is important to this thesis and is introduced in the next subsection.
- **Unified Modeling Language (UML)** [10] is a general-purpose modeling language adopted by OMG. The UML specification defines how UML models should be constructed including modeling concepts, rules to combine the modeling concepts as well as notations to represent them. The data structure of UML models, i.e., the UML abstract syntax (see Subsection 2.2.1), is defined by the UML metamodel. This metamodel uses a subset of UML constructs identified by MOF (see Subsection 2.1.3).

With the release of the major revision UML 2.x, the language unit[11] is introduced to partition UML into a modular structure. A UML language unit is a set of tightly coupled modeling constructs focusing on a specific aspect of systems, which can describe models in a particular type of diagrams. In UML version 2.5.1, these units are referred to as semantic areas as shown in Figure 2.1. They are divided into two categories, i.e., the structural semantics that define the meaning of structural elements and the behavioral semantics that define the meaning of behavioral elements. Specific structural constructs for modeling are based on a common base of fundamental concepts. The common behavioral semantics are then built on the structural constructs, which provide a framework to model behaviors. Actions are the fundamental units of behaviors. They can be used in higher-level behavior modeling formalisms such as Activities. Besides, UML also provides supplemental modeling constructs to describes use cases, deployments and information flows.

Supplemental Modeling	Use Cases	Deployments	Information Flows
Behavioral Modeling	State Machines	Activities	Interactions
	Actions		
	Common Behavior		
Structural Modeling	Values	Classifiers	Packages
	Common Structure		

Figure 2.1: Semantic Areas of UML. [10]

Profiles are introduced in UML2.x as a lightweight standard mechanism to extend the UML.[11] They are defined through Stereotypes which are specialized modeling elements in UML confined by Tag definitions and Constraints. It is not possible to remove existing Constraints from a Stereotype in a Profile which applies to the model element it extends. Rather, Profiles are intended to adapt the existing UML model to a specific domain with additional Constraints.

- **Object Constraints Language (OCL)**[12] is a declarative language which is used to describe rules applying to UML models. It becomes a part of the UML specification in UML 2.x[11]. This

language is used to express additional constraints on UML models which cannot be expressed by the UML syntax. OCL is a pure specification language that is side-effect-free. Thus, evaluating OCL expressions cannot alter the state of a system. These expressions are usually invariant conditions that the modeled system must hold or queries on a UML model. Same as UML, OCL is independent of programming languages. It is of great importance to MDD since many model transformation languages are developed based on it.

- **XML Metadata Interchange (XMI)**[13] is an OMG standard to support MOF. It is an interchange format for exchanging metadata information using Extensible Markup Language (XML)[14]. XMI is often used to exchange UML models. Although, this format can be applied to all metadata whose metamodels are described in MOF. It defines representations of objects via XML elements and attributes, as well as the standard mechanisms to link objects. XMI documents uses XML Schema for validation.
- **QVT (Query/View/Transformation)**[15] is an MOF Specification defining transformation languages. It includes three related languages as shown in Figure 2.2 which is redrawn from the QVT specification. Among these languages, QVT-Relations and QVT-Core are declarative languages with the same semantics at different abstraction levels. The former is a user-friendly language that supports complex object pattern matching. Its semantics can be mapped to QVT-Core. The latter is a small language that can be directly implemented. It supports pattern matching over a set of variables and its trace model must be explicitly defined. These two languages enable black-box implementation via MOF operations. The third language QVT-Operational extends OCL to provide means to define imperative mappings. It brings QVT the ability to express procedural transformations.

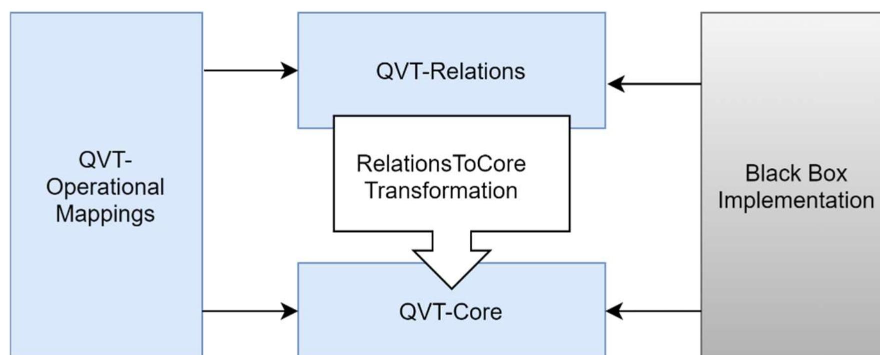

Figure 2.2: Relationships between QVT metamodels. [15]

2.1.3 Multilevel Metamodeling

An angle to view the levels of model abstraction other than the MDA layers introduced in Subsection 2.1.1 is to determine layers of models based on the linguistic “instance-of” relationship [16]. It emphasizes the strict metamodeling[17] paradigm: “if a model A is an instance-of another model B, then every element of A is an instance-of some element of B”[18]. In this case, the model B is a metamodel of the model A whereas the model A is said to conform to the model B. The instantiation of a metamodel results in an instance model of it at one layer lower. The instance model may be further instantiated. This leads to a multilevel modeling structure. A model in an intermediate layer is an instance model of some model at one layer higher, and the metamodel of some models at one layer lower. For instance, the class “Element” in the UML metamodel is a metaclass to describe UML models, but an instance of the MOF metamodel. Models that cannot be further instantiated are usually considered as a thing in the real world[6]. They can range from physical entities on the earth to runtime objects in software applications.

MOF forms such a modeling structure which is referred to in various OMG specifications as having four modeling layers from M3 to M0. The M3 meta-meta layer locates the metamodel of MOF used for defining metamodels. It is a self-describing model that conforms to itself. The metamodel of UML locates at its M2 meta layer. M2 metamodels describe models at the M1 layer which can be further instantiated as M0 real-world objects. However, the current MOF specification has clarified the ability of MOF to handle any number of metamodel layers recursively following the “instance-of” relationships by using “Classifier” and “Instance” concepts in UML, which may introduce more than one meta layer

in practice.[9] For instance, defined UML Profiles in a domain-specific modeling solution shall be located as a separate meta layer, which pushes UML itself further to a higher meta layer.

Layers in the multilevel modeling are based on linguistic “instance-of” relations. Within one layer, terms cannot be each other’s instance. Researchers [6], [16], [19]have stated that multilevel modeling could bring mismatches between ontological and linguistic modeling layers. For example, two classes representing real-world concepts are included in one software model since they need to be instantiated at runtime. But one presented concept could be conceptually the instance of another, even these two classes are located in the same modeling layer. To clarify the confusion, this thesis focuses on engineering-oriented solutions. Models involved in this thesis serve as abstract syntax models of formal languages or language instances. They are located in a certain linguistic modeling layer according to how many times they need to be instantiated in the development and the use of the system.

The MDA layers introduced in Subsection 2.1.1 and the metamodeling layers described in this subsection are orthogonal. For example, both an M1 model and an M0 model can be platform-independent. Both the layered structures are used to coordinate models used in this thesis. To avoid confusion, the remaining text in this thesis refers to an architectural layer of MDA as an MDA “layer” (e.g., the CIM layer) and a metamodeling layer as a modeling “level” (e.g., the M2 level). Readers can see in Chapter 4 that a transformation path in this thesis crosses MDA layers but remains in the same metamodeling level.

2.2 Computer Languages

First, this section provides a short summary of fundamental concepts related to computer languages in general, based on classic lecture books [20]–[24]. To keep the focus, the introduction only covers relevant concepts that are essential to understand this thesis. Second, it also briefly introduces basic knowledge on Domain-Specific Languages (DSLs)[25], as this thesis develops DSLs as part of its solution.

2.2.1 Elements of a Computer Language

A computer language’s definition is usually determined by its syntax and semantics. The syntax is a set of rules specifying how the language looks like. The abstract syntax of a language specifies the data structure which holds the semantically relevant information of a program in this language. It is typically a tree structure that represents how this language looks like from the language processor view. On the other side, the concrete syntax of a computer language represents how this language looks like from the language user view. It specifies the notations that are used by users to express programs in this language, e.g., keywords and symbols, as well as the rules of how these notations should be used to write a program.

A computer language implementation requires both the abstract syntax and the concrete syntax. It is not necessary at the specification level though. Depending on the usage and characters of a language, its specification may contain only one of these two syntax components. In text-based programming languages, the concrete syntax is usually considered as an obligatory part of a language specification. The concrete syntax of such a language is usually defined by formal language-generation mechanisms such as a context-free grammar[26]. An implementation of this language then uses an abstract syntax holding the data structure of the implementation. The opposite cases are often observed from modeling and description languages. Specifications of these languages may only contain abstract syntax. High-level concepts and data models behind programs are the most important to users of these languages. By leaving the concrete syntax to the implementation level, such a language shall be implemented with various notations adapted to different user groups. Besides, abstract syntaxes of such languages are often specified in terms of other more general languages, whose notations are usually standardized. This enables unambiguous definitions of their abstract syntaxes.

The semantics of a computer language define the meaning of its programs, which consists of two aspects. The static semantics specify constraints and/or rules of the type system that a program in this language must conform to. A program has to be validated against the static semantics before it is processed. The dynamic semantics (also called execution semantics or operational semantics) specify the

behaviors of a computer when processing/executing a program in this language. Dynamic semantics are implemented in language processors which will be introduced in the following paragraphs.

An implementation of a computer language includes several programs that enable the use of this language. This thesis refers to such programs as the language tooling. Common components of the language tooling are introduced below. They can be standalone applications. Though, they are often built into an Integrated Development Environment (IDE) of this language in practice.

A source program editor (may be referred to as a language editor in this thesis) is a program for editing source programs in a concrete syntax of a computer language. It usually also has a simple interface to pass the edited program to back-end components. Language editors may have different forms. A source code editor is a text editor specialized for editing programs written in some textual concrete syntax of a computer language. This type of editors is the fundamental tooling component for many general-purpose programming languages. Such an editor usually provides support functions for editing specific languages in addition to normal free text editors, e.g., syntax highlighting, autocompletion and so on. A structured editor (also called a projectional editor) allows to directly manipulate the structure of the program, i.e., the abstract syntax tree. This kind of editor is not popular in the tooling of general programming languages. It is often seen as the implementation strategy of a modeling language whose underlying data structure is of importance to language users. Such a modeling language can be defined by its abstract syntax in terms of a metamodel. The editor renders a textual or graphic representation of the program based on projection rules. Users edit on the projected representation. Their editing actions modify the abstract syntax tree. Programs written in this language are stored as its abstract syntax trees, which is usually encoded in XMI.

A reader is a program that relates a concrete syntax of a language to its abstract syntax. For a parser-based language implementation which often uses some textual concrete syntax, it usually consists of two parts. First, a scanner (also called a lexical analyzer) takes a piece of a program written in a concrete syntax and transforms it into a stream of tokens. During this process, information that is not relevant to the meaning is removed, e.g., white spaces and control characters. Then, a parser (also called a structural analyzer) consumes these tokens to build up a representation that conforms to the corresponding abstract syntax. This representation is usually in the tree style which is called a syntax tree. It reflects the data structure of the program. If the input breaks the rules of the concrete syntax, an error will be generated by the reader. A projectional language implementation does not need these components. This thesis will not distinguish detailed components of different implementation approaches while it is beyond the focus. A component set that takes the input program as the syntax tree to the back-end components is referred to as the reader as a whole.

Not all computer languages are executable by specification, such as some configuration languages and markup languages. However, programs written in computer languages are meant to be processed by computers in some way. A back-end program that performs such processing tasks is called a language processor or an execution engine. Processor implementations of high-level languages often use two approaches. One of them is to implement a compiler that takes a program in this language and translates it into other artifacts. Execution semantics are described by relationships/mappings between inputs and outputs of the compiler. In the case of modeling languages, this approach is commonly referred to as transformation when the outputs are other syntax tree-based models. Or, it is referred to as generation when the outputs are in the textual form which is often general-purpose programming language code that can run on some infrastructure. A compiler that creates high-level programming code is commonly referred to as a code generator. Transformation and generation are conceptually the same process. Both of then create other artifacts from input programs. In contrast, a processor can also be an interpreter, which loads input programs and acts on it. In this case, execution semantics are described by explaining what semantic actions should be performed with respect to specific language elements. Such actions for a high-level modeling language shall be encoded by general-purpose programming language.

The term “translator” is often used to refer to a processor of a programming language in the programming language theory, including both interpreter and compiler, as well as other types of processors. It is also observed in modeling language literature that the term “translator” only refers to compilers. To avoid confusion, this thesis avoids using this term but sticks to the terms introduced above.

2.2.2 Domain-Specific Languages

A Domain-Specific Language (DSL) is a computer language specialized for a given class of problems, which is called a domain[24]. A realization of a DSL normally has the same components as introduced in Subsection 2.2.1. However, the abstract syntax of such a language is based on abstractions that are aligned with concepts used by the domain for which this DSL is developed. The concrete syntax should also be suitable for expressing these abstractions.

According to Völter's book dedicated to DSL [24], there are two approaches to define a domain. The author defines a program p as "a conceptual representation of some computation that runs on a universal computer (Turing machine)"[24]. Based on this definition, the **inductive (or bottom-up) approach** identifies a domain D as "a set of programs with common characteristics or similar purpose"[24]. Commonalities among the programs in this set can be described by a set of domain-specific patterns and idioms. Then, this set of programs from all conceivable programs P can be written in a domain-specific language l based on these patterns and idioms, denoted as P_l. The other approach, i.e., the **deductive (or top-down) approach** views a domain D as "a body of knowledge about the real world"[24] for which the software support needs to be provided. Developing a DSL for a domain in this definition is much harder than the inductive approach since the nature of D has to be understood precisely in order to identify the interesting programs in this domain from P.

Nevertheless, a domain D can be ultimately specified by a set of programs P_D in the realm of software. P_D can be expressed in multiple languages, whereas a language may only be able to express some part of P_D. Based on this understanding, the author defines a DSL for a domain D, denoted as l_D, as "a language that is specialized for encoding programs from P_D"[24]. It means that this language is able to represent programs in P_D more efficiently than other languages.

The boundary of a domain is often fuzzy. This is especially true in the deductive approach since if a program belongs to a domain is determined by human understanding and interpretation. The same program may also be considered as belonging to the intersection of two or more roughly orthogonal domains. Each domain covers one aspect of this program. For example, a program allowing users to fill online questionnaires belongs to the domain of web applications and the domain of questionnaire forms. It can be also considered as a member of a domain that is specialized for the online questionnaires. The coverage of a domain depends on the common purpose of its member programs. It consequently decides which abstractions should be included in a DSL expressing this domain.

Using DSLs shall bring various benefits that have been outlined by researchers and engineers[24], [27], [28]. Among them, the following points are particularly of interest to this thesis. They are the main reasons that the thesis chooses DSLs for its solution. These benefits are summarized in the following paragraphs in a general manner. The benefits that DSLs bring to the solution in this thesis are discussed at the end of this thesis in Chapter 8.

First, the DSLs can **serve as a thinking and communication tool** during development processes. This function particularly matches the idea of MDD that uses models as communication vehicles among different roles. Thus, defining and using DSLs is an important ingredient of MDD. These DSLs are used to express models or model-based programs. Their syntaxes play the role of metamodels. On one side, terms and structures used to build a DSL have higher abstraction levels than general-purpose computer languages and are aligned with the domain the DSL focuses on. This allows language users to separate essential logical structures in this domain from complicated low-level implementation details. Domain-specific programs can be described in a more declarative and concise way. These programs can be then read more clearly and be discussed more easily. On the other side, the expressiveness of a DSL beyond its focused domain is limited. This reduces the chances of language users to make mistakes.

Second, using DSLs **enables non-developer involvement**. The domain experts can understand programs in DSLs that focus on domains they are familiar with, while expressions in these languages are aligned with how they express the corresponding domains. All irrelevant low-level implementation details are hidden from them. They can read or even write code in DSLs and be involved in validation and review of the products expressed in DSLs.

Further, DSLs **bring productivity and efficiency** in development. Users of DSLs are free from low-level coding work. The amount of DSL code that has to be manually written for a product is supposed to be much less than the code that has to be written in a general-purpose language on a certain target platform. A processor of a DSL takes the responsibility to remove domain-specific abstractions and generate the code in a less abstract language. It may also run the programs on a target platform without having to compile a separate underlying general-purpose language program for the whole software every time. In this way, the processor shall parse DSL programs and invoke corresponding pre-compiled libraries which are based on the common concepts in the DSL. Besides, an implementation of a stand-alone DSL also comes with the language tooling (see Subsection 2.2.1) specialized for this language to support writing programs more productively.

Moreover, the higher level of abstraction DSLs and its separation of concerns make DSLs can be defined at an implementation-independent level. This enhances the **portability of the programs (thus, also the underlying models)** they express, which is important in MDD. A DSL expresses the application logic at an abstraction level that is meaningful to the domain. Programs in well-defined DSLs can be executed on different technical platforms by replacing the implementation of its processor. Besides, underlying models of DSL programs can be easily transformed into other representations.

2.2.3 Language Workbenches

Modern language engineering has been greatly simplified by so-called language workbenches[29]. They are toolsets that provide various high-level mechanisms to efficiently define computer languages and to implement language tooling.[30] They are the basics that make the solution in this thesis technically feasible for various development teams since they make the development of DSLs required in this thesis to be an affordable amount of work. Thus, this subsection briefly introduces the origins and features of language workbenches, as well as existing implementations of language workbenches that are ready to use. The term "language workbench" was proposed by Martin Fowler when he used it to refer to the tools that support building software around a set of DSLs [29]. However, nowadays, the capabilities of mature language workbenches are beyond the limitation of only supporting the DSL development. They are also suitable to develop general-purpose languages. A mature language workbench mainly has capabilities as summarized below, which are the technical foundation that enables the realization of the proposed framework in this thesis. More comprehensive reviews on language workbenches can be found in [30]–[33].

Define and modify language syntactic models: nearly all language workbenches provide relatively simple user interfaces and editing supports for language developers to formalize syntactic models of computer languages. This feature is usually supported by small declarative meta languages. It shall allow language developers to build an abstract syntax with modeling languages and automatically generate the default concrete syntax from the defined abstract syntax, or the other direction around. In most cases, both the syntactic models can be modified and improved by designers. Relations between them are maintained by the workbench. Thus, language developers only need to focus on abstractions and notations they want to define in their languages. This greatly simplified the development of language models.

Create language tooling: one of the most powerful functions of language workbenches is to automatically generate infrastructures of language tooling from user-defined language models. A default editor to edit programs in a concrete syntax can be generated. This editor could be either graphic/tree-based such as supported by MPS[34] and default editor generation facility of Eclipse Modeling Framework (EMF)[35], or in free-text style such as supported by XText[36], EMFText[37] and Spoofax[38]. It usually embeds support functions such as syntax highlighting, error-detection, etc. The editor usually comes with reader components such as parsers for textural concrete syntaxes. In addition, workbenches may also generate skeletons for language processors with default behaviors. These generated infrastructures are often in the form of general-purpose programming languages and can be optimized by language developers. They are easy to be integrated with customized pieces of code or programs in other high-level languages such as transformation rules in ATL [39]. Thus, language developers save a great amount of work. They are free from the coding of infrastructures that are common to multiple computer languages and can focus on implementing the semantics of their own languages.

This separation also makes the modification of an implemented language more easily, which is important especially for small DSLs that are often incremental with new understandings of its domain.

Build and test languages: language workbenches are essentially development frameworks for implementing computer languages as a set of computer programs. Thus, these workbenches integrate with continuous build tools that compile these programs. Besides, some of the workbenches also provide facilities to test developed DSLs. For example, EMF-based language workbenches often generate default Junit test code based on which language developers can add their unit-testing cases. Some workbenches such as MPS provide their own DSLs to write the test.

3 Related Works

This chapter introduces the existing researches that are related to the solution of this thesis. First, Section 3.1 reviews researches on CIM-PIM transformations that the thesis aims to establish for its proposed development framework. Second, Section 3.2 pays attention to the context domains that are of interest to this thesis, i.e., the simulated environment in spatial simulations. It reviews the existing spatial conceptualizations and representations, which has the potential to be used to model spatial entities in the simulated environments from both the human perspective and the system perspective, as well as the common forms of the component that provide simulated environments in the computer simulations. The lessons learned from the existing researches and the missing points which have to be fulfilled by this thesis are summarized at each subsection.

3.1 CIM-PIM Transformations

One main goal of this thesis is to bridge the gap between expected simulated environments from the human view and models of simulated environment components in computer simulations from the system view. It is comparable to the perspective switch from domain-oriented CIMs to system-oriented PIMs in MDD. Thus, this section reviews existing researches in CIM-PIM transformations to summarize the useful findings as well as the missing points that this thesis needs to fulfill.

MDA recommends automating PIM-PSM transformations and mature tools have been developed for this purpose, e.g., the EMF[35] that generates Java-specific model code from XMI-encoded Ecore models. This type of transformations is free from the domain context of the transformed models. It becomes a technical investigation once the target platform is fixed and can be solved at a domain-independent range. Compared to it, CIM-PIM transformations involve the perspective change. This makes this type of transformations more complex which requires much manual work. Aiming at reducing the human effort in the development, the review especially pays attention to what the existing researches achieved by their automatable CIM-PIM transformations.

3.1.1 CIM-PIM Transformations in Specific Domains

Mazon et al.[40] developed an automatic CIM-PIM transformation method specialized for the data warehouse (DW) development. Their approach defines CIMs of DWs using a UML Profile for the i* modeling framework[41]. This profile is used to describe actors in a business process and business goals that need to be achieved through this process. Information requirements are identified corresponding to the most concrete goals, i.e., information goals. Further, PIMs of DWs are described based on a UML Profile for multidimensional modeling. This profile organizes information into facts and dimensions. These PIMs are derived from the CIMs by a set of QVT rules.

Koch et al. developed the UML-based Web Engineering (UWE)[42] approach as an MDD process which can automatically traverse models from high-level functional requirements way down to prototypical Web applications [43]. In [44], transformations from CIMs to PIMs in this approach are introduced. The authors developed a metamodel as a UML Profile for the Web Requirements Engineering (WebRE)[45] and specified graphic icons for Stereotypes in this profile. These Stereotypes extend both the structural metaclasses (e.g., Node as a specialized Classifier) and the behavioral metaclasses (e.g., Browser as a specialized Action) of UML. The WebRE Profile is used to express the requirements of Web applications as CIMs. At the design phase, PIMs of Web systems are created. For this side, the authors developed the UWE Profiles to express the content of a Web system.

Transformation rules between the two layers of models based on these metamodels are defined in QVT. First, instances of the Stereotype Content in a requirements model are turned to classes in a UWE content model. Then, instances of Stereotypes that extend the Action metaclass are transformed into a navigation model as navigation classes or access structures. Another transformation derives a UWE process model from the UserTransaction together with a related Content in the requirements model. Finally, a presentation model specifying the layout of the application is derived from the navigation model

and the process model. Manual refinement of the intermediate models shall be made among the transformation steps.

Later in [43], the requirements metamodel is included in the UWE Profiles and an implementationof the UWE based on Atlas Transformation Language (ATL)[39], [46] is presented. Relations among the requirement package and other packages remain the same.

Fatolahi et al. presented in [47] a whole MDD process to generate web-based applications from use cases following the UWE principles. In their approach, State Machines at the PIM layer are generated by a tool called UCEd [48] from Use Cases at the CIM layer. These transformations are performed semi-automatically with inference from developers. The resulting State Machines are then further transformed into other PIM-layer models.

3.1.2 Analytic Approaches for CIM-PIM Transformations

Kherraf et al. [49] proposed a disciplined approach for CIM-PIM transformations. Their approach builds a CIM consisting of a Business Process Model (BPM) and a Requirement Model. The BPM is built by Elementary Business Processes (EBPs)[50] which represent well-delimited user tasks. The Requirement Model is derived from the BPM and expresses system requirements to optimally support the business. Both models are expressed with notations in UML2 Activity Diagrams (ADs). Then, a PIM is obtained from the Requirement Model. It represents system components to support a business process and involved business entities. The concept "archetype" in this paper represents a specialized term for describing model elements in the component models at the PIM layer. It plays a similar role as a Stereotype in other introduced approaches. The authors did not yet provide an implementation for automating the transformations but stated the possibility. Other analytic transformation approaches include the work of Kardoš and Drozdová [51] who use Data Flow Diagrams (DFDs) and textural descriptions to express CIMs. Use Cases, Activity Diagrams, Sequence Diagrams and Domain Class Diagrams in UML are used to express PIMs in their approach.

Zhang et al. [52] presented an approach for CIM-PIM transformations in a feature-oriented and component-based view. At the CIM layer, a feature model is used to structure system requirements. It contains a set of features and their relations. At the PIM layer, models are described by software architectures. This approach aims to bridge the gap between CIMs and PIMs in a disciplined manner. The authors introduced the concept of "responsibilities" to connect features and components. A feature model is operationalized into responsibilities, resource containers as well as relations among them. These elements are then clustered to construct the software architecture, based on responsibilities being assigned to components. No formal transformation pattern was defined, while this approach provides a basis to specify it.

3.1.3 Automatable CIM-PIM Transformations

Rodriguez et al. [53]–[57] extended the metamodel of UML2.0-AD and Business Process Diagram (PBD) of Business Process Modeling Notation (BPMN) for expressing security issues in business processes. Their work resulted in BPSec-Profile [57] which supports expressing security requirements as CIMs. Secure Business Processes (SBPs) described by BPSec-Profile are fed to a set of transformations to create analysis-level classes and use cases as PIMs. These transformations are defined by QVT rules[56]. First, a horizontal transformation within the CIM layer is performed to generate refined CIMs from a BPMN-BPD model. The output consists of a normal UML2.0-AD model representing business processes and a model conforming to the BPSec-Profile which represents all security issues. Then, the process model is fed to a vertical transformation as the input to generate the first version of a UML2.0-Class Diagram (CD) at the PIM layer without security issues being considered. This CD and the BPSec model are then transformed into a refined CD including security issues. Further, the refined CIMs from the horizontal transformation step are transformed to generate UML2.0-Use Case models, followed by a manual refinement step.

Gutierrez et al. [58] automated the generation of Activity Diagrams via model transformations from use cases, whose output can be refined by hand. They defined a metamodel for describing input use cases.

These use cases are encoded in an XML-based concrete syntax and represent the functional requirements of the system under development. The output Activity Diagrams are conformed to a selected subset of the metamodel of UML2.0 Activity Diagram. Transformations in this approach are defined by the QVT-Relational language. A transformation generates an Activity for each requirement in the input use cases and an Action for each main step. Exceptional steps that indicate conditional choices are transformed into decision nodes. Elements are then chained through control flows. The authors did not explicitly locate their metamodels regarding the MDA layers. Nevertheless, their work is comparable to transformations from CIM to high-level PIMs.

Hahn et al. [59] developed a semi-automatic approach called SHAPE (Semantically-enabled Heterogeneous Service Architecture and Platforms Engineering) to bridge the gap between business requirements at the strategic level and the execution models. This approach links CIMs of business requirements expressed by a metamodel called CIMFlex to PIM-layer models. CIMFlex combines BPMN and Architecture of Integrated Information Systems (ARIS) notations. It supports expressing issues to achieve business goals such as business rules, processes and contracts. The PIM layer in this approach has two sub-layers. The sub-layer linked to the CIMs uses the Service-Oriented Architecture Modeling Language (SoaML)[60] to describe models. These models represent services in distributed environments.

A set of transformation rules was defined to transform CIMFlex models into SoaML models. The initial version of CIMFlex supports transformations via ATL. Then, the service models are further transformed into more comprehensive multi-agent system models at the other PIM sub-layer. A PIM metamodel called PIM4Agents[61] was developed for expressing models in this sub-layer. Transformation rules were defined between SoaML and PIM4Agents.

De Castro et al. [62] applied CIM-PIM transformations in developing information systems with the service-oriented development method(SOD-M). Their approach separates the business in which the system is involved from the functional requirements of the system. The focus lays on modeling the former one at the CIM layer. CIMs used in this approach are modeled by the value model[63] and BPMN. The value model expresses business cases as value exchanges and value activities of business actors. BPMN describes processes related to the environment in which the system is used. A PIM in this approach consists of following models: a use case model identifying business services in UML Use Case Diagram(UCD), an extended use case model identifying functional services to carry out the business services in UML UCD, a service process model expressing workflows of activities to perform business services in UML Activity Diagram, and a service composition model extending the service process model by identifying fundamental behavioral units of each activity. These models are expressed in a set of DSLs based on well-defined metamodels.

The authors then proposed a methodological process to define semi-automatic mapping rules for CIM-PIM transformations based on the metamodels. The mappings range from natural language descriptions of mappings to formal or half-formal transformation rules. They proposed to use the weaving models [64] to integrate the mappings that cannot be fully formalized. The transformations are implemented in ATL integrated with weave models.

Bousetta et al.'s approach[65] models both the behavioral and the static aspects of a system at the CIM layer. These two aspects are captured by two models: a Use Case model that represents business actors of the system and functionalities to be realized; a BPM that represents the behaviors of use cases. The BPM includes three descriptive views, namely, the functional view presenting activity flows, the behavioral view presenting conditions under which activities are performed and the structural view presenting involved objects in a process. This approach starts with building the BPM. The three views are expressed in one BPMN diagram. The resulting BPM has multiple levels. In addition, models at the CIM layer contain template-based Business Rules. At the PIM layer, a Domain Class Diagram (DCD) is used to represent the static aspect of a modeled system. A PIM also includes sequence diagrams of the system's external behavior (SDSEB) which represent high-level behaviors of systems. An SDSEB is a UML sequence diagram that shows only interactions between actors and the whole system. It is transformed later into a sequence diagram of the system's internal behavior (SDSIB). An SDSIB represents interactions between objects within a system.

Transformations from CIMs to PIMs are realized through a series of transformations. First, high-level BPMs are transformed into Use Case models. Then, low-level BPMs that present sub-processes in more detail are transformed into SDSEBs. Further, Input/output data objects in the low-level BPMs are mapped to classes in DCDs. These classes are completed with terms and facts derived from the Business Rules. Models in the approach are at least semi-formal. The authors also presented mapping rules for each step which enables a semi-automatic transformation from CIMs to PIMs.

Kriouile et al. [66]–[68] proposed that a CIM should consist of BPMN models in BPD and Use Case models in UCDs. The former part represents exchanges of information between actors, while the latter part identifies features and good functioning conditions of a system. A BPD is modeled at first and a Use Case model is derived from the BPD by a horizontal transformation. The transformation from the Use Case model to a BPD is also possible during the refinement of CIM-layer models. Then, a behavioral model at the PIM layer is obtained by System Sequence Diagrams (UML SSDs) via vertical transformations from UCDs. Similarly, PIM-layer static models expressed in Domain Class Models (UML DCMs) are vertically transformed from BPDs. These static models represent the structure of modeled systems.

Transformation rules in this approach are based on the equivalence between the concepts in corresponding metamodels. These rules are specified in QVT and thus enable the possibility of automation. The two parts of CIM-PIM transformations for behavioral models and static models are described in detail in [67] and [66], respectively.

Rhazali et al. have done a series of work regarding CIM-PIM transformations. In [69] and [70], business models are expressed in BPMN and UML2 Activity Diagram at the CIM layer. Class Diagrams and Package Diagrams are used for expressing the static view of PIMs, while State Machine Diagrams are used for expressing the dynamic view and the functional view. Their approach was developed analytically at the beginning which obtained a set of qualitative guidelines to construct CIMs in a way that is easy to be transformed into PIMs, e.g., the average numbers of Activities that a CIM model represented in ADs should have. A set of analytic mapping rules was used in their early work for CIM-PIM transformations. Later in [71]–[73], these mapping rules are formalized in ATL to enable automatic transformations. In their most recent work [74], [75], SoaML is used to express CIMs. The authors also introduced an additional model at the PIM layer to represent the web view. This additional model is expressed by the Interaction Flow Modeling Language (IFML)[76] and is transformed from the other PIM-layer models. Through this strategy, transformations can be applied to generate the front end of web applications.

3.1.4 Summary and Relation to This Thesis

The introduced researches in this section are summarized in Table 3.1 based on evaluation approaches on CIM-PIM transformations introduced in 3.1.4.1, followed by a discussion on how the existing work is related to this thesis.

3.1.4.1 Evaluation

A short introduction to evaluation approaches on CIM-PIM transformations is given below to provide reference and to support readers' understanding.

Yue et al.[77] designed a conceptual framework to describe transformations from requirements to PIM-layer models[2]. It includes taxonomies to express requirements, restriction rules that are used for regulating requirements, and a taxonomy to express PIM-layer models. In addition, they developed a taxonomy for describing transformations between CIMs and PIMs as well as a process model for describing transformation processes. This framework does not provide models and mappings for specific transformations but focuses on describing transformation approaches. Requirements in this framework rather refer to the information gathered prior to the constructions of CIMs. Nevertheless, it can serve as

[2] This paper refers to PIMs as "analysis models" and PSMs as "design models", while this thesis refers to PIM models as "(system) design models".

a basis to compare the transformation approaches from high-level requirements to PIMs by structuring them into a comparable structure. The authors derived a list of evaluation criteria to give suggestions on constructing good CIM-PIM transformations.

Later, Sharifi et al. [78] used a simplified and adapted version of this framework for reviewing CIM-PIM transformations. This version only considers high-level components of rule-based transformations with CIMs as inputs. This provides an overview of the possible components of CIM-PIM transformations. Figure 3.1 is a merged redrawn of Figure 6-9 in [78] to provide a concise presentation of their taxonomy. The term "Traceability", according to IEEE Standard Glossary of Software Engineering, is defined as "the degree to which a relationship can be established between two or more products of the development process…for example, the requirements and design of a given software component match" [79]. Other concepts have been introduced in Chapter 2.

Evaluation criteria for transformations are derived from the model. Kriouile et al.[80] conducted a criteria-based evaluation based on the taxonomy shown in Figure 3.1 and the guidelines of MDA. The authors evaluated input CIMs of a transformation approach regarding their coverage, i.e., if a CIM covers the static view with business objects, the behavioral view with business processes, and functional view that considers requirements. An output PIM is evaluated against its completeness to check if it includes both the structural aspect and the behavioral aspect of the system. Then, the transformation process is checked to see if the transformation is automatic, if the transformation rules are complete, as well as if the traceability is maintained.

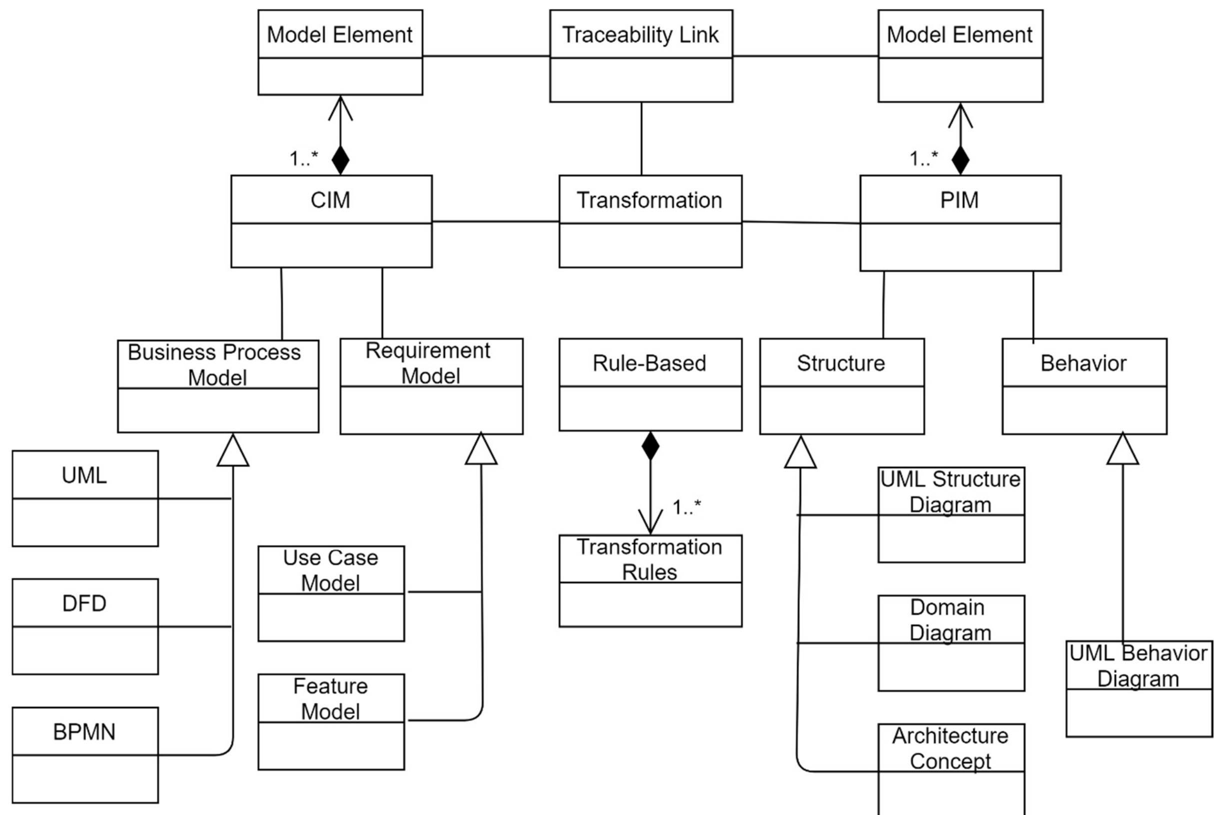

Figure 3.1: Taxonomy of CIM-PIM Transformation Components. [80]

Table 3.1 summarizes the related work introduced in Subsection 3.1.1-3.1.3 based on the transformation components identified by the taxonomy shown in Figure 3.1 and the derived criteria in [80] as introduced above. Since describing business processes can hardly be done without denoting involved objects, this table summarizes business model-related information in one column. For each component in an approach, the table denotes the modeling languages used to describe models (i.e., the metamodel), if a specialized metamodel is defined in addition to the standard modeling notations, and if this part is transformed from other models at the same layer.

	Focused Domain	CIM Presentation and Metamodel		PIM Presentation and Metamodel		Form of rules
		Business Process	Requirement	Structural	Behavioral	
Mason et al.[40]	data warehouses	N/A	UML Profile	N/A	UML Profile	QVT
Koch et al. [42]–[45]	web application	UML Profile based on Activity Diagram	UML Profile based on Use Case	UML Profile based on Class Diagram	UML Profile based on Activity Diagram	QVT, ATL
Fatolahi et al. [47]	web application	Use Case	Use Case	Profiles used by chosen code generation tools, transformed from State Machine	State Machine	QVT
Kherraf et al. [49]	N/A	Activity Diagram based on BPMN	Activity Diagram, transformed from BPMN	Archetypes	Archetypes	N/A
Kardoš et al.[51]	N/A	Data Flow Diagram	N/A	Domain Class Diagram	Sequence Diagram	N/A
Zhang et al.[52]	N/A	Feature Model	N/A	Software Architecture	N/A	N/A
Rodriguez et al. [53]–[57]	security business	BPMN-BPD	BP-Sec Profile based on Activity Diagram, transformed from BPD	Class Diagram	Use Case	QVT
Gutierrez et al. [21]	N/A	N/A	Metamodel to describe Use Case	N/A	Selected subset of UML Activity Diagram	QVT
Hahn et al. [59]	N/A	CIMFlex metamodel based on BPMN	CIMFlex metamodel based on BPMN	SoaML transformed from CIM, PIM4Agents transformed from SoaML	PIM4Agent metamodel	ATL
Da Castro et al. [62]	N/A	Value Model and BPMN	N/A	Domain Specific Language	Domain Specific Language	ATL
Bousetta et al. [65]	N/A	BPM, Template-based Business Rules	Use Case, transformed from high level BPM	Domain Class Diagram	Sequence Diagram	Rules defined
Kriouile et al. [66]–[68]	N/A	BPD	Use Case, transformed from BPD	Domain Class Diagram, transformed from BPD	System Sequence Diagrams, transformed from UCD	QVT
Rhazali et al. [69]–[75]	N/A	BPMN and Activity Diagram or SoaML		Class and Package Diagram, IFML from other PIMs	State Machine	ATL

Table 3.1: Summary of CIM-PIM Transformation Researches.

3.1.4.2 Lessons Learned and Missing Points

Existing CIM-PIM transformation approaches provide inspirations and guidelines for the construction of the view bridging in this thesis. Related observations from these researches are listed below.

1. Intermediate models in the transformation chain of an approach do not have to be unified with other approaches. As Subsection 3.1.1 described, in approaches with horizontal transformations within the same layer, intermediate models such as Use Cases generated from models of business processes are observed being classified both as CIMs[65] and as PIMs[53]–[57].
2. Obtained system models from the transformation should contain structural and behavioral aspects of the system. This is recommended by the introduced evaluation approaches. Most of the reviewed approaches consider both aspects at the PIM layer and present a path of transformations to achieve their planned outputs.
3. The transformation automation relies on well-defined mapping rules among formal metamodels, expressed by QVT or ATL. In newer researches, ATL is more frequently used thanks to its good tooling support. This proves the feasibility to use ATL for implementing mapping rules between CIMs and PIMs.
4. Most automatable approaches recommend a semi-formal transformation process as described in Subsection 3.1.1 and Subsection 3.1.3. Human roles may manually transform unstructured information, evaluate and optimize the generated results of automated transformation steps. Human interference shall improve the quality of production. Thus, a practical approach should minimize the human effort, while also provide a guide on how the human roles interact with the automatic process.

Despite that various CIM-PIM transformation approaches exist, none of them satisfies the needs of view bridging that this thesis is interested in. Some approaches focus on a restricted domain. For instance, Mason et al.[40] worked on the data warehouse and thus paid more attention to the structure of information pieces. The work from Koch et al. [42]–[45] supports generating components and operations in the Web. The domain of spatial-aware simulation is orthogonal to these two domains. Although a simulated environment component can be settled in a web environment or can involve data storage, the existing solutions do not cover models and transformations dealing with the generation of spatial-temporal varied data.

Approaches that do not state a focused domain fail to satisfy the needs due to a similar reason. Their CIMs pay attention mainly to enterprise and organizational aspects that the systems under development are involved in, which is one reason that BPMN is most frequently chosen to express their CIMs. These approaches orient themselves to the business modeling domain. Functionalities in generated models are often user-triggered operations, such as submitting a form via clicking a button. While business goals and user operations are of importance to any system development, this thesis, however, is interested in expressing functional requirements at a more specialized level as well as has a different runtime environment. Functionalities of the interested systems are restricted to provide simulated environments during spatial-aware simulations. CIMs in this thesis should focus more on expressing the expected behaviors of spatial phenomena in simulation processes than on organizational processes. Metamodels that are more expressive for the dynamic spatial phenomena than BPMN or basic UML are needed to reach more specialized software structure at the PIM side via automatic transformation.

3.2 Spatial Conceptualization and Data Representation

Simulated environment components provide digital representations of environmental phenomena which are often spatial information. During an MDD process, such contexts need to be expressed by models at different layers from the human perspective and the system perspective. Thus, this section briefly reviews the existing researches on the expression and representation of the spatial information from these two perspectives, as described in Subsection 3.3.1 and 3.2.2, respectively. Their relations to this thesis are summarized in Subsection 3.2.3.

3.2.1 Spatial Conceptualizations

The underlying conceptualizations of spatial information have been thoroughly investigated since the early years of Geographic Information Systems (GIS). It has been well-accepted that two fundamental

approaches exist when humans conceptualize spatial phenomena, i.e., object-based and field-based conceptualizations. The former ones view spatial phenomena as distinct individualities carrying various characteristics, while the latter ones view spatial phenomena as a set of locations that carry characteristics (i.e., as a function that maps locations to thematic values). [81]–[83] Each approach stands for an angle to view the real world, but neither of them represents the "truth" of the world. Duality exists even when people conceptualize the same phenomenon. For instance, a river can be viewed as an object bounded by its bed. It can also be viewed as a field, with each spatial location within its boundary have a certain water depth. Frameworks that integrate both conceptualizations to manage the spatial data have been established [84], [85].

High-level conceptual models and description languages of spatial phenomena using the fundamental spatial concepts as constructs have been proposed. Günting et al.[86]–[88] have done a series of work to specify database modeling and query languages based on the concept of moving object. Carmara et al.[89] use the concept of field as a general datatype to represent and to operate on spatiotemporal data. Other researchers consider both conceptualizations. Kuhn and Ballatore[90] designed a language for spatial computation by specifying core spatial computation operators based on a set of basic spatial concepts.

Specifications of these models and languages are essentially human-oriented. Constructs of these languages are based on common-sense conceptualization rather than specialized knowledge in spatial data, even though they may be referred to as "datatypes". They are defined at the cognitive level and are independent of how their represented information is logically organized. Thus, these works are comparable to the models at the CIM layer**.** They mainly aim to provide richer semantic to spatial data, as well as to raise the usability of spatial database and information systems. Implementations of these works are often embedded in spatial DBMS and GIS software, or as code libraries that enable general spatial computing by people with limited programming experience. In this situation, implementations of these languages locate at the M1 model level from the model-driven engineering view, which encloses a fixed implementation.

3.2.2 Representation of Spatial Data

Logic-level data and service models that are implementation neutral have been developed to support building, managing and exchanging the spatial data and spatial information services. These works are roughly comparable to the models at the PIM layer. Significant works in this area are standards from Technical Committee 211 of the International Organization for Standardization (ISO/TC 211)[3] and the Open Geospatial Consortium (OGC)[4]. Most data models in these standards have their groundings on academic research in relevant domains such as computer graphics, spatial-temporal databases, etc. Thus, they reflect well the theoretical work in this area.

The ISO/TC 211 focuses on the standardizations related to geographic information. The ISO 191XX series are developed by ISO/TC 211 for information associated with a location relative to the earth[91]. These standards cover a variety of topics regarding the acquirement, management, processing and exchange of geographic information. Among these standards, the family that is particularly of interest to this thesis are the data model standards. They provide conceptual schemas to represent different aspects of the geographic information as components of features. The term "feature" is defined by the domain reference model of ISO19101, which is an "abstraction of real-world phenomena"[91]. UML is used as the conceptual schema language within these standards[92].

Among the data model standards, ISO19107 [93] specifies a schema with geometric and topological types to represent the spatial characteristics of features. To support low-cost implementations, ISO19137[94] also provides a small core profile of the spatial schema defined in ISO19107. ISO19108[95] provides a schema with geometric and topological datatypes to represent temporal characteristics of features. Further, ISO19141[96] extends ISO19107 to enable the expression of moving geometries. A spatial feature type can be composed of these datatypes together with thematic attributes. Alternatively, ISO19123[97] provides a schema to represent features as coverages. A direction position

[3] https://www.isotc211.org/
[4] http://www.opengeospatial.org/

within the geometric representation of a coverage feature has a single value for each thematic attribute of this feature. These standards schemas correspond to the M1 level models. The ISO19109[98] provides a metamodel called General Feature Model (GFM) to integrate various components described by these standard schemas as features. Based on this metamodel, this standard specifies rules to develop application schemas that describe feature types in particular application fields. These rules are descriptively documented in terms of UML requirement classes.

The OGC is an international industry nonprofit organization that develops open, free standards to improve geospatial data sharing. It has the main focus on promoting interoperability specifications for geospatial content and services. Standards from OGC are developer-oriented technical documents that specify interfaces of software and web services that can work together without further debugging[99]. Information models are often provided with these implementation specifications in the form of XML schemas in OGC's schema repositories[5].

The OGC also developed an architecture called OGC Abstract Specification to provide conceptual models for developing OGC standards which are referred to as Implementation Standards.[100] This part is closely related to the model abstraction levels of this thesis as the OGC's intention states. OGC Abstraction Specification consists of an essential model to describe the conceptual links between the software and an abstraction model to describe how software should work at an implementation-independent level [99]. The ISO TC211 has a co-operative agreement with the OGC, through which ISO TC211 standards have been incorporated as part of the OGC Abstract Specifications while the OGC submits Implementation Specification to ISO for adapted as ISO International Standards.[101] The OGC Abstraction Specification is divided into topics. The ISO 19107 is identical to OCG Abstract Topic 1 – Feature Geometry, and the ISO 19123 is identical to OCG Abstract Topic 6. Similar to various ISO standards, these documentations are not exactly at the same abstraction level. For instance, Topic 1 and 6 specify standard schemas of datatypes to represent the components of features as mentioned above. Topic 5-Features[102] expresses nine levels of abstractions, which implies the process of modeling the real world. Its abstraction model of features acts as a very general metamodel of feature types.

3.2.3 Summary and Relation to This Thesis

As summarized in the previous two subsections, models of spatial phenomena from the human view and from the system view both exist. Language models in the framework proposed by this thesis make use of these concepts and datatypes resulting from these researches. However, existing works have purposes that are different from this thesis. Thus, their models cannot be directly used for this thesis. Observations of these differences and how these observations inspire this thesis are listed as follows:

1. A conceptualization of spatial phenomena in existing researches reflects an angle to view real-world entities and is associated with various possible functions. Metamodels at the CIM layer in this thesis describe requirements of expected context generated by various simulated environment components under development, which could be viewed from different angles. For instance, in a simulation, the system of interest component needs to be informed of the hazardous area of a storm. In this simulation, the storm is viewed as an object moving in the space. In another simulation, the same storm may need to be viewed as a wind field that the system of interest travels through. Thus, CIM metamodels should avoid presuming a fixed conceptualization that states "what it is" for a described phenomenon but focus on supporting expressing the expected characteristics and behaviors that have to be preserved in models in more specific layers.
2. Standardization organizations like ISO and OGC focus on enabling data sharing and system interoperability. Their models include interfaces and public attributes that datatypes should expose to external components. These models aim to cover possible properties and operations that a datatype shall expose to external systems, but do not concern much with the development cost of a separate system. This thesis has a different focus that aims to facilitate application development processes. It prefers to provide easily usable models in small size and only considers simulation relevant aspects. PIM-layer metamodels specified in this thesis utilizes the common geometry types from existing models, e.g., point and polygons, to express spatial characteristics. Besides, it introduces stereotypes based on widely used

[5] http://schemas.opengis.net/

geometric representations in spatial simulations, such as gridded sites in cellular automata[103]. Developers shall further map PIMs to existing implemented code utilities based on these types.

3. Most of the spatial data models are defined from a database view, i.e., information exists somewhere that can be checked or requested. Spatial entities in this view may be modeled as hybrid spatio-temporal digital objects since historical states of these objects need to be stored. In contrast, models describing a computation process of simulated phenomena include runtime objects that are being updated during executions. These objects need to be modeled as situating in space and changing over time, who do not necessarily hold the past states.

4. Existing spatial data (meta-)models are defined at different abstraction levels without alignment with the process of development & the use of the software. This hinders the efficient choice and use of these models. It is not useful if developers use a very general algebra to model a software which they need to implement. Exposing too many details of data structure in the user interface makes the products quite complicated to users. Thus, the development framework in this thesis should locate the used models properly in the software development cycles according to their abstraction levels, so that involved roles shall gain maximum benefits from applied models.

5. The traceable view switch is absent among existing models from various perspectives with different levels of abstractions, although its importance has been stated by OGC[102]. It leads to heavy manual work when applying MDD principles using these models. Transformations from high-level user understandable models to implementable software design models cannot be automated since the mappings among them are missing. Also, since complicated logic-level data standards introduced in Subsection 3.2.2 are not easily readable to non-expert users, no control exists to preserve user expectations of the final products when developers design the application using these standards. These gaps have motivated this thesis, as pointed out in the introduction chapter.

3.3 Simulated Environments in Software

This section summarizes the common styles to provide simulated environments in simulation programs. From a technical perspective, a simulated environment is some digital data. These data are fed to the component that simulates a system of interest during its execution, or to some intermediate digital system which further presents them to non-digital participants (e.g., human) in physical simulations. They could be either externally provided or simulated within the working community based on a computational model. Forms of the environment components in these cases are summarized in Subsection 3.3.1, followed by Subsection 3.3.2 that denotes their relation to this thesis.

3.3.1 Styles of Simulated Environment Components

It is very common to use existing data for representing simulated environments. These data are normally collected by a trustworthy external provider, or from the fieldwork by the working community. The data source could also be synthesized by some external providers that are professional in modeling and simulating required environmental phenomena. Modern simulation toolkits with spatial extension, e.g., NetLogo[104], GeoMason[105], Simulink[106], support loading spatial data used as the simulated environment in simulations, which is sufficient for relatively simple simulation programs. In a more complicated simulation, simulated environments may be provided through a more loosely-coupled component which could be a connection to a spatial data store.

The development of simulated environment components with integrated computational models are more complicated. It needs to notice that the "simulated environment" is a concept relative to the "system of interest model". A simulated environment includes abstraction of some real-world systems, which are conceptually viewed as phenomena in the situated world of the system of interest. In this case, they are provided as a component that embodies computational models to produce necessary data representing the environmental phenomena. Within this component, the phenomena being computed become the system of interest model which may have their own simulated environment.

This case could happen when the required data are not available or only partially available from external providers, or when the working community desires to control the environmental conditions more freely. Another possibility is the available data is about a phenomenon that does not influence the system

of interest directly but drives the behavior of another phenomenon that influences the system of interest. Further, a working community may have subgroups that develop computational models for simulating various real-world systems, some of which produce output that can be used for other models as simulated environments.

In spatial-aware simulations, models providing simulated environments are executed to draw data samples of spatial patterns changing over time, which are then sent to other components. Within the models, these spatial patterns may emerge and change through different mechanisms, depending on the underlying methods. The conceptual schemes of various spatial simulation paradigms are summarized in the following list. Many practical models use a mixture of these schemes, with each of their building blocks following one scheme.

1. The behaviors of a model are described based on the discretization computation of continuous functions (often in the form of differential equations) on a set of regular-spaced sample locations (i.e., based on finite-difference methods). The spatial patterns are determined by the top-down view equation-based relationships between thematic values and spatio-temporal locations. In the results of computation, A spatial pattern is the difference of computed sample values of some attribute from location to location. It is updated over discrete time steps through the update of the sample values. These top-down equation-based methods are widely used to simulate phenomena. A famous example is the sea surface simulation based on a function called the wave variance spectrum[107]. The theme to be computed is the elevation of the sea surface related to the sea level. Its patterns are described by the spectrum and the Fourier transformation from the spectrum to the elevation. In the computation, state values of the elevation are calculated on a set of grid points at a two-dimensional space at a necessary pace.[108] Other cases can be found in the simulations of temperature fields based on the heat equation[109], concentration distributions of chemical substances simulated based on advection-diffusion equations[110], etc.
2. The behaviors of a model are described based on a set of unmovable units, each of which shall hold a set of thematic attributes. The macroscopic spatial patterns that are of interest emerge from the state differences of the thematic attributes from unit to unit. In many model descriptions of this kind, the states of exposed themes are implicitly referred to as the states of these units. Such a model focuses on expressing the rules to update the state of a unit about these themes with the current states of this unit and its neighborhood units. This view is often taken by cellular automata models where the units are often referred to as "cells" [111]. The updates happen iteratively, and the macroscopic spatial pattern is changed over time through the local updates.

   Different from the first case, this type of models is bottom-up. They describe local processes that consider the neighborhood of the location where a state value is computed. No central control exists in the core paradigm. Nevertheless, the first two cases have similar schemes in the sense that the values reflecting spatial patterns of interest are carried by a set of fixed locations, despite that the finite-difference methods often work on a regular grid of points, while the "cells" of cellular automata could have irregular geometry [112]. A collection of the units is often referred to as "lattice" or "sites" in both cases, which is conceptualized as discretized spaces experiencing some states. Researchers have also reported that the models based on the finite-difference methods can also be transformed into the form of cellular automata[113].
3. The behaviors of a model are described based on a set of moving units, each of which shall hold a set of thematic attributes. The macroscopic spatial patterns that are of interest emerge from the distribution of these units over space. Such a model focuses on expressing the rules to update the location of a unit at each step. The existence of these units implies a theme. In a more complicated case, the macroscopic spatial patterns of interest may be exhibited from the state differences of thematic attributes from unit to unit. This view is often taken by the multi-agent models of mobile entities, while each unit is conceptualized as a mobile entity[103]. A simple model could consist of a set of random walkers, whose moving direction and speed (or distance) at a step are drawn from a probabilistic distribution.[114] In practical cases, the random walker rules often are more decorated, which may consider additional effects such as the correlation between consecutive steps[115]. Further, efforts of the movement goals of these units and the background information can be also specified in the rules to define purposeful agents, which yields the computation of search problems in the spatial context.[103]
4. The behaviors of a model are described based on a set of moving units that actively change the thematic attributes of a set of unmovable units. The macroscopic spatial patterns of interest to external components emerge from the state differences of thematic attributes carried by the unmovable units. Such

a model focuses on expressing the rules of two aspects, i.e., the rules of updating the location of a moving unit and the rules of updating the thematic states of the unmovable units when a moving unit arrives at its location. This view is often taken by the variants of percolation models[116]–[119] and Eden growth models[120], [121], which are bottom-up view models to simulate the spread of substances such as forest fire, oil spill and so on[122].

This case is closely related in both 2 and 3. These unmovable units are normally conceptualized as the "underlying space" or "landscape" in such model descriptions (as the lattice or sites in 2). Movements of the moving units (as some random walkers or rational agents in 3) shall be expressed in terms of discrete steps from unit to unit of the underlying space instead of continuous coordinates, which significantly simplifies the computation.

When a model based on the above-described schemes is developed as an independent simulation program, it is used to produce multiple sample draws. The summarized characteristics (often in the form of describing functions and statistic indexes) of multiple samples are analyzed to provide answers and suggestions for real-world problems. The purpose of usage changes when this model is used to compute simulated environments for other components. In this situation, it is supposed to be already "correctly" established. A sample draw from this model is fed to other components during a simulation run. The summarized characteristics may be fixed in the implementation or be implemented as modifiable parameters. In this case, the development of simulated environment components involves two main tasks. First, the computational models of environmental phenomena should be developed and be implemented with the technologies that are compatible with the current simulation platform. Second, the models should be integrated into a target frame as the environment component of the target simulation.

3.3.2 Summary and Relation to This Thesis

Ways of providing simulated environments in simulation programs introduced in Subsection 3.3.1 can be categorized into three styles as shown in Figure 3.2. A system of interest component that consumes the provided environmental data is referred to as the "client system" of the simulated environment component in this subsection.

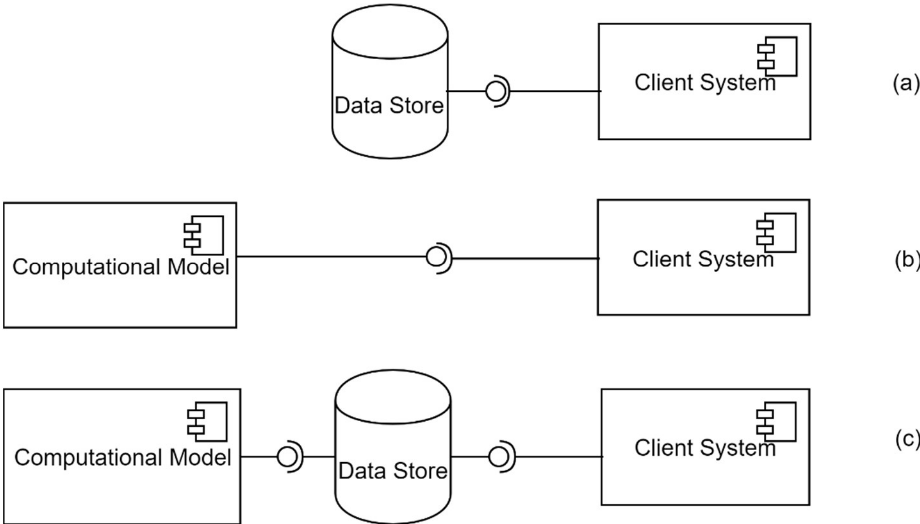

Figure 3.2: Styles to Provide Simulated Environments.

The case (a) is to use externally provided data. Simulated environments produced by integrated computational models shall be provided to client systems in two ways. The computation shall run synchronously with client systems and send computed values to client systems at each step as the case (b) illustrates. The case (c) shall apply when the computation of environmental data requires long process time, in which computation results are exported to a data store at first and be used by client systems later for efficiency. In both cases, an environment component needs a frame to enclose computational models.

A computational model shall be implemented as a continuous function that calculates thematic values of a phenomenon when given a spatial-temporal location. It does not contain the logic to compute a chosen set of discrete samples. Each calculation is performed independently that does not require results from previous calculations. Simulated environments provided in such a way are conceptually like analytic coverages as defined by ISO19123[97]. The computation is performed reactively upon the request of client systems. The frame enclosing such a model provides access to this model with a location as the parameter. Running this model with client systems needs to determine a finite structure of the required output at a computation step. This situation also applies to the case (a), in which the analogy of the mathematic calculation is the connection and query functions to the data source.

Computational models shall also ground on process modeling methods that calculate the next state of a phenomenon based on its current states, such as the spatial simulation models summarized in Subsection 3.3.1. In this situation, a simulated environment component needs a data structure to hold the current state of its computed phenomena at runtime. Besides, since such a model computes discrete outputs, the component also needs a frame to enclose the logic for evaluating values at a location in the continuous space from computed values. This situation also applies to the case (a), when the logic is not provided with the data source.

The identification of common schemes and the necessary composition of simulated environment components are the basis to specify the design-level metamodels that express these components in this thesis. Concepts of the structural metamodel are introduced based on the analysis in this subsection. These metamodels are presented in Section 5.2.

4 Language-Driven Development Framework of Simulated Environment Components

As Chapter 1 has addressed, the overall goal of this thesis is to provide a domain-specific solution to facilitate developments of software components that provide simulated environments in simulations, respected to identified research objectives. This solution is provided in this thesis as a development framework built on top of the language-driven development[123]. The proposed framework reflects the language-driven paradigm in two aspects. First, it involves domain-specific meta languages that describe simulated environment components in views of different roles throughout a development process. Second, it constructs software that has the structure of a computer language. This applies to the structure of the proposed framework itself, as well as for structures of simulated environment components to be built within the framework. These two aspects are explained in the first two sections of this chapter, followed by a section that presents a development process using the framework. Benefits brought by this framework are clarified during describing this development process, with comparison to domain-independent solutions.

The framework specification consists of the following parts. Corresponding sections that present these parts in this thesis are also listed below.

- **A set of formal meta languages used within the framework:** an overview of these languages at the architecture level is given in Section 4.1, including purposes to use them in the framework, perspectives from which they are specified, their target users, and relations among them. Chapter 5 is dedicated to explaining modeling principles and detailed specifications of these languages, including their syntactic models, structural semantics, and transformations among them.
- **A system architecture to guide the realization of the framework:** the architecture specifies obligatory and optional components of this framework. It is presented in Subsection 4.2.1, including the functionalities and places of these components in the framework, as well as how they are connected to build up this framework. The specification of the system architecture is implementation neutral, same as the meta language models. Implementation suggestions are given during the introduction and an exemplary implementation for use case validation is provided in Chapter 6.
- **A development process to guide the use of the framework:** this process explains how to use the proposed framework to develop a software application to generate simulated environments. It is presented in Section 4.3. This process specifies transitions of artifacts among different components, input/output artifacts of each component, as well as activities that need to be performed in each development phase.

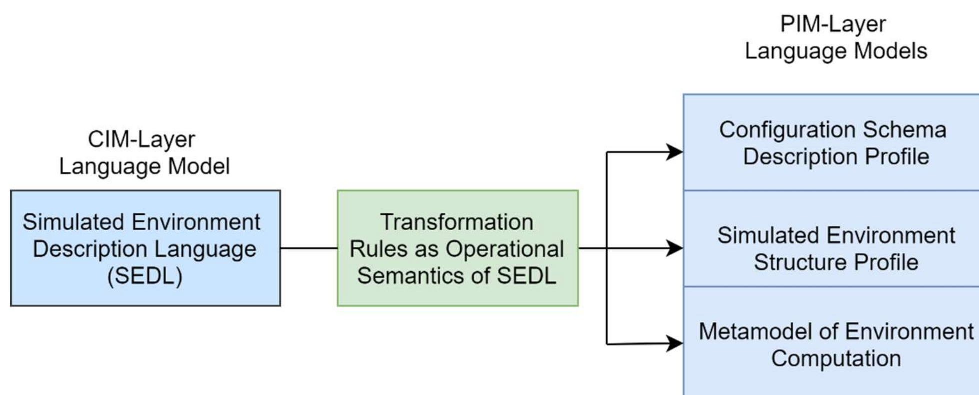

Figure 4.1: DSL Models in the Proposed Framework.

4.1 Executable Meta Languages

Domain-specific meta languages are the backbone of the proposed framework. This framework specifies four language models for describing CIMs and PIMs of simulated environment components as

shown in Figure 4.1. Each of these languages is specialized for one separate concern during the development process. They target different involved roles in developments and include the expressions in their syntaxes that the target roles are familiar with. Low-level implementation details that can be accomplished with general-purpose programming languages are left out. An instance model described by one of these languages represents a specific software component that generates simulated environments for a simulation application, from a view corresponding to the used language. The CIM-layer language Simulated Environment Description Language (SEDL) is used to describe what kinds of simulated environments should be produced by such a component in a structured manner. The other three PIM-layer language models are used to describe the configuration schema, the data models and the computation flow of such a component, respectively. They are mainly specified as UML Profiles.

The main reasons to use these languages in the proposed framework are to bridge various views and to enable automation of transforming software descriptions at a higher abstraction level to more concrete ones. These language models have nested structures and serve as the SEDL Core Language Model component within the proposed framework, which are introduced in Subsection 4.2.1 in more detail. The metamodel of the SEDL is mapped to the PIM-layer language models via transformation rules. These rules play the role of the operational semantics of SEDL. Thus, executing a software description in SEDL should perform these transformation rules to produce three models, each of which is expressed by one of the PIM-layer languages. These three models are inter-related and represent the same component as described by the SEDL description, at an abstract level that is independent of the implementation programming language.

A brief overview of these language models is given in the subsections of Section 4.1. The later sections in Chapter 4 introduce where these languages are placed in the framework architecture and how they work in the development process using the framework.

4.1.1 Simulated Environment Description Language (SEDL)

Purpose: SEDL is a description language for expressing functionality requirements. It supports expressing the context of simulated environments that should be provided by environment components in simulations under development and functional behaviors of these components in simulations. An SEDL description serves as the input of transformations to generate a set of software models that can be further turned into implementation code or code skeletons. It is the anchor language based on which the proposed framework is built and serves as the start point to develop simulated environment components using this framework. Language users can scratch a description of required simulated environments in SEDL.

Perspective: the syntax of SEDL is designed to be understandable by people without software development or data structure expertise. It provides an intuitive description structure for simulated environments in spatial-aware simulations from a human observer perspective. It supports describing expected compositions of simulated environments required by systems of interest in simulations, as well as changes that phenomena in these environments should exhibit during simulations. A description in SEDL does not express a specific environment with a fixed evolving path. It describes all possible behaviors of an environment that a component should be able to produce, denoting conditions that should be modifiable to users for different runs. More modeling principles of SEDL are introduced in Subsection 5.1.1 within its specification.

Humans conceptualize a spatial phenomenon based on its properties or behaviors which they are interested in. A specific property or behavior type is associated with a specific conceptualization at a certain level of detail. Thus, humans may switch conceptualizations implicitly when describing multiple aspects of an identical phenomenon. Besides, aspects of a phenomenon type that are of interest to the simulation decide functionalities of software that produce or record phenomena of this type in the digital form. These functionalities require certain kinds of data structure and operations in software. While the aspects of interest are clear, fixing one conceptualization is meaningful. For instance, databases using the “moving object” concept shall support recording and querying movements of spatial objects and their topological relationship[87]. In contrast, SEDL is intended to be used to develop different software programs that may require software artifacts based on different conceptualizations. The most suitable one

cannot be foreseen when specifying the language model. Due to the above-mentioned relations, SEDL does not fix a conceptualization to describe environmental phenomena. Instead, it supports classifying changes of phenomena required by simulation scenarios at the system analysis phase based on which software artifacts are derived in the following development phases.

Relations to other DSLs: the grammar model of SEDL locates at a higher abstraction level than the one at which the other three meta languages locate at within the framework as will be explained in 5.1.1.1. The language specification of SEDL is defined with operational semantics in terms of transformation rules. These rules map the abstract syntax model of SEDL to the models of the other three meta languages. The other three languages are often referred to as output metamodels/meta languages in the following text. The abstract syntax models of the output meta languages can be viewed as parts of its operational semantics specification. Transformations of an instance model in SEDL results in a set of instance models described by the other languages.

Target users: SEDL is intended to be used in the system analysis phase. First, it is used to document and communicate the high-level functional requirements of software components. Component users can describe in SEDL their expected environment to be generated by a software component. Developers can also write the description according to the discussion with users. The resulting descriptions can be checked and confirmed easily by involved roles. More importantly, an SEDL description is a half-formal model, which enables its automatic transformation to more concrete models. In a realization of the proposed framework, executing an SEDL description will provide developers with software models or code skeletons of the component under development described by this SEDL description.

4.1.2 Configuration Schema Description Profile

Purpose: Configuration Schema Description Profile is a description language specified as a UML Profile. It is used to describe user-software interfaces. An instance model of this profile is a configuration schema for a software application, which contains groups of configurable parameters exposed to users. In the proposed framework, this profile serves as one of the output metamodels of the SEDL description execution or the input metamodel of transformations to platform-specific user interfaces.

Perspective: this language is purely declarative. In the proposed framework, an instance model of this language describes a user interface to configure a simulated environment component from a data model perspective. Second, the structure of this instance model is aligned with the structure of simulated environments that is described by a corresponding SEDL description.

The first point means the abstract syntax of this language is not a model denoting how an input field of a parameter should be visually presented to users. It focuses on the content which can be set by users and be passed to the back end of the component under development. A concept in the profile either presents a parameter or a group of parameters with a certain structural pattern. For example, a configuration model shall state a parameter about the wind speed called “initial”. Then the configuration interface should allow users to set different values for “initial”. However, the model does not state that these values must be set through a certain GUI entity, e.g., a textbox. This prevents user interfaces from being restricted to a specific visual representation. An additional mapping layer can be established to transform an instance model described by this profile to a graphical or textual configuration editor in some specific platform. This point will be brought up later in Subsection 4.2.1.

The second point means a configuration model described in this profile is organized in the same structure which corresponds to an SEDL description. This configuration model is supposed to get its hierarchy when it is derived from that description. It also gets names of elements and characteristics in the SEDL description. It keeps the same structure when users form the simulated environment in their minds to describe their requirements. Thus, the resulting user interface can be easily understandable.

Relations to other DSLs: a model described by this profile is supposed to be initially generated when executing an SEDL description. Links exist between this model and a model described by the metamodel of environment computation. The latter is generated from the same SEDL description. A component developed from that computation model consumes an instance configuration to initialize settings for a

simulation run. Modifications of the initial generation from either side of these two models should retain these links.

Target Users: this language is of interest to front-end developers who should develop an interface that allows users to interact with the component under development. They should bind a configuration model (i.e., an instance model of this profile) to a user interface of the component. They shall map this profile further to suitable visual representations to accelerate UI development. Though a developed UI, users can add or remove a phenomenon for a simulation execution conforming to the cardinality restriction, assign values to parameters, etc. Names of elements within the configuration model are exposed to users to denote which values they configure.

4.1.3 Simulated Environment Structure Profile

Purpose: Simulated Environment Structure Profile is a language model to describe abstract data structure[124]. It serves as one of the output metamodels of the SEDL description execution, as well as provides some PIM-layer constructs for developers to describe application-specific structural models. It may also serve as the input metamodel of transformations to the platform-specific models.

Perspective: this profile is also used to describe structural models for components that produce simulated environments in simulations. Different from the configuration profile targeting user interfaces, this language focuses on the back-end data structure of the component programs. An instance model in this language reflects how characteristic values of environmental phenomena are organized and stored within programs. It contains classes to store values of environmental entities during computation and classes representing data objects that being sent to systems of interest from the environment components. Objects modeled by this model are created and updated during component runtime.

An environmental phenomenon described by this profile is essentially a complex datatype with multiple aspects. These aspects are represented by geometry datatypes for its spatial representation and non-spatial primitive datatypes for other properties, respectively. The spatial representations covered by this language model are based on existing data models and simulation researches. This enables seamless mapping from this language to encoding libraries which are based on these works.

Relations to other DSLs: a model described by this profile is supposed to be initially generated when executing an SEDL description. A behavior model described by the metamodel of environment computation generated from the same SEDL description is associated with it. Links between these two models must be retained when modifying the initial generations. Computation classes implemented from the behavior model are in charge of creating, updating or destroying objects from corresponding datatypes in this model.

Target users: this profile and its instance models are hidden from component users. It is mainly for developers who implement the component back ends. Developers should turn a generated model described by this profile to functional code. They can benefit from general code generation facilities to get model code skeletons from this model. They shall modify the generated model to optimize the software design before the code generation. Further, they shall develop or utilize default implementations for spatial representations of meta types in the language metamodel (i.e., stereotypes in this profile) and reuse it for all model elements of the same meta types.

4.1.4 Metamodel of Environment Computation

Purpose: the metamodel of environment computation describes behavioral models of simulated environment components. It serves as one of the output metamodels of the SEDL description execution, together with the other two introduced in the previous two subsections. It also provides some PIM-layer model constructs for developers to express application-specific behaviors of the component under development. It could also serve as the input metamodel for computation flow code generation.

Perspective: different from the previous two output metamodels which focus on structural aspects, constructs of behavioral models are process-oriented. An instance model presents computation processes

to compute states of a simulated environment at a simulation step and simulation routines in which the component under development participates in.

Meta elements that express such models come from several sources, which include basic UML behavioral metal elements, behavioral elements owned by stereotypes in the structural metamodels, and action stereotypes defined in the behavioral metamodel specification. They are all viewed as part of the behavioral metamodel used in this thesis at the PIM layer. The detailed composition is summarized in the specification in 5.2.3.1.

Relations to other DSLs: a model described by the computation metamodel is supposed to be initially generated by executing an SEDL description, together with two other output models in previously introduced languages. These three models together present a software design model. The computation model has associations with both the other two models. Modifications of the generated models should not break these associations. During the runtime of a developed component, the implemented computation model gets its initial setting from an instance of the configuration model and operates on objects that are the instances of the data structure model to produce outputs.

Target users: this language and its instance models are also designed for back-end developers and thus are hidden from component users. Developers should turn an instance of the computation metamodel into a sequence of computation units in code. This shall be assisted by code generation facilities in an implementation of the proposed framework as explained in Section 4.3.

The body of each computation unit can be individually implemented based on mathematical formulizations provided by modelers of environmental phenomena. Thus, instances of the computation metamodel also serve as communication media between software product developers and environmental phenomena modelers, when these two roles are taken by different persons. Modelers do not need to bother with the structures that are specific to programming languages, but only the computation logic within computation units. Since the input parameters and output types of these units are fixed in a computation model, modelers shall identify the corresponding parameters and types in their formulas. Product developers then can turn the formula into the method code with correct input and output to chain the computation process correctly.

4.2 Build Software Applications as Computer Languages

The proposed framework emphasizes building software following the structure of an executable computer language. This strategy brings two benefits. First, developed meta languages in the framework can be more comprehensive used than merely serve as description syntaxes. Language users do not only benefit from the high-level grammars of these meta languages, but also components of computer languages that implement their operational semantics. Second, the realization of software can use the basic language infrastructure generated by workbenches (see Subsection 3.2.3) from language models, which ease the implementation in practice.

In the following text, Subsection 4.2.1 presents how the architecture of the proposed framework follows such a structure, followed by Subsection 4.2.2 explaining that how a simulated environment component developed within this framework is coordinated as a set of computer language components.

4.2.1 Development Framework as IDE of SEDL

At the architecture level, components of the proposed framework are organized in the way to build an IDE of the anchor language SEDL. Thus, a realization of this framework should provide an infrastructure to write and execute SEDL descriptions. The logical components of the framework are shown in Figure 4.2 and explained below, among which the **SEDL Core Language Model** and the **Basic SEDL Tooling** are the mandatory components for a minimal realization.

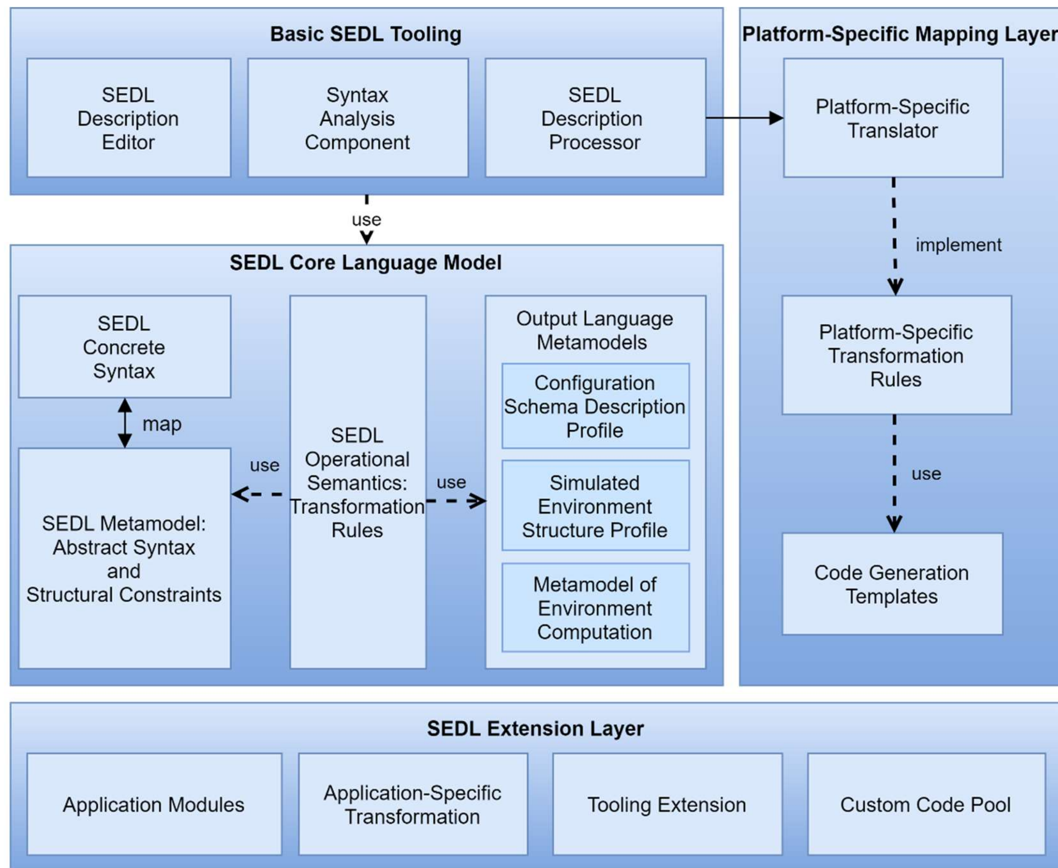

Figure 4.2: Logical Components of the Proposed Framework.

4.2.1.1 SEDL Core Language Model

The **SEDL Metamodel** is the grammatic model for creating descriptions of analysis-level requirements of simulate environment components. It is defined in the core SEDL specification using UML terminology to keep implementation neutral. The **Abstract Syntax** of SEDL provides concepts and structures to describe simulated environments, whereas its **Structural Constraints** provide constraints that a validate description must conform to. The **SEDL Metamodel** is a domain model that does not represent software structure. One class in SEDL should not be confused with one class in a software program. Rather, it corresponds to a set of artifacts in software. Classes in SEDL metamodel shall be used as software model classes only when implementing the SEDL tooling.

SEDL Operational Semantics are essentially transformation rules from SEDL descriptions to software design models at the platform-independent layer. While the input side of the transformation conforms to SEDL Metamodel, outputs of the transformation are expressed by the **Output Language Metamodels**. It comprises models of the other three meta languages summarized in Section 4.1. These models can be viewed as being nested in SEDL and are also defined in the core SEDL specification, mainly in the form of UML Profiles. The design of language models in the proposed framework follows principles of the strict multi-level metamodeling as introduced in Subsection 2.1.3. Artifacts related to simulated environments can be derived from these models. These language models, their instances and derived artifacts, relations among them, as well as their relations to general modeling languages are shown in Figure 4.3. This figure also shows their position in metamodeling levels and MDA layers.

Both the **SEDL Metamodel** and the **Output Language Metamodels** are designed at the M2 meta level. This pushes the general modeling language, i.e., UML in this case, further to the M3 meta-meta level. The SEDL syntax describes simulated environments. It results in high-level domain models of simulated environments in the human view, which are CIMs. In contrast, output meta languages describe models of software systems at the logical level, which are PIMs. Mapping rules between these two sides

which constitute the **SEDL Operational Semantics** are established at the M2 level. An SEDL description is an instance of the **SEDL Metamodel** and locates at the M1 model level.

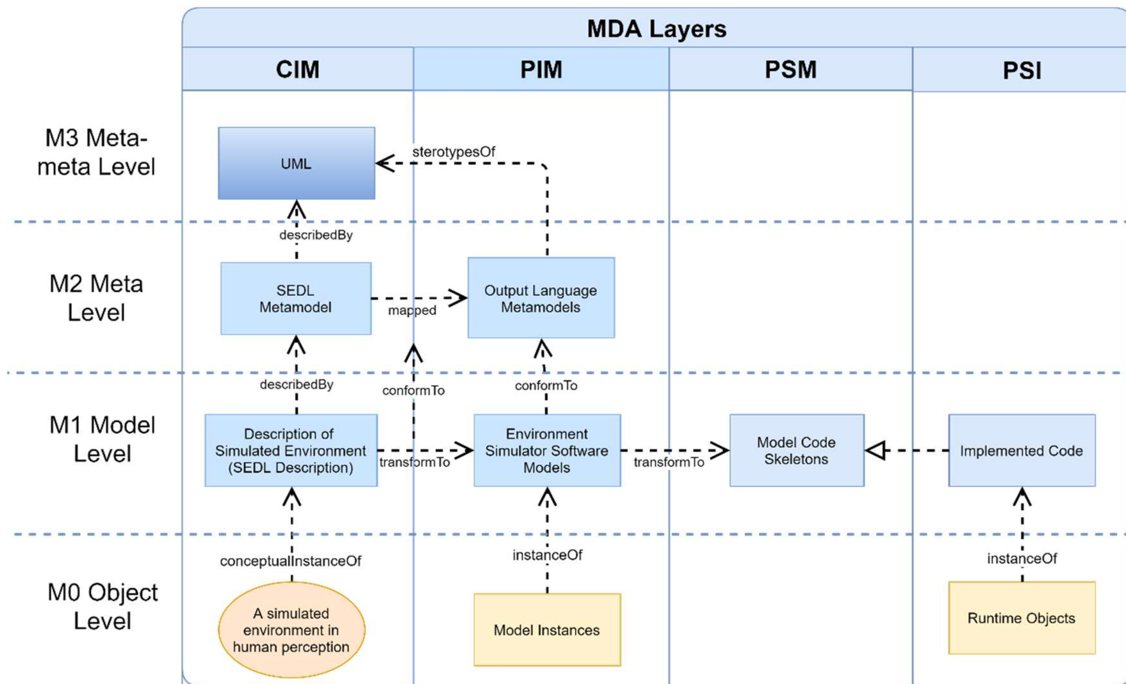

Figure 4.3: Model Related Artifacts Correspondence to MDA and Metamodeling Levels.

This M1 model is essentially a conceptual description of a set of possible simulated environments that need to be created by a component under development and how they should be provided to the system of interest component during simulation executions. These environments have structures and property types in common, while may differ in the qualities of their properties. Thus, an M0-level object at the description side corresponds to a simulated environment in a specific simulation run in human perception. It should be made clear that an SEDL description does not need to be further instantiated in an information system. Thus, the SEDL descriptions at the M1 level are the end instances at the CIM layer in the framework.

At the M1 level, an SEDL description is transformed into a set of software models of an application that produces simulated environments. These output models conform to the output metamodels. In other words, they are the instances of corresponding metamodels. The transformation at this level follows the M2 level mapping rules. The M0 level of the output side locates runtime objects of these M1 models.

The SEDL specification is presented in Chapter 5. To keep the proposed framework independent of implementation, this specification provides neither a concrete syntax for SEDL descriptions nor a concrete syntax for the output meta languages. However, concrete syntaxes of these languages are mandatory to realize any language tooling with which users are can create and modify instance programs of them. The **SEDL Concrete Syntax** could be either in textual source code style or in graphic notation which is similar to the underlying UML-based grammar model, while the UML graphic notation and its exchange format XMI can be applied for encoding the output software design models at PIM layers.

4.2.1.2 Basic SEDL Tooling

The language tooling is implemented based on SEDL language models. The **SEDL Description Editor** is used to create and modify description programs that conform to the **SEDL Concrete Syntax** being used for the framework implementation. This editor is integrated with the underlying **Syntax Analysis Component** to provide editing support functions such as syntax highlighting and validation. The composition of this component depends on the type of implemented concrete syntax.

The editor should also implement save functions to store created SEDL descriptions in files and reopen them for further modification or execution. Since the **Abstract Syntax** of SEDL is defined in terms of UML, it is recommended to also implement the save function for storing a parsed version of these programs encoded in the UML interchange format XMI. Thus, the written programs can be used in different implementations of the proposed framework, as well as other UML-based modeling tools.

In addition, the editor should have an easy interface that allows users to pass an SEDL description to the **SEDL Description Processor** for execution. This processor implements the **SEDL Operational Semantics**. An execution process for a minimal realization of the proposed framework is a model-to-model transformation. It produces and saves XMI-encoded models conforming to the **Output Language Metamodels**. This transformation can be combined with an optional platform-specific mapping which is introduced in the following subsubsection.

4.2.1.3 Platform-Specific Mapping Layer

This layer a worth-to-have optional component that extends the basic tooling with a **Platform-Specific Translator**. It implements the **Platform-Specific Transformation Rules**. Models described by the output meta languages are independent of specific implementation technologies. This translator further maps these models to some specific implementation platform, e.g., the programming languages used in a development team. For example, an output model from the basic tooling may have a class with a property "a". When the implementation technology is set to be JavaBeans[6], this class can be further mapped to a Java class with a private field "a" and public getter/setter methods to access this field.

This layer usually involves code generation facilities. It can be implemented by model-to-code transformation languages as **Code Generation Templates** or make use of existing UML-based code generation tools. In this case, the execution of an SEDL description in the combined toolchain of basic tooling of SEDL and the platform-specific mapping layer produces code skeletons as the PSMs at the M1 level as shown in Figure 4.3. This is especially recommended for computation model transformations, while computation models enclose application-specific behaviors of a component that needs to be implemented by developers. PSM-layer outputs provide developers with architectural code which they can fill in these application-specific behaviors. To facilitate the PSM code generation, transformation details specified in Chapter 5 express the output models in a structural view in terms of operations owned by activity classes and model elements relevant to it.

4.2.1.4 SEDL Extension Layer

This thesis focuses on developing new components rather than reuse of developed components. Also, the SEDL core language is designed to be independent of specific scenarios or phenomenon types. This strategy keeps the core language having a concise size and being easy to use. Further, it ensures that the core language to be applicable in different applications. However, a collection of developed simulated environment components emerges and evolves during long-term developments. This results in an extension layer of the implemented framework as shown in Figure 4.2.

For groups whose work is dedicated to specific application areas, frequently appeared environmental phenomenon types shall be identified within a certain application area, e.g., sea wave in a marine environment. It is worth to add concepts to describe these reoccurred phenomenon types as **Application Modules** of SEDL. Besides, the implemented code in the **Custom Code Pool** to produce specific environmental phenomena has the potential to be reused. Different from other components of this framework, this layer is not a self-contained component but a collection of application artifacts. To reuse the implemented code of an application module, the corresponding **Application-Specific Transformation** and the **Tooling Extension** need to be realized. While this is beyond the focus of this thesis on the core SEDL-based specification, Section 8.4 in the discussion chapter briefly provides a conceptual design of the integration architecture that integrated these artifacts into the core framework infrastructure.

[6] https://www.oracle.com/technetwork/articles/javaee/spec-136004.html

4.2.2 Simulated Environment Specification by Configuration Language

The proposed framework guides the development of simulated environment components to reach software products as syntax-directed applications[125]. A simulated environment component is organized following the structure of a light-weighted configuration language.

This strategy is achieved by CIM-PIM transformations from SEDL descriptions. Models that locate at the PIM layer and the PSM layer in the proposed framework take the view of software systems. The basic structure of a simulated environment component under development is first provided by the generated PIMs at the M1 level. Output PIMs transformed from an input SEDL description include three inter-related models, i.e., a user interface model, a data structure model and a computation process model. They represent together the preliminary design model of the component expressed by the input SEDL description. This set of models is coordinated in the way to provide functionalities to read and process a light-weighted application configuration language as shown in Figure 4.4. Refinements and further transformations should maintain the fundamental structure. More details are explained below.

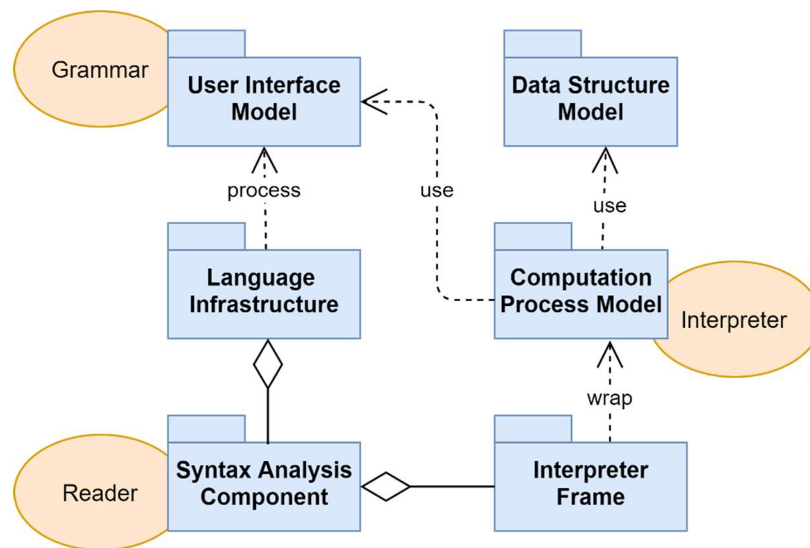

Figure 4.4: Developed Components as Syntax-Directed Applications.

Grammar: the generated UI model is a configuration schema, which corresponds to the grammar model of a light-weight configuration language for simulated environments. The structure of this model is aligned with the input SEDL description and its terms are transformed from the chosen names of elements in this description. Both are expressed by component users or with the involvement of component users which ensures its understandability. In a realization of the SEDL, it is recommended to pass this model to a language workbench to create basic language infrastructure as reviewed in Subsection 2.2.3, which provides a frame to integrate the three output models together. The infrastructure has various components that can process instances of the configuration schema.

Reader: a reader parses information written in a certain grammar and performs post-processing on parsed objects to translate them into the inner data structure of the software. Thus, this term roughly refers to the scanner, the parser and the post-processor of a computer language. Here, the reader corresponds to the syntax analysis component in the language infrastructure created based on the configuration schema as mentioned above. It wraps methods that parse an instance of the schema and support functions that initialize a computation process according to a configuration instance. The initialized computation process is an instance of the computation process model.

Interpreter: the computation process model corresponds to an interpreter. It provides functionalities that consume a configuration instance, initialize necessary instances of the data structure model and compute simulated environments during simulations. The generated computation process model includes computation classes and computation flows associated with it. In a realization, this part of the models

can be wrapped by an interpreter skeleton in the language infrastructure generated from the configuration model as introduced above.

An advantage of this strategy is that it enhances the usability of developed components. By applying a syntactic model whose terms and structures are derived from user requirements to a configuration interface, component users are assisted with the intuitive vocabulary support in the same way they used to describe the to-be produced environments. Further, treating the configuration schema as a grammar model can benefit from language development facilities to create frames that bind different parts of component models together. Manual development work can be reduced through this strategy.

4.3 Development Process with the Proposed Framework

This section provides recommendations for aligning the development process of a simulated environment component with the framework. It presents each development phase that the framework contributes to, starting from determining functional requirements way down to getting executable code ready in chosen programming languages. These development phases chained together through transitions of intermediate artifacts among them to construct a whole development process with the framework. Figure 4.5 provides a non-normative simplified overview of this process with the emphasis of the transition flow among different components. The solid arrows show the flow that the information or artifacts being passed between two components. Actions other than simple passes are labeled near corresponding arrows. In practice, each step shall be iterated to adapted to real software development lifecycles, which will be summarized in Subsection 4.3.4.

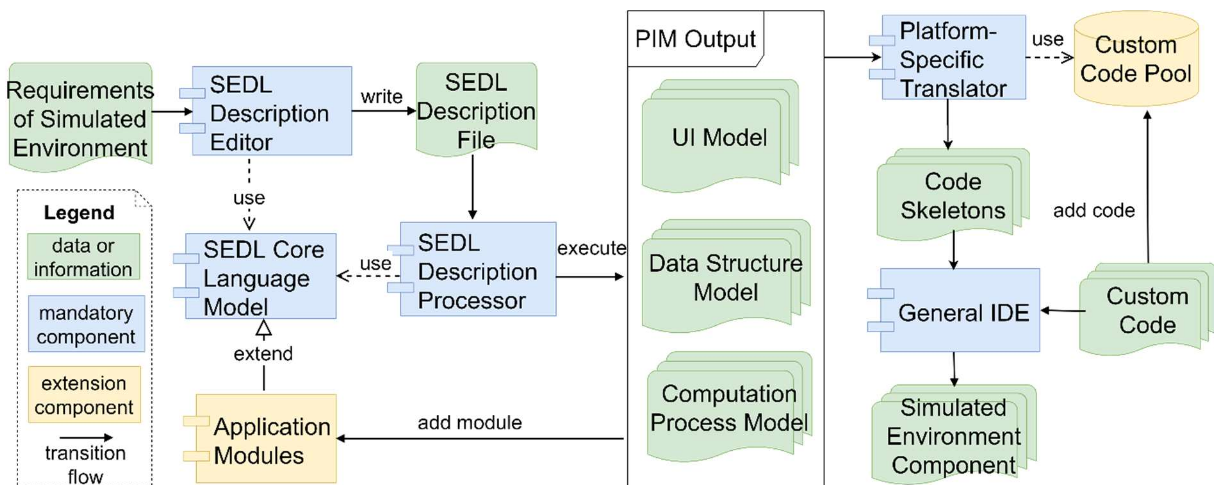

Figure 4.5: Development Process with Processed Framework.

This development process is grounded on MDD. In the following sections, involved components and the main working goal of the corresponding phase are summarized at the beginning with respect to MDD. Then, situations of using domain-independent solutions are briefly described, followed by comparison to the proposed framework for explaining the specialty of this framework. Finally, tasks that should be performed in this phase with the framework, as well as the types of artifacts produced for the following phase, are presented.

4.3.1 System Analysis

In the System Analysis phase, the **SEDL Metamodel** and the **SEDL Description Editor** are involved. The goal of using the framework at this phase is to determine the functionalities that a simulated environment component should have in the form of an SEDL description (located at the M1 level of the CIM layer as illustrated in Figure 4.3).

In a conventional software development lifecycle, required functions are often recorded in informal documentations. When it comes to an MDD process with domain-independent modeling tools support,

System Analysis often produces high-level use cases and business activities diagrams as CIM models to document these requirements, as summarized in Table 2.1. General modeling languages do provide constructs for structuring informal messages to a certain extent to create half-formal models at this layer. However, model elements expressed by them shall have very different levels of detail since these constructs allow to express context in any application domain. For instance, actions in a high-level business activity flow may range from a customer making an order to the system popping up a warning message on a screen. Information that reflects the domain context may be expressed by any natural language terms and can hardly be systematically transformed. Such models require manual interpretation when being used in further development phases. Besides, communication difficulties caused by different domain terminologies used by roles from different backgrounds may remain.

Within the proposed framework, SEDL descriptions play the role of functional requirement models at the CIM layer. Different from the domain-independent languages, the **SEDL Metamodel** provides constructs with limited expressiveness. Each of these constructs does not allow arbitrary context, but only allows expressing a specific type of context about simulated environments. This metamodel plays two roles in the analysis phase.

On one side, it regulates communicating terminologies and structures to express a CIM of simulated environment components. During the analysis, involved human roles sketch and exchange opinions to determine functionalities of the component under development with the support of the SEDL syntax. The SEDL metamodel is specified with terms based on common-sense perception to provide a common ground for various roles with different levels of technical skills to exchange the information. In principle, an SEDL description expresses the following aspects of simulated environments that should be produced by a component under development:

1. **Composition**. What kind of phenomena should be included in the simulated environment? Which relevant characteristics of them should be included in the digital representation? How should such a phenomenon occupy the virtual space at a time instant?
2. **Variations.** What kind of changes that the simulated environment may occur in space and time, e.g., movement in two-dimensional space, increase/decrease of a global property value, value difference over space? Which phenomena and properties are involved?
3. **Execution Conditions**. Which aspects of the included phenomena should be controllable by users to set different scenarios? In which condition information of environments should be computed by software during a simulation? How should the computed information of environments be provided to a system of interest component at a simulation step?

On the other side, it serves as a communication vehicle that wraps functional requirements to be passed to the next development phase. By the end of the analysis phase, the required context of simulated environments is written in an SEDL description file via the **SEDL Description Editor**. This description is passed to other components that perform tasks in further phases. By doing this, the model transformation process within this framework is triggered.

4.3.2 Software Design

The Software Design is the development phase in which the meta languages summarized in Section 4.1 are most heavily involved. At this phase, the goal of using the framework is to produce a set of software design models (located at the M1 level of the PIM layer as illustrated in Figure 4.3) described by output meta languages. The tool used to perform tasks in this phase is the **SEDL Description Processor.** It is related to the **SEDL Core Language Model**, as well as available extensions of the model.

Aligned to the MDD, CIMs will be turned to PIMs at this phase. In a domain-independent solution, due to the heterogeneities of conceptual structures in a potentially infinite number of domains, it is hard to determine the correspondence to software structures for an arbitrary CIM. In this case, CIMs resulting from system analysis often do not have sufficient information to be automatically turned to a software design model at the PIM layer.

Different from a general MDD solution, the proposed framework contributes to the Software Design phase by enabling automatic domain-specific transformations from CIMs to PIMs. An SEDL description from the System Analysis phase serves as the input of the design phase. As have been indicated, it is a

functional CIM. The first step of the design phase is to pass this description to the **SEDL Description Processor** for transformation. The processor parses it to an abstract syntax tree following the SEDL abstract syntax definition and translates it to a set of software design models described by output meta languages. This automatic transformation is built on the following theoretic foundations. First, the **SEDL Metamodel** has restricted the way to construct an SEDL description. Second, **Output Meta Languages** provide profiles to describe the component design models with a finite number of constructs. Third, the previous two finite sets enable establishing well-defined mapping rules between them, which ensures unambiguous transformation outputs from items in an SEDL description.

Outputs from the automatic transformation already have sufficient details to be passed to the next phase. However, in practice, the automatic transformation is usually followed by one or more rounds of manual refinement. During the refinement, initial generated models are optimized. For instance, artifacts for satisfying detected missing requirements and for managing non-functional concerns shall be added. More specific artifacts may replace general ones with the knowledge of computation logic at this step.

As will be seen in the specification of meta languages in Chapter 5, PIM-layer metamodels also provide some domain-specific constructs to support developers to model more application-specific behaviors since the PIM layer. They are used for describing behaviors that are not formally derivable by automated transformations from CIMs. These constructs are specified as behavioral elements carried by other model elements derived from CIMs. Developers can invoke these constructs through its belonged model elements to model behaviors. In practice, it is recommended first to transform the PIMs into a PSM version and begin to use these constructs at the PSM layer for implementation. Since at either layer, these behaviors are supposed to be brought in by the same roles, this strategy avoids unnecessary complications in PIM-PSM transformations.

The proposed framework emphasizes the use of software prototyping[126], [127] during the design and implementation phases. Instead of forming a simple sequence, iterations of these two phases exist in practice. All intermediate versions of the design models during the optimization are described by the output meta languages. Each of them or parts of them can be passed to the next phase, i.e., **Implementation**, for prototyping and refinements of prototypes. The initial generated design models are passed as the base for the semi-automatic generation of the first version of the prototypes.

4.3.3 Implementation

In the Implementation phase, the design model is turned to an executable program by chosen technologies (located at the M1 level of the PSI layer as illustrated in Figure 4.3). At this phase, the component in action is the **Platform-Specific Mapping Layer**. Its main job is to transform design models at the PIM layer to PSMs.

Automatic transformation in MDA usually starts from PIMs to PSMs. Transformations from PIMs to PSMs, especially for structural models, are relatively straightforward compared to transformations at higher abstraction layers. Context-wisely, models at the two sides of a PIM-PSM transformation have similar levels of detail. Such a transformation refines PIMs with support artifacts for a specific technology, which does not require domain-specific knowledge. Thus, transformation automation between the PIM layer and the PSM layer is feasible to realize at the domain-independent range. It often becomes a key feature of a general modeling tool based on MDA standards.

An example is the default code generation facilities of EMF. The same model shall be used at different MDA layers when serving for different purposes. In an EMF-created application, underlying Ecore models often serve as a domain model that can be used for different technical platforms.[35] However, at the stage of creating tooling for this application, the default model code generator provided by EMF uses Ecore models as software models at the platform-independent layer. As Section 3.1 points out, a transformation from the CIM layer to the PIM layer turns a model from the conceptual business view to the logical information system view, which is seldom a one-to-one mapping process. In contrast, the default code generation of EMF only adds Java-specific facilities to existing Ecore elements, e.g., access methods for private attributes and factory classes. For instance, when an Ecore Class “Book” is transformed into a Java Class, the resulting Java class will have an “author” attribute if the “author” attribute is explicitly defined in the Ecore class. This process does not add new domain-specific

information into the Java classes. Thus, it is more a transformation that refines models from the PIM layer to the PSM layer. Modeling languages provide rules to structure a PIM here.

Readers may notice that generated code in the above example is referred to as "model". The reason is given as follows: although Figure 4.3 illustrates the platform-specific model and the implementation (PSM and PSI) separately for clearance, the boundary between these two is not rigorous in practice. Transformation facilities in domain-independent modeling tools often generate code of interfaces together with a default implementation, such as the EMF case. Nevertheless, this transformation cannot catch the application-specific functionalities needed for the final application. The main goal for this code generation is still to provide platform-specific models, which are presented in the form of skeletons of generated code.

Since sophisticated PIM-PSM transformation tools have been developed in domain-independent modeling platforms as explained above, the proposed framework is designed to be integrated with existing technologies, especially for its **Platform-Specific Mapping Layer**. The metamodels of PIM-layer languages are defined based on UML Profiles recommended by MDA. Thus, instances models of these metamodels, i.e., the output models from the design phase, are UML-based. They are compatible as the input of model transformation and code generation facilities, which are also based on UML.

Once the PIM-layer models have been derived from SEDL descriptions, they shall be fed to domain-independent MDA-based modeling tools, to generate PSMs according to the tool implementation. These tools serve as the **Platform-Specific Translator**. This translator, however, can be enriched by adding missing functions of an existing tool, especially for the behavioral model transformation and transformation to structures implied by stereotypes, as specified in Chapter 5.

It is recommended that the automatic generated PSMs are encoded in the form of program code skeletons. Under this condition, the Implementation phase comprises iterations of two steps. First, a version of design models or some part of them from the Software Design phase is passed to the Platform-Specific Translator for execution. PSMs are generated as code skeletons through the execution. Then, developers fill the function body of these code skeletons to provide the executable program. During PIM-PSM transformations, domain-specific constructs for modeling application-specific behaviors are transformed, e.g., in the form of operations owned by a class in an object-oriented language. Thus, in this step, developers can further use these constructs to implement application-specific behaviors.

4.3.4 Development Activity Flow with Iterations

This subsection provides a graphic summary of main activities during the development process as shown in Figure 4.6, which have been explained in previous subsections. The activity flow is presented in Business Process Model and Notation (BPMN)[128] and explained below. BPMN is an OMG standard that provides graphic notations to describe high-level business procedures.

This figure emphasizes the flow of tasks that a development team shall follow and the steps where user participation comes to play. To keep the focus, it only includes activities related to the proposed framework. It identifies developers and users of the component under development but does not distinguish more specific roles at each side for providing a clear overview. As have been addressed in Chapter 1, these two groups usually both belong to a bigger community that develops and uses simulations (as "Simulation R&D Community" in the figure). Assigning individual tasks of developing a specific model function are decided within the developer teams since the expertise of developers is different from team to team. The assignment can be supported by the generated models, which help to identify functional units of the component under development.

Subsection 4.3.2 points out that one benefit that the proposed framework contributes is to support rapid prototyping. In practice, the development process of a simulated environment component with this framework is not a simple linear sequence, but includes iterations of the introduced activities, as shown in Figure 4.6.

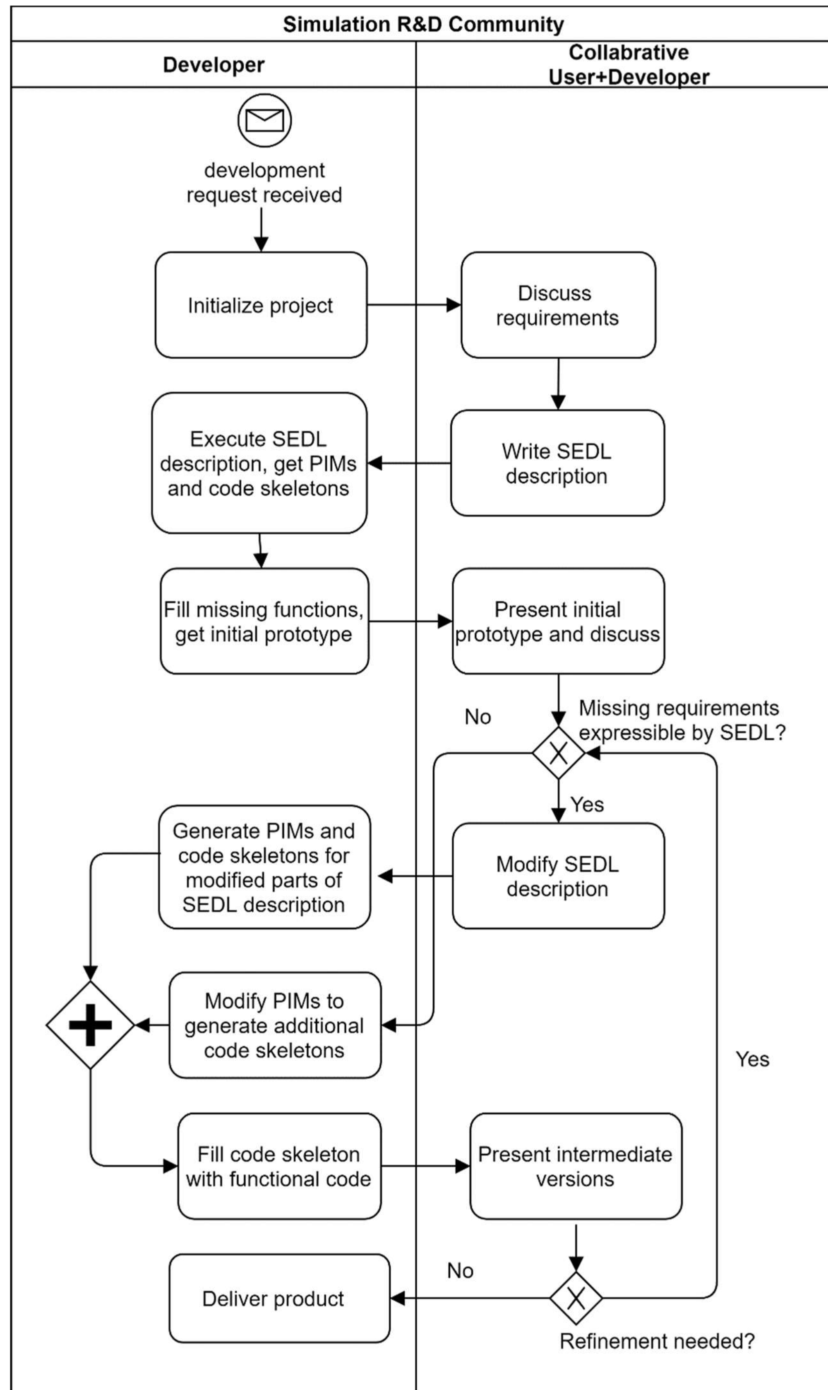

Figure 4.6: Main Activities in the Development Process.

The coding of the component shall start when the first automatic generation result from an SEDL description is ready. CIM-PIM transformations and PIM-PSM transformations shall be combined as a chain to create component skeletons. Further, the Platform-Specific Translator shall create default ready-to-run functional code from design models, e.g., using pre-assumed return values or an illustrative body for generated operations. An initial runnable prototype of the component can be provided to users at a fast speed to confirm the satisfaction of their needs and to identify missing functionalities. Then, the prototype is refined during the necessary iterations of previously introduced development steps. At the end of each iteration, an intermediate version of the component is presented to users to discuss if further refinement is needed and in which step the next iteration should start. This process continues until the component satisfies the requirements of the simulation.

5 Simulated Environment Description Language

The framework proposed by this thesis specifies domain-specific languages to describe simulated environment components at the CIM layer and the PIM layer. They enable domain-specific transformations during the MDD process using the framework. This chapter presents the specification of the anchor language of this framework, namely Simulated Environment Description Language (SEDL). The models of other languages in the framework are embedded in this specification. Section 5.1 introduces the language model of SEDL. Section 5.2 presents the models of the other three languages. They describe PIM models that are transformed from SEDL descriptions through the transformation chain in the proposed framework. The execution semantics of SEDL, which is a set of transformation rules, is presented in Section 5.3.

5.1 SEDL Language Model

This section presents the modeling principles of SEDL, as well as the specification of its abstract syntax and descriptive semantics. The specification does not fix a concrete syntax for SEDL, which leaves the freedom to different implementations.

5.1.1 Conceptual Modeling Principles

Before the SEDL model is presented, this subsection provides a brief introduction to the modeling principles of this model to help readers to understand the following subsections.

5.1.1.1 Level of Modeling

SEDL is a language used in the system analysis phase to describe simulated environment components of simulation applications under development. In this phase, high-level functional simulation scenarios are identified. These scenarios are further specialized to guide the design and implementation of final simulation programs. The major part of SEDL is specified to support expressing the context of simulated environments that forms a part of simulation scenarios.

As Figure 5.1 illustrates, a program in SEDL is a high-level description created in the system analysis phase of the development. It is an instance of the SEDL language model. In the following phases, this program should be turned to digital structures that represent the described environment and program functions to compute environmental data. These representations are implemented as a component of the simulation application with alterable parameters. By fixing these parameters, each execution of this application produces a concrete simulated environment instance of the simulation as a set of data objects.

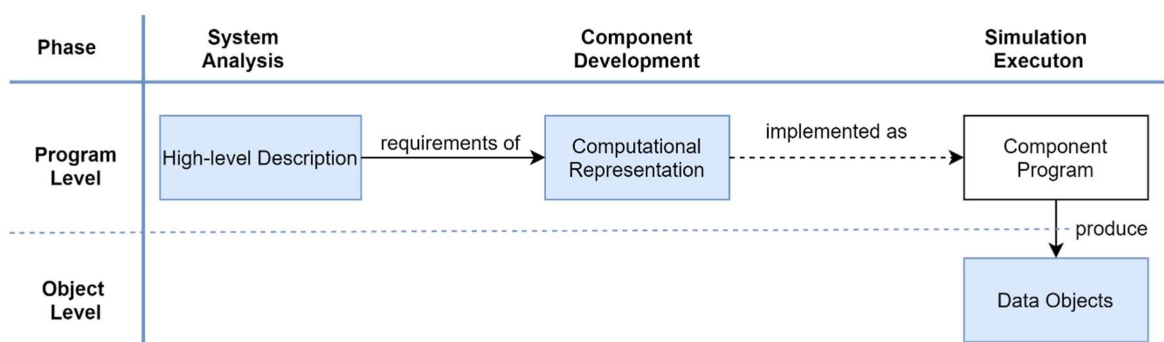

Figure 5.1: Forms of Simulated Environments at Different Development Phases.

Thus, an SEDL description locates in the same modeling level of simulation programs. A phenomenon expressed in this description corresponds to all possible simulated phenomenon instances that can be produced from the executions of a simulation program. SEDL descriptions express simulation

components and are transformed into models or code of these components instead of concrete sets of environmental data.

5.1.1.2 Perspective of Modeling

An SEDL description reflects the functional requirements of the simulated environment component in a simulation program from the system of interest modelers of this simulation. SEDL is introduced for developing components in complex spatial-aware simulations, in which simulated environments could be produced through a different paradigm from the paradigm used to simulate the system of interest component. The environment component may be developed by other specialties than the system of interest modelers. However, scenarios of a simulation depend on the goal for which its system of interest is modeled and should be decided by the system of interest modelers. The simulated environments in these scenarios should also match the expectation of the system of interest modelers. Thus, their expectation needs to be correctly identified, communicated and preserved in the development, for which SEDL comes to action.

An SEDL description expresses the environment component of a simulation from the user perspective in a computation-independent manner. Users here refer to modelers that use this component to provide the simulated environment to their system of interest component. It describes the context they expect from this component while neglects how the context is produced. This results in a domain model of described environments in the view of a human observer. It also describes high-level behaviors of this component which users are aware of, such execution routines, configurable options, etc.

Terms used by the SEDL syntax are based on common-sense knowledge. Creating SEDL descriptions does not require the expertise of environmental modeling, e.g., data structure or computation methods for the described phenomena. These descriptions are supposed to be communicated among roles with diverse expertise. The common understanding of used terms should exist or be easy to be established among these roles. This means that the SEDL model should ground on common sense and remain simple.

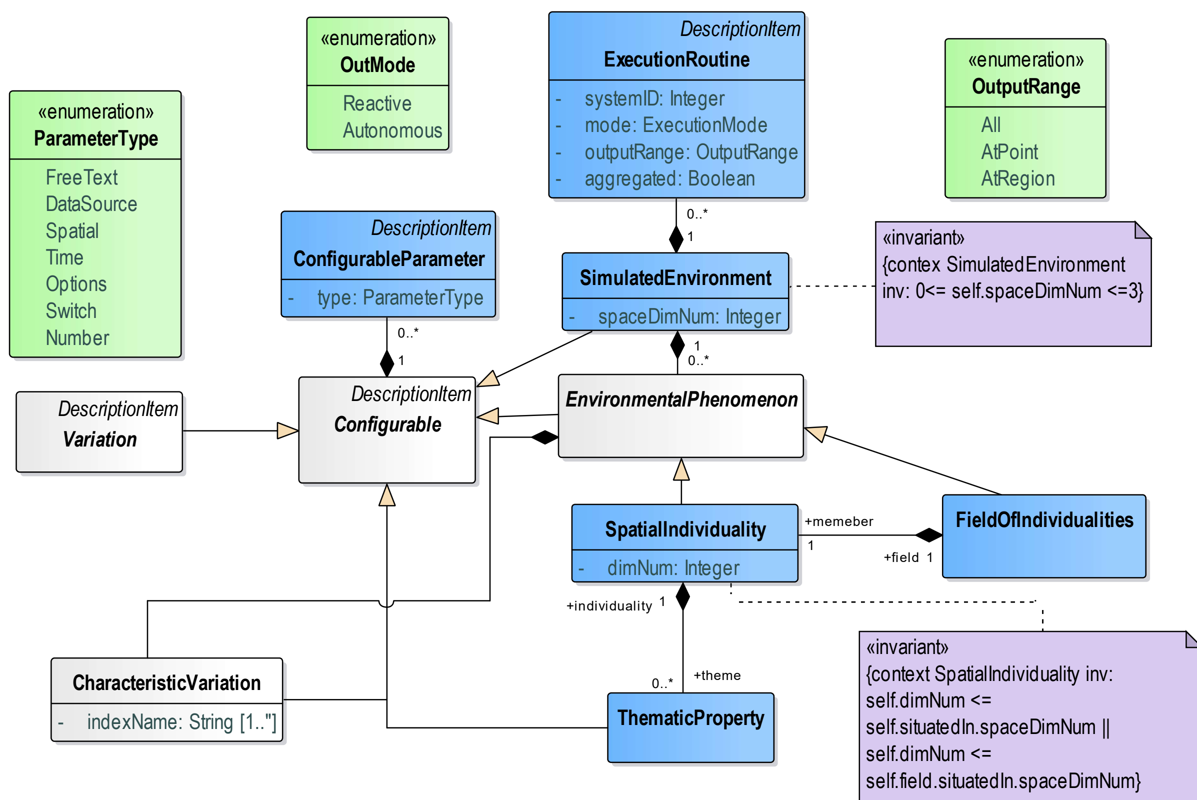

Figure 5.2: SEDL Abstract Syntax.

5.1.2 Abstract Syntax

The abstract syntax of SEDL is specified in UML to keep implementation neutral. It defines legitimate terms and the valid structure of SEDL descriptions. Figure 5.2 presents the abstract syntax that expresses the main structure of an SEDL description. Additional structural constraints are presented in OCL and will be explained in the next subsection. Other terms are omitted due to the limitation of space. They will be presented by figures in the next subsection.

A UML class in this syntax model represents a term (may also be referred to as a type in this specification) in SEDL. SEDL specification only introduces terms when they are conceptually different. The class names in the abstract syntax model are more used as identifiers for back-end processing. Elements in the abstract syntax trees of SEDL descriptions are instances of these terms. Relationships among these terms specify their instances' relations in a valid description. The syntax model should not be confused with a digital data structure model.

Several abstract terms are introduced into the SEDL language model for specification convenience. They are named based on common sense or conceptual approximation and thus should not be exposed to language users to avoid confusion. These terms are marked italic in the figures.

5.1.3 Descriptive Semantics

This subsection introduces the meanings of SEDL terms to guide the use of SEDL.

5.1.3.1 DescriptionItem and Configurable

All terms in SEDL are used to enclose pieces of analysis-level descriptions about computer applications. Not all aspects of applications can be formally captured at this level. An abstract term DescriptionItem is introduced to deal with this issue. All other terms in SEDL are subtypes of DescriptionItem and inherit its attributes. A DescriptionItem instance must have one attribute *name* of the String type to identify it in the SEDL description and transformed artifacts in the following development phases. Besides, it may, and is recommended to, have one attribute *description*. The value of this attribute is some text to express application-specific requirements that cannot be captured by the formal part of this instance.

Another abstract term Configurable is specified to facilitate the specification. It may contain 0 or more ConfigurableParameter as specified in Subsubsection 5.1.3.2.

5.1.3.2 ConfigurableParameter

Environment instances generated by a component can vary among executions. To adjust the output for various scenarios, users may want to have some control over the component to produce outputs that match some desired index values about the whole environment, some phenomenon, or some change patterns of a phenomenon. These indexes reflect the characteristics of the generated environment instances. They shall be some summarized descriptors, e.g., the lifespan of a phenomenon instance, the mean value of some property during a phenomenon's lifetime, etc. They may also be some special state, e.g., the initial value of the phenomenon location. Their values are used to initialize a simulation run. A ConfigurableParameter expresses such an index about a Configurable whose value should be set by users before an execution.

SEDL made all terms that describe composition and changes of the simulated environment as subtypes of Configurable. Thus, ConfigurableParameter-s can be related to corresponding Configurable-s in SEDL descriptions. A ConfigurableParameter is mapped to a parameter of the resulting simulation program that users can set through some user interface or a configuration file.

The *name* of a ConfigurableParameter inherited from DescriptionItem should be given by component users to identify the characteristic controlled by this parameter in the simulation program. Besides, it must have one attribute *type* that describes the form in which the parameter value should be set. This type

can be chosen from an Enumeration named ParameterType. Values in this Enumeration express parameter types in a way that is looser than datatypes in a programming language so that users do not need to precisely know the supported datatypes of the implementation platform precisely. The available options are listed in Table 5.1.

Value	Explanation
FreeText	The parameter is some text. This type is rarely used in practice due to its difficulty of being processed. It should only be used if no more restricted type can be identified yet.
DataSource	The parameter points a location of some existing data that users want to feed to the component.
Spatial	The parameter represents some location in space.
Time	The parameter represents some point or interval in time.
Options	The parameter is a condition whose status can be in one of several alternative options for one execution. Different options lead to different generation behaviors.
Switch	The parameter is a condition that can be set to be one of two statuses for one execution, such as true or false, on or off, yes or no, etc.
Number	The parameter is a numeric value.

Table 5.1: Available Options of ParameterType.

5.1.3.3 Composition of Environment

Terms specified in Subsection 5.1.3.3 are used to describe which kinds of phenomena should present in a simulated environment generated by a component under development. They are subtypes of Configurable.

SimulatedEnvironment is the entry term of an SEDL description that holds description items together as a standalone unit. A description within the scope of a SimulatedEnvironment expresses the user-required constituents of simulated environments and their possible behaviors, which should be produced by a simulation component under development, as well as high-level operational behaviors of this component in simulation executions.

By specification, a SimulatedEnvironment must have one attribute *spaceDimNum* that constrains the number of spatial dimensions in which the environment should be simulated. This attribute decides the number of coordinates needed to represent a spatial point in the described component. Its value can be the integer between 0 and 3 as explained in Table 5.2. When a working community has fixed the number of dimensions it intends to work on, this attribute can be predetermined in an implementation of SEDL and hidden from language users.

Value	Explanation
0	The spatial locations of the environmental phenomena are irrelevant to the simulation. The component simulates time series data without spatial locations.
1	Only the location along one direction, e.g., along a road, is relevant to the simulation. The component simulates phenomena data with one-value coordinates.
2	The space is abstracted as a projection from the three-dimensional physical world into a two-dimensional surface. The case is typical in spatial simulations at the geo scale that the altitude is negligible or treated as a thematic domain.
3	The representation of a spatial location should be composed of three-value coordinates to simulate a three-dimensional space. It is often the case in spatial-aware simulations about a small area that movements in all three dimensions are relevant.

Table 5.2: Possible Number of Dimensions of a SimulatedEnvironment.

The simulated environment described by a SimulatedEnvironment can contain phenomena of an arbitrary number of types. Each type can be described by an EnvironmentalPhenomenon contained in this SimulatedEnviornment. Besides, a SimulatedEnvironment may have zero or more ExecutionRoutine-s, each of which regulates the functional routine of the described component in a bigger simulation (See Subsubsection 5.1.3.9).

An EnvironmentalPhenomenon is a piece of description that describes a constituent element type of the simulated environment. When a type of phenomena should appear in an environment instance generated from the component described by a SimulatedEnvironment, an EnvironmentalPhenomenon should be added to this SimulatedEnvironment given a name to denote the phenomenon type. An EnvironmentalPhenomenon contains more-detailed description pieces about this type as specified in the following text of this section. The EnvironemtnalPhenomenon is an abstract term.

In practice, one of its subtypes, i.e., SpatialIndividuality or FieldOfIndividualities, should be used.A SpatialIndividuality encloses analysis-level descriptions about a phenomenon type that should appear in a simulated environment as an identifiable individuality or distinct substance in space, whose boundary may be recognizable or unrecognizable within the extent of the simulated environment. A phenomenon of this type behaves on its own during simulations.

By specification, a SpatialIndividuality must have one attribute *dimNum* which denotes the form of its described phenomenon's geometry required by the simulation. Its value can be one of the following integers: -1(non-spatial), 0(point), 1(line), 2(region), 3(3D volume). This value must not be greater than the *spaceDimNum* of the SimulatedEnvironment it is contained in, either directly or indirectly.

When only one or a few significant instances of a phenomenon type appear in a simulation execution and each of them is supposed to be configured and created independently, this type can be expressed by a SpatialIndividuality that is directly contained in a SimulatedEnvironment. For instance, the storm needs to be included in a simulation as the extreme environmental phenomenon that vehicles should avoid. Before an execution, users want to be able to configure one or two storm instances, each with a determined path and some constant moving speed. In this case, user expectation about how the storm should be can be expressed by a SpatialIndividuality contained in a SimulatedEnvironment.

A SpatialIndividuality can also be the *member* of a FieldOfIndividualities as specified in the following text. The containment relationship ensures that an instance of SpatialIndividuality is only in one of these two situations.

A FieldOfIndividualities expresses a phenomenon type that appears in a simulation as a swarm or a group of individualities of an identical kind. For each generated environment instance, individualities of this kind appear as an integrated whole with no significant single members when observing at the whole environment scale. Forms and behaviors of each individuality in this field follow the same set of regulations. All members in this field together exhibit some spatial patterns at a time instant. A FieldOfIndividualities must have exactly one *member* which is a SpatialIndividuality to express the spatial form, characteristics and behavioral modes of member individualities in an instance field of the described type. SEDL regulates that a SpatialIndividuality is not allowed to have ThematicValueDistribution (See Subsubsection 5.1.3.6) if it is used to describe members of a FieldOfIndividualities.

This term can also be used when users wish to systematically create and update a set of individualities of an identical kind during a simulation run. For instance, a set of moving ships need to be included in a simulation to create some traffic flow in the environment, whose initial locations are placed randomly. These ships together can be described as a FieldOfIndividualities, which consists of multiple non-significant ship members.

Which term to use depends on the spatial scale of the simulation and the way that users want to express and create individualities of a type. It does not reflect the "true" form of real-world existence. A forest may be expressed as a SpatialIndividuality occupying an area for one simulation with the changing density of trees, but as a FieldOfIndividualities consisting of multiple individual trees for another simulation.

Terms for expressing environmental phenomena are specified from a simulation perspective, in which an individual phenomenon is viewed as existence that is recognizable at each time instant and evolves in a temporal process. The time is treated as a locating frame of simulation processes, which is independent of the existence of phenomena. During its lifetime, an individual phenomenon exhibits various non-locational characteristics that are perceivable at its current spatial location. The term ThematicProperty is used to express such characteristics. When a non-locational characteristic of a phenomenon type is relevant to simulation scenarios, a ThematicPropety that expresses this characteristic is added to a corresponding SpatialIndividuality as its *theme*. The presence of a ThematicProperty in an SEDL description means the component under development should provide the digital representation of its described characteristic. For a FieldOfIndividualities, all members of the described field have the same non-locational characteristics as the *theme*-s of its *member* regulates.

In SEDL descriptions, different EnvironmentalPhenomenon-s contained in a SimulatedEnvironment shall have ThematicProperty-s with the same name, while the composition relationship between these two terms excludes the ambiguity. However, in practice, automatic transformations using SEDL descriptions as inputs need to be aware of this issue and avoid naming conflicts in transformation outputs.

Properties for recording spatio-temporal locations of phenomena in described component do not have to be explicitly denoted in SEDL descriptions. They can be derived in later development phases based on conceptual forms and expected changes of the described phenomenon type. The term "thematic property" in SEDL applies to all non-locational characteristics of phenomena. It is not restricted by which domain such a property is about or by which datatype it is recorded. For instance, the length of time since a phenomenon comes to exist could be relevant to a simulation. The validation domain of its values is restricted by the locational characteristic of this phenomenon, while this characteristic itself is about time. It acts as a theme of this phenomenon and can be expressed as a ThematicProperty (e.g., as the "age") in an SEDL description, even though it may be produced as some temporal datatype in an end application.

5.1.3.4 Expression of Exhibited Changes

Changes of phenomena in simulated environments should match corresponding simulation scenarios. A simulated environment component should be able to provide data representing these changes for adequately executing these scenarios. SEDL provides terms to describe requirements about changes that environmental phenomena should exhibit in simulation scenarios. These terms are identified based on the basic components of a change expression in natural language. Such an expression consists of the following three components underlying unstructured expressions.

1. Variant: the domain space in which the change is observed, together with the reference based on which the amount of change is measured.

A domain space consists of all possible states that a phenomenon's characteristic can be in. Each state corresponds to a point location in this domain space. A phenomenon has the inherent structure with the characteristics in three types, i.e., spatial, temporal and thematic[129]. Thus, a domain space can be one of the three types.

Fundamentally, the change about a characteristic of a phenomenon can be viewed as some difference in the phenomenon's location in a domain space of this characteristic's states. The first component of a change expression denotes in which domain this difference is perceived, i.e., what the expressed change is about. For the change of wind speed, it is the speed of wind.

Locations in a domain space need to be expressed based on a reference location. If the phenomenon's location relative to the reference does not alter, the difference amount is 0, i.e., no change is observed. This reference in computation is often specified as a part (e.g., the origin) of a reference system that regulates the meaning of coordinate values assigned to locations. The amount of difference becomes the difference between the coordinates SEDL does not regulate the reference location down to the technical level. It simply distinguishes two types of reference locations people implicitly use when they conceptually view a phenomenon change in spatial and temporal domain spaces, namely, self and external. For instance, the change of a phenomenon's spatial extent means that its spatial location relative to a point tied to itself (e.g., its geometric center) has some difference. In contrast, its movement in space

means that its spatial location relative to an external reference location (e.g., the geo-centroid of the earth) has some difference[7].

Types of the first component based on the above principles are listed in Table 5.3, each of which assigned to a name that is conventionally used in daily language for this specification.

Name of Component Type	Type of Domain Space	Type of Reference
Geometry	Spatial	Self
Location	Spatial	External
Duration	Temporal	Self
Time	Temporal	External
Theme	Thematic	N.A.

Table 5.3: Types of Variable and Variant.

2. Variable: the domain space in which observations of the expressed change are made, together with the reference based on which the difference of observation locations is measured.

Differences in a domain space do not exhibit along. It may only perceive when observing the states of a phenomenon at different locations in another domain space. For instance, the wind force difference may be observed at different time points or at different spatial locations. This second component of a change expression denotes in which domain these observations are taken to perceive this change. Types of Table 5.3 can also be applied to this component.

3. Variant = R(Variable): the mode of this change.

The third component expresses how the state of the phenomenon in the Variant should change when altering its locations in the Variable. It deterministically or stochastically regulates a relationship R between the first two components. In natural language expressions, it may be fuzzy or missing. The speed change of wind over time could be "constantly increase over 3 hours". This component is case-specific, which have an infinite number of possible pattern types. In computation, it is often abstracted as a mathematic model in terms of some functions.

5.1.3.5 Conceptual Approximation in the Formulation of Change Expressions

At the abstraction level of SEDL, the simplicity of terms is more important than the mathematic accuracy. Thus, several conceptual approximations are applied to specify the components of the change expression. They are clarified in this subsection to avoid confusion.

First, the terms "Variant" and "Variable" used to denote the first two components of a change expression are only analogies of the same terms in math. They emphasize the role that the corresponding domain space plays in the expressed change, which reflects the angle that humans view the change mode. At the analysis phase, the relationship between these two components is not perceived and described as accurate as a function in the sense of math. Some change mode may not be described as mathematic function at all when being modeled in the following development phases. For instance, a thematic value change over time of some individual phenomenon may be determined based on the mean of a phenomenon's neighbors.

Second, the distinction between external-referenced differences and self-referenced differences is a conceptual approximation. For instance, when the spatial extent of a phenomenon becomes different (self-referenced), it is impossible that this phenomenon still occupies the exact same spatial location (external-referenced). Each case of the approximation emphasis one angle to view and express the state differences in a domain.

Third, the change expression composition considers only two domain spaces, while the overall change of a phenomenon exhibited in the real world is normally the combined result of relationship patterns

[7] A technical analogs term is the body-fixed frame for spatial coordinates.

among its states in multiple domain spaces. This strategy supports decomposing these complicated patterns into simpler aspects. Thus, language users can express the essential context of their expected changes from their angles of thinking, without the need of paying attention to how they should be computationally combined in a developed program.

5.1.3.6 Chang Types of an Individuality

This subsubsection introduces terms to support describing expected changes of individual environmental phenomena in simulation scenarios. These terms are subtypes of Configurable. Each term represents a type of change about an individual phenomenon that can be perceived in the macro world.

These types are identified based on three aspects, i.e., the types of the first two components in a change expression as introduced in the previous two subsubsections and the level of perception at which changes are observed. Two levels of perception, i.e., the individuality level and the non-locational characteristic level, are considered. Every possible combination of the three aspects' types is evaluated. A specific individuality in the common-sense world only has one duration. Thus, combinations with duration involved are not applicable.

Each rational combination identifies a term introduced in this subsubsection. Figure 5.3 shows the abstract syntax of the identified terms in SEDL. They are described in detail in the remaining of this subsubsection. These terms are related to SpatialIndividuality or ThematicProperty via composition Associations, since their instances are description pieces about a phenomenon type or its thematic property, which should be contained in the description of that phenomenon or property.

When component users require a type of individual phenomena to exhibit a particular type of changes in simulation scenarios, they can add an instance of a suitable term specified in this subsubsection to the SpatialIndividuality instance that describes the phenomenon type or to a ThematicProperty of this SpatialIndividuality. Changes expressed by these terms are relevant to data structures that hold runtime values of simulated phenomena during execution, and functions that should exist to update these values in a resulting application. Common structures implied by the applied change types can be derived during follow-up development phases, leaving only the application-specific functions that compute the change mode, i.e., the third component of the change expression to be formalized. This third part at the analysis-level can be documented by some free text in the *description* attribute inherited from DescriptionItem.

Individuality level: changes at this level involve geometry or location of phenomena whose difference necessarily changes the whole individuality at the human-recognizable scale (given that the timeline is viewed as an independent frame in simulations). Values in spatial domains are viewed as properties of individualities in these cases.

Five change types are introduced into SEDL to categorize changes at this level, resulting in 5 terms in SEDL as specified in Table 5.4. To keep the SEDL model concise, mirror cases with the Variable type and the Variant type exchanged are denoted by the same term. The roles of involved domain spaces for a described change are denoted by attributes of these terms as introduced below.

An abstract term IndividualityChange is included in the SEDL model to facilitate the specification. This term is related to the SpatialIndividuality via a 1: n composition Association. An IndividualityChange must be the *hasChange* attribute of a SpatialIndividuality. All terms in Table 5.4 are subtypes of IndividualityChange. Each of their instances must be linked to a SpatialIndividuality in an SEDL description.

Since a SpatialIndividuality may locate in more than one thematic domain spaces as they have multiple ThematicProperty-s, a GeometryThemeDependency or a LocationThemeDependency needs to denote the involved thematic domain spaces of their described changes. Thus, both terms have a non-containment Association to the ThematicProperty as *involvedTheme*. An instance link of these two Associations must satisfy the following restriction: its *involvedTheme* must be a *theme* of its other member end. Besides, both of the terms have an attribute *roleOfTheme* to denote if the involved theme is viewed as Variable or Variant. The roles of the two involved domain spaces have to be denoted in a GeometryLocationDependency as well. In contrast, the simulation is a temporal process in which the

time is considered as the fully independent variable. For the change types that the time is involved, the mirror cases in descriptions do not change the underlying role of time as the Variable. In other words, the mirror case of a description of the individuality change involving time is conceptually the same to itself.

Term I.1	**GeometryLocationDependency**
Description	The exhibited pattern of difference in a phenomenon's geometry, when the spatial location of this phenomenon varies in a controlled way. **Mirror:** the exhibited pattern of difference in the spatial location of a phenomenon, when its geometry varies in a controlled way.
Example in natural language	"Its spatial extent should become larger when it moves to higher latitude". **Mirror:** "it should move to higher latitude when its spatial extent becomes larger".
Term I.2	**Deformation**
Description	The exhibited pattern of difference in a phenomenon's geometry with the time goes on in a controlled way. **Mirror:** the exhibited pattern of difference in time, when the geometry of a phenomenon varies in a controlled way (i.e., speed of deformation).
Example in natural language	"The influential area of the storm should shrink linearly over time". **Mirror:** "t hours passed for its radius of influential area shrink r".
Term I.3	**GeometryThemeDependency**
Description	The exhibited pattern of difference in a thematic characteristic A of a phenomenon, when the geometry of this phenomenon varies in a controlled way. **Mirror:** the exhibited pattern of difference in a phenomenon's geometry, when its thematic characteristic A varies in a controlled way.
Example in natural language	"Its spatial radius should shrink when its speed decreases". **Mirror:** "Its speed should decrease when its radius shrinks".
Term I.4	**RigidBodyMovement**
Description	The exhibited pattern of difference in a phenomenon's spatial location with the time goes on. **Mirror:** the exhibited pattern of difference in time, when the spatial location of a phenomenon varies in a controlled way (i.e., speed of movement).
Example in natural language	"The hurricane moves uniformly over time along a recorded path". **Mirror:** "1 hour passed for every 5 km of its walk".
Term I.5	**LocationThemeDependency**
Description	The exhibited pattern of difference in a phenomenon's spatial location, when the value of its thematic A varies in a controlled way. **Mirror:** the exhibited pattern of difference in a thematic characteristic A of a phenomenon, when its spatial location varies in a controlled way.
Example in natural language	"It should move to higher latitude when its speed decrease". **Mirror:** "Its speed should decrease when it moves to higher latitude".

Table 5.4: Individuality Level Change Types.

Non-locational characteristic level: changes at this level involve a non-locational characteristic of an individuality (expressed by a ThematicProperty in SEDL). A thematic domain space must be involved in a change at this level. Further, geometry should not be involved since its difference influences a whole individuality. Three terms to categorize changes at this level are introduced into SEDL, as specified in Table 5.5. As the level suggests, changes at this level are normally viewed as the change about thematic values, i.e., the involved thematic domain is viewed as the Variant. The spatial and temporal domains in these cases are often viewed as spaces for evaluating the change amount.

ThematicValueDistribution and ThemeDynamics are both associated with ThematicProperty via a 1: n composition Association. An instance of these two terms must be the *hasChange* attribute of a

ThematicProperty, which denotes the involved thematic domain space of the change expressed by this instance.

If component users require a thematic property to exhibit some spatial heterogeneity, this requirement needs to be expressed by a ThematicValueDistribution of the corresponding ThematicProperty. Spatial heterogeneity of a phenomenon can also be perceived as heterogeneity among a set of spatially distributed individualities of some type. In this view, such a set can be described as a FieldOfIndividualities, and heterogeneity within the extent of each individuality is often neglected. To avoid over-complexity, it is not allowed to add a ThematicVlueDistribution to a ThematicProperty of some SpatialIndividuality that is the *member* of a FieldOfIndividualities.

The ThematicValueDistribution and the LocationThemeDependency describe relationships between a thematic domain space and a domain space of spatial locations about an individuality at the two levels of perception, respectively. The distinctions between these two types are that the "domain of spatial locations" and the conceptual form of the described phenomenon type in these two types are different. When humans describe a LocationThemeDependency, the phenomenon is conceptualized as an object that can move in a frame of space. In this case, the domain of spatial locations consists of all possible parts of space that can be occupied by this object in the frame at some instant. When human describes a ThematicValueDistribution, the phenomenon is conceptualized more like a field[82] that holds a set of spatial locations. In this case, the domain of spatial locations consists of all possible elementary spatial locations within the part of space that is occupied by this phenomenon.

A ThemeDependency involves two thematic domain spaces. For a clear hierarchical description structure, this specification regulates that a ThemeDependency instance must be contained by a ThematicProperty that describes its Variant, as its *dependOn* attribute. Besides, this term has a non-containment Association with the ThematicProperty as its *variable*, which denotes the Variable of the described ThemeDependency. An instance of ThemeDependency must satisfy the following restriction: both of its linked ThematicProperty-s must be *theme*-s of the same SpatialIndividuality.

Term T.1	**ThematicValueDistribution**
Description	The exhibited patterns of difference in values of a thematic characteristic of a phenomenon over different locations within the spatial extent of this phenomenon.
Example in natural language	“The wind speed decreases from the center of the storm”.
Term T.2	**ThemeDynamics**
Description	The exhibited pattern of difference in a thematic characteristic of a phenomenon with the time goes on.
Example in natural language	“The air temperature increases during the day after sunrise”.
Term T.3	**ThemeDependency**
Description	Patterns of difference in a thematic characteristic A of a phenomenon exhibited at different values of another thematic characteristic B of the same phenomenon. **Mirror:** Difference in a thematic characteristic B of a phenomenon exhibited at different values of another thematic characteristic A of the same phenomenon.
Example in natural language	“Its temperature increases while its weight increase”. Mirror: “Its weight increases while its temperature increase”.
Implication	Implies both A and B have ThematicValueDistribution, or both A and B have ThemeDynamics.

Table 5.5: No-locational Characteristic Level Change Types.

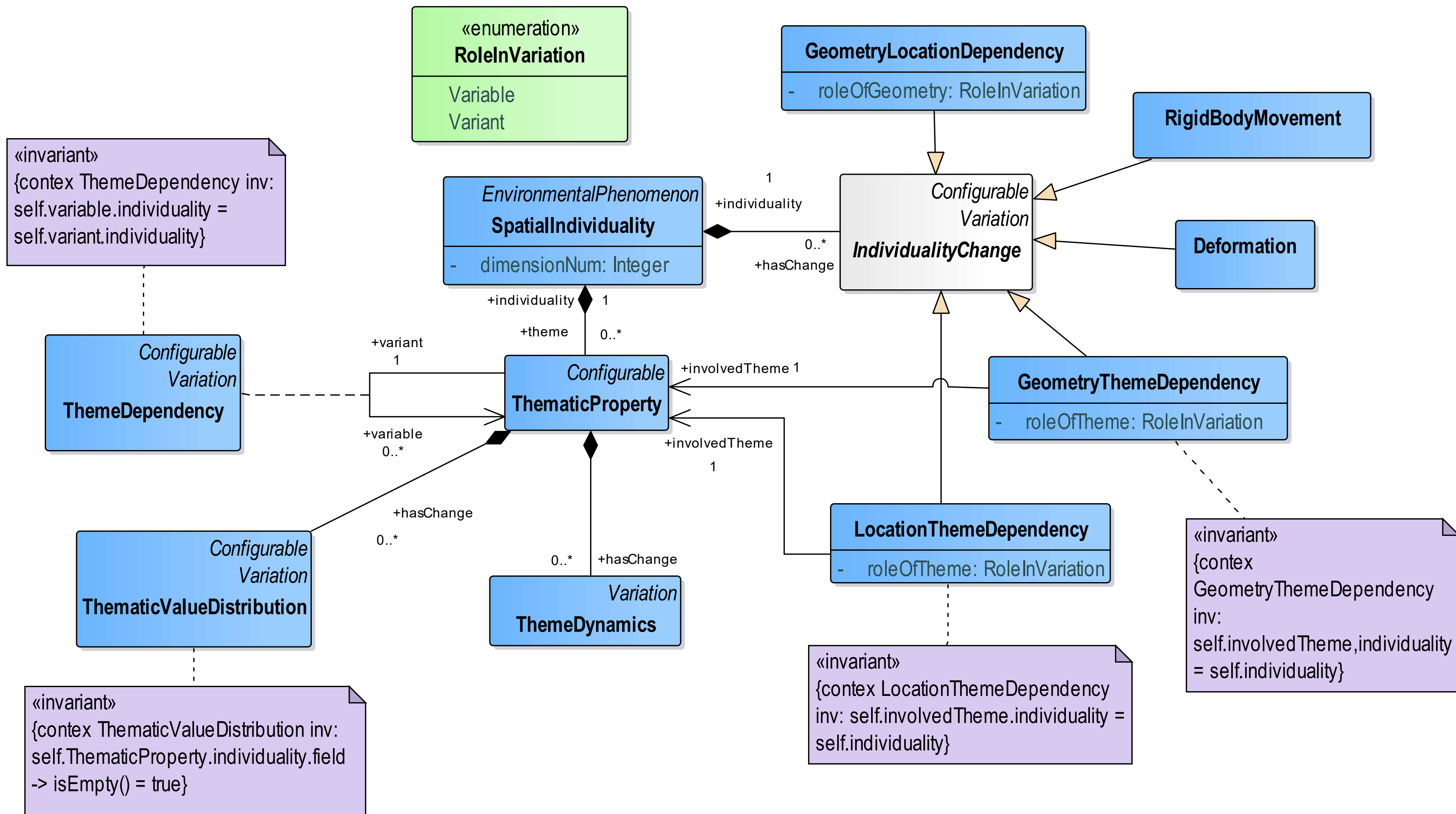

Figure 5.3: Abstract Syntax of Individuality Change Types.

5.1.3.7 Alternative Change Modes

An EnvironmentalPhenomenon shall have more than one change of one type. The described phenomenon type exhibits a change pattern which reflects the overall effect of all its described changes. However, in some cases, a type of change of a phenomenon type may have several possible modes. An instance phenomenon only exhibits one of them. For instance, in different seasons, the density of a resource may follow different distribution forms in space.

An above-mentioned case shall be documented with a single instance of a Variation in SEDL with a ConfigurableParameter of an Options type to denote possible modes. However, in this way, the alternative modes can only be documented within the ConfigurableParameter in free text. Transformations specified in Section 5.3 does not recognize them separately to generate corresponding computation units for them. Also, while these modes reflect some application-specific structural options, each of them may still have a set of alterable conditions that are execution-specific. For instance, in the same season, the density of that resource may still vary from year to year due to different average temperatures, etc. This means a component to produce such resources should be configured by different sets of parameters when choosing different modes. When these modes have been documented within a ConfigurableParameter, users have no means to express controllable conditions for each mode other than the free text.

SEDL introduces the term AlternativeMode for describing these alternative change modes. It is a subtype of the Configurable. Figure 5.4 shows the abstract syntax of this term. Terms from Subsubsection 5.1.3.6 are subtypes of Variation in this figure. The term Variation is an abstract term that is purely for specification and implementation convenience. It denotes the exhibited changes as described in Subsubsection 5.1.3.4 in the specification but is not exposed to language users.

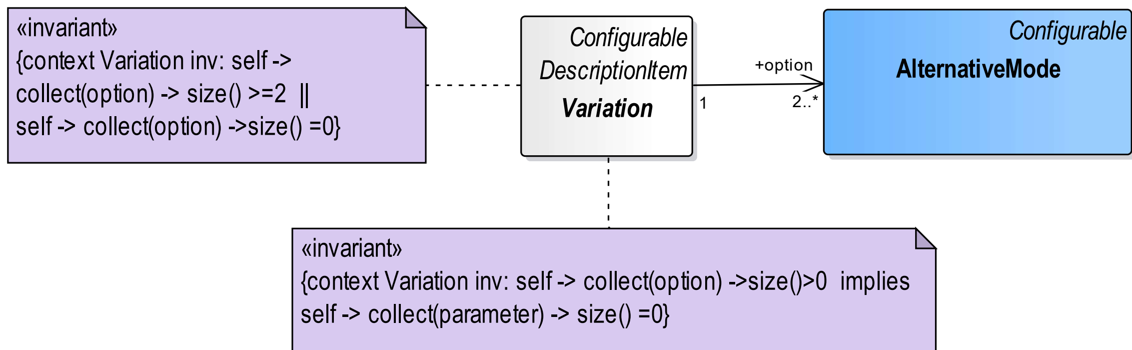

Figure 5.4: Abstract Syntax of AlternativeMode.

An instance of Variation may have either no or at least two *options* which are instances of AlternativeMode. Each AlternativeMode instance is a piece of description that expresses a possible mode of the Variation instance. An instance of the described phenomenon type only exhibits one of the modes which should be selectable for an execution. To avoid over-complication, this specification regulates that when a Variation has *options*, it cannot directly own ConfigurableParameter-s. This constraint should be implemented in the validation function of an SEDL editor. The current specification brings a drawback: a parameter that is conceptually common to all options needs to be repeatedly described for each option.

5.1.3.8 Characteristic Variation among Instances of an EnvironmentalPhenomenon

The term CharacteristicVariation is used for describing user-expected variations of some characteristic values among the whole set of a phenomenon type's instances. Such a characteristic can be summarized by some index (see explanation in Subsubsection 5.1.3.2). The value of an index is fixed to a phenomenon instance, no matter if the value is only revealed after the phenomenon's lifetime completes (e.g., the lifespan of this phenomenon). A characteristic variation of some phenomenon type can be represented as the value distribution of some index.

Suppose the set S contains all digital instances of a phenomenon type that can be generated by the component under development. If users require that all values of some index from instances in S to follow a distribution, they can add a CharacteristicVariation to the EnvironemtnalPhenomenon that describes the phenomenon type, denote the involved index and provide some information about the distribution. The developed component should have the function to initialize instances of this type with index values drawn from the expected distribution.

In the current SEDL version, each CharacteristicVariation is independently described. The variation about a set of correlated indices should be described in one CharacteristicVariation. It is essentially a distribution of phenomenon instances in a combined domain space of these indices' domains. A CharacteristicVariation must have one or more *indexName*-s, each of which denotes the name of a described index.

Component users may have requirements on the variation among generated instances at different detail levels and may require having control at these levels, e.g., for a single run, a set of runs or multiple sets of runs. SEDL supports users in expressing requirements of controlling over the executions in a structured way through CharacteristicVariation and ConfigurableParameter specified in Subsubsection 5.1.3.2. A guide about the usage of these two terms is provided in the following paragraphs, supplemented by Table 5.6 which summarizes these requirements at each detail level and the term to use for describing them.

For a single run, users may expect an individuality to be in some certain condition. In a program, this can be expressed with some desired value of an index about this individuality. Thus, users would require the component to provide some interface through which they can give the desired value to the program. In this case, they can add a ConfigurableParameter to a SpatialIndividuality to give a name and a value type to this index.

An additional level of characteristic variations exists for individualities that are members of a group expressed by a FieldOfIndividualities. A FieldOfIndividualities denotes that, behaviors of its members in a simulation execution should conform to the same set of rules which are supposed to be parameterized at the whole field level. During an execution, members of a FieldOfIndividualities instance are iteratively updated by the same set of computation functions. The diversity of state values among members originates from different index values[8] assigned to these members. The values of such an index are not supposed to be manually configured one by one. Instead, they should be systematically drawn from some distributions. Users can express their expectations on how values of some index for individualities in the whole group (i.e., an instance of this FieldOfIndividualities) should distribute in a domain value space. In this case, they can add a CharacteristicVariation to the member SpatialIndividuality of a FieldOfIndividualities.

A high-level functional scenario is often executed by a set of runs. In these runs, users may expect that the values of some index from all digital instances of an EnvironmentalPhenomenon match a certain distribution. For instance, some type of summarized descriptors of these instances should match the observed distribution of real-world individualities. This requirement can be expressed by a CharacteristicVariation of this EnvironmentalPhenomenon. When the phenomenon is a FieldOfIndividualities, each run should compute an index value from the expressed characteristic variation f(e) about the whole field. Table 5.6 and Figure 5.5 illustrate how this value is connected to more detailed level computations.

[8] very frequently, initial values of their properties

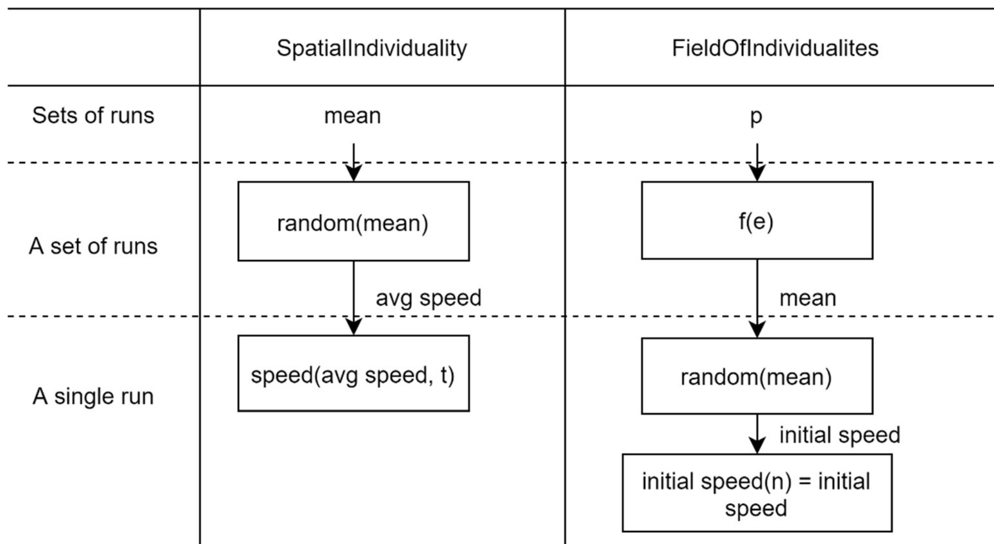

Figure 5.5: The Computation Chain of Characteristic Variations.

Further, a component may be used in multiple functional scenarios, each of which corresponds to a set of runs. For instance, in an environment with blowing wind, the initial wind speed in one execution should be generated by a random function. This can be described by a CharacteristicVariation. This environment can be used in various functional scenarios with calm, windy and stormy weather, each of which makes a set of n runs. For each set, the randomly generated initial speed should have a different mean value, which reflects the character of the corresponding functional scenario. At this level, users may require to be able to set their desired mean value. They can add a ConfigurableParameter of a corresponding CharacteristicVariation to describe this requirement in this case. Similarly, the CharacteristicVariation of a FieldOfIndividualities f(e) may also have a ConfigurableParameter p to change the form of f(e) for different sets of runs. The connections between illustrative examples at different levels in Table 5.6 are shown in Figure 5.5. Essentially, a CharacteristicVariation corresponding to some level leads to some function to generate a characteristic value for the computation of the level below, which otherwise shall be exposed to users as a configurable parameter. Characteristic variations at some level may not be recognized by or interesting by users. They can still be embedded in computation functions by developers without being exposed by users.

		SpatialIndividuality	FieldOfIndividualities
One individuality in one run	**Req.**	Each SpatialIndividuality instance should be initialized with an index decided by users. **E.g.**, avg speed = x, where x should be decided by users.	The n-th member of a FieldOfIndividualilites instance should be initialized with an index value drawn from F(n). **E.g.,** for n-th member, initial speed (n) = random ().
	In SEDL	The index, i.e., "avg speed", as a **ConfigurableParameter** of this SpatialIndividuality.	The F(n), i.e., the "random ()", as a **CharacteristicVariation** of the member SpatialIndividuality of this FieldOfIndividualities.
One run	**Req.**	Each instance is configured separately.	A parameter of the above F(n) should be configured with a value decided by users. **E.g.**, for n-th member, initial speed (n) = random (mean), in which the value of mean should be decided by users.
	In SEDL	Implied by the term "SpatialIndividuality".	The parameter, i.e., the "mean", as a **ConfigurableParameter** of the CharacteristicVariation that expresses F(n).
Multiple runs	**Req.**	The value of an index to initialize a SpatialIndividuality instance should be drawn from f(e). **E.g.**, avg speed (e) = random ().	The value of a parameter of the above F(n) should be drawn from f(e) for a run. **E.g.,** the value of the "mean" in the random(mean) from the above example, should be drawn from f(e).
	In SEDL	The f(e), i.e., random (), as a **CharacteristicVariation** of this SpatialIndividuality.	The f(e) as a **CharacteristicVariation** of this FieldOfIndividualities.
Multiple sets of runs	**Req.**	For each set, a parameter of the above f(e) should be configured with a value decided by users. **E.g.**, avg speed (e)= random(mean), where the value of "mean" should be decided by users.	For each set of runs, a parameter p of the above f(e) should be configured with a value decided by users.
	In SEDL	The parameter, i.e., the "mean", as a **ConfigurableParameter** of the CharacteristicVariation that expresses f(e), i.e., the random(mean).	The parameter p as a **ConfigurableParameter** of the CharacteristicVariation that expresses f(e)

Table 5.6: Express Required Characteristic Variation and Alterable Conditions in SEDL.

5.1.3.9 ExecutionRoutine

One or more ExecutionRoutine-s can be contained in SimulatedEnvironment. The term ExecutionRoutine provides some simple support to describe how the computed data from the component described by this SimulatedEnvironment should be provided to a system of interest component of a larger simulation. Such execution routines are not parts of functionalities of a simulated environment component itself but are the way that this component participates in a larger simulation. Thus, a SimulatedEnvironment can be a valid description without an ExecutionRoutine. On the other hand, a developed component shall be used for more than one simulation with different systems of interest. Thus, a SimulatedEnvironment shall have multiple ExectionRoutine-s.

An ExecutionRoutine has an attribute *systemID*. It denotes to which component that the described routine sends data. Besides, it has an attribute *mode* that denotes how communication between this component and the system of interest component is triggered, i.e., at which time the state data of simulated environment should be sent to the system of interest component. Two execution modes are available in the current version, as defined in Table 5.7. They are defined within the Enumeration ExecutionMode as the type of the *mode* attribute. These modes are independent of how the values are computed within the simulated environment component.

Value	Explanation
Autonomous	The environment component runs autonomously as a concurrent process in the bigger simulation once an execution starts. The interval between two time points of communication from this component to the system of interest component is determined before the execution, which could be fixed by the component function, controlled by a moderate component, be fixed by user configuration, and so forth. At each communication time, state values of the simulated environment corresponding to this time should be sent to the system of interest component. For instance, this mode can be used to simulate the behaviors of a system, which is regularly informed by some weather service with the current weather information.
Reactive	The environment component should be kept available during the execution of the bigger simulation. The communication happens in two directions. The environment component should send data upon the request from the system of interest component during the execution. For instance, this mode could be used to simulate the behaviors of a ship, which is influenced by the force of water. During executions, the ship model is fed with environmental data upon its at-moment location.

Table 5.7: Available Options of ExecutionMode.

An ExecutionRoutine also has an attribute *outputRange* which specifies the range of data that should be sent to the system of interest component at a communication time of the described routine from the environment component. The available values of this attribute are defined within the Enumeration OutputRange, as explained in Table 5.8.

Value	Explanation
All	All produced data from the simulated environment component at the time corresponding to the communication time should be sent.
AtPoint	All produced data about a point location from the component at the time corresponding to the communication time should be sent. If the *mode* of the ExecutionRoutine is Reactive, this point location is determined upon the request of the system of interest component.
AtRegion	All produced context at a region from the component at the time corresponding to the communication time should be. If the *mode* of the ExecutionRoutine is Reactive, this region is determined upon the request of the system of interest component.

Table 5.8: Available Options of OutputRange.

In addition, an ExecutionRoutine has an attribute *valueAggregation* of the Boolean type. In spatial simulations, computation functions may simulate different values for a phenomenon property at different locations. If the system of interest component in the described routine needs a single aggregated value for each property from the computed ones, the *valueAggregation* should be set to *true*. The values of this attribute may influence the derivation of structure for the data being sent to the system of interest component.

5.2 PIM Layer Metamodels

Executions of SEDL descriptions perform transformations that generate PIM-layer software models of simulated environment components in simulations. The transformation rules are specified from the SEDL language model to the metamodels that are used to describe the output models. To enable domain-specific outputs with more specialized and more concise elements than models consisting of basic UML elements, three metamodels for describing the outputs are presented in this section. Elements in these metamodels extend corresponding UML elements with additional descriptive semantics. From a computer language perspective, these metamodels define domain-specific languages specified by UML. As the UML metamodel, their execution semantics are not restricted by specifications. Thus, these metamodels can be mapped to different technical platforms for various implementations. The version of UML used in this specification is UML2.5.1[10]. Modeling elements in UML2.5.1 being mentioned in this specification are written with the prefix "uml" for identification, e.g., uml:Class.

5.2.1 Configuration Schema Description Profile

The Configuration Schema Description Profile is a small UML Profile that specifies stereotypes to describe form-based configuration schemas of software programs. Stereotype-specific properties in this profile and other metamodels in this chapter are marked *italic*.

5.2.1.1 Summary

A configuration schema expressed by the Configuration Schema Description Profile (i.e., an instance model of this profile) is an M1 model within the framework in this thesis. This schema describes the complete set of parameters that is necessary to be set before running a software program. These parameters should be accessible to the users of this program, which allows them to communicate with the program. An M0 instance of such a schema holds a set of values of these parameters provided by users. This instance must be passed to the back end of the program before executing this program. Figure 5.6 presents this profile expressed in the UML graphic notation.

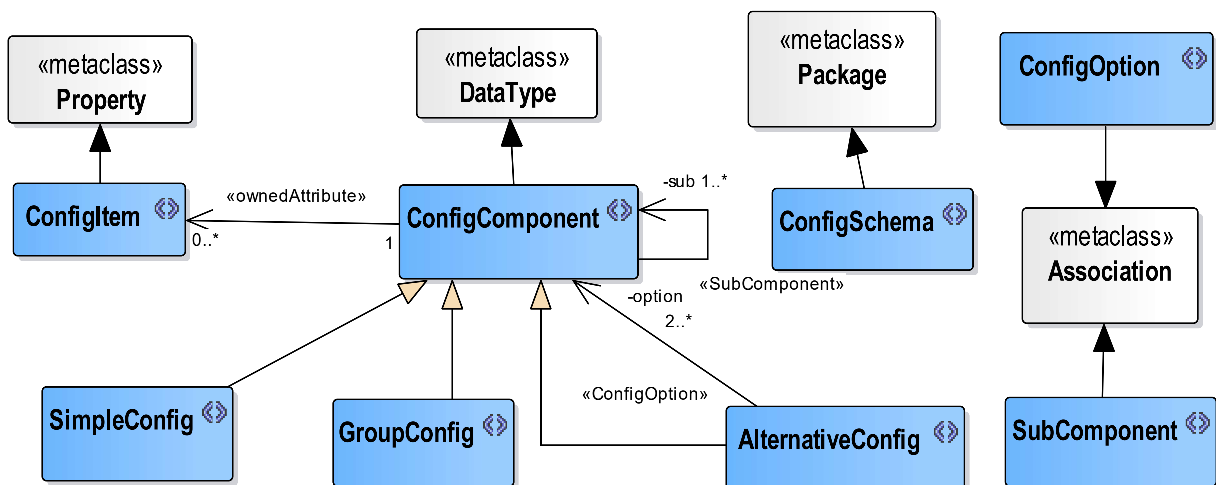

Figure 5.6: Configuration Schema Description Profile.

The following paragraphs in this subsection present its descriptive semantics. Different from the other two metamodels specified in this section, stereotypes in this profile do not regulate implicit structures of model elements. Applying this profile to an M1-layer schema mainly help further transforming this schema to some platform-specific model, especially for visual front ends of components, e.g., graphic interfaces where users can edit configurations. This specification does not restrict presentation or behavioral semantics of the defined stereotypes while they depend on the chosen platform.

5.2.1.2 ConfigSchema

A ConfigSchema extends a uml:Model, which is a specialized uml:Package that "describes a system from a certain viewpoint"[10]. An instance of the ConfigSchema[9] is a model package that contains specifications of all modifiable parameters of a computer program, which are exposed to end users. Parameters in this package are grouped into one or more ConfigComponent-s (see Subsubsection 5.2.1.4). These parameters specify arguments that need to be passed to the program before execution. The model package presents the boundary of these specifications, i.e., a configuration schema of the program. An instance of this schema is a specific configuration with fixed parameter values. It holds a piece of information that can be passed to and processed by the program.

Within the thesis, a model package of this type is initially generated when an SEDL description is executed. In general, for each SimulatedEnvironment in an SEDL description, an instance model of this type is generated. Further generated elements related to the configuration are added to this package.

5.2.1.3 ConfigItem and Primitive Types

A ConfigItem is a UML Property of a primitive type or an enumeration type. A ConfigItem must be owned by a ConfigComponent which is specified in Subsubsection 5.2.1.4. This item essentially describes a "name-value" pair (i.e., a parameter) in a software configuration schema. Changing the value of such a parameter alters the computation behaviors of the software. This stereotype adds the following two constraints to the metaclass uml:Property:

1. an element that applies the ConfigItem stereotype must be the *ownedAttribute* of a model element that applies the ConfigComponent stereotype,
2. the values of this element must be either one of the primitive types OR a uml:Enumeration type. Primitive types used in this profile include the types defined in the PrimitiveTypes package of UML[10], i.e., Integer, Boolean, String, Real and UnlimitedNatural, OR one of the primitive types defined in this subsubsection that is a uml: PrimitiveType.

Three additional primitive types are defined to express special types of strings. A SourceString represents a string that should be interpreted as representing a location where a piece of data is stored. A GeometryString represents a string that should be interpreted as representing geometric objects. A TimeString represents a string that should be interpreted as representing a time instant or a period. No encoding formations are specified for these three string types in this specification since they may vary among platforms and among applications, e.g., some applications require configurations of UTC coordinates, while some others only need an integer value denoting the index of a time unit. They provide vocabularies for more clearly expressing the context of values in design models that facilitate the communication and implementation choice. It depends on developers to determine a further encoding standard, either by implementing the modeled application, or by implementing a platform-specific model translator to turn the design models into models that restrict these string types to more specific forms. Same as the other UML primitive types, these types themselves, rather than their instantiation, will appear in M1 models that apply this profile.

An M1 instance of the ConfigItem uses the notation of UML Property. Its type is denoted by the name of corresponding primitive types in this subsubsection. At the M0 level, this item is a named slot that holds a specific value.

---

[9] Strictly speaking, it is an instance of the UML metaclass Model that applies the ConfigSchema stereotype. The following text uses the same shorter convention to refer to model elements with stereotypes.

5.2.1.4 ConfigComponent Types

Two configurations with the same structure (i.e., conform to the same schema) are identified by their values. When all the values at the same location respected to the structure are equal, these two configurations can be treated as equal since they provide the same input for a program. The program can run multiple times with the same configuration. Even when its functions that generate output values are stochastic, the characteristic information (e.g., the standard deviation of all values) of each generation remains the same, which matches the situation of the given configuration. Thus, stereotypes specified in this subsubsection to express partial structures of configuration schemas extend uml: DataType, whose instances are identified only by their values. Each stereotype represents a possible type of substructures within the schema.

A ConfigComponent is a part of a configuration schema that embodies a group of user-modifiable parameters. This group of parameters is considered as related to the same aspect that is denoted by its name. A ConfigComponent shall aggregate another ConfigComponent or shall represent one option of an AlternativeConfig (see this subsubsection below). It shall also be aggregated by ConfigItem-s and/or other ConfigComponent-s. Three descendants of ConfigComponent are defined to express the components with different aggregated structures as follows.

A SimpleConfig is a ConfigComponent that only owns only ConfigItems. The types that apply SimpleConfig are the leaf groups of parameters in configuration hierarchies. A model element applying this stereotype must satisfy the following constraint:

1. it shall only be the substructure of another ConfigComponent, i.e., when it is linked by a SubComponent which specified in Subsubsection 5.2.1.5, it must be the *sub* end.

A GroupConfig is a specialized ConfigComponent that contains and only contains one or more other ConfigComponent-s. It does not directly own any ConfigItem. A model element applying this stereotype must satisfy the following constraints:

1. it must have at least one *sub* property which is a ConfigComponent, linked to it via a SubComponent association.
2. it must NOT have any *ownedAttribute* which applies the ConfigItem stereotype.

An AlternativeConfig represents a part of a configuration schema related to some aspect denoted by its *name*. This part can be configured by one of several possible parameter sets. Each of the sets is an *option* of this AlternativeConfig. Such a set itself can be expressed by a ConfigComponent type, which may have its own substructure. In an M0 configuration instance, only one of the sets can be picked and configured. A model element of this type must satisfy the following constraints:

1. it must have at least two *option-s* which are ConfigComponent, each linked to it via a ConfigOption which is specified in Subsubsection 5.2.1.5.
2. It shall only be the substructure of another ConfigComponent. i.e., when it is linked by a SubComponent, it must be the *sub* end.
3. it must NOT have any *ownedAttribute* which applies the ConfigItem stereotype.

5.2.1.5 SubComponent and ConfigOption Associations

Two stereotypes that extend the uml:Association are defined in this profile to describe the relationships between ConfigComponent types, as introduced in this subsubsection. An Association that applies one of these stereotypes links two ConfigComponent-s in a configuration schema at the M1 level.

A SubComponent represents the aggregation relationship between two ConfigComponent-s. The component being aggregated have the other component as its substructure. A ConfigComponent cannot be aggregated by itself either directly or indirectly. An Association applying this stereotype must satisfy the following constraints:

1. it must be a binary association whose two *memberEnd*-s are properties of two different ConfigComponent types AND
2. it has a *memberEnd “sub”* with the *aggregation = AggregationKind:none* AND

3. Associations stereotyped with SubComponent in a configuration schema cannot form a cycle.

A ConfigOption links an AlternativeConfig type to one of its options. An instance of this stereotype in the M1 layer must satisfy the following constraints:

1. its *memberEnd* with *aggregation = AggregationKind:shared* property must be a property of an AlternativeConfig type AND
2. the other *memberEnd "option"* must be a property of a ConfigComponent type.

Constraints of stereotypes in this specification can be used to validate a schema at the M1 (type) level. For instance, e.g., checking if an AlternativeConfig in a configuration schema is linked at least by two ConfigOption associations. The multiplicity of a ConfigOption association can also constrain that only allowed number of instances (usually 1) of its linked option (which is a ConfigComponent) appears in an instance configuration. However, it cannot restrict that only one option of an AlternativeConfig is picked in a configuration instance, i.e., only one of the ConfigOption associations linked to an AlternativeConfig is instantiated. An object diagram with two alternative options picked is still a valid UML graph and thus cannot be detected by a general UML validation tool. The semantics of "alternative options" is descriptively defined in the specification. It must be implemented either by some programming language or strategies, as shown in Chapter 6.

5.2.1.6 Usage Outside the Framework

Within the framework proposed by this thesis, creations of configuration schemas are triggered by executions of SEDL programs. Therefore, the contexts of these schemas are always related to simulated environments. However, this profile does not define any stereotype that is specific to simulated environments. Thus, the specification does not rule out the possibility of using this metamodel for programs other than simulated environment components. In general, this profile describes hierarchical structures of software configurations. It shall be used as a concise input metamodel for implementing automatic user interface generators or renderers from configuration schemas. The stereotypes can be mapped to a description in languages with representation semantics such as HTML&CSS[10], or a structure of GUI widgets. The resulting generators/renderers are context-free, which are also applicable for schemas whose contexts are not related to simulated environments. Nevertheless, the usage outside the framework is not emphasized in this thesis.

5.2.2 Simulated Environment Structure Profile

The Simulated Environment Structure Profile is a PIM-layer UML Profile that is used to describe structural aspects of programs that provide simulated environments.

5.2.2.1 Summary

Graphically, models that apply the Simulated Environment Structure Profile can be presented in UML class diagrams. Elements in these models can also be instances of non-stereotyped UML constructs since this profile does not add additional restrictions on which core UML constructs can be used in a model. For a clear model hierarchy, models that apply this profile are referred to as an instance of it. Such a model locates at the M1 level to express the M0 objects of this program, e.g., objects that hold values at runtime and data objects that being sent to other components.

Stereotypes defined in this profile provide additional constructs for concisely expressing building blocks of simulated environment components in structural models. Each stereotype regulates the common structure to all classes that apply it as well as derivable structures for these classes based on their associations. Implementations of these classes should realize these structures on a chosen platform. In a design model expressed by this profile, such structures can be made implicit. These stereotypes are extensions of elements that are used in UML class diagrams, including specializations of uml: Classifiers

[10] https://www.w3.org/standards/webdesign/htmlcss

(Datatype and Class) and uml: Associations. Their descriptive semantics are described in the following subsubsections.

For a phenomenon type being computed, a geometric representation to hold its spatial location is needed. This thesis mainly focuses on geo-scale simulations in which space is often abstracted as two dimensional, and the height shall be treated as a thematic property. To remain the focus, this profile defines stereotypes with geometric representations in a two-dimensional context. Consequently, transformations in Section 5.3 are specified for SEDL descriptions of simulated environments whose space dimension number is set to two. Nevertheless, geometric representations in this profile are based on well-established researches, each of which has its 3D counterpart. A realization of the framework that supports 3D contexts shall be implemented based on these counterparts. For a SimulatedEnvironment in SEDL whose *spaceDimNum* is set to 3, the transformations create the same type of elements as specified in Section 5.3, except that their geometries and geometry-related operations are replaced by a 3D version.

5.2.2.2 Utility Datatypes

The following datatypes are used in this profile to support the specification of attribute types that hold an elementary geometric location on the timeline and in the two-dimensional space. They are not defined as stereotypes since these types are rather M1 level instances of the uml: Datatype. These types are very basic types in geometry or temporal data models and standard, as well as in their implementations. Thus, this specification does not unnecessarily redefine the complete structure of these types. In a realization of the proposed framework, it depends on the choice of the working community to map them to a more concrete standard or a technical implementation. These types are listed as follows with reference to a comparable datatype in the ISO standard.

TemporalPosition: an instance of this class is an undividable location on the timeline in the view of a simulation. This position can be both an instant on the real timeline or a period whose length equals to the minimum recognizable unit by the simulation. This location can be represented by a coordinate in some coordinate reference system[130]. A comparable datatype can be the TM_Position[95] in ISO19108. It could also be mapped to a temporal datatype in some platform, e.g., the DateTime in Eclipse Platform Java API[11].

Point: an instance of this represents a 0-dimensional geometric object which can hold a single point's location in the space. In a two-dimensional abstraction, its location can be represented by a pair of coordinates in some coordinate reference system, one for each spatial dimension. A comparable datatype is the GM_Point[93] in ISO19107.

Polygon: an instance of this class is a 2-dimensional geometric object which can hold a location of a regional subset of the space. In a two-dimensional abstraction, its location can be represented by the underlying plane bounded by a closed circle that is formed by a finite set of connected straight-line segments, while the location of each segment is determined by its two endpoints. A comparable datatype can be the GM_Polygon[93] in ISO19107.

5.2.2.3 Runtime Simulated Feature Types

Subsubsection 5.2.2.3-5.2.2.6 introduce stereotypes in the structural profile to express the classes (at the M1 layer) representing runtime objects (at the M0 layer) during executions of simulated environment components. They are used to regulate common structures and spatial representations of classes that hold state values of environmental phenomena at runtime, which are used by the computation functions of the components.

Classes applying these stereotypes should not be confused with the datatypes that are used to model data stores of spatio-temporal information. These classes provide the necessary structure to hold state values of phenomenon instances that should be computed at a simulation step. A storage region in memory is allocated to an M0 object of these types at runtime upon its creation. Object values are initialized and updated during the execution process. Older values are discarded in an update that affects

[11] https://help.eclipse.org

these properties. Their instances exist at runtime and only hold state values of the represented objects at "present" that the simulated process is passing. In contrast, a spatio-temporal data store needs to deal with historical values that do not reflect the "now" of its represented world. It may store a history of property values of a spatially identifiable object as in a moving object database[87], or store values in a matrix with a temporal dimension as in the implementation of some array-oriented data form such as NetCDF[131]. Although there is no technical restriction to implement a property of a runtime data object that holds historical values at all computation steps of another property in memory, normally it is not necessary. When the size of computed values at one step is large, keeping such a property in memory is also not an efficient strategy.

An abstract term SimulatedFeatureType is specified to facilitate the specification. It is the supertype of all stereotypes that express digital entities which are representations of participants in simulated processes. An instance of the subtypes of SimulatedFeatureType is a uml: Class that provides the data structure of state values about a type of environmental phenomenon being computed or queried during simulation executions. Time is not an integrated domain of these entities but as a separate reference dimension, which is unfolded with the program execution.

The SimulatedFeatureType is introduced based on correspondence among entities and processes in different worlds related to a simulation. Conceptual links among these entities are shown in Figure 5.7. The "simulated world" in the figure distinguishes itself from its digital representations computed in software. The digital entities are the actual existence that imitates the real world, while the simulated world is what these entities are interpreted in perception. A simulated world and phenomena in it shall be conceptually continuous, while their digital representations are discrete.

An execution process p_e of a simulation program corresponds to a simulated process p_{sim} which imitates a real-world process in a simulated world. A computation step s of the p_e corresponds to a temporal location on the timeline of p_{sim}. An instance i_{sf} of a class applying the SimulatedFeatureType stereotype created by p_e is a digital entity. It represents a simulated phenomenon that imitates a real-world phenomenon. Its state values are updated during p_e. The state values of i_{sf} at s reflect the conditions of its represented phenomenon at the "current moment" in p_{sim}, i.e., the temporal location that the step s corresponds to.

A runtime instance i_{sys} representing a simulated system uses the output from p_e. The computation process that creates i_{sys} does not necessarily happen synchronically with p_e, but it also corresponds to p_{sim}. At a computation step of this process, i_{sys} accesses the state values of i_{sf} corresponding to the same time in p_{sim} as this step does. To the simulated system represented by i_{sys}, the simulated phenomenon represented by i_{sf} is a part of its simulated environment that influences its behaviors. This is an imitation of the real-world that an environmental phenomenon has influences on a system of interest.

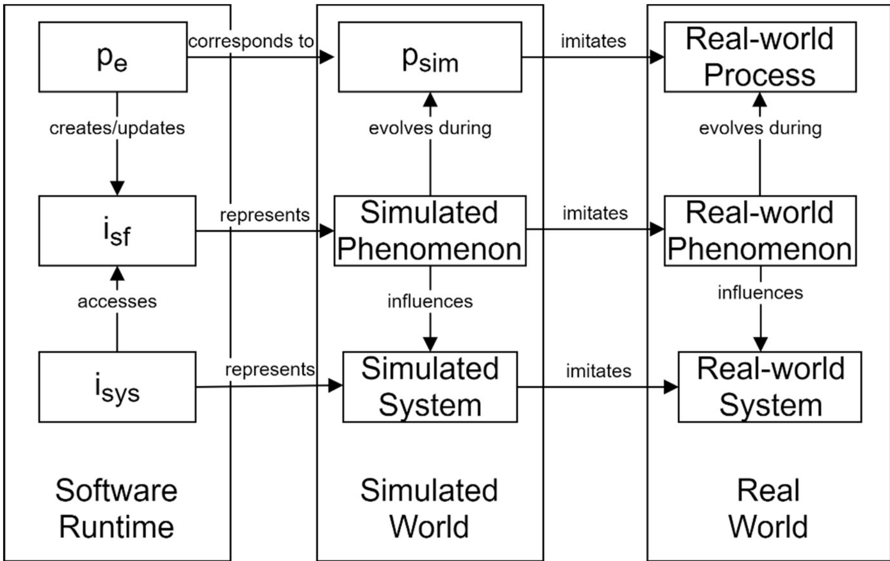

Figure 5.7: Conceptual Links of Entities Related to a Simulation.

Since the existence of i_{sf} is observable and identifiable at each step in its lifespan during p_e, it represents a simulated phenomenon as a substance. This means the existence of its presented simulated phenomenon is not dependent on the time domain. It "lives in" the simulated world and "evolves" over time during $p_{sim.}$ Conceptually, the state of i_{sf} provides a "snapshot" of its represented phenomenon at a corresponding time on p_{sim}. Thus, properties representing characteristics of simulated phenomena in the classifier of i_{sf} (i.e., a class applying the SimulatedFeatureType stereotype) normally only need to provide structure to hold values valid for an instant (more precisely, a minimum identifiable temporal unit). In this class, properties representing characteristics with a temporal type should be viewed in the same way as the properties representing thematic characteristics[12]. It means that the classifier of i_{sf} shall have a property of a temporal type to represent a characteristic of simulated phenomena. Nevertheless, a state value of this property owned by i_{sf} at s reflects the condition of represented characteristic valid at an instant in $p_{sim.}$

A class applying the SimulatedFeatureType stereotype must have an attribute *timestamp: TemporalPosition* (See Subsubsection 5.2.2.2). At an execution step, this attribute of an instance holds a location in time, which represents the "current time" that the represented phenomenon is experiencing in the simulated process. This attribute is not considered as a characteristic of the simulated phenomenon but the coordination on the timeline of the simulated process. The existence of the timeline is independent of the existence of the phenomenon in a simulation. In a computation step, the state value of this attribute owned by an instance denotes the valid time in the simulated process of other state values owned by the same instance.

The SimulatedFeatureType stereotype does not add restriction to prohibit adding thematic properties to a class applying this stereotype to hold a series of historical values. However, it is not recommended as described at the beginning of this subsubsection. No transformation rules defined as the SEDL execution semantics (see Section 5.3) leads the creation of such properties in a SimulatedFeatureType class. Functions of a specific application that must be satisfied with such properties cannot be determined at the domain level and thus are not captured by this profile. When they are needed by a specific application, they have to be added by developers at later phases.

This profile specifies specialized subtypes of the SimulatedFeatureType to express a class in more detail. Each of the subtypes restricts one representation for the spatial extent of represented phenomenon types. These subtypes are specified based on the runtime schemes that are needed for different spatio-temporal data synthesis methods and spatial simulation paradigms. A specialization of the SimulatedFeatureType specifies the following aspects of a SimulatedFeatureType class: 1) how the extent in space of a simulated phenomenon type is represented by this class; 2) how the property values representing its thematic characteristics are linked to its spatial extent. The available stereotypes for runtime simulated feature types are shown in Figure 5.8 and are specified in detail in the following subsubsections.

5.2.2.4 Single-Valued Feature Types

A class that applies a stereotype introduced in this subsubsection should provide slots to hold a single value for each thematic characteristic of the represented type. The "single" means, despite that this value shall be composed by several numbers or other primitive types or a SpatialFunction class as defined in Subsubsection 5.2.2.7, all its parts together represent the condition of this characteristic. In practice, each thematic characteristic should be represented as an attribute owned by this class.

A GlobalFeature is a SimulatedFeatureType whose represented phenomena type is computed as pervasive and homogenous in the simulated world. In the view of the computation functions that use this class, the boundary of its represented phenomenon type is not reachable in the simulated world. Its real-world counterpart may be non-physical by nature or has a spatial extent that is much larger than the world part being simulated. Spatial heterogeneity of its thematic characteristics is either not of interest to the target simulations or not recognizable at the resolution of simulations. Thus, thematic characteristics of a

[12] For simplification, a property representing a thematic characteristic is referred to as "thematic property" in the following text, and a value of a thematic property is referred to as a "thematic value".

phenomenon of this type are viewed as homogenous at all spatial locations. The computation functions calculate one value to represent one of its thematic characteristics at each execution step.

No spatial representation is needed in this class since its represented phenomenon type is conceptually everywhere to the computation functions. For an instance of this class, the state value of such an attribute is considered applicable to represent the condition of the represented characteristic by the attribute at any spatial location, and this condition is valid at the time represented by the state value of the *timestamp* attribute of this instance.

A LocalFeature is a SimulatedFeatureType whose represented phenomena type is computed as occupying some identifiable part of the space. For the computation functions that use this class, the spatial extent of its represented type is recognizable in the simulated world and may be calculated by the model at each computation step. One value is calculated to represent one of its characteristics at each execution step. Spatial heterogeneity of its thematic characteristics is either not of interest to the simulations, or not recognizable at the resolution of simulations this type serves.

A class that applies the LocalFeature stereotype must have a *geometry* attribute that should be a geometry type from the utility datatypes in Subsubsection 5.2.2.2. It is the only spatial representation of the represented phenomenon type. Other attributes with a geometry type should be interpreted in the same way as thematic properties. This class should represent a thematic characteristic as a single value, i.e., as an attribute owned by this class with the multiplicity of 1.

Instances of a LocalFeature appear and evolve independently from each other in a simulation. This stereotype does not restrict the relation among instances of the same class or additive effect of multiple instances at a location since they are specific to an application. They are described at the M1-level application models. The opposite case is covered by the CollectiveFeatureType stereotype introduced in the following two subsubsections. A class applying a subtype of CollectiveFeatureType expresses a set of phenomena whose behaviors can be modeled by some common regulations. Such a set of phenomena can be viewed as an integrated whole that exhibits some spatial pattern at a time instant.

5.2.2.5 Collective Feature Types

A CollectiveFeatureType is an abstract SimulatedFeatureType whose represented phenomenon type is computed as a set of units, each of which occupies a spatial location in the simulated world. All these units have the same logical structure, which consists of a geometric representation and a set of attributes representing thematic characteristics. The CollectiveFeatureType stereotype regulates a structure that holds multiple state values about a set of thematic characteristics at runtime. A set of values of a thematic characteristic is calculated at each execution step. Each value is held by an attribute representing this characteristic owned by a unit. This value is paired with a spatial location through this unit's geometric representation. Since this attribute is owned by all units, values of this attribute from all units together reflect the represented characteristic.

Computations that generate spatial patterns changing over time often require a runtime structure as regulated by CollectiveFeatureType. The structure is discrete to be handleable by computers so that computation functions using such a structure can perform on a finite set of individuals (corresponds to units). Thus, the CollectiveFeatureType stereotype is specified based on the concerns of spatial simulation models. To distinguish these models with models that express software and data, simulation models in a mathematic context are mostly referred to as "computational models" in this thesis.

A CollectiveFeatureType must have one and only one *unit* whose type is a class applying the CollectiveFeatureUnit stereotype. This CollectiveFeatureUnit class represents the structure of its units. A CollectiveFeatureUnit must have a *geometry* attribute whose type can be one of the geometry types from Subsubsection 5.2.2.2. A CollectiveFeatureType should also provide access operations to its units so that developers can design and implement application-specific computations on these units, as specified below and in the next subsubsection for the subtypes of CollectiveFeatureType. These operations are stereotypes of uml: Operation. For different CollectiveFeatureType classes, their units have different attributes representing thematic properties and lead to different return types of such an operation. They cannot be simply represented as a fixed operation within the stereotype representation in

Figure 5.8. Complementary information about these operations' stereotypes is given in Subsubsection 5.2.3.1, where behavioral elements in the PIM-layer metamodels are summarized. These operations are specified at the PIM layer to emphasize the logical functionalities a stereotyped class should have. This specification does not regulate a fixed choice of implementation signature for these operations.

A CollectiveFeatureType class links the view of computational models and the view of external components. Computed patterns of a collective feature are exhibited as thematic properties of the macro phenomenon type represented by the CollectiveFeatueType class. Thus, a CollectiveFeatureType is viewed as has the same list of properties as its units. For each of the properties P, an access method *getP(Point p)* should be provided by the class. It returns the state value of the property P at the input point location.

Locations of units during computation may be denoted by simple internal coordinates (e.g., integer index) that reflect units' locations related to the spatial span of computation. External components cannot recognize these coordinates and thus cannot relate associated thematic values to their own world. Each CollectiveFeatureType should provide an operation *getUnit(Point p)* to return an object of its unit type. This operation returns the unit of its belonged feature, which the input point location intersects with, or null if there isn't any.

Computations of a collective feature are performed on a set of units. A CollectiveFeatureType should provide a method *unitsIterator()* which returns an iterator object of its units. Then, developers can use this iterator to traverse through units to perform update functions on a unit. The implementation form of this iterator depends on the chosen implementation platform.

5.2.2.6 Subtypes of CollectiveFeatureType

The spatial representation of units in a collective feature includes the geometry of computed units and spatial relationships among them. The choice of units' geometry type influences computation results and exhibited spatial patterns of computed thematic characteristics[103]. Spatial relationships among units such as the distance and the connectivity are even more important, since behaviors of units are often computed based on distances and connectivity among units.

To support the concise expression of units' spatial representations, this profile introduces concrete subtypes of the CollectiveFeatureType as described in the following paragraphs. Each subtype restricts the CollectiveFeatureType with a specific spatial representation that is often used by simulation modelers, which includes the geometry type of its units and the way that these units are distributed. Figure 5.9 shows an illustration of these representations with one unit highlighted to support understanding and choosing these stereotypes. The representation is chosen by modelers who develop the computational models of environmental phenomena and can only be included in the software design models with their involvement. Thus, these subtypes are specified with the notations that are close to the terminology in computational schemes of spatial simulations.

A central issue to the computational models using a CollevctiveFeatureType is to specify the spatial closeness and the neighborhood of a unit[103]. They determine which other units are considered for updating state values of a unit and their weight of influence on this unit. Spatial closeness among units is described in terms of distances from the unit to other locations in the space, especially to the locations of other units. Two types of distances may be used by these models. The first is the geometric distance that can be calculated based on the geometric coordinates of units. The second is the graph distance (also called the network distance) that is described in terms of the minimum steps connecting two units, given that the distance between two directly connected units is one step. The graph distance can both be grounded on geometric adjacency or artificially created connectivity among units. The former can be derived from the spatial distributions of units, while the latter can only be additionally specified. More complex networks may also include variable costs for different steps, which vary from features to features.

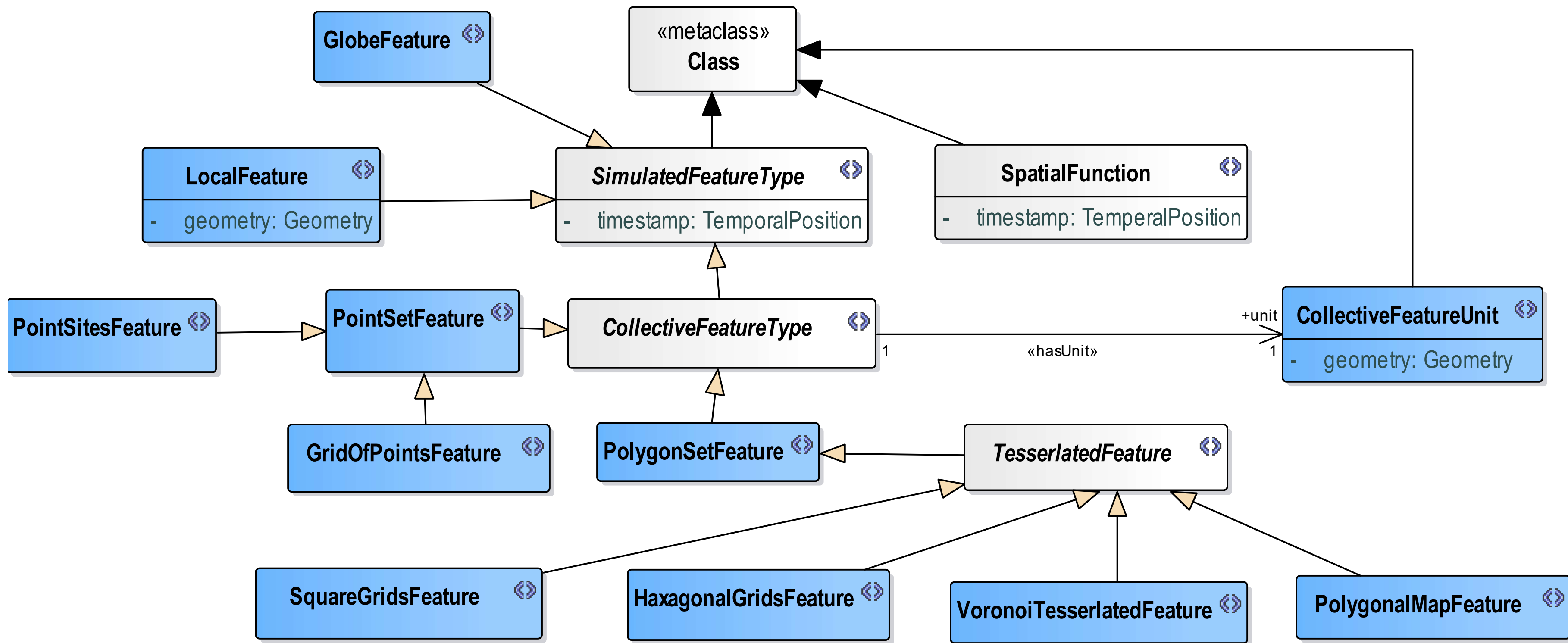

Figure 5.8: Stereotypes of Runtime Simulated Feature Types.

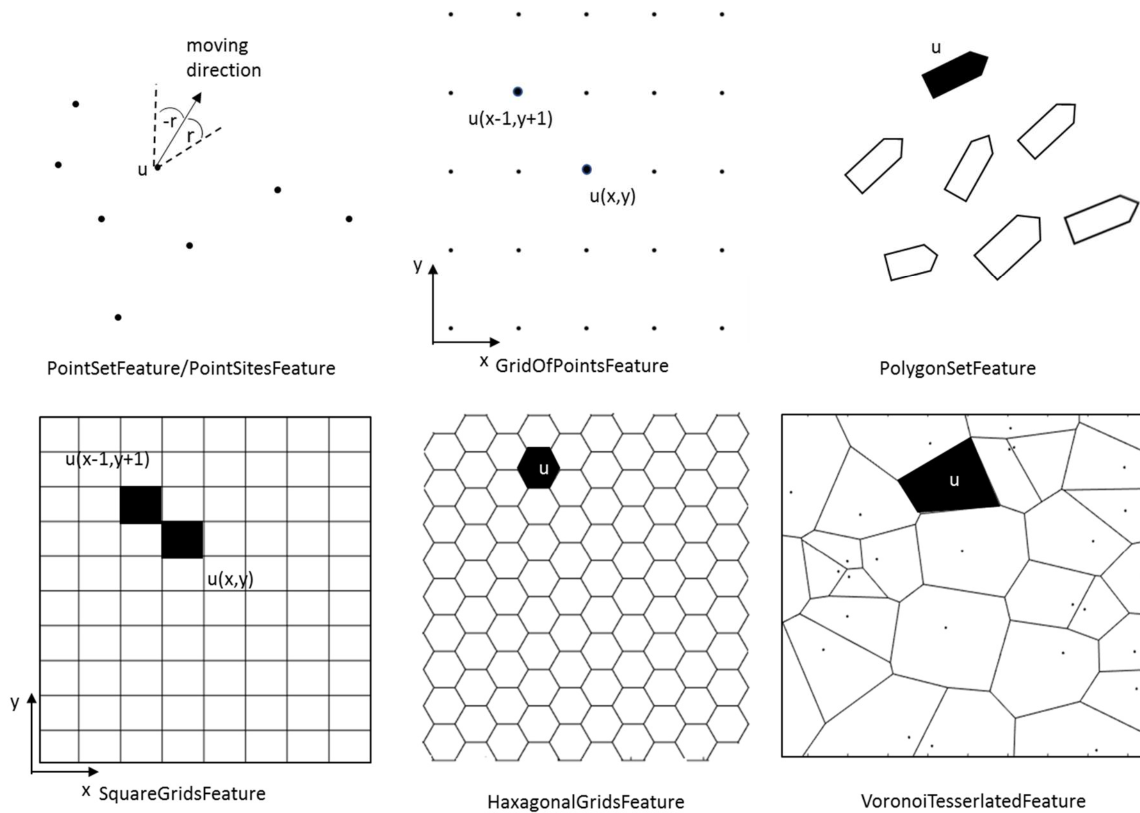

Figure 5.9: Geometry Illustration of CollectiveFeatureTypes.

Spatial distances between units that are determined from the geometry and topology of units can be computed without knowing the application-specific thematic properties carried by these units. Thus, relevant methods on accessing the neighborhood of a unit based on such distances are specified for each subtype of CollectiveFeatureType. They can be realized at the stereotype-level and be integrated into the implementation of stereotyped classes via transformations so that developers can utilize them to implement application-specific behaviors of units.

A PointSetFeature is a CollectiveFeatureType whose geometry is computed as a set of freely movable points. The geometry of a unit is presented by a point in this set, which holds the unit's location. No spatial relation among points is fixed by this stereotype. The location of each unit can be updated separately during computation. Thus, the neighbor units of a unit shall change with its location change. The neighborhood of a unit is determined via geometric closeness by computational models.

A PointSetFeature class should provide an operation *getNeighbors (Unit u, float distance, float -r, float r): Set <Unit>*. The type *Unit* is the unit type of this PointSetFeature, which is a class applying the CollectiveFeatureUnit stereotype. In the following paragraphs, unit types are denoted as *Unit*. It needs to notice that these types are different from model to model. This operation returns a set of neighbor units of u within the geometric distance of *distance*, which has the same measuring unit as the one used by geometric coordinates of this class. The parameters *-r* and *r* are optional, which further restricts the range of returned neighbors based on the at-moment moving direction of u as shown in Figure 5.9. Their values shall be ignored when u is not moving.

This representation is chosen by bottom-up computational models simulating multiple autonomous moving entities, which have been briefly introduced in Section 3.3.1.

A PointSitesFeature is a PointSetFeature whose units are represented as a set of points with fixed locations in the view of computational models. The geometric distance and the relative direction between each pair of points in a PointSitesFeature are fixed during computations. Each unit in such a set corresponds to a "site" in the spatial simulation terminology[103]. The geometry of this stereotype is comparable to the domain of CV_DiscretePointCoverage[97] in ISO19123.

This structure is chosen when simulating time series of one or more themes at a limited set of irregularly distributed locations. Models of this kind draw a temporal process at each site and take the spatial autocorrelation[132] among sites into account. Spatial relations among units that have effects on computations of themes can be calculated or assigned upon instance initialization and are stored in memory during a computation. The effect on space is often stored in the form of a spatial weight matrix with numerical weights[132].

Bottom-up simulations shall also use this structure. Thematic values of a unit are calculated based on only neighbor points. Two types of distances can be derived from the units' spatial locations of a PointSitesFeature to specify the neighborhoods of its units, i.e., the geometric distance inherited from PointSetFeature and the graph distance of the TIN[13] based on its member points.

In addition to the inherited *getNeighbors()*, a PointSitesFeature class with the unit type *Unit* should have an operation *getTINNeighbors(Unit u, int distance):Set<Unit>* to return the neighbor units of u within the graph distance of *distance* on the implicit TIN by this PointSitesFeature.

A GridOfPointsFeature is a PointSetFeature whose units are represented by a regular grid of even-spaced points. The computed thematic values of a unit are related to the point location of this unit. During computation, the thematic values of units shall be stored in a value matrix and are accessed via indexes denoting the relative locations of the units within the grid. The geometry of this stereotype is comparable to the domain of the CV_DiscreteGridPointCoverage[97] in ISO19123, while the geometry of its unit is comparable to CV_GridPoint[97].

This representation is often chosen when the computational models are based on the finite difference scheme as introduced in Subsection 3.3.1. Points in such a grid correspond to discretized sampling points of differential models. Computations calculating thematic values on a grid that are valid in grid cells should use a SquareGridsFeature class, which is introduced in later paragraphs.

Bottom-up simulations may also use this type. Graph distances are used to compute neighborhoods of units in a GridOfPointsFeature. Two definitions of adjacency can apply to this type for computing the graph distance. The first considers only the points that are orthogonal next to a point as its adjacent points (i.e., Von Neumann neighbors). The second additionally includes the points that are diagonal next to it (i.e., Moore neighbors).[111] The distance between two adjacent units are equal to 1 in this specification. A GridOfPointFeature should have two operations to return these two types of neighbors of a unit within a graph distance of *distance*, i.e., *getVonNeumannNeighbors (Unit u, int distance): Set<Unit> and getMooreNeighbors(Unit u, int distance):Set<Unit>*.

Besides, a GridOfPointsFeature class should provide an operation *getUnit(Unit u, int x, int y)* to access a nearby unit of u based on their relative positions in the grid. As shown in Figure 5.8, for a unit *u* with the grid index (x, y), *getUnit(u, -1, 1)* of this feature returns its unit with grid index (x-1, y+1). It returns null when no such units exist.

A PolygonSetFeature is a CollectiveFeatureType whose geometry is computed as a set of freely movable polygons. The geometry of a unit is presented by a polygon in this set, which holds the unit's location. The location of each unit can be updated separately during computation.

A PolygonSetFeature class should provide an operation *getNeighbors (Unit u, float distance): Set <Unit>*. This operation returns a set of neighbor units of *u* within the geometric distance of *distance*, which has the same measuring unit as the one used by geometric coordinates of this class. This specification regulates that this geometric distance should be computed between the boundaries of two units.

Computations of a set of moving entities normally calculate locations of entities based on some reference points on the entities (e.g., centroids of entities). Entities in flops with higher spatial geometric dimensions in an application shall be built using computed points as centroids of entities. This stereotype is defined here for completeness reason, in case some deformation of units is documented in SEDL. When no change about the units' shape needs to be computed, the PointSetFeature stereotype should be used.

[13] see the last paragraph of this subsubsection for the definition of TIN

A TersserlatedFeature is an abstract PolygonSetFeature whose geometry is represented by a set of polygons covering a spatial area without gaps or overlaps. The term “tessellation” implies the subdivision of space[103]. The geometry of each unit is represented by one of the regional divisions (i.e., the polygons) in this tessellation. Thematic values of the represented phenomenon type are paired with regional divisions. Each unit is treated as an individuality (i.e., a site or a cell in the spatial simulation terminology[103]) during computations.

An instance of a TesserlatedFeature class can be viewed as an implicit graph, with each unit corresponds to a vertex, and each pair of adjacent units are connected by an edge. Distances among units during computations are represented in terms of graph distances. Two adjacent units have a distance of one step to each other. The neighborhood of a unit within a range of r includes all units in the tessellation, which have a graph distance that is not greater than r to this unit.

This structure is chosen by cell-centered computational models such as bottom-up cellular automata and models about underlying space altered by active entities (See Subsection 3.3.1). The following subtypes support concise expressions of common styles of tessellations used by these models. These subtypes have different definitions of adjacency.

A SquareGridsFeature is a TesserlatedFeature whose geometry is represented by a grid of square cells. This grid is created through regular tessellation by two perpendicular sets of even-spaced lines. The geometry of each unit (i.e., a site or a cell) is represented by one of the square divisions. This geometry is the most fundamental tessellation type used as lattices in spatial-explicit simulation algorithms. It is comparable to the underlying discrete geometry of the CV_ContinuousQuadrilateralGridCoverage[97] in ISO19123, while the geometry of its unit is comparable as CV_GridCell[97].

Similar to a GridOfPointsFeature, the thematic values of an instance of this class are stored in a value matrix during a computation. The values of a unit can thus be accessed via indexes in the matrix, which denotes the relative location of the unit within the grid. The GridOfPointFeature and the SquareGridsFeature are made two stereotypes for clear distinction of the unit geometry which influence the evaluation of thematic values at a spatial location based on computed unit values.

It is important for cell-based computational models that all thematic values are recorded at the same scale. As the units of a SquareGridsFeature are created purely based on the geometric subdivision, all units should have the same size to be at the same scale. More complex computations are decomposed into hierarchies, while thematic values at each hierarchy are still simulated in a lattice with the same unit size. This stereotype does not consider the grid structure with variable cell size as may appear in the data storage.

The two types of adjacency that can be applied to GridOfPointsFeature can also be used for computing graph distances among units in a SquareGridsFeature instance. In the case of Von Neumann neighbors, a cell is adjacent to the four cells that are orthogonally connected to it. In the case of Moore neighbors, a cell is adjacent to the eight units that surround it. Similar to a GridOfPointsFeature, a SquareGridsFeature should have two operations to return these two types of neighbors of a unit within a graph distance *distance*, i.e., *getVonNeumannNeighbors(Unit u, int distance):Set<Unit> and getMooreNeighbors(Unit u, int distance):Set<Unit>*.

Also, a SquareGridsFeature class should provide an operation *getUnit(Unit u, int x, int y)* to access a nearby unit of u based on their relative position in the grid as shown in Figure 5.9. For a unit *u* with the grid index (x, y), *getUnit(u, -1, 1)* returns the unit of its belonged feature with grid index (x-1, y+1). It returns null when no such unit exists.

A HaxagonalGridsFeature is a TesserlatedFeature whose geometry is represented by a tessellation that is composed of regular hexagons (i.e., 6-sided polygons whose sides have the same length) with the same size. Hexagons are 6-sided polygons. The geometry of this stereotype is comparable to the underlying discrete geometry of the CV_HexagonalGridCoverage[97] in ISO19123. The typical orientations for hex grids are vertical columns (flat-topped) and horizontal rows (pointy-topped). The advantage of computing a phenomenon using the hexagonal grids compared to using the square grids is

that only one possible adjacency exists among units in the hexagonal grids. Each pair of adjacent units share a common borderline.

A VoronoiTesserlatedFeature is a TesserlatedFeature whose geometry is generated from a set of irregularly distributed points (referred to as “seeds”, “seed points” or “generating points”) as follows: the spatial space is divided into cells represented by polygons; each cell is associated to a seed; each cell covers the region containing all points in the space which are closer to its associated seed than to any other seed points. Such a geometry is often called a Voronoi tessellation or a Voronoi diagram which is comprehensively introduced in [133]. Another name of such divisions is the Thiessen polygons known in the spatial analysis domain.[134] The geometry of this stereotype is comparable to the underlying discrete geometry of the CV_ThiessenPolygonCoverage[97] in ISO19123. Thematic values of the phenomenon type represented by a VoronoiTesserlatedFeature are paired with the polygonal cells as units during computations. Two Voronoi units that share a common borderline are adjacent and have a graph distance of one step to each other.

Classes applying above two stereotypes should have an operation *getGraphNeighbors(Unit u, int distance):Set<Units>*. It returns neighbor units of a unit within a graph distance of *distance* based on the adjacency definition of the applied stereotype.

A dual diagram named Delaunay triangulation[134] can be created for each Voronoi diagram by connecting all pairs of seeds associated with two adjacent Voronoi units. It results in a collection of triangles. For any of the triangles in this collection, no other point in the seed set is inside its circum-ball[133]. This triangulation is often named TIN (Triangular Irregular Networks) in spatial analysis.[134] The graph distance between two Voronoi units is equal to the graph distance among their associated seed points in its dual TIN. In a TIN, thematic values are paired with the seed points that are vertices of triangles but do not fall inside to any divisions. This structure is often used for interpolating values of points other than seed values. However, conceptual confusion appears when it is used as the lattice by cell-based models that compute the values of divisions. Thus, the Delaunay triangulation is not introduced for lattice in this profile, but only as a network type for computing graph distances between units in a PointSitesFeature. Discussions regarding network features can be found in Subsection 8.3.1.

A PolygonalMapFeature is a TessellatedFeature whose geometry is represented by irregular polygons imported from existing polygonal map data. The geometry of this stereotype is roughly comparable to the domain of CV_DiscreteSurfaceCoverage[97] in ISO19123. Visually, it looks like the polygonal map. Such a tessellation type provides a more flexible alternative to regular geometric subdivisions, which is particularly useful for simulations based on artificially determined regions, such as simulations based on data of administrative divisions.

Two units in a PoygonalMapFeature instance that touch each other are considered as adjacent, i.e., as direct neighbors with the graph distance of one to each other. A problem may be introduced by imported polygonal map data when isolated polygons that do not touch any other polygon may exist. This situation violates the strict definition of the tessellation since empty spaces exist among divisions. Thus, this stereotype uses a relaxed definition: the geometry of one or more units in a PolygonalMapFeature may be an isolated polygon that does not touches any other unit. Such a unit has no geometric neighbors by default.

A PolygonalMapFeature should have an operation *getGraphNeighbors(Unit u, int distance, Boolean includeIsolatedUnit):Set<Units>*. It returns neighbor units of a unit within a graph distance of *distance* based on the adjacency definition of PolygonalMapFeature. When the third parameter is set to *true*, this operation should treat the units that have the shortest geometric distance to u as its adjacent units. By this specification, this geometric distance should be computed between the boundaries of two units.

5.2.2.7 SpatialFunction

A SpatialFunction is a class that holds the computational function that represents the form of a spatially heterogeneous theme. It can be used in one of the following situations: 1) a thematic property of a phenomenon is represented by a continuous function from spatial locations to thematic values at a time instant. No regulation about discrete spatial samplings is fixed yet. 2) information about this

property is acquired from a spatial data source by external providers, which differs from location to location. In this case, this class serves as a wrapper to hold spatial queries to the data source.

Same as other stereotypes for runtime objects representing environmental phenomena in simulations, a SpatialFunction conceptually represents some existence in simulated space that evolves over the timeline. To external components, this class represents a thematic property whose values can be "asked" by given the location of a spatial point at each time instant. Since points in a conceptually continuous space are infinite, these values are only computed (i.e., calculated or queried) when necessary upon request from client systems or moderation functions. A SpatialFunction at a time instant is comparable to an analytical coverage in ISO19123 that maps spatial locations to thematic values via a mathematic function[97].

Formally, a class applying the SpatialFunction stereotype must have an attribute *timestamp: TemporalPosition*. Similar to the *timestamp* of a SimulatedFeatureType, it provides a slot to hold the "current time" during execution. Besides, this class should have an operation *eval(point)*. This operation takes a spatial point as the input and returns the value of its represented property at this point. Developers should implement the spatial function f(s) representing a thematic property in this operation.

A SpatialFunction class could be generated by the transformation from an SEDL description (See Section 5.3) as the type of some SimulatedFeatureType's attribute, when some ThematicValueDistribution in the input exists. If its represented property is also expected to be dynamic, the transformation generates operation to update the state of this attribute at an execution step. This operation updates necessary parameters of the embedded spatial function $F_t(s)$ in the *eval(point)* of this SpatialFunction to alter its at-moment form. The parameters of the spatial function can be implemented as attributes of this SpatialFunction class. At an execution step, invoking the *eval(point)* via an instance of a SpatialFunction class with parameter *s* returns the represented property value of this instance, at the location of *s* and at the time that is the state value of its *timestamp*.

5.2.2.8 Requested Snapshots

A snapshot is the state of a system at a temporal location. At a step of a simulation execution, the system of interest component consumes snapshot data of its simulated environment at the temporal location corresponding to that step. This subsubsection specifies additional stereotypes as shown in Figure 5.10, which are used to express the information structure requested by the system of interest component. They are subtypes of the stereotype Snapshot that extends the uml: Datatype metaclass.

A Snapshot represents the data structure about a simulated environment being sent to a system of interest component at a simulation step. Different from the subtypes of SimulatedFeatureType which are also defined from a snapshot view, the Snapshot stereotypes describe static pieces of message data that are sent between components. These stereotypes shall be used in design models in the following situations.

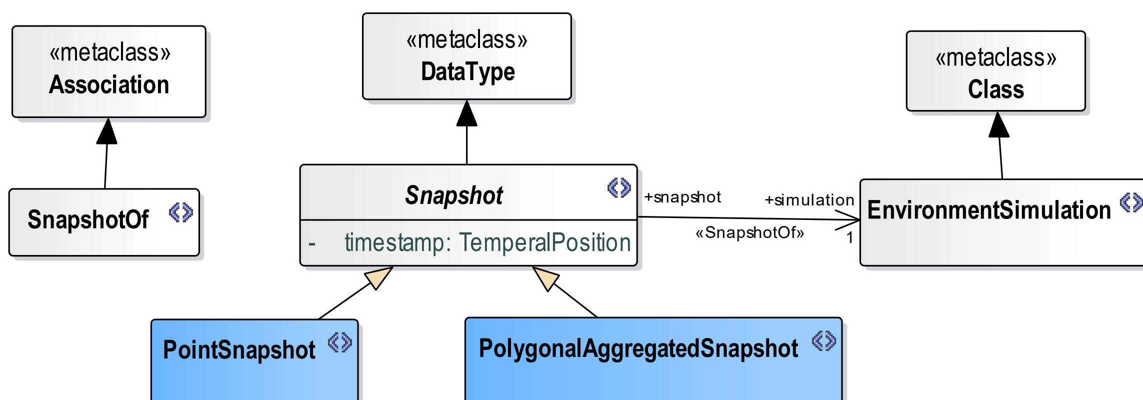

Figure 5.10: Snapshot Types.

1. The system of interest component requires a subset from all computed values or some post-processed values based on computed values. Structures of required values expressed by snapshot types are used by communication functions between components, which is specific to a simulation process, while runtime data structures expressed by simulated feature types are needed for computations of phenomena, which is specific to a simulated environment component.

This could happen when computed values are consumed by multiple functionalities. It may also happen when environmental phenomena are computed as CollectiveFeatureTypes. Thematic values at a location may not be independently updated but depending on nearby values, which requires a runtime data structure to hold values from multiple locations. Although, the requested snapshot data by a client component shall be only from one location.

2. Some phenomenon property is implemented as a SpatialFuction (see Subsubsection 5.2.2.7). Since computable state values from this function are infinite at each execution step, Snapshot types are used in this situation to guides its execution loop in an application. In this case, a Snapshot expresses which set of values is requested by another component and thus should be computed by the embedded spatial function at an execution step.

A Snapshot must have an attribute *timestamp: TemporalPosition* that denotes the time when a snapshot is taken. In the design model, it must be exactly one *memberEnd* of an association that applying the SnashopOf stereotype which is introduced later in this subsubsection.

To facilitate the object-oriented modeling, a stereotype EnvironmentSimulation extending uml:Class is introduced to provide a model construct that holds behavior elements for simulation execution control. Same as the Snapshot types, EnvironmentSimulation classes are more related to the simulation that the component under development participates in, but not the computation models of a phenomenon type. As a meta element that aims at expressing behaviors, it is specified in more detail in Subsubsection 5.3.3.6. Relevant information for this subsubsection is that an EnvironmentSimulation class maintains several sets, each of which maintains computation instances for a SimulatedFeatureType class. The structure of a Snapshot type depends on the SimulatedFeatueType computation that can be maintained by the EnvironmentSimulation class and thus are partially derivable from this class.

To support concise expressions of this dependency, the profile defines a stereotype SnapshotOf that extends the UML metaclass Association. A SnapshotOf is a binary association. It must have a *memberEnd* (*simulation*) which is an EnvironmentSimulation class and the other *memberEnd (snapshot)* which is a Snapshot class. A Snapshot must be exactly linked by one SnapshotOf association. A SnapshotOf association indicates instances of its *snapshot* end includes snapshot information from the instances of its *simulation* end. Thus, the derivable structure of its *snapshot* end can be implicit in a design model applying this profile. It results in more concise model representation. During transformations from the design model to implementations or intermediate models in general modeling languages, derivable attributes of a Snapshot type should be made explicit in the outputs, as specified below for each subtype.

A PointSnapshot is a Snapshot representing the snapshot data structure of a simulated environment from a simulation at a point location. It has an attribute *location* of the *point* type that represents the location at which a snapshot instance is taken. A PointSnapshot has the following implicit structures:

1. For each feature type class (see Subsubsection 5.2.2.4) whose computation instances can be maintained by the *simulation* end linked to the PointSnapshot, a member datatype resembling this feature type should be included in this Snapshot type. An instance of this snapshot type can contain one or more instances of the member datatype.

2. When the feature type is single-valued, the member datatype implicitly has all attributes representing thematic properties of that feature type. If some attribute of the associated feature type is a SpatialFunction, the type of the corresponding attribute in this member datatype should be changed to the return type of the embedded function of the SpatialFunction.

When the feature type is a CollectiveFeatureType, the member datatype implicitly has all thematic attributes representing thematic properties of the feature type's *unit*.

A PolygonalAggregatedSnapshot is a Snapshot representing the data structure of the aggregated snapshot of a simulated environment from a simulation within a region. It has an attribute *location* of the

polygon type which represents the region at which a snapshot instance is taken. For each single-valued feature type class whose *geometry* type is *point*, a member datatype is included to this Snapshot type with a *location:Point* attribute and all attributes representing thematic properties of that feature type. Other of its implicit structure is derived in the same way as the PointSnapshot. An attribute value of such a snapshot instance is computed via some kind of aggregation (e.g., mean) from the same attribute of its snapshot feature within its *location* at its *timestamp*. The specific way of aggregation is left to application implementation.

5.2.3 Metamodel of Environment Computation

Meta elements for describing PIM-layer behavioral models of the environment computation in this thesis have a more complex composition than the metamodels for describing structural models. They are not defined simply within a profile but are from different sources as summarized in this subsection.

5.2.3.1 Summary

CIM-PIM transformations in the proposed framework map SEDL elements to computation units and chain these units together. They create computation flows for simulated feature types at an execution step. Given connected units that are formally expressed, the generation of architectural code and object flow code shall be automated to create computer program skeletons for enclosing application-specific implementations. Developers can focus on implementing computational logic and arithmetic functions. The generated units are application-specific elementary functions that do not necessarily contain domain-level common structures. Thus, instead of defining redundant stereotypes, the transformation rules map relevant SEDL language elements to the behavioral elements in the UML metamodel. Besides, since these functions operate on data objects about environmental phenomena, the transformed behavioral models may also contain elements from the structural models that are expressed in previously introduced metamodels.

Elements from the above two sources could appear in the output behavioral models by CIM-PIM transformations from SEDL descriptions. Further, stereotype-specific behaviors for SimulatedFeatureType-s have been specified in the structural profile, in terms of stereotyped uml: Operation that must be owned by a class applying some subtype of SimulatedFeatureType. These operations are common in spatial simulations but could be used differently in different applications depending on computational methods in these applications. Thus, the specified CIM-PIM transformation in this thesis does not involve the generation of behaviors that invoke such operations. When objects of stereotyped classes have been created in a behavioral model by transformations at the PIM layer or a further-mapped PSM layer, developers can invoke these operations through the objects to construct application-specific behaviors within a generated unit.

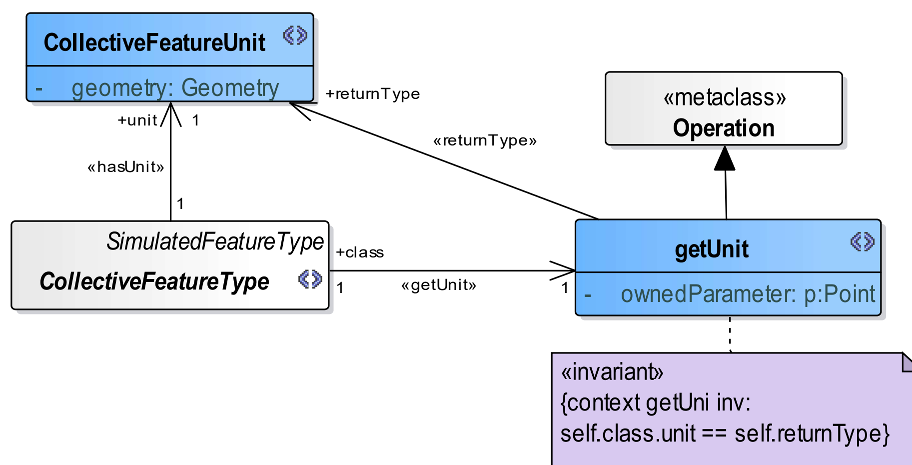

Figure 5.11: Define the getUnit() Operation for the CollectiveFeatureUnit Stereotype.

At the profile level, operations owned by stereotypes of uml:Class are viewed as stereotypes of uml:Operation. Figure 5.11 gives an illustration that defines *getUnit()* for the CollectiveFeatureType in Subsubsection 5.2.2.5 using the graphic notation, with additional constraints expressed in OCL. This figure shows, each class applying the CollectiveFeatureType stereotype (or one of its subtypes) should own a special uml:Operation named *getUnit*. This operation has a parameter *p* of a *Point* type and returns an instance of a class that represents the units of this CollectiveFeatureType class.

Besides, a stereotype EnvironmentSimulation is specified to express model artifacts that enclose behaviors of an environment component in a bigger simulation. Such behaviors include communication routines between the environment component and a system of interest component, create necessary data messages, and so on. These behaviors are simulation-specific and shall be partially derived from a SimulatedEnvironement and one of its ExecutionRoutine. The derivation is explained in Subsubsection 5.3.3.6. An EnvironmentSimulation class and its further transformed code skeletons are used by developers to implement the intended execution routine.

At last, Subsubsection 5.2.3.3 recommends a set of stereotypes as meta elements for describing behaviors to modify the existence of a simulated phenomenon at the PIM layer. Same as the operations owned by SimulatedFeatureType stereotypes, the main audiences of these stereotypes are developers of the modeled components. They provide concise constructs for developers to model and implement simulation processes. Transformations from SEDL descriptions to PIMs do not involve the creation of instances of these stereotypes, since the way to use them is specific to different applications that cannot be captured formally by SEDL.

5.2.3.2 Two Views of Behavioral Models

Behavioral models can be presented in two views when using the UML graphic notation, and so do the behavioral meta elements. An element may be expressed by different UML metaclasses in these two views. Figure 5.12 illustrates how behavioral elements are presented in two views, in which elements with the same name represent the same model element. Behavioral meta elements in this chapter are specified in one of the views. They can be switched to their counterparts in the other view as explained below.

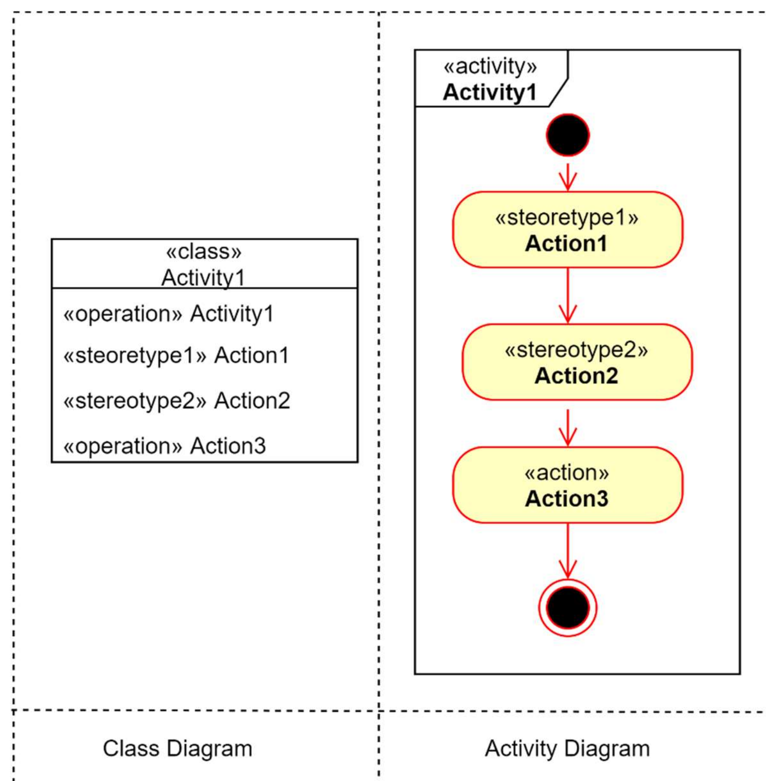

Figure 5.12: Elements in Instance Models of Behavioral Metamodels.

First, a computation procedure of a simulated environment component can be modeled as an instance of the uml: Activity metaclass presented in a UML activity diagram. It is composed of instances of available UML elements in activity diagrams. Stereotypes expressing behaviors are extensions of available UML metaclasses in activity diagrams, especially the extension for uml:Action. Second, each Activity can also be presented as a uml:Class in a UML class diagram. Each Action of this Activity can be then presented as an uml: Operation owned by the activity Class. Thus, an instance of some uml:Action stereotype can be presented as a stereotyped uml:Operation in class diagrams, while a basic uml:Action instance can be presented as a non-stereotyped uml:Operation. The Activity class normally also owns a uml: Operation that contains the behavior of this Activity.

5.2.3.3 Life Cycle Control of Simulated Features

This subsubsection recommends a set of PIM-layer stereotypes to provide additional constructs to express behaviors that control the lifecycles of SimulatedFeatureType instances or their units. They are summarized in Table 5.9 as extensions of the uml: Action metaclass. It is recommended to realize these constructs at the PIM layer and more specific layers as modeling/programming utilities in the proposed framework. They are not involved in CIM-PIM transformations specified in the framework.

CIM-PIM transformations in the framework proposed by this thesis create behavior models from SEDL descriptions in a logical structure that each instance of a computation class can be initialized with a configuration object of its computed phenomenon type[14]. A computation instance holds behaviors that compute a phenomenon instance. Its member object of a corresponding SimulatedFeatureType class hold state values of this phenomenon instance. Thus, all configured phenomena from a configuration are supposed to have been initialized at the beginning of an execution, even though they may be conceptually not alive for a while (state values of the feature data object remain zero or null).

However, the existence of phenomena may change during simulations. Some computation instances may not exist anymore, while some others may be initialized to compute newly emerged phenomena. In the current framework version, such behaviors can be added to the behavioral model since the PIM layer as actions expressed by stereotypes in this subsubsection. Instances of these actions, however, do not appear in the outputs of the CIM-PIM transformations from SEDL descriptions since no formal SEDL terms are defined for capturing such context.

Stereotype	Description
FeatureEmerge	An Action that makes a simulated feature come to life in the simulation. It often technically means to create and initialize a new instance of a computation instance for a SimulatedFeatureType.
FeatureTermination	An Action that terminates the life of an existing simulated feature in the simulation. This feature will not exist in the simulated world and will not participate in the simulation process anymore. Depending on the implementation strategy, it could technically mean to release a computation instance in the memory or exclude this instance in further computation. This Action should be distinguished from an Action that changes the simulated feature to state “dead”, which still has influence in the simulation. The latter Action should be viewed as an Action that computes thematic changes of a simulated feature.
MergeFeatures	An Action that merges multiple simulated features of the same kind into one. It takes two or more instances of the same SimulatedFeatureType as input and performs a FeatureEmerge Action to create a new instance of this type. The initial values of this new instance are computed based on the values of input instances. The input features are then terminated via a FeatureTermination.
FeatureAbsorb	An Action that one simulated feature absorbs one or more other features of the same kind. It takes two or more instances of the same SimulatedFeatureType as

[14] This PIM layer structure, however, does not have to strictly remain the same when mapped to a specific technical platform, as long as the mapped structure provides the same logical functionalities.

	input and updates state values of one of the features according to the input features. Other input features are then terminated via a FeatureTermination.
SplitFeature	An Action that splits one simulated feature into multiple features of its kind. It takes one instance of a SimulatedFeatureType as input and performs a FeatureEmerge Action a needed number of times to create two or more instances of the same type. The initial values of the new instances are computed based on the values of the input instance. The input is then terminated via FeatureTermination.
ReplicateFeature	An Action that a new simulated feature is created by an existing feature of the same kind. It takes one instance of a SimulatedFeatureType as input and performs a FeatureEmerge Action to create another instance of the same SimulatedFeatureType.
DuplicateFeature	A specialized ReplicateFeature Action that duplicates an existing simulated feature. It takes one instance of a SimulatedFeatureType as input and performs a FeatureEmerge Action to create another instance of the same SimulatedFeatureType with exactly the same state values as the input instance. The input feature is not altered by this Action.

Table 5.9: Stereotypes for the Simulated Feature Life Cycle Control.

5.3 Transformations of SEDL Descriptions

This section specifies transformation rules from SEDL descriptions to PIM-layer design models expressed by metamodels that are specified in Section 5.2. These rules regulate what happens when an SEDL description is executed by an implemented SEDL Description Processor.

5.3.1 CIM-PIM Transformation Process

The transformation from an SEDL description to design-level software models is a multi-steps and semi-automatic process, as shown in Figure 5.13. An implemented SEDL Description Processor should be able to perform automatic steps (denoted as small round shapes in Figure 5.13) in this process. An implemented SEDL IDE should also have functionalities that facilitate manual operations in this process.

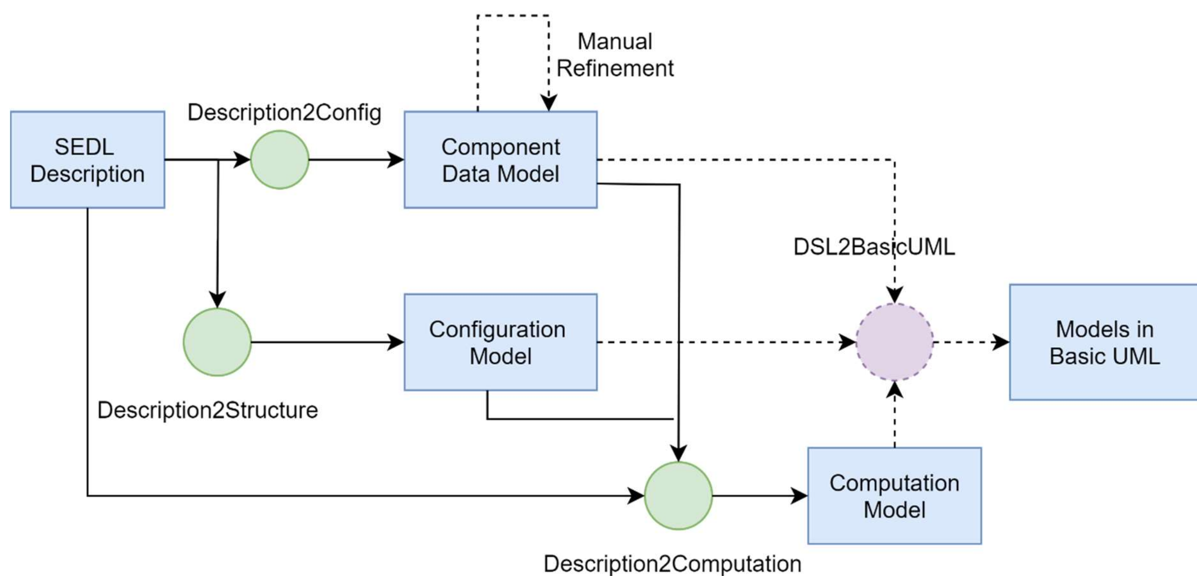

Figure 5.13: CIM-PIM Transformation Process.

First, two automatic transformations, i.e., **Description2Config** and **Description2Structure** are executed to derive two structural models from an input SEDL description. It generates a configuration

schema applying the profile specified in Subsection 5.2.1 and a data structure model applying the profile specified in Subsection 5.2.2. This step shall be followed by a manual refinement step (denoted as a dashed line as other optional elements in Figure 5.13), which is recommended in practice. At this step, developers of environment computation can bring more details that do not implicitly exist in high-level functional requirements into design models. Then, the two generated structural models together with the input SEDL are fed to a third transformation, i.e., **Description2Computation**. This transformation generates a behavioral model expressed by modeling constructs specified in Subsection 5.2.3. Data objects used by behavioral elements in this model are instances of the output model elements from the previous transformation steps.

The CIM-PIM transformations specified in this section has been completed by the previous three transformations. Detailed transformation algorithms of these three steps recommended by this thesis can be found in Appendix A.

The development process of a simulated environment component can be further automated from here through an optional transformation **DSL2BasicUML**. This transformation turns outputs of previous transformations to models expressed by basic UML constructs. This is archived by explicitly adding necessary structures to model elements applying stereotypes of profiles specified in Section 5.2. These structures are implied by the applied stereotypes as defined in their descriptive semantics. This step can be made implicit to language users by chaining it with a step that uses domain-independent generation tools to create PSM model or code skeletons. In this case, this optional step creates input for these general tools that do not recognize DSLs defined in Section 5.2.

5.3.2 Description2Config

The output of the first transformation **Description2Config** is a configuration schema of the component under development. Relevant aspects in an input description are the description hierarchy, names of Configurable and ConfigurableParameter-s. In a nutshell, it generates a hierarchical structure aligned to the input description. Each Configurable is transformed into a ConfigComponent with the name of this Configurable and is nested by the component transformed from its belonged Configurable. Each ConfigurableParameter is mapped to a ConfigItem of a corresponding ConfigComponent. Configurable-s with no parameter are excluded in this schema. Based on the component structure, a subtype of ConfigComponent may apply to this component. The complete transformation can be found in Appendix A.1**.**

This transformation is intentionally kept simple to create intuitive user interfaces for users who intend to use the developed component without modification, i.e., the "original" users who give the requirements as documented in the input descriptions. An output model of this transformation can be used to regulate the configuration structure of a developed component.

Since stereotypes in the configuration profile do not imply additional structure to its extended metaclass, removing the applied profile from an output schema model results in a valid model expressed by basic UML. When only the configuration structure is of interest, implementation of this transformation can be simplified. Instead of generating an element applying a stereotype, it generates an instance of the UML metaclass that the stereotype extends. The prototype implementation in Chapter 6 uses this strategy.

5.3.3 Description2Structure

The second transformation **Description2Structure** creates design-level structural models of environment components. For each EnvironmentalPhenomenon in the input SEDL description, it creates a uml:Class stereotyped with a suitable subtype of SimulatedFeatureType from Subsection 5.2.2. The choice of applied stereotype regulates the spatial representation of the phenomenon in the simulation. It is mainly derived from the declared dimension in the input description and IndividualtiyChange-s of the EnvironmentalPhenomenon. As explained in Subsubsection 5.2.2.1, this section focuses on specifying transformations for two-dimensional spatial simulations in which the highest dimension of a phenomenon is set to 2. Table 5.10 shows the primary step that applies a stereotype to the data structure Class SIData generated from a SpatialIndividuality SI. The applied stereotype needs to be refined with a more specific

subtype by developers for further transformation. Further attributes of the Class, which is not implied by its applied stereotype, are derived from the *theme*-s and enclosed Variation-s of the input SpatialIndividuality. Appendix A.2 provides the detailed logic of this transformation.

1	**If** dimNum of SI equals to 2 **Then**
2	Apply the *LocalFeature* stereotype to *SIData*;
3	Set *geometry* of *SIData* to *Polygon*;
4	**Else if** SI has change involving geometry **Then**
5	Apply the *LocalFeature* stereotype to *SIData*;
6	Set *geometry* of *SIData* to *Polygon*;
7	**Else if** dimNum of SI equals to 0 **Then**
8	Apply the *LocalFeature* stereotype to *SIData*;
9	Set *geometry* of *SIData* to *Point*;
10	**Else if SI has** RigidBodyMovement **or** LocationThemeDependency
11	Apply the *LocalFeature* stereotype to *SIData*;
12	Set *geometry* of *SIData* to *Point*;
13	**Else**
14	Apply the *GlobalFeature* stereotype to *SIData*;
15	**End if;**
16	**End if;**
17	**End if;**
18	**End if;**

Table 5.10: Apply a Stereotype to the Data Structure Class.

Contradictions may exist among different pieces of an SEDL description that influence the derivation of the necessary number of spatial dimensions in computation. They cause conflicts among different transformation rules. The conflicts are eliminated by the following three mechanisms: 1) Some contradictions are forbidden by the structural restrictions that are introduced with the SEDL model in Section 5.1. They can be eliminated by an SEDL editor with validation functions. 2) Change types involving space need a certain minimum number of dimensions. If the declared dimension of a phenomenon is lower than what its changes require, an SEDL editor should give warning. The contradiction should be clarified and eliminated through discussion. There, SEDL plays its role as an analysis-phase communication tool. 3) if the contradiction is not solved by step 2, the transformation generates a geometric representation of this phenomenon with a minimum necessary number (or a pre-determined higher one) of dimensions.

Given a chosen implementation platform, classes in an output model from this transformation can be further mapped to a spatial data structure implementation according to the applied stereotypes and attributes derived from the input description.

5.3.4 Description2Computation

The third transformation **Description2Computation** has two main purposes: 1) generating computation units that update states of simulated phenomenon instances during simulation; 2) building activity flows of the component under development with these units.

Design-level outputs from the third transformation can be presented in two views as explained in Subsection 5.2.3.2. To facilitate the further transformation to object-oriented code structure, this subsection expresses this transformation in the structural view in terms of uml:Class and uml:Operation[15]. The details of this transformation are written with the programming terminology which is closed to Java-like coding convention in Appendix A. For an Operation that represents a decomposable uml:Activity, each elementary uml:Action is expressed as a statement that invokes the action Operation within this activity Operation. Due to the abstraction level of platform-independent models, some artifacts are rather denoted descriptively in the appendix, so that it can be adapted to different technical platforms for creating

[15] The prefix „uml" of these two metaclasses are omitted in the left text in this section for simplicity.

code skeletons for these artifacts. An implementation of the transformation to the platform-specific layer should replace the general forms of output elements in the appendix to comparable ones in the target platform. It should also adapt the elements' names to the naming convention of the platform. The overall steps of **Description2Computation** for an EnvironmentalPhenomenon Ep are summarized in Table 5.11 and explained in the following paragraphs.

1	Create a *Class ComputeEp*;
2	Create *Attributes* in *ComputeEp* for each Ep's:
3	1) ConfigurableParameter,
4	2) indexName of each CharacteristicVariation;
5	Create an *Operation* in *ComputeEp* for each of following description items within Ep:
6	1) CharacteristicVariation,
7	2) individual Variation,
8	3) AlternativeMode;
9	Create other support structures;
10	Generate a dependency graph G for computation of Ep's properties;
11	Traverse G starting with node t to get a topologic sequence of nodes;
12	Add *Object epData* of Ep's generated datatype;
13	Create *Operation computeEp()* in *ComputeEp*;
14	Create statements in *computeEp()* to invoke the previously generated Operations to initialize and
15	update the properties of *epData* (or a unit in the *epData* if Ep is a FieldOfIndividualities),
16	following the sequence of nodes from Line 11;
17	Create necessary iterations when Ep is a FieldOfIndividualities;

Table 5.11: Overall Steps of Description2Computation.

First, this transformation generates a Class for each EnvironmentalPhenomenon to hold the computation behaviors. Each CharacteristicVariation or Variation within the scope of this EnvironmentalPhenomenon is transformed into an Operation (i.e., an uml:Action in the behavioral view) in the computation Class. ConfigurableParameter-s within the scope of the EnvironmentalPhenomenon and in some cases, also parameters of Operations that compute characteristic variations, are transformed into attributes in the computation Class. This step also adds member instances of the configuration and data structure Classes from the previous two transformations to the computation Class to hold information for executions. This step is presented in detail in Appendix A.3.

Next, a directed graph for each EnvironmentalPhenomenon is derived from its Variation-s. Nodes in this graph represent the described properties of this phenomenon. Edges in this graph correspond to the relations between their connected nodes that are expressed by a Variation. The graph reflects the computation dependency among these properties, i.e., an edge a → b denotes that the state value of a is required for determining the state value of b. It is an intermediate artifact to derive the computation order of the computation units generated by the previous step, which is used to form activity flows for updating states of this phenomenon.

The previous step of the graph construction allows cycles in this graph. If representing relations in such a cycle with equations, this loop corresponds to an equation system that determines the values of the nodes in the loop. However, the solving order of the computation units cannot be derived when loops exist. Thus, the construction is followed by a step that detects the cycles in the graph and replaces each cycle with a compound node. The primary steps of the graph generation for a SpatialIndividuality are shown in Table 5.1, and the detailed logic of it can be found in Appendix A.4. It applies to every SpatialIndividuality-s in the input SEDL description. Operations on graphs can be implemented using standard graph data structure and algorithms.

1	Create a directed graph G;
2	Add nodes t, l, g to G; //represents time, location, geometry
3	**For each** ThematicProperty p of SI
4	Add a node p to G;
5	**End for**;

6	**For each** Variation: a → b within the scope of SI
7	Add an edge a → b to G if it does not exist;
8	Store a reference to the generated *Operation* from this Variation with this edge;
9	**End for**
10	Search for cycles in G;
11	**For each** found cycle cyc
12	Add a node cyc to G to replace the subgraph of the cycle;
13	Add *Operation cyc()* to *ComputeSI*;
14	Store a reference to the subgraph of the cycle with cyc;
15	**End for;**

Table 5.12: Generation of the Dependency Graph for a SpatialIndividuality.

The dependency graph for an EnvironmentalPhenomenon implies the appropriate computation order of properties during a simulation step. The next step generates an Operation to compute states of a simulated feature object at a step. This Operation invokes the generated Operations for computation units in sequence to update the phenomenon data object hold by the computation class. The timestamp is updated at first as the time is the only fully independent variable in a simulation. Then, the properties that only depend on time are computed by the Operation referenced to the incoming edges of the node representing these properties. The computed values are then fed to the computation units represented by its outgoing edge. This process continues until all properties are updated. It applies to all SimulatedIndividuality-s in the input SEDL description. For a SpatialIndividuality that describes members of a FieldOfIndividualities, the state computation function updates a unit of the transformed CollectiveFeatureType instance. The transformation additionally generates iteration over units of this instance. Recommended details of this step are presented in Listing A.5.

For a clear specification, the transformation process is separated into steps in different listings based on logically different tasks. An implementation does not have to follow the sequence strictly. For instance, the generation of computation units, corresponding nodes and edges in the dependency graph are often performed spontaneously when parsing through the input. The same applies to the pseudo-code of each step. Outputs from this transformation are limited by the SEDL expressiveness and the information that can be determined at the analysis phase. These limitations are discussed in Section 8.2.

5.3.5 Map Description of Spatial Heterogeneity to Design Models

SEDL provides two ways to indicate that the spatial heterogeneity of an environmental phenomenon needs to be computed, i.e., through the ThematicValueDistribution of a ThematicProperty or spatially distributed members of a FieldOfIndividualities. They indicate the spatial heterogeneity perceived from different angles. However, different angles of perception do not necessarily result in different modeling decisions at the design phase by component developers. This issue brings complexity and limitations to the transformation automation, which requires developers' involvement. This subsection introduces the mapping principles to transform SEDL pieces relevant to the spatial heterogeneity to component design models and the necessary manual interference during this process. Transformation details are presented in listings in Appendix A.

The term FieldOfIndividualities implies a discrete view. A FieldOfIndividualities is mapped to a CollectiveFeatureType in the design-level structural model by **Description2Structure**. For the behavioral model generated by **Description2Computation**, change descriptions associated with its member individuality are transformed into behaviors that update states of a unit in the transformed CollectiveFeatureType's instances. The transformation also generates an activity that iteratively executes the update function to update all units.

The term ThematicValueDistribution, however, does not imply any discretization. A ThematicValueDistribution reflects the spatial derivative of a ThematicProperty, which treats the relevant property as a spatially continuous entirety. This term has duality in nature. On one side, it represents some function that determines a value at a given spatial location about this property. On the other side, in a temporal process, the form of the distribution at a time instant plays the role of the state value of the

property. Other changes which involve this property influence the distribution form. Thus, without manual interference or pre-restricted logical structure, a ThematicValueDistribution in an SEDL description should be mapped to a SpatialFunction that holds the distribution function with the spatial point as the parameter. The attribute type transformed from its associated ThematicProperty is set to be this SpatialFunction. An instance of this SpatialFunction represents the value of an attribute of some simulated feature instance. It is updated by updating its attributes that control its embedded distribution function. Table 5.13 shows the logic to generate a SpatialFunction Class P_Dist from the ThematicValueDistribution-s of a ThematicProperty P.

1	Create *Class P_Dist* applying the *SpatialFunction* stereotype;
2	**For each** ThematicValueDistribution Dist of P
3	Add *Operation dist()* to *P_Dis*t;
4	**For each** of its ConfigurableParameter CP
5	Add *Attribute cp* with the declared type to *P_Dist*;
6	**End for;**
7	**If** Dist has options **Then**
8	**Add** *Attribute dist_op* to *P_Dist*; // to mark the active option
9	**For each** option Option of Dist
10	Add *Operation option()* to *P_Dist*;
11	**For each** ConfigurableParameter CP_O of Option
13	Add *Attribute cp_o* with the declared type to *P_Dist*;
14	**End for;**
15	**End for;**
16	Add a conditional brunch to *dist()* which:
17	1) checks the value of *dist_op*,
18	2) invokes the generated *Operation* from the corresponding option denoted by the
19	value of *dist_op* as the behavior of *dist();*
21	**End if;**
22	Add a *Constructor* to *P_Dist* that can initialize its instance with a configuration object;
23	**End for;**

Table 5.13: Generation of a SpatialFunction.

5.3.6 Generate Execution Routine

The behavioral model of a simulated environment component considers two aspects, i.e., the behaviors that produce the context of environmental phenomena, and the behaviors of the component in a bigger simulation. The focus of this thesis stays on the first consideration. Various supports are provided to express the expected context of simulated environments and to create models of computation behaviors.

Nevertheless, SEDL also provides ExecutionRoutine to document expected behaviors of the second aspect for completion. Application artifacts that are specific to a system of interest component and the environment component described by a SimulatedEnvironment shall be derived from an instance of ExecutionRoutine owned by this SimulatedEnvironment. This subsection recommends the derivable component structures from an instance of ExecutionRoutine and its belonged SimulatedEnvironment. Their forms could be quite different in different technical paradigms and thus are specified descriptively.

For a SimulatedEnvironment instance, a Class applying EnvironmentSimulation (See Subsubsection 5.2.3.2) is supposed to be created for each of its ExecutionRoutine with a name resembling the input SimulatedEnvironment. If no ExecutionRoutine is presented, one default EnvironmentSimulation is created for the SimulatedEnvironment. Then, the following structures for this Class are created.

First, this Class maintains several collections of computation instances, each collection for one phenomenon type. These collections represent existing phenomena in the environment simulated by this EnvironmentSimulation. Thus, developers can implement a simulation process with phenomenon instances being added or removed during the process. For each EnvironmentPhenomenon in the input SimulatedEnvironment, a list is added to this Class. The type of the instances held by this collection is

set to the computation Class generated from the input EnvironmentPhenomenon by transformation steps in Appendix A.3.

Then, a constructor Operation is added to this Class. This constructor has a parameter of the ConfigSchema type generated from the input SimulatedEnvironment by SEDL2Config in Appendix A.1. This constructor should initialize instance of this class in the following way: 1) for each configured phenomenon in the schema, it creates and initializes a suitable computation instance with the configured values; 2) after that, it adds this computation instance to the list that maintains the corresponding computation type.

Following structures of this EnvironmentSimulation Class are further generated, when it is not a default one but is transformed from an ExecutionRoutine.

Snapshot types may be generated and linked to this Class via a SnapshotOf association. The recommended subtypes are: 1) a PointSnapshot type when *outputRange* is *atPoint*, or 2) a PolygonalAggregatedSnapshot when *outputRange* is *atRegion* and *valueAggregation* is *true*. This Snapshot type has the implicit structure as specified in Subsubsection 5.2.2.8. More complicated snapshot structures are not defined at the stereotype level in the current framework. Consequently, this EnvironmentSimulation should also contain the behavior to make a snapshot of the current state of the computed simulated environment for feeding a system of interest component. One of the following Operations needs to be implemented in this Class.

1. *snapshot():* it fetches state values of all existing phenomena maintained by the EnvironmentSimulation instance, puts them in a message and returns it.
2. *snapshot(Point p):* it retrievals current state values of all existing phenomena maintained by the EnvironmentSimulation instance at *p*, puts them in an instance of the PointSnapshot datatype derived from the running simulation and returns this instance.
3. *snapshot (Polygon pol):* it retrievals state values of all existing phenomena in the EnvironmentSimulation instance within the area of *pol*, puts them in a message and returns it.
4. *aggregatedSnapshot(Polygon pol):* retrieves a subset of state values of all existing phenomenon maintained by the EnvironmentSimulation instance within the area of *pol*, replaces values of multi-valued properties or of SpatialFuntion properties to aggregated values, puts them in an instance of the PolygonalAggregatedSnapshot datatype derived from the running simulation and returns this instance.
5. *aggregatedSnapshot():* fetches current state values of all existing phenomena maintained by the EnvironmentSimulation instance, replaces values of multi-valued properties or SpatialFunction properties to aggregated values, puts them in a message and returns it.

Besides, an Operation *update(t)* should be added to this Class to allow developers to implement the behaviors that update all computation instances of existing phenomenon computation instances maintained by this EnvironmentSimulation to the state at time *t*.

After that, when the *executionMode* of the ExecutionRoutine is *Autonomous*, an Operation that contains a loop is added to implement continuous communication behaviors with the system of interest component. The loop body executes *update(t)* to compute the state values of a simulated environment at t and a suitable snapshot Operation to make a snapshot, as well as sends the snapshot to the system of interest component in the way depending on the technical platform. Parameter sets shall be added to this Operation to control the incremental *t* by predetermining a combination when implementing this transformation. Recommended parameter combinations are as follows: 1) start/end time and temporal length of a step, or, 2) start time, temporal length of a step, and the number of loops.

Otherwise, when the *executionMode* is *Reactive*, a trigger handling Operation should be added. This Operation is triggered by receiving a message from the system of interest component, which includes: 1) a value of the *TemporalPosition* type; and 2) a value of the *point* type if *outputRange* of the ExecutionRoutine is *AtPoint* or a value of the *polygon* type if *outputRange* is *AtRegion*. This operation should execute a suitable snapshot operation and send the snapshot data to the system of interest component when triggered. In this mode, the implementation of the execution pace of *update(t)* during a simulation is left to developers.

Additional information about discretization may also be derived from an ExecutionRoutine. When *outputRange* of an ExecutionRoutine is *region* or *all* and valueAggregation is *false*, an additional Class for each generated SimulatedFeatureType with attributes of some SpatialFunction type are suggested to be added to the design model. This Class is stereotyped with a default subtype of CollectiveFeatureType. This step also generates a Class applying CollectiveFeatureUnit for this additional Class. For each attribute of the SimulatedFeatureType with a SpatialFunction type, it adds a corresponding attribute of a default type to the unit type.

6 Prototypical Implementation

This chapter presents a Java-based prototypical realization of the proposed framework in this thesis as specified in Chapter 4~5. It illustrates how the framework shall be implemented based on available tools. The following sections are based on involved tools, ordered by the sequence when they are used during the implementation, with explanations of which framework components are implemented by them and how to do the implementation.

6.1 Eclipse Modeling Framework

In this prototype, the SEDL Abstract Syntax Model is realized as an Ecore model using Eclipse Modeling Framework (EMF)[35]. The PIM-layer structural profile is implemented using the EMF-based UML2 plugin[16] since the Ecore itself does not support the profile notion.

The EMF is a modeling framework based on Eclipse[17]. It supports encoding model specifications in XMI. The logical metamodel used by EMF to describe models is called Ecore. It is a simplified and de-facto reference implementation of the EMOF (Essential Meta-Object Facility)[9] by OMG, which is grounded on UML. The names of the elements in Ecore start with E, such as EClass that corresponds to Class in UML. Thus, the UML-based SEDL abstract syntax model can be easily adapted as an Ecore model. Further, the encoded Ecore models can be used with rich EMF facilities for further implementations. EMF provides a tree-based editor that allows modelers to write models and store them in XML files. Figure 6.1 shows the classes in the SEDL abstract syntax model encoded as EClasses using this editor.

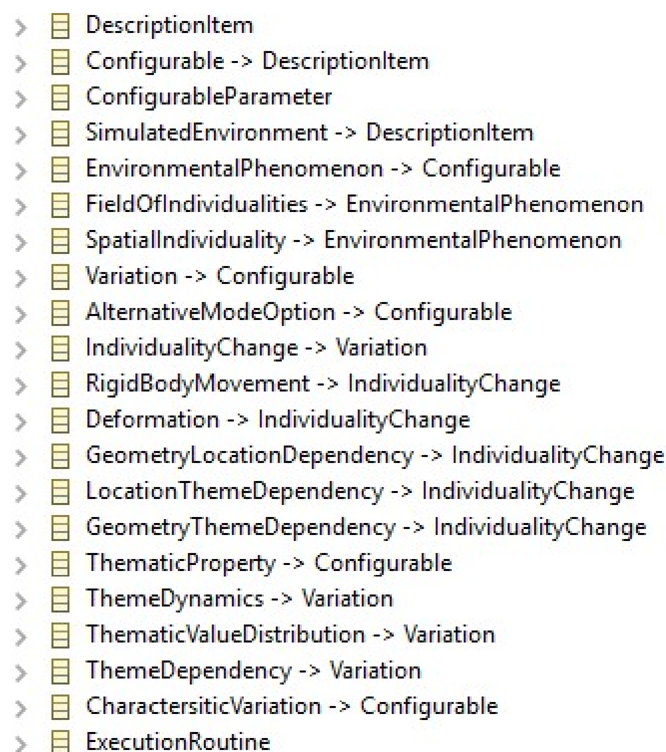

Figure 6.1: EClasses in the SEDL Ecore Model.

The class names in the abstract syntax model are rather used as identifiers for back-end processing. They do not necessarily be strictly identical to the keywords in a concrete syntax of SEDL that will be

[16] http://www.eclipse.org/modeling/mdt/uml2
[17] https://www.eclipse.org

used to write an SEDL program in that syntax. The implementation of the textual concrete syntax in the prototype is introduced in the next section.

The EMF-based UML2 plugin provides a visual editor that supports creating profiles, as well as adding stereotypes and other profile-related UML elements to profiles. After a profile is created, an Ecore-conformed XMI encoding for this profile is generated by this plugin. Through this step, an Ecore model is created for this profile, in which an EClass is created for each stereotype. Such an EClass has an EAnnocation denoting its corresponding stereotype and an ERference to its extended UML metaclass in an Ecore version of the UML metamodel. Figure 6.2 gives an illustration of the stereotype SpatialFunction definition and the created EClass.

6.2 EMFText

After the Ecore version of the SEDL Abstract Syntax Model has been written, it is used with EMFText[18] to create the infrastructure of the Basic SEDL Tooling in the framework.

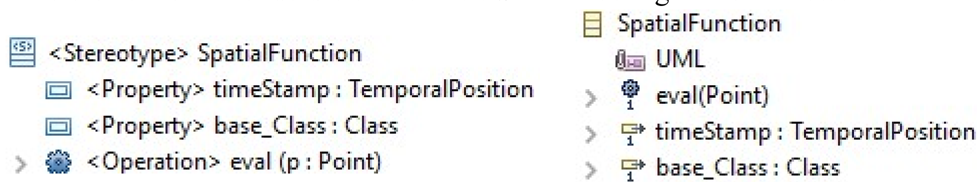

Figure 6.2: SpatialFunction(Left) and Created EClass (Right) in the UML Editor.

EMFText is an EMF-based language workbench implemented as an Eclipse plugin. EMFText provides an editor to help language developers write textural syntaxes for Ecore models. Using this editor, a textual SEDL Concrete Syntax is specified in a file with the “cs” extension, which has references to the SEDL Ecore model. For each term in the Ecore model, a syntactic rule is specified, which regulates a grammar in Extended Backus–Naur Form[135] with some keywords. When a phase in an SEDL textual description matches this rule, it should be recognized as an instance of this term. Rules and keywords in this prototype are specified closer to natural language expressions so that descriptions in this concrete syntax can serve as human-readable requirements documentation. The concrete syntax for a term can be modified and adapted in different implementations.

EMFText also has functionalities that can generate a language infrastructure from a concrete syntax file referenced to an Ecore model. The “cs” file of SEDL is fed to EMFText to create an infrastructure of the SEDL tooling. This infrastructure is created in the form of Java programs. It supports writing and processing SEDL descriptions in the specified concrete syntax, as explained below. Model code from the Ecore SEDL model is also generated to be used by the SEDL tooling.

First, EMFText generates an SEDL Description Editor for the specified concrete syntax. It supports writing an SEDL description and store it with a file extension as specified in the concrete syntax. In the prototype, the extension of an SEDL textual description file is “sedl”. This editor can recognize files with this extension. When such a file is open in this editor, keywords referenced to underlying Ecore abstract syntax are highlighted by the editor. It also denotes errors in an opened file when somewhere in the description does not conform to the specified concrete syntax. These functionalities are enabled by the underlying Syntax Analysis Component. Figure 6.3 illustrates the prototypical textural editor. The specified concrete syntax in the prototype focuses on describing two-dimensional spatial simulations at the geo scale. The keyword “Spatial Individuality” in the concrete syntax corresponds to the term SpatialIndividuality in the abstract syntax, and the “type” corresponds to its *dimNum* with the following supported values: -1(“Global”), 1 (“Point”), 2(“Regional”).

EMFText also generates various facilities for syntax analysis. On one side, they are connected to the SEDL editor and enable the above-mentioned supportive functions of the editor. On the other side, it

[18] https://github.com/DevBoost/EMFText

parses valid SEDL descriptions written in the specified concrete syntax to abstract syntax trees as instances of the SEDL Ecore model for further processing.

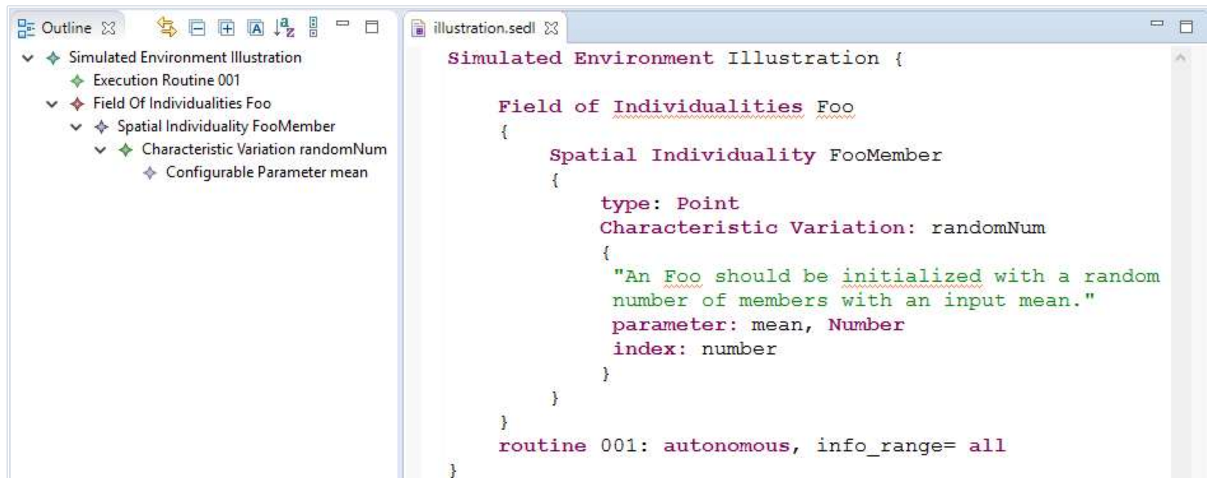

Figure 6.3: Textual Editor for SEDL Description.

In addition, EMFText generates a frame for the SEDL Description Processor. The implementation of the processor functions can be held by the frame as will be introduced in the following sections. The frame is integrated with the SEDL Description Editor so that the processor can be invoked through the editor to execute an SEDL description file through some user actions such as right-click on the description file and choose to run the file.

6.3 OCLInEcore

By default, syntax analysis facilities generated by EMFText can recognize errors in an SEDL description that breaks structural constraints encoded in the SEDL Ecore model, such as conflicts with the cardinality. Additional structural constraints of SEDL specified in Chapter 5 are beyond the expressiveness of UML, as well as the Ecore metamodel. These constraints are formalized in OCL in the SEDL specification.

The prototype uses the Eclipse implementation of OCL[19] to integrate the specified OCL constraints into the SEDL Ecore model. This implementation provides the OCLInEcore[20] editor to open Ecore models in a textual form and to write constraints in OCL syntax into the Ecore model. After the OCL constraints are added to the SEDL Ecore model, the EMF code generation facilities generates a validator class when creating the model code for this Ecore model. This class is placed in the “util” code package of this model. For each OCL constraint in this model, a validate method is generated within the validator class. It checks if some context in an SEDL model instance breaks and returns true or false. Figure 6.4 shows an example of the encoded OCL constraints. This example regulates that when a piece of change description (represented by the abstract term “Variation”) can either have parameters or options. And, when it has alternative mode options for a specific execution, it must have at least two.

```
abstract class Variation extends Configurable
{
    property option : AlternativeMode[*] { ordered composes };
    invariant numOfOption:
        self->collect(option)->size()>=2 or self->collect(option)->size()=0;
    invariant parameterConstraints:
        self->collect(option)->size()>0 implies self->collect(parameter)->size()=0;
}
```

Figure 6.4: OCL Constraints for Change Description.

[19] http://www.eclipse.org/modeling/mdt/ocl

[20] https://wiki.eclipse.org/OCL/OCLinEcore

The generated Java method to validate the constraint about the number of options is shown in Figure 6.5. It utilizes a basic validation method on Ecore model instances. Then, the default editor generated by EMFText is improved with the assists of the validator class. The functional code of the editor mainly locates in the "ui" package within the EMFText-created resource code of the SEDL tooling. The presented editor invokes the validate methods from its validation function and its hover text provider. The validation results guide the editor to display different customized hover text to SEDL users, warning them when some of their editing contexts break a specific OCL constraint.

```
protected static final String VARIATION__NUM_OF_OPTION__EEXPRESSION =
        "self->collect(option)->size()>=2 or self->collect(option)->size()=0";

public boolean validateVariation_numOfOption(Variation variation,
        DiagnosticChain diagnostics, Map<Object, Object> context) {
    return
        validate
            (SedlPackage.Literals.VARIATION,
             variation,
             diagnostics,
             context,
             "http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
             "numOfOption",
             VARIATION__NUM_OF_OPTION__EEXPRESSION,
             Diagnostic.ERROR,
             DIAGNOSTIC_SOURCE,
             0);
}
```

Figure 6.5: Validation Method Example.

Figure 6.6 illustrates the warning message from the SEDL editor when the context does not satisfy the constraint shown in Figure 6.5.

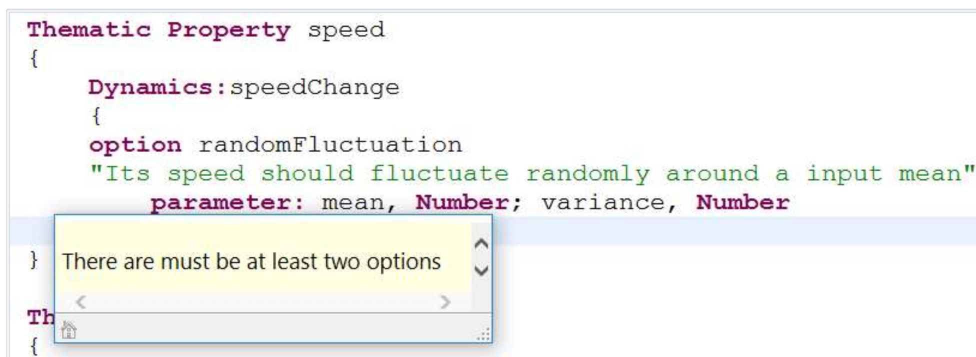

Figure 6.6: Warning Message for OCL Constraint Violation.

Another worth-to-mention implementation strategy for dealing with these constraints is to introduce additional support classes into the Ecore model to explicitly regulates alternative forms of a term, e.g., a class for change descriptions with no options but parameters, and a class for the ones with options. These support classes only benefit at the syntactic level but do not brings new concepts to SEDL conceptual model and are not specified as one part of SEDL specification. Through this strategy, the constraints can be expressed by the Ecore model itself. However, it complicates the implementation of model transformations from SEDL instances since the abstract syntax trees of these instances become more complicated due to the inclusion of additional terms.

6.4 ATL

The transformations Description2Config and Description2Structure are implemented as ATL

programs using the ATL component of the MMT project[21] based on Eclipse. Transformations from SEDL generate more detailed model elements in the outputs that cannot be formalized by simple matched rules. These elements are created with assists of imperative statements in the "do" section of ATL rules.

The presented prototype does not focus on graphic interface mapping. As explained in Subsection 5.3.2, Description2Config in the prototype is simplified. The stereotyped elements are replaced with instances of corresponding base metaclasses to create the data structure of configuration objects. In the EMF-based implementation, the Ecore metamodel is used as the output metamodel. An EPackage is created as a ConfigSchema. Each stereotyped class in the output is replaced with a normal EClass, while its ConfigItems are created as its EAttributes. Each SubComponent association is generated as a containment EReference of the container configuration EClass, typed by its *sub* configuration EClass. For ConfigOption associations, an abstract EClass is created as an AlternativeConfig. Then, each of its ConfigOption is created as a non-abstract subclass of this EClass. An instance of the configure schema will have to pick one of these subclasses to set values.

The Description2Structure implementation transforms an SEDL instance to a UML model that applies the implemented Simulated Environment Structure Profile as introduced in Section 6.1. Outputs of this transformation are set to conform to the UML metamodel implementation within EMF. The profile implementation is marked as a metamodel of the input in the ATL program so that its stereotypes can be retrieved and applied to the output elements using the profile-related library provided by EMF. Figure 6.7 shows an ATL rule example that creates a SpatialFunction class to hold the distribution function for a spatially heterogeneous property. It is a lazy rule which is only executed when being called by other rules. This rule also calls another rule "Distribution2Operation".

```
lazy rule Distributions2Class{
    from
        th:SEDL!ThematicProperty,
        pa:UML!Package
    to
        c:UML!Class(
            name <- th.refImmediateComposite().name.startWithUpper()+th.name.startWithUpper()+'_Dist',
            package <- pa
        )
    do {
        c.applyStereotype(thisModule.getStereotype('SpatialFunction'));
        -- Add a Operation to the created class for each Distribution of this ThematicProperty
        for(dis in th.spatial){
          c.ownedOperation <- c.ownedOperation.including(thisModule.Distribution2Operation(dis,c));
        }
    }
}
```

Figure 6.7: ATL Rule Example.

As Section 5.3 specifies, some transformation may lead to a TesserlatedFeatureType in the output, which needs to be refined by developers since it is a choice to make at the design level. While the "TesserlatedFeatureType" is conceptually abstract by specification, it cannot be instantiated if being implemented as an abstract class. Thus, in the implemented UML Profile introduced Section 6.1, it is made concrete and marked with a denotation to remind developers that its instances in transformation output needs to be refined.

After the transformation rules have been written in the ATL program, an Eclipse plugin can be created from it. Then ATL program can be invoked through the SEDL processor as introduced in Section 6.2. By starting the execution of an SEDL textual file, the processor feeds a parsed SEDL instance from this textual file to the transformation to create PIM models.

6.5 Acceleo

The transformation Description2Computation is implemented by the template-based code generation tool Acceleo[22]. Acceleo is an EMF-based implementation of Model to Text Language (MTL)[136]

[21] https://www.eclipse.org/mmt
[22] https://www.eclipse.org/acceleo/

standard from OMG. It can transform EMF-based models to text according to generation templates, which usually is textual code in some computer language in practice.

This implementation also includes the functionalities of a Platform-Specific Translator since it generates platform-specific code. The implemented Description2Computation takes SEDL descriptions as input and produces as Java files as outputs. PIM-layer outputs are made implicit. Acceleo is also used for implementing the Platform-Specific Translator from refined PIM-layer outputs of the Description2Structute to Java files.

Outputs of these Acceleo transformations include architectural code and datatypes that are specific to the chosen target platforms. For this proof-of-concept implementation, the Java-based multi-agent simulation library Mason[23] and its extension for geospatial data GeoMason[24] are chosen as parts of the target platform.

Take an instance of FieldOfIndividualities whose PIM-layer data structure should be mapped to a PointSet class at the PIM layer as an example. Its unit type is generated as a JavaBean with all thematic properties as private attributes with setter and getter methods. The geometry of the unit type is generated as a MasonGeometry provided by Mason.

The ComputeFoI class transformed from this FieldOfIndividualities, as documented in Appendix A, is adapted to the chosen target platform. The computation class of its unit is mapped as a class that implements the Steppable of Mason. This class holds an object of the unit datatype and private Java methods that are generated computation units about individuality changes. The methods are marked as "//TODO should be implemented". The free-text description of each change in the input SEDL is transformed as Java comments and is placed above the body of the corresponding computation method to guide the implementation. The computeM() operation in Appendix A.3 is generated as the step() method of the Steppable class. It contains statements to update the unit object held by an instance of this class at a simulation step in the sequence derived based on Appendix A.4~5.

This demonstrative implementation assumes the target platform decides that different simulated feature types are managed by a Java class that extends the SimState of Mason. This SimState class corresponds to the EnvironmentSimulation class at the PIM layer as specified in Subsubsection 5.2.3.1 and Subsection 5.3.6. The current implementation does not generate a separate data structure class for the whole PointSet, which could be better for a more loosely coupled implementation in practice. A GeomVectorField object from GeoMason is generated and is placed in this SimState class to hold all unit geometries. The GeomVectorField from Mason provides neighborhood search utilities as the specification of a PointSet type requires.

The SimState class also holds Java methods generated from CharacteristicVariation-s of the input FieldOfIndividualities and its units, as regulated by Appendix A in the ComputeFoI class. A "for" loop skeleton is generated within the start() method of the SimState and marked as "//TODO should be completed". This loop is used to initialize a set of instances of the unit's Steppable class and add their geometries to the GeomVectorField object. Derivable actions that are relevant to the initialization of units are generated as statements in this loop. Iterations over units in this implementation are generated by adding the unit Steppable instances as repeating events to an instance of Schedule provided by Mason.

Acceleo code generations are template-based. Each code template is enclosed within a pair of the "[template][/template]" markup. Figure 6.8 shows a snippet from the SEDL2Computation implementation for illustration. This piece of template creates Java code from Variation-s of a FieldOfIndividualities with some AlternativeMode-s. It generates an attribute for each Variation in the computation class, which is used as a flag to denote the current mode of the computed change corresponding to that Variation during execution. It also generates attributes to hold configurable conditions during execution from ConfigurableParameter-s of this FieldOfIndividualities, since this implementation has decided that these conditions are allowed to be changed during execution. The

[23] https://cs.gmu.edu/~eclab/projects/mason/

[24] https://cs.gmu.edu/~eclab/projects/mason/extensions/geomason/

createAtt() in the snippet is an implemented Acceleo Query. The “p.createAtt()” returns a string that is a Java private attribute generated from the ConfigurableParameter p.

```
    [for (iv: Variation | f.eAllContents(Variation)) separator('\n')]
        [if (iv.option ->notEmpty())]
private String [iv.name.toLowerFirst()/]_option;
            [for (o: AlternativeMode | iv.option)]
                [for (p:ConfigurableParameter| o.parameter)]
[p.createAtt()/];
                [/for]
            [/for]
        [/if]
    [/for]
```

Figure 6.8: Acceleo Code Generation Example.

6.6 Other Involved Tools

The imperative functions which deal with graph and other operations on the input SEDL for SEDL2Computation are implemented by Java. They are called by the generation templates as services that are wrapped in an Acceleo query. For the simplicity of the implementation structure, this demonstration builds strings for computeSI() and computeM() in Appendix A.5 within the Java service and passes them to the Acceleo templates after that. Yet, it is beyond the scope of this thesis to evaluate if this is the optimal way of implementation on the target platform. For a full implementation, the template of architectural code should be determined by the technician who is familiar with the chosen platform to ensure that the created code and datatypes provide expected structures and functions.

It is also recommended to optimize the programming interface to access the to-be-implemented computation units. Thus, developers can focus on implementing the computational logic within the units. This issue depends on the implementation technologies and is beyond the scientific concern of this thesis.

The configuration models from implemented Description2Config are simple, static data models in Ecore, which do not include behavioral elements. The platform-specific translator of the demonstrative prototype uses the EMF code generation facility to create model code for these models as JavaBeans, which are needed by the other generated model code as described in Section 6.5. Different from the OMG standard-based code generation implemented by this prototype, the EMF default code generation is based on JET (Java Emitter Templates)[25].

[25] https://www.eclipse.org/modeling/m2t/?project=jet

7 Use Cases

In Chapter 4 and 5, a domain-specific, language-driven framework to assist the development of environment components in simulation applications have been specified. This chapter presents use cases to demonstrate the usage of this framework based on the prototype implementation of the framework introduced in Chapter 6.

7.1 Focus of the Use Cases

The use cases aim at demonstrating the following functionalities of the proposed framework in the development process to build a component that provides the simulated environment for a simulation application:

- Document and communicate the simulated environments required by the high-level functional simulation scenarios using the analysis-level language SEDL.
- Generate design models of simulated environment components described by PIM-layer metamodels from an SEDL description via the implemented CIM-PIM transformations.
- Generate PSM-layer models as code skeletons of the simulated environment component from PIM design models via the implemented platform-specific mapping.

To remain the focus, the simulated environment in each case is restricted to a small number of phenomena. Requirements and computations of the environmental phenomena are simplified to reach presentable cases within the length of the thesis. The aim is to cover different aspects that can be expressed by SEDL. Thus, the included phenomenon types, relationships and computational models may not be a complete and optimized practical solution. Nevertheless, since SEDL model is developed based on object-orientated notations and the generation leads to self-contained subcomponents for different phenomenon types, more types can be added in the same way. Besides, the PSM-layer code snippets in this chapter are illustrated by the Eclipse IDE. Providing an optimized user interface specifically for programming with these PSM-layer models is useful in practice but beyond the scope of this thesis.

The covered aspects in the simulated environment described by SEDL and produced artifacts by the framework for the use case are summarized at the end of each case.

7.2 Use Case 1: Sea Environment for the Path Assessment

The first use case is motivated by accidents that containers fell off from a cargo ship due to heavy weather. To better understanding these accidents and to avoid them in the future, researchers build a model of the cargo ship to observe and to analyze its behaviors through computer simulations. In the use case, this model is used to assess a newly-planned path of the ship between two harbors, which is theoretically more cost-effective than the current routine path. It passes an area which the current path does not intersect with, where no historical information is available and may be potentially risky for the cargo ship. Thus, before executing this plan in the real world, voyages using this path is simulated by computers with the ship model.

The functional scenario of this use case is that a cargo ship executes a planned path passing the area of interest under various weather conditions. The maneuvering and seakeeping behaviors of the ships are simulated with the data representing its situated environment fed to the ship model. Ship states during the simulated voyages about its stability are computed to analyses the probability of the ship to encounter an accident of container fall. The simulated environment component developed in this use case should simulate the necessary environmental data with user-desired conditions for this simulation. This includes the information informed by Vessel Traffic Service (VTS) or its own sensors (e.g., the current wind level), and influential forces from the surroundings (e.g., the force from the wave).

The extent of the simulated environment is restricted by the area of interest. The environmental data is provided at the geo scale that spatial locations are abstracted in two dimensions. The requirements[26]

[26] simplified for demonstration

about phenomena in the simulated environment from the ship modelers are summarized in Table 7.1. Time-invariant information during the simulation should be taken from the survey data and included as the knowledge of the ship model. e.g., the bathymetry data. They are not provided by the simulated environment component.

Wind	1. The ship model should be fed with the speed and direction of the wind in the simulated area at each step. 2. The wind direction should be modifiable by users among executions. It roughly stays the same with random turbulence over time. 3. The wind speed should change over time with one of the following patterns chosen by users: • random turbulence with a modifiable mean value and a modifiable variance; • linear change with a modifiable initial value and a changing rate.
Wave	1. The height of waves should be fed to a visualization engine to display the sea surface. The height values should be different from location to location. 2. The wave should be computed considering wind influence.
Background Traffic	1. A set of ships should be included in the area of interest to add some marine traffic influence. The number of ships should be modifiable by the user to observe the cargo ship's behaviors with various traffic density. 2. The cargo ship should be informed by the locations and moving direction of the ships at each step. The geometry of these ships can be neglected. 3. The background vessels move randomly. Each of them has a constant speed, which depends on the type of the ship. 4. The moving direction of a ship should turn a random angle within a range relative to the direction of the last step. The angle should be drawn from a normal distribution with modifiable mean and variance.

Table 7.1: Initial Requirements of Environmental Phenomena in Use Case 1.

7.2.1 SEDL Description

At the system analysis phase, the simulated environment in the functional scenario is documented in an SEDL description using the textual editor of the prototypical implementation introduced in Chapter 6. The wind and wave are both documented as a SpatialIndividuality, while the BackgroundTraffic is documented as a FieldOfIndividualities whose members are described by "Ship". Each of their thematic properties whose information is required by the system of interest component should be explicitly added as a ThematicProperty, so that the transformations can be informed to generate the necessary component structure for these properties. The screenshot of the full description can be found in Appendix B.1.

The current SEDL does not yet provide terms to explicitly classify changes involving derivatives, which is, the relationship between a characteristic C and a Variation: A → B. In computation, such a relationship constrains the computation order between C and B, since the value of C will be used as parameters of this Variation to compute B. Due to the expressiveness limitation, the C → (A → B) is documented in an instance of a Variation subtype that can classify the change pattern C → B.

In this use case, the moving direction of a vessel in the background traffic is required by the system of interest component and thus is described as a ThematicProperty. It changes the way of ship movement and thus has to be updated before the vessel location. A LocationThemeDependency "ChangingDirection" of the field member "Ship" is added to inform the automatic transformation about this constraint. This description item is added by component developers and marked with "Derived by developers". The effort of this item in later development phases is shown in the next subsection.

7.2.2 Transformed Artifacts

When the SEDL description satisfies the involved roles in the system analysis phase, it is fed to the implemented transformations to generate program skeletons of the simulated environment component for this use case. First, the SEDL2Configuration ATL transformation creates a configuration model of the component in Ecore. It is shown in Figure 7.1, which is open in the graphic view of EcoreTools[27] for illustration.

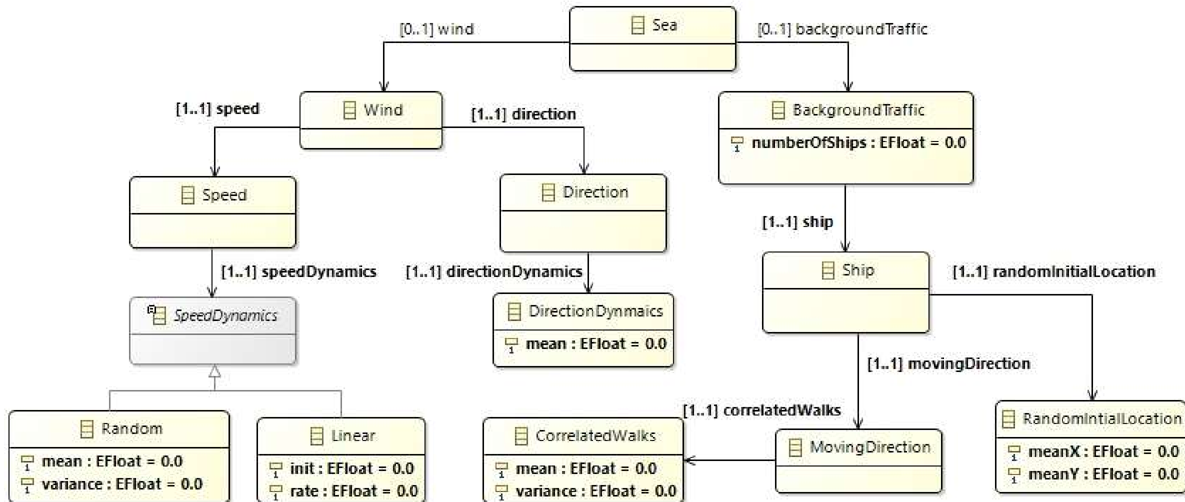

Figure 7.1: PIM-Layer Configuration Model in Use Case 1.

The PIM-layer output from SEDL2Structure is shown in Figure 7.2. It is a UML model encoded in XMI and is opened in the UML2 Plugin used in Chapter 6.

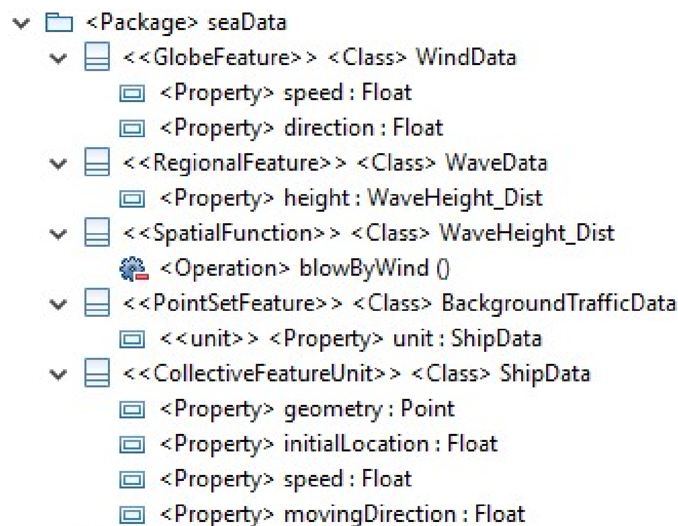

Figure 7.2: PIM-Layer Data Model in Use Case 1.

The above two ATL transformations are chained with code generation. The configuration model code files generated by EMF are placed in the "seaCon" package and the subpackages of this package as shown in the following figures[28]. The configuration for one execution can be parsed to instances of these classes and used by computations.

[27] https://www.eclipse.org/ecoretools/

[28] Classes in seaCon are not shown in the figure due to the limitation of space. They are the same as can be expected from the EMF default code generation using the input Ecore model.

JavaBeans are generated from the data structure model in Figure 7.2 by the Acceleo transformation implemented by the framework prototype. They are placed in the "seaData" package as shown in Figure 7.3. As Chapter 6 introduced, the prototype maps the PointSetFeature to a GeomVectorField managed by a Mason SimState class (i.e., the ComputeSea class in Figure 7.5). Thus, no Java class "BackgroundTrafficData" is created in the package.

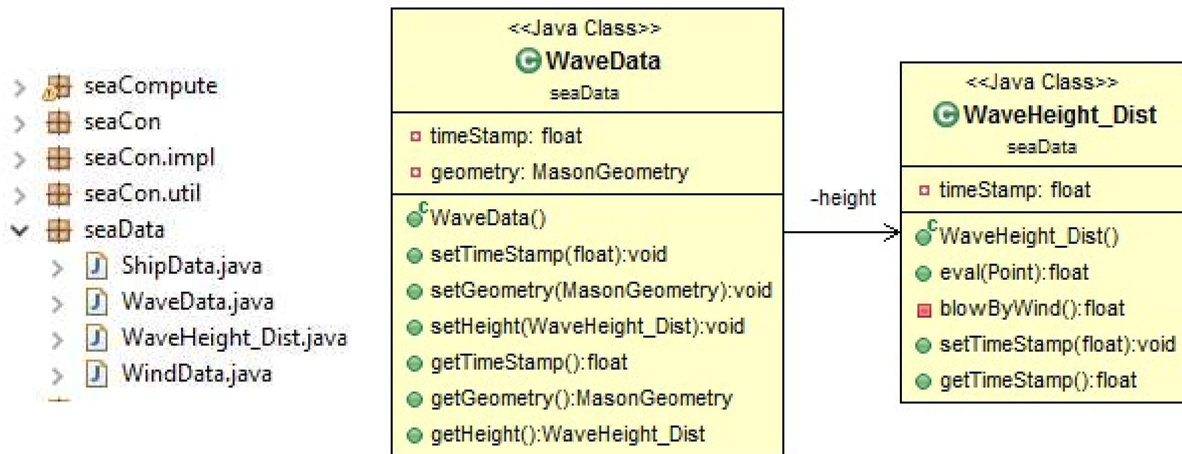

Figure 7.3: PSM-Layer Java Classes of Data Model in Use Case 1.

For illustration, the right side of Figure 7.3 shows a class diagram of the "WaveData" Java class generated by the code visualization tool ObjectAid[29]. Instances of the class hold the state values of the wave during simulations and provide access methods to them. The characteristic "height" is spatially heterogeneous, whose states are represented by the "WaveHeight_Dist" class. Its value at a point location is supposed to be accessed via the "eval(Point)" method of this class, which needs to be implemented. Its pattern should be implemented in the private method "blowByWind()". This method is similar to other

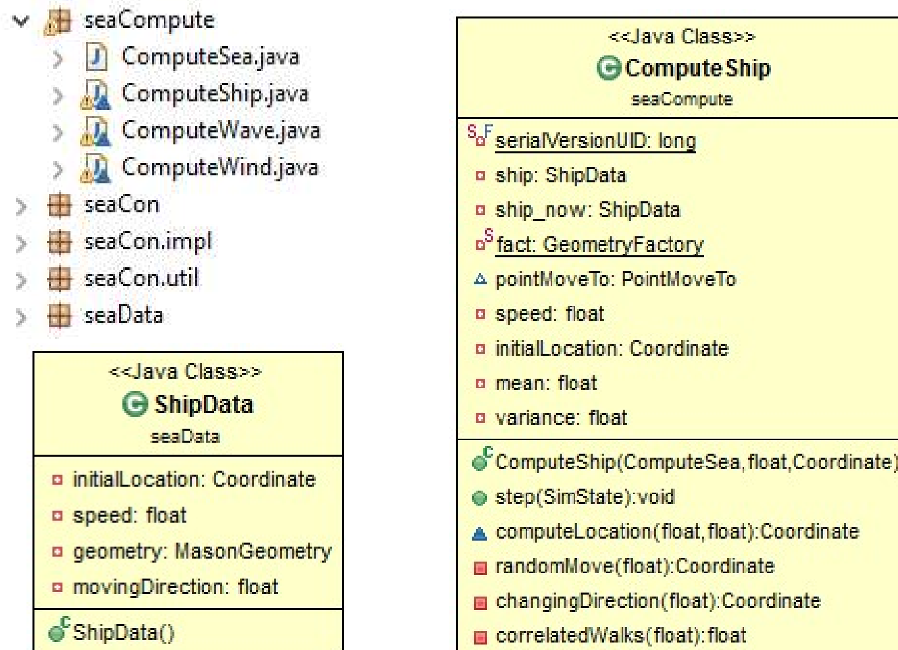

Figure 7.4: PSM-Layer Java Classes of Compute Model in Use Case 1.

[29] https://www.objectaid.com

private methods derived from Variation-s in the input description, as illustrated in the following paragraphs.

The implemented SEDL2Computation transformation directly creates Java code. They are the PSM-layer component skeleton. Outputs of this transformation are placed in the “seaCompute” package, as shown in the up-left part of Figure 7.4. In addition to the computation classes for all SpatialIndividuality-s, the “ComputeSea” (whose name is got from the input SimulatedEnvironment) class is generated to manage all simulated features and simulation routines. As the GeomVectorField generated from the BackgroundTraffic is held by the “ComputeSea” class, the computation code about the whole BackgroundTraffic (e.g., the iteration over all unit ships in it) are also placed in this class. This is an implementation decision made at the PSM layer.

The other parts of Figure 7.4 show the visualized class diagram of the generated “ComputeShip” class and the “ShipData” class used by it. The “ComputeShip” is a Mason Steppable that is used to implement Mason agent behaviors. It includes Mason-specific structures and utilities such as the “GeometryFactory fact” used to create the geometry of the ship, etc. This part of the PSM-layer outputs could vary in different technical platforms. The transformation also generates objects to hold states of the computed ship, i.e., “ship” and “ship_now”, as well as attributes to hold indexes of the ship, i.e., “speed” and “initialLocation”. All the objects and attributes are made private with access methods following the JavaBeans specification. The access methods are left out in the figure due to the limitation of space.

The following Java methods are generated in the “ComputeShip” class: the constructor “ComputerShip(ComputeSea, float, Coordinate)”; the “step(SimState)” method of Steppable to update the states of the “ship” object, which corresponds to the “computeShip()” at the PIM layer in Appendix A.5; private methods whose body should implement a computation unit, or implement the combined effort of relevant units to compute an attribute of the ship data object.

The constructor is shown in Figure 7.5. A “TODO” mark is generated to remind developers to complete the application-specific initialization. In SEDL, the meaning of an index in a CharacteristicVariation is captured by the free text. It may not be directly assigned as some initial state of a simulated feature. Thus, the automatic transformation does not generate code for such assignments.

```
31 //==============================Constructor==========================================
32     public ComputeShip(ComputeSea state, float speed, Coordinate initialLocation){
33         this.speed = speed;
34         this.initialLocation = initialLocation;
35 
36         // TODO initial states of data objects should be implemented
37         Coordinate location = null;
38         MasonGeometry geometry = new MasonGeometry(fact.createPoint(location));
39         geometry.isMovable = true;
40         this.ship.setGeometry(geometry);
41     }
```

Figure 7.5: Constructor of the “ComputeShip” Java Class.

Different from the other two types of phenomena in the simulated environment, the background ships are the members of the background traffic. As the transformation rules specified, iterations over these

```
55         for(int i=0; i< this.numberOfShips; i++){
56         // TODO data types of indexes should be refined
57             float speed = randomSpeed();
58             Coordinate initialLocation = randomInitialLocation(this.meanX,this.meanY);
59             ComputeShip cShip = new ComputeShip(this, speed, initialLocation);
60             ships.addGeometry(cShip.getShip().getGeometry());
61             schedule.scheduleRepeating(cShip);
62         }
```

Figure 7.6: Code for the Initialization of Ships in the Background Traffic.

ships for initialization and update are generated in component models and code. Such code snippets are placed in the "ComputeSea" class. Figure 7.6 illustrates the generated code snippet which initializes a set of ships, adds them to the GeomVectorField of background traffic (i.e., the "ships" object in the code), as well as adds them to the simulation schedule which is Mason-specific. The indexes of these ships are created using the methods generated from corresponding CharacteristicVariation-s of these indexes, i.e., the"randomSpeed()" and "randomInitialLocation(float, float)".

The three generated private methods of computation units are application-specific and need to be implemented. Some generated attributes are supposed to be used in the methods and remain unused in the outputs, which leads to the warning sign on the computation classes in Figure 7.4. Figure 7.7 shows the code skeletons of these methods. They are marked with the "TODO" mark. The free-text description of their corresponding Variation-s in the transformation input is generated as Java block comments to guide the implementation.

```
76 //================================Changes of individual units================================
77
78        /*The moving direction of a ship should turn a angle relative to the moving direction
79        of the last step. This angle should be chossen from a normal distribution.*/
80        private Coordinate randomMove(float time){
81             // TODO should be implemented
82
83            //Create an object to hold return value
84            Coordinate c = this.ship.getGeometry().geometry.getCoordinate();
85            return c;
86        }
87
88        /*Derived by developers - the ship moves towards the new moving direction*/
89        private Coordinate changingDirection (float movingDirection){
90             // TODO should be implemented
91            Coordinate c = this.ship.getGeometry().geometry.getCoordinate();
92            return c;
93        }
94
95        /*The moving direction of a ship should turn a angle relative to the moving direction
96        of the last step. This angle should be chossen from a normal distribution.*/
97        private float correlatedWalks (float time){
98             // TODO should be implemented
99
100           return 0;
101       }
```

Figure 7.7: Methods for Computation Units of a Ship in the Background Traffic.

The "computeLocation(float, float)" shown in Figure 7.8 needs to implement the function that combines the effects of all relevant computation units to compute a new state of the ship location. It is generated with the "TODO" comment and a default implementation. The default implementation simply executes the unit methods one by one, which results in an additive effect.

```
67 //================================ Update states of an property ==========================
68        private Coordinate computeLocation(float time, float movingDirection){
69            // TODO effect of units should be combined
70            Coordinate c = null;
71            c = changingDirection(movingDirection);
72            c = randomMove(time);
73            return c;
74        }
```

Figure 7.8: Method for Compute the Location of a Ship in the Background Traffic.

Finally, Figure 7.9 shows the "step (SimState)" method in the "ComputeShip" class. The computation sequence of this use case is relatively simple since only the "movingDirection" attribute and the

geometry[30] of the ship need to be updated. The update of the timestamp is managed by the “ComputeSea” class. As Appendix A.4~A.5 specify, it can be derived from the LocationThemeDependency “ChangingDirection” in the input SEDL description that the “movingDirection” attribute of the “ship” object should be updated at first. The prototype does not generate the “computeN()” method when only one computation unit is needed to update the property N but directly uses this unit as the update method. Thus, the “correlatedWalks(float[31])” is invoked in the code snippet.

```
51  @Override
52  public void step(SimState state) {
53      ComputeSea computeSea = (ComputeSea) state;
54          this.mean = computeSea.getMean();
55          this.variance = computeSea.getVariance();
56
57          //update movingDirection
58          float movingDirection = correlatedWalks(computeSea.getTimestamp());
59          this.ship.setMovingDirection(movingDirection);
60
61          //update geometry
62          pointMoveTo.setCoordinate(computeLocation(computeSea.getTimestamp(), this.ship.getMovingDirection()));
63          this.ship.getGeometry().geometry.apply(pointMoveTo);
64          this.ship.getGeometry().geometry.geometryChanged();
65  }
```

Figure 7.9: Code for Computing New States of a Ship in the Background Traffic.

7.2.3 Summary

This use case involves following kinds of phenomena in simulated environments which can be described in an SEDL description: the phenomenon that has a global effect (i.e., the wind) in the simulation area, the spatially heterogeneous phenomenon within its extent (i.e., the wave), and a set of spatial entities of the same kind with no significant members (i.e., the background traffic).

The system analysis phase is a communication process in which component developers also participate. They can add description items by identifying implicit requirements or information hidden in the free-text descriptions to reduce manual work in later phases, while automatic transformations derive models from the documented description items. It is recommended to mark these items with some denotations to avoid confusion. The use cases mark such items with “Derived by developers” in its free-

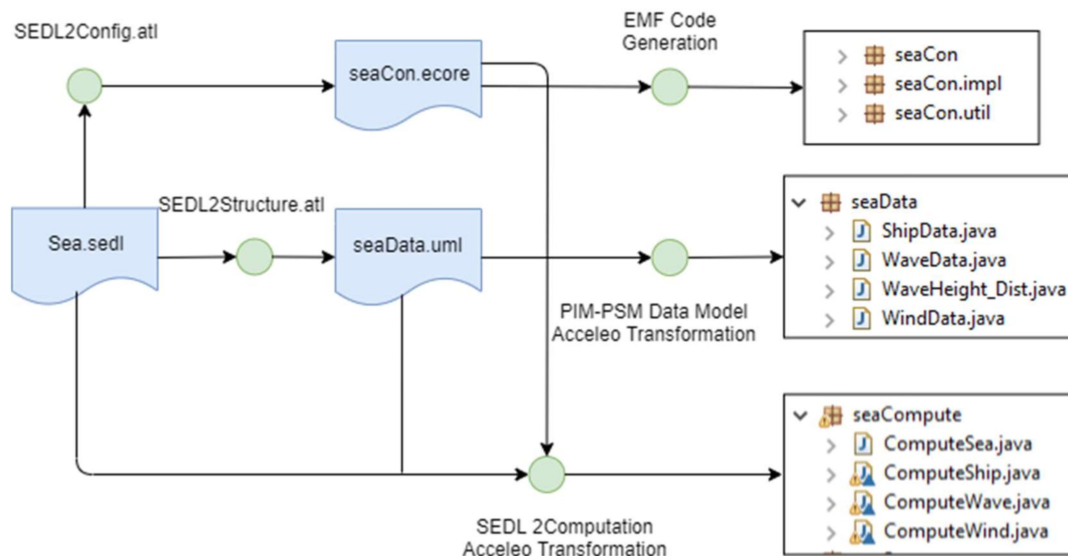

Figure 7.10: Transformations of Use Case 1.

[30] The coordinates of the geometry naturally hold its spatial location.

[31] The time in this prototype is represented by simple float numbers, which shall be replaced by more a sophisticated representation of time in practice.

text description, such as the “ChangingDirection”. More sophisticated supports shall be provided in practice, such as including keywords for the denotation in the SEDL implementation.

Figure 7.10 summarizes the transformation chain in this use case, as well as the generated artifacts during this process. The green round shapes represent the automatic transformation steps. Manual work is not shown in the figure except the input SEDL that needs to be written by involved roles. A step of the model/code generation costs the time at an order of magnitude between $1 \sim 1 \times 10^1$ seconds, which can be neglected compared to the total development time.

As explained in Subsection 7.2.1, the “ChangingDirection” in the SEDL description describes a relationship between a characteristic and a Variation. It is added to the description to deal with the current expressiveness limitation of SEDL. Even though this item only influences the update sequence derivation for the ship attributes, it brings side effects in the transformation output. As Figure 7.7 and Figure 7.8 show, a computation unit is generated from it and is used for updating the ship location. This unit does not contribute any effect to directly alter the ship location. The implementation should remain as the default generation, which simply returns the current value of the ship location.

In practice, trade-offs between explicitly describing such relationships or manually adjusting the update sequence need to be made. The choice depends on the number of attributes of a phenomenon type to be computed in the simulation. The more attributes are involved, the more likely that the first one is the more convenient strategy.

7.3 Use Case 2: Storm Avoidance Strategy Evaluation

In this use case, the goal of the simulation is to evaluate the robustness of a set of storm avoidance strategies of ships during voyages. A ship model is developed with the embedded logic of the strategies under evaluation. Voyages are simulated with the ship model set in the simulated environment with severe weather phenomena. The ship model uses the embedded logic to adjust its behaviors in reaction to the weather condition.

The functional scenario starts when the ship departs from a port. During the voyage, some storm forms on the sea. The ship is informed with the current status of the storm and adjusts its behaviors to avoid the influential area of the storm. The trajectories of the ship and other relevant data are simulated by multiple executions. These data are then analyzed to provide information to domain experts, such as the average length of detours that the ship takes to return to its planned path or reach a nearby safe port, the probability that this ship fails to avoid the extreme weather using the current strategy, the potential consequence of failure and the factor that may cause the failures, etc. Domain experts can then use the information to evaluate the applicability of these strategies and improve them.

This use case is simplified to focus on the extreme weather phenomenon, i.e., the storm. The simulated environment component needs to simulate the storm data required by the ship model during the simulation execution. When developing this component using the framework prototype introduced in Chapter 6, a minimum possible number of Java files are generated at the PSM layer. The small size output is easy to present and reflects the structure of the generated model code from the implemented transformation in the framework prototype. Subsection 7.3.2 presents these files for this use case and uses them to explain the PSM-layer generation structure by the prototype in more detail. The initial requirements about the storm generated from the simulated environment component are briefly summarized in Table 7.2.

Storm	1. For each execution, ship modelers should be able to configure an initial location of a storm with an unnavigable area to the ship. 2. The evolving pattern of the storm should match the normal behaviors of the storms in the Atlantic Ocean in a season which can be chosen by users for one execution. 3. The maximum wind speed of the storm, the moving speed, and the direction of its center should be computed and provided to the ship model at each simulation step.

	4. The unnavigable area to the ship caused by the storm is an area with the wind speed higher than a threshold. This area should be computed by the component under development at each simulation step based on its at-moment status. Users should be able to set a threshold for an execution.

Table 7.2: Initial Requirements of Environmental Phenomena in Use Case 2.

7.3.1 SEDL Description

The simulated environment of this use case is documented in an SEDL file using the textual editor of the framework prototype. The full description is shown in Appendix B.2. During the voyage, the ship should avoid entering the area under some unnavigable conditions to the ship. In this simplified case, the risky condition caused by a storm is defined as the wind speed higher than a threshold w_{max} m/s. The ship modeler decides that the simulated environment component should in charge of informing the current area with the wind speed higher than w_{max} to the ship at each simulation step. The storm is conceptually abstracted as an individual object with spatial extent representing its unnavigable area to the ship. It is documented as a regional SpatialIndividuality named "Storm". As summarized in Table 7.2, the ship model still needs to know the maximum wind speed, the moving direction and the moving speed of the storm's center. These characteristics are explicitly listed as the ThematicProperty-s of the "Storm".

Ship modelers expect that characteristics of a storm change over a simulation execution, which should follow normal behaviors of storms that appear in the Atlantic sea in a season chosen by users for this execution. These requirements are documented as instances ThemeDynamics belonging to these ThematicProperty-s. The alterable condition "season" is recorded as a ConfigurableParameter of the "Storm". During the analysis phase, developers of the simulated environment component shall explain to ship modelers the overall idea of how the required changes could be computed and make agreements on these ideas with them. Based on the agreements, some preliminary information about the computation idea can be added to the SEDL description, such as the refined name and free-text description of the ThemeDynamics instances in this use case. The free-text description pieces are transformed as comments within computation units in follow-up transformation steps to guide the implementation. More discussion about the developer-refined information can be found in the summary of this use case in Subsection 7.3.3.

The size and the location of a storm's unnavigable area should also change during executions as the evolvement of the storm. The center of the storm should move in a way that matches normal behaviors of the Atlantic storms. Similar to the Use Case 1, two LocationThemeDependency-s are added to inform the transformation that the moving speed and direction of the storm are needed for computing its movement. Developers then derive that this area should be computed based on the current state of the maximum wind speed, which has been described as a ThematicProperty. Thus, the dependency between the unnavigable area and this property is documented by the GeometryThemeDependency "UnavigableArea" marked with "Derived by developers".

7.3.2 Transformed Artifacts

This subsection presents the transformation outputs of this use case in a similar way as Subsection 7.2.2 does, starting from the SEDL description in Appendix B.2. The support tools used for illustration are the same as Use Case 1.

The PIM-layer outputs from the input SEDL description generated by the two ATL transformation, i.e., SEDL2Configuration and SEDL2Structure, are illustrated in Figure 7.11. In the configuration Ecore model on the left side of the figure, an EEnumeration named "Season" is generated for adding options of the configuration item "season" that is an EAttribute in the Ecore model. The "season" is generated from the ConfigurableParameter "Season" which is the Option type. SEDL descriptions do not formally document options of such a parameter. The transformation generates an EEnumeration with a default EEnumeration Literal, which should be refined by developers. The data structure model (on the right side of the figure) in this use case is simple, while only one phenomenon type needs to be computed according to the input SEDL description.

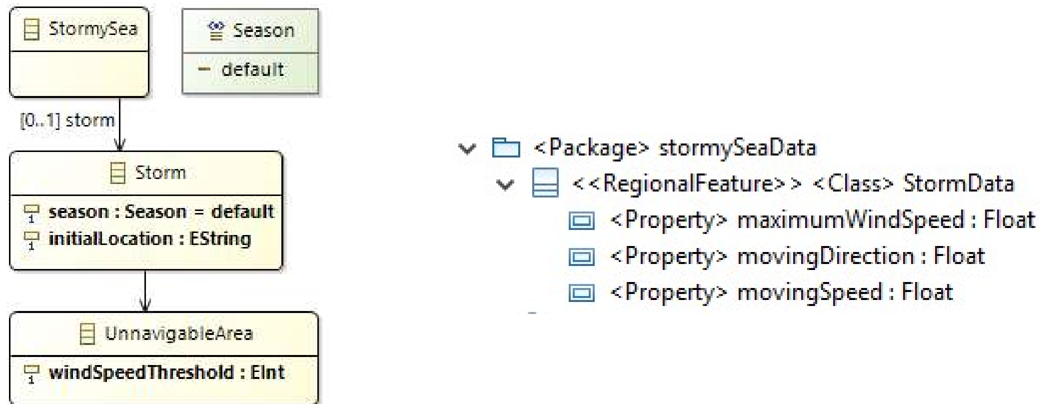

Figure 7.11: PIM-Layer Configuration Model and Data Model in Use Case 2.

Four EEnumerarion Literal representing four seasons are added to the "Season" by developers. Then, both outputs are fed to platform-specific code generation. As shown in the left part of Figure 7.12, the files of the configuration model code are placed in the "stormySeaCon" package. They are generated by the EMF code generation facilities and are not presented in detail. The Java class "StormData" generated from the RegionalFeature with the same name in Figure 7.11 is placed in the "stormySeaData" package. The right side of Figure 7.12 shows the visualized diagram of this class. An instance of this class holds state values of the relevant storm characteristics during the simulation execution. In this prototype, the geometry of the PIM-layer RegionalFeature is mapped to a MasonGeometry.

The PIM-PSM Acceleo transformation of data structure models turns each single-valued feature type specified in Subsubsection 5.2.2.4 into a JavaBean and adds necessary stereotype-specific attributes to it in the Java form. It also generates a JavaBean for each SpatialFunction and each CollectiveFeatureUnit, as Use Case 1 has shown. The geometry attributes of these classes are created as attributes that have the MasonGeometry type of Mason library. Besides, statements in the computation model code skeleton are generated by the transformation. These statements initialize the wrapped geometry of such a MasonGeometry instance as a Point or a Polygon based on the applied stereotype of the input PIM class.

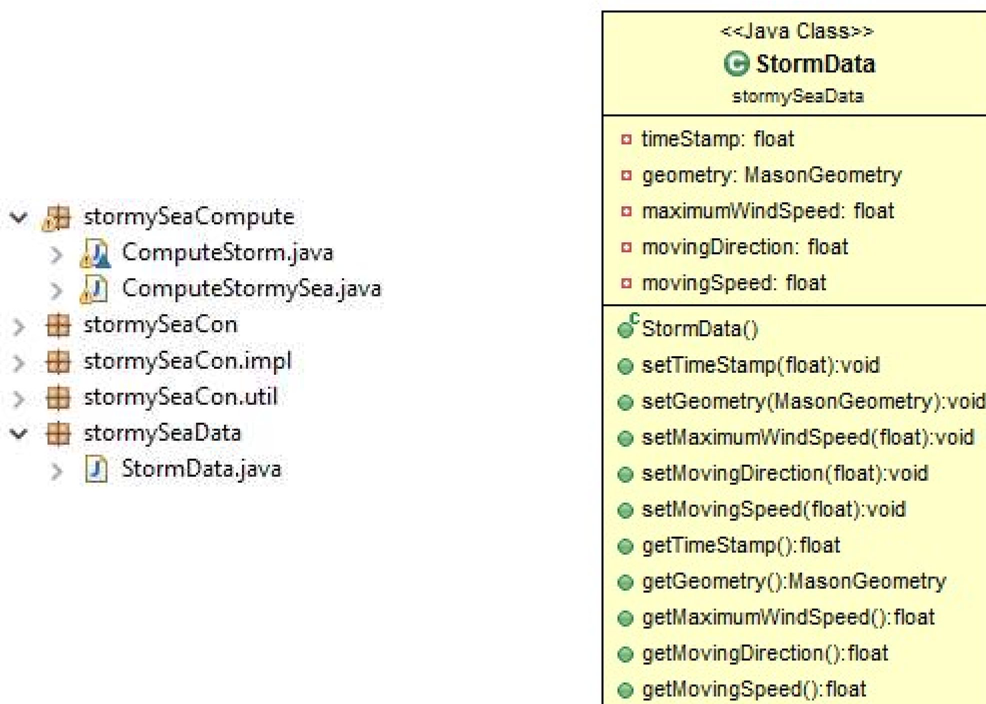

Figure 7.12: PSM-Layer Java Classes of Configuration and Data Model in Use Case 2.

The output code of the SEDL2Computation Acceleo transformation is placed in the "stormySeaCompute" package as shown in Figure 7.12. In this simplified case, two classes are created as visualized in Figure 7.13 as explained below.

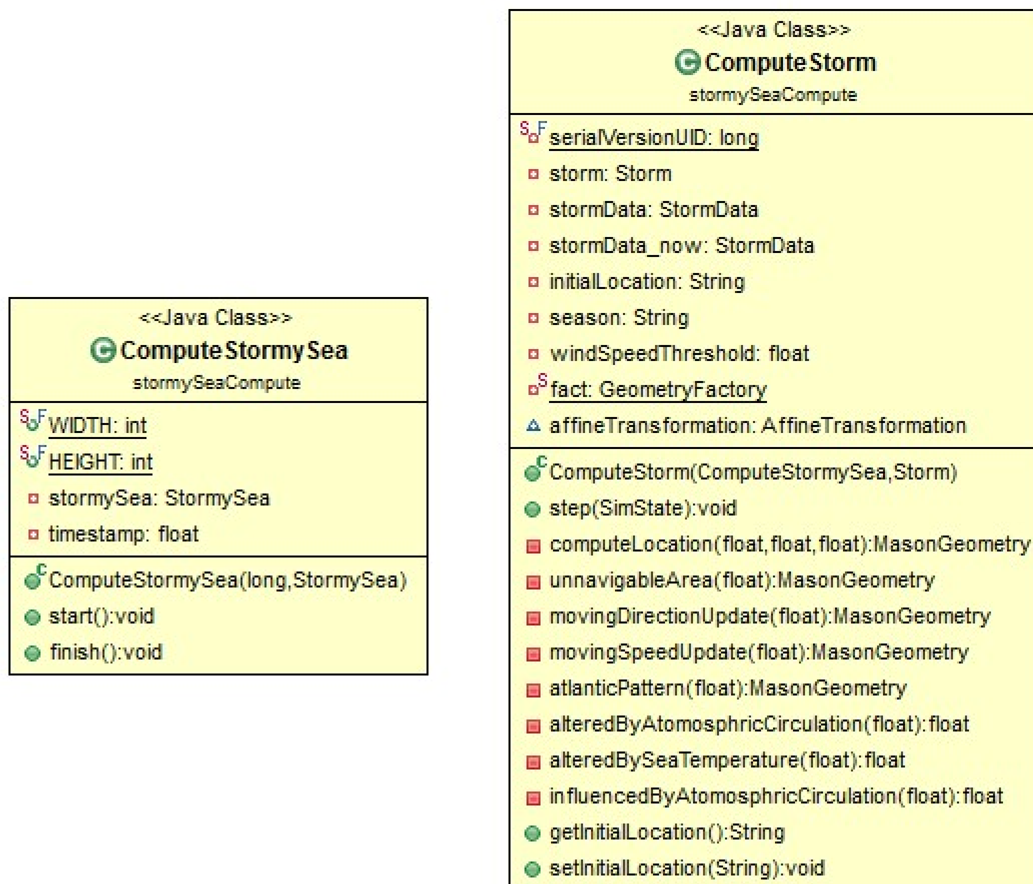

Figure 7.13: PIM-Layer Java Classes of Computation Model in Use Case 2.

First, the "ComputeStormySea" class manages the routine that simulates the whole simulated environment. This class extends the Mason SimState which represents a simulation. Its Constructor initializes an instance of this class with a configuration object. The configuration object is an instance of the "StormySea" class in the configuration model. The "start()" method of the "ComputeStormySea" initializes computation objects for configured phenomena in the configuration object and adds them to the schedule of this simulation. These computation objects are instances of classes that implement the Mason Steppable interface, e.g., the "ComputeStorm" class in this use case.

The SEDL2Computation Acceleo transformation generates a Mason SimState class for each simulation and names it after a SimulatedEnvironment in an input SEDL description. In Use Case 1, this SimState is the "ComputeSea" class.

Second, the "ComputeStorm" class holds the skeleton of the code that computes a storm during the simulation. This class implements the Mason Steppable interface. It holds following data objects and attributes which are generated based on the transformation rules specified in Appendix A.3~A.5: 1) instances of the "StormData" class in the data structure code to hold states of the storm being computed, i.e., "storm" and "storm_now"; 2) attributes to hold configured parameters of the storm being computed, i.e., "initialLocation", "season" and "windSpeedThreshold".

This class also holds the following behavioral elements as Java methods, which are transformed based on the rules in Appendix A.3~A.5: 1) the constructor "ComputeStorm(ComputeStormySea, Storm)", which initializes an instance of this class with a configuration object of the "Storm" type; 2) skeletons of private methods (marked with red squares in Figure 7.13) whose body should implement computation

units according to Appendix A.3, with the free-text description of the input SEDL pieces transformed as Java comments within the methods; 3) private methods whose body should implement the combined effort of necessary units to compute an attribute of the storm, i.e., the computeLocation()”; 2) the “step(SimState)” method of Mason Steppable, which contains the computation flow to update the state of the storm at a simulation step, as specified in Appendix A.5.

Third, platform-specific structures and utilities are also generated in this class, which are only necessary and/or useful in the chosen technical platform of the framework prototype. These structures and utilities include: 1) the public “get” and “set” methods to access the data objects and attributes held by this class, which follow the JavaBeans conventions, e.g., the “getIntialLocation()” and setIntialLocation()[32]; 2) the “sereialVersionUID” that is used for Java serialization control; 3) the Mason utilities “fact” and “affineTransformation”, which can be used to handle the creation and the update of the simulated storm’s geometry.

The SEDL2Computation generates a Mason Steppable class with a similar structure as the “ComputationStorm” class for each SpatialIndividuality to hold the computation model, as also have been illustrated in Use Case 1.

When a SpatialIndividuality describes the members of a FieldOfIndividualities, it is transformed into a CollectiveFeatureUnit in the PIM-layer data structure model. The FieldOfIndividualities is transformed into a CollectiveFeatureType. Developers should replace the applied stereotype of the CollectiveFeatureType to a more specific subtype when necessary. Then, the PIM-layer data structure model is also used as the input of the SEDL2Computation code generation, as shown in Figure 7.14. For such a CollectiveFeatureType, the transformation generates the following code in the above-explained SimState class: 1) a GeomVectorField object or a GeomGridField depending on the applied stereotype of this CollectiveFeatureType, which represents the whole field; 2) a private method for each CharacteristicVariation of the SpatialIndividuality 3) loop code in the “start()” method to initialize a set of the Steppable class instances for computing members in the field and add them to the simulation schedule (Mason-specific). Indexes of these instances are created using the methods of corresponding CharacteristicVariation. The “BackgroundTraffic” in Use Case 1 provides an illustration.

This implementation let SimState classes manage the relationships and variations among units of collective features for simplification reason. However, the SimState class is related to a particular simulation. This implementation decision brings the drawback that the computation model of a CollectiveFeatureType is completely decoupled from the simulation as the PIM-layer models do. This hinders the reuse of such computation models in other simulations. This part of PSM-layer output structure should be optimized in the future so that all computation logic of a CollectiveFeatureType can be maintained in a self-contained subcomponent.

7.3.3 Summary

This use case involves phenomenon type (i.e., the storm) that is conceptualized in the following way. The alterable conditions (e.g., its location when the simulation starts) of such a phenomenon for a simulation execution should be controlled individually. This phenomenon has a regional effect on the system of interest, which in this specific case is an area that the ship should not enter in. Its influential area changes during the simulation with the evolvement of its other characteristics which are also of interest to the ship (e.g., the maximum wind speed).

Similar to Use Case 1, developers also contribute to the SEDL description in this use case, as explained in Subsection 7.3.1. These refined pieces of description have two primary purposes. First, it is used to document the overall implementation idea that has been agreed with the component users, i.e., the ship modelers. Second, it is used to add information that guides further developments. The added information is preserved in relevant pieces of code skeletons by the transformation to guide the implementation.

---

[32] Other similar access methods are omitted from this figure due to the limitation of space.

An SEDL description is used to describe the phenomenon types whose information is required by, but not handled by the system of interest. In this use case, the SEDL description includes the requirements about the simulated environment from the view of the ship modeler. As a result, the sea surface and the circulation of the air over the ocean are not documented as environmental phenomena in this description, since this use case does not require their information to be sent to the ship model during the simulation. Instead, the developer-refined description items "AlteredBySeaTemperature" and "AlteredByAtmosphericCirculation" record the agreed idea to use the sea temperature data and atmospheric circulation data for computing the states of storms.

It also needs to notice that the terms "system of interest model" and "simulated environment" are relative. In the view of the storm modeler, the sea temperature and the atmospheric circulation should be part of the simulated environment for their system of interest model, i.e., the storm model. The storm simulation shall also have an external environment component when necessary. Requirements of this component can be expressed in an SEDL from the view of the storm modeler. In this use case, information about these two phenomena is supposed to be provided by the static data that are manageable by the storm computation component. Thus, no additional component needs to be developed. In Use Case 1, the bathymetry is not included in the SEDL description for the same reason, as mentioned in Section 7.2.

The transformation chain and the generated artifacts of this use case are summarized in Figure 7.14. The working flow and execution time of transformations are similar to Use Case 1.

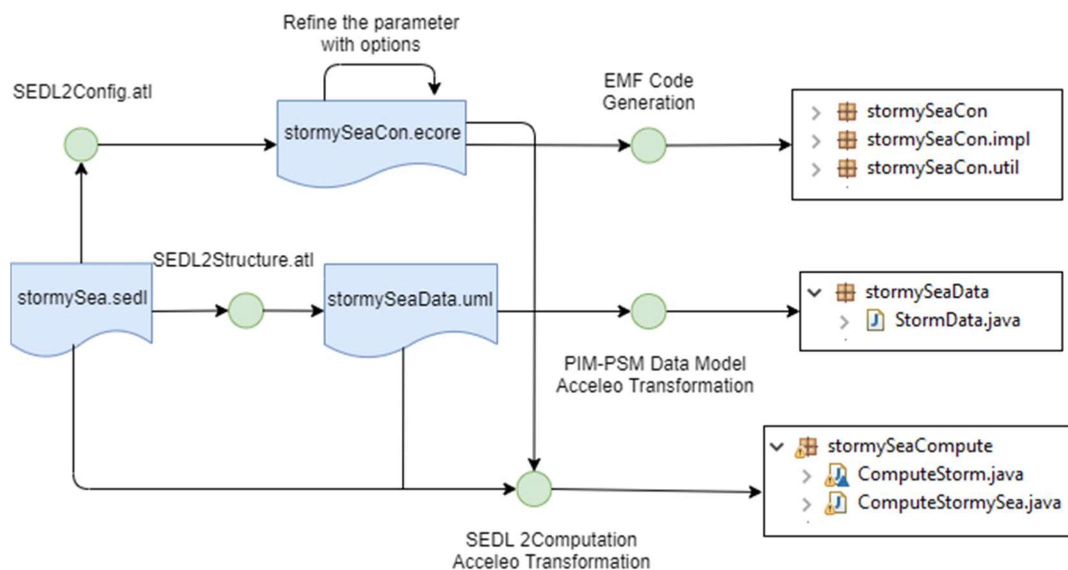

Figure 7.14: Transformations of Use Case 2.

8 Discussions

This chapter provides a summary of this thesis and discusses related issues. Section 8.1 summarizes the contributions of this thesis and denotes that how the research objectives identified in Chapter 1 are fulfilled. Section 8.2 discusses the limitation of transformations in this thesis due to the nature of the model-driven development and strategies to deal with it. Section 8.3 provides visions and preliminary conceptual design for possible extensions of the anchor language SEDL and its transformations. Section 8.4 further discusses the emerging reuse issues with the evolvement of developed components by the framework and needed upgrade of the current framework to handle these issues.

8.1 Contributions and Objective Fulfillment

As introduced in Chapter 1, the development of simulated environments in multi-component spatial simulations encounters difficulties such as miscommunication among various roles, huge development efforts and possible failures of integration with systems of interest due to not correctly preserved requirements. To overcome these difficulties, this thesis develops a domain-specific, language-driven framework. After investigations on existing works to build theoretic foundations and identify missing points as summarized in Chapter 2 and Chapter 3, the framework architecture is specified in Chapter 4. This is followed by Chapter 5 that specifies the domain-specific languages that form the backbone of this framework. The developed framework assists in overcoming the identified difficulties during developments as summarized below. The framework is demonstrated by use cases in Chapter 7 on a prototype implementation as described in Chapter 6.

First, domain-specific languages ease communication among involved roles in the development of simulated environment components.

The analysis-phase language SEDL in Section 5.1 assists in documenting requirements about these components in a structured form. It provides a communication tool to exchange and discuss produced context and behaviors of the component under development. An SEDL description categorizes the expected context of simulated environments into pieces, each of which enclosed in an instance of a formally defined term. These pieces are organized in a simple hierarchical structure as the language model specified, which fulfills Objective 1.1 that captures a cognitive-level description structure of simulated environments. Terms in SEDL are mainly specified based on conceptual forms of spatial phenomena and types of changes they may exhibit. These terms of change types are systematically derived from a common definition of the change expression to fulfill Objective 1.2. The language model is derived from common-sense perceptions and has a small size so that it can be understood by various roles that have different expertise with small learning effort.

At the software design phase, the domain-specific profiles specified in Section 5.2, especially the two describing back ends, assist in describing models of environment components concisely. They cover both structural and behavioral aspects of these components to fulfill Objective 2.1 that captures a generic metamodel for computer simulation components producing simulated environments. Stereotypes in these profiles are specified based on well-established datatypes and actions in spatial computations and simulations. Each stereotype regulates the common structure to model elements applying it, as well as derivable structure for an element through some associations, e.g., the structure of a Snapshot should be derived from its linked SimulatedFeatureTypes. These common structures can be made implicit in a model applying this profile. Thus, such a model has a higher-level of abstraction than a model of the same context in basic UML. They are more readable and easier to be discussed. Besides, model elements represent meaningful software units and are loosely coupled, each of which can be treated as a self-contained task assigned to suitable developers.

Second, the framework enables automated software engineering that reduces development efforts of simulated environment components through various mechanisms.

This thesis categories possible conceptual context of simulated environments and analyses necessary artifacts in software in order to produce them. In Chapter 5, These two perspectives are modeled in the SEDL language model and design-level profiles, respectively, mapped to each other by formally specified

transformation rules that can be automated. These rules fulfill Objective 2.2. By realizing the language tooling based on these models and rules as specified in Subsection 4.2.1, preliminary software models can be created by the automated transformations to save manual work when the expected simulated environments are documented in SEDL. These software models can be further transformed to generate architectural code skeletons to reduce coding work. This specification regulates how the proposed framework should be built, which fulfills Objective 3.1. An implementation of the proposed framework is presented in Chapter 6. The guide of the development process with the framework to fulfill Objective 3.2 is presented in Section 4.4 and is demonstrated with use cases in Chapter 7.

Further, stereotypes in the two back-end profiles in Subsection 5.2.2 and Subsection 5.2.3 regulate stereotype-specific structures and operations. These stereotype-specific artifacts only need to be implemented once at the platform-specific layer on a chosen platform and be embedded in the framework implementation. For a component under development, these implementations are invoked by generated code in addition to the architectural code created by automated transformations to the PSM layer. This strategy further reduces the coding work. The same advantage applies to the configuration profile, while each component or item type in this profile can be implemented once at a platform-specific representation layer as visual interface elements. Besides, since stereotypes in these profiles are determined based on existing works, they guide developers to utilize existing implementation libraries.

Third, transformations from human-perspective descriptions to software models preserve functional requirements of environments component under development.

Domain-independent, formally specified transformations often start from the PIM layer. They can preserve existing model units but do not have mechanisms to help to ensure that the analysis-phase identified requirements have been modeled. With SEDL being introduced to provide a way to express structured human-perspective models of simulated environments, transformations specified in Section 5.3 already starts from the CIM layer. For each piece of SEDL description, these transformations create corresponding model elements associated with free-text descriptions about them. Through this way, documented requirements in SEDL are preserved in formal design models whose structure can be preserved (although refined to be platform-specific) through further formally specified transformations, together with the free-text descriptions that shall guide implementations.

Besides, the specified transformations create configuration schemas aligned to SEDL description with parameters denoted by user-specified names. This ensures user-understandable configuration interfaces. They also create behaviors to pass each configuration item derived from some part of an SEDL description to back-end functions derived from the same part. It helps to make sure that user-configured values being used at intended places.

Assisted by the automated transformations in the realized framework as explained in Chapter 4 and illustrated in Chapter 6, prototypes of environment components can be fast created and be presented to users. Based on this, the recommended development process in Section 4.3 that follows the rapid-prototyping paradigm[126], [127] can be executed. Component users are kept informed and give their feedback during the development while the prototypes are recursively improved, so that functions deviating from their requirements can be identified and implemented.

8.2 Limitation of Model Transformations in Development

A model-driven development process for a computer system consists of a chain of transformation steps, each turning more abstract models of the system to more concrete models. More details and restrictions are included in the output models at each step until a complete system is developed. This nature limits the transformation automation in development, since the information that determines some part of a concrete model does not exist (both explicitly and implicitly) in the corresponding abstract models. This part is not derivable from the more abstract models but needs to be brought in by developers.

Model transformations in this thesis are not exceptions. Transformations specified in Chapter 5 only recognizes the formal part of an SEDL description. This part is mapped to elements that form a component skeleton, or further to architectural code in chosen computation paradigms and/or

implementation platforms. Application-specific behaviors[33] enclosed in these elements are supposed to be formalized manually in later phases, e.g., as code in function bodies in the final implementation.

To overcome this limitation, this thesis recommends a manual refinement step in the CIM-PIM transformation chain, since the PIM-layer profiles proposed by this thesis contain stereotypes implying information that cannot be formally captured by an SEDL description. For instance, while a FieldOfIndividualities may be mapped to a CollectiveFeatureType, the geometry of its units may not be totally captured by the formal part of this FieldOfIndividualities. The general CollectiveFeatureType may be replaced with a subtype of it by developers at the PIM layer, which could then enable transformation automation in further steps. Besides, Subsections 5.2.2~5.2.3 also introduce various PIM-layer stereotypes that extend Actions/Operations. They provide more concise constructs to support developers modeling application-specific behaviors that are not formally captured by SEDL and transforming them into more specific layers.

Adding component structures that are not derivable from the more abstract models via automated transformation requires embedding such knowledge into the automated function. Two strategies are recommended for this mission as introduced below.

Over-generation: in this way, the transformation generates both derivable artifacts and some artifacts that are only possible to be used. The recommended CIM-PIM transformation details at the specification level in Appendix A uses this strategy. For instance, Listing A.5 suggests adding an additional object of a phenomenon's datatype to the class that holds the computation model of the phenomenon type. At the beginning of an execution step, this object is assigned with the current values of the object that holds the state values of the being-computed phenomenon. It is used for "update" style computations which need to calculate the difference between the current value and the value from the previous step of some phenomenon's property, for updating another property. However, it may be redundant for some other computation methods and may not be used in the final implementation.

Pre-determination: a type of elements in a more abstract model may be modeled in various ways in a more concrete model. Following this strategy, the automated transformation assumes that this type is always transformed into elements modeled in one of these ways. The choice is determined when the automated transformation is implemented.

For instance, ExecutionRoutine in SEDL expresses that some data are communicated between simulated environment components and system of interest components. The communication process is not application-specific but still can be modeled in multiple paradigms, e.g., through messages between two components, or a component dealing with global communication among other components. An automated transformation may be developed to generate model artifacts following one of the paradigms, often depending on the platform for final implementation. Also, instead of manually choosing a subtype of TesserlatedFeatureType at the design phase, when the implementation platform uses a fixed type of tessellation, an automated transformation can generate elements applying the stereotype from Subsection 5.2.2 corresponding to that tessellation type.

Since the choice of mapping options often depends on the applied paradigms of the implementation platform, this thesis suggests that automated transformations using the pre-determination strategy to embed the mapping from PIMs to the platform-specific application model/code skeletons.

For transformation realizations, the trade-off between the manual work needed and the freedom given to application developers has to be made. The model structure generated by an automated transformation may not be optimized for a particular component, which needs developer interference for optimization. Besides, SEDL does not contain vocabularies that can formalize application-specific behaviors, e.g., the mode of a movement is usually denoted by the name and the free-text comment of an instance of Movement but not an interpretable formal representation. Thus, it is not recommended to directly interpreting a description expressed by the core SEDL constructs, since the possible result is limited due to the restricted expressiveness.

---

[33] Some of them may be derived from the free-text part of an element in SEDL.

Another reason that causes incomplete application model comes from the not-identified requirements that are not documented in an input SEDL description. This issue does not influence transformation execution itself. It cannot be resolved at the technical level but should be resolved via communications among various roles.

8.3 Visions of SEDL Extension

When this thesis provides a core set of terms and a syntactic structure of the analysis-level language SEDL, several useful extensions shall be made for this language. This section discusses potential useful terms that can be formalized in SEDL in future work, as well as a short proposal on how to formalize them and consequent PIM-layer mapping.

8.3.1 Environmental Phenomenon as Network

A straightforward extension that can be made is to add terms for describing a set of individualities that forms a visually recognizable network, such as networks of roads or waterways. The geometry of an individuality in such a network is often abstracted as a polygon or a polyline.

In current SEDL, individuality groups can be described by a FieldOfIndividualities. Their structure at the design phase may be represented by a class applying a subtype of CollectiveFeatureType. As Subsubsection 5.2.2.2 specifies, relation networks among individualities can be derived from geometries of existing CollectiveFeatureType stereotypes and be used in the computation. The key difference between the network features and instances of these existing types is that a member individuality in a CollectiveFeatureType is treated as a vertex in its implied network. When not specified otherwise at the application level, edges and their weights in such a network are derived based on either adjacency and/or geometric distances among individualities. These edges often do not have geometries. In contrast, an individuality in a network feature is conceptually perceived as an edge. It should also be digitalized in this way when being modeled in computer applications.

Networks are common phenomenon types in some domains, such as traffic simulations. An example is to simulate voyage behaviors of ships along dynamic waterways in a small area, in which the ship model is allowed to move beyond the area of the waterways as the trigger of some adjustment behaviors in simulations. State values of edges (i.e., waterways) need to be updated rather than the vertices (i.e., intersections of waterways). Besides, the geometry of waterways cannot be ignored to reduce the simulated environment representation to a non-spatial graph.

The current SEDL does not have terms to denote that an individuality in a described group is perceived as a network edge. Subtypes or attributes of FieldOfIndividualities shall be included to enable this distinction so that more specific transformations can be performed from SEDL. As discussed above, CollectiveFeatureType stereotypes also have not covered the network features. With SEDL being extended, subtypes of CollectiveFeatureType representing network feature types shall be added to the structural profile. Then, CIM-PIM transformations can be established to map the extended terms to these subtypes and further to stereotype-specific implementations through the transformation process as the current framework specifies.

Two stereotypes for representing network feature types are suggested to be included in the future in the structural profile, i.e., a type whose units are abstracted to lines, and a type whose units are abstracted as polygons while the shape of network edges matters to the simulation.

8.3.2 Influence Between Phenomenon Types

A term to express requirements about influences between EnvironmentalPhenomenon can be added to SEDL in the future. In the current version of SEDL, the requirements of each phenomenon type are described separately within the scope of an EnvironmentalPhenomenon. The computation of each type is developed as a self-contained subcomponent. However, the phenomenon of some type in a simulated environment may be influenced by other types. This situation leads to dependency between subcomponents of the resulting application, in which some type's instances must be computed at first and

are fed to the subcomponent that computes another type's instances. To identify and to implement these connections are often the task of environmental modelers. Nevertheless, at the analysis phase, introducing the suggested term could be useful in the following cases.

First, users of the simulated environment component may express such a requirement: when behaviors and states of EnvironmentalPhenomenon A are computed, states of some other EnvironmentalPhenomenon B should be taken into consideration. Data transition between subcomponents can be derived from such descriptions in design models. However, this influence may turn out to be negligible in the simulation or not exists, decided by environment modelers via communication or in later phases. Thus, the transformation for such a piece of description follows over-generation strategy as introduced in Subsection 8.2.1.

Second, component developers may also identify missing requirements when reviews an SEDL description. They can add an influential type that is not required by the system of interest. In this case, automated transformation shall save their later work.

The suggested term in a UML-based specification shall be modeled as an Association class referencing two involved phenomenon types. Attributes can be added to it to denote the relevant properties of the influential phenomenon type. An instance of this term is conceptually an *influencedBy* link between two EnvironmentalPhenomenon instances as shown in Figure 8.1. The link in the figure implies that in the component under development, the computation functions of the phenomenon "wave" requires the state of phenomenon "wind". Transformation generating computation sequences via these links shall be specified following a similar logic as in Listing A.5 of Appendix A to generate a property update sequence for a phenomenon.

Figure 8.1: An influencedBy Link in an SEDL Description.

8.3.3 Using Spatial Predicates at the System Analysis Phase

Conceptual-level spatial predicates evaluate if a topological relation between two spatial entities satisfies or not. Sets of spatial predicates have been comprehensively investigated and modeled in researches such as [137], [138]. They are applied in spatial databases to form powerful query languages that use topological relations as filters[139]. These predicates may as well benefit the development of simulated environment components when being introduced to SEDL.

In spatial-aware simulations, locations of phenomena are often changing, and so do their topological relations. Some of their behaviors exist when they are in a certain relation, while some other behaviors are triggered when their relation switches to another. A set of terms, each based on a spatial predicate, can be introduced to extend SEDL. An instance of these suggested terms in an SEDL description holds the description of expected behaviors of involved EnviromentalPhenomenon-s, when its represented relationship satisfies (for relations that may last such as "overlap") or occurs (for instantaneous relations such as "meet").[87] This subsection suggests considering the following aspects when formalizing a term in this set as SEDL terms.

First, this term denotes a topological relation that guides the described behavior, which can be taken from existing models such as [87].

Second, slots need to be provided to reference involved types, which may be one EnvironmentalPhenomenon denoting the described behavior should happen when the predicate is true between two instances of this type, or to two EnvironmentalPhenomenon which means that the behavior should happen when the predicate is true between two instances, one for each type.

Third, some behavior types can be identified to formally describe expected behaviors when the predicates are evaluated as true. These behaviors could include: behaviors that change the existence of phenomena, e.g., appear, disappear, merged and split; behaviors that alter geometries like a conditional

Deformation; behaviors that alter thematic property like a conditional ValueDynamics. These behaviors can be recognized by transformations to create necessary subactions within computation units transformed from SEDL Variation-s, which have to be added by developers in the current framework.

At the logical design or more specific layers, spatial predicates are often performed on geometries, which can be implemented independently from the application-specific feature types. Thus, with these terms being added to SEDL, transformations can be established to transform instances of these terms to conditional actions (or further operation code skeletons) based on the evaluation of corresponding predicates on the suitable simulated feature instances.

8.4 Reuse of Implemented Environmental Phenomena

This thesis focuses on the development of environment components in simulations, whose models have not been implemented within the framework proposed by this thesis. With the evolvement of developments, various phenomenon types would be implemented within a chosen platform. It brings new issues of managing and reusing implementations. This section discusses the functionalities needed by the proposed framework to handle these issues.

In transformations specified in Section 5.3, computation models of phenomena types and execution routines for a specific simulation are generated into different classes. It means that developed phenomenon types could be re-grouped into another simulated environment component. To-enable user-oriented reuse of these types, the following upgrade of the current framework could be needed. A conceptual illustration of the potential upgrade is shown in Figure 8.2.

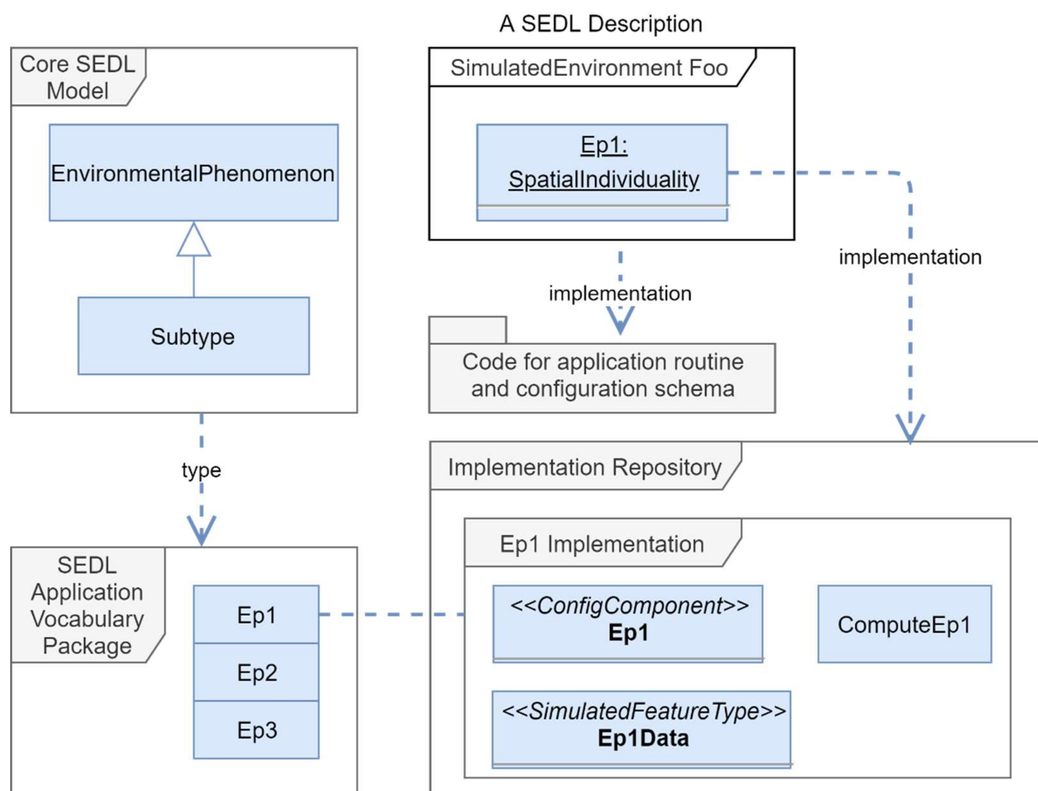

Figure 8.2: Conceptual Architecture for Implementation Reuse.

First, transformations need to create fully de-coupled subcomponents per EnvironmentalPhenomenon and organize them in a repository. This improvement needs to be made on the current transformation SEDL2Config, whose outputs have not been fully de-coupled. In the upgraded version, code for routines and configuration schemas that are specific for simulation applications should be separated from the model code for an EnvironmentalPhenomenon and its user-configurable parameters.

Then, application vocabulary packages need to be integrated into the syntactic model of SEDL. Each term in these packages represents an implemented EnvironmentalPhenomenon. It enables analysis-phase descriptions that some implemented types should be used for a described component. A key problem to be solved here is to deal with terms that cross modeling-levels since these terms are instances of the EnvironmentalPhenomenon. It could be solved by extending SEDL with a special subtype of EnvironmentalPhenomenon (denoted by the "Subtype" placeholder in Figure 8.2), whose values can be picked from an extensible list.

Further, mechanisms need to be provided to link terms in application packages to corresponding implementations, e.g., via annotations. Transformations from such a piece of description should not generate any new computation skeletons or data structures, but only invocation code in an execution routine and a ConfigSchema transformed from its belonged SimulatedEnvironment. For those descriptions which only require using implemented phenomenon types, a platform-specific transformation should generate ready-to-run applications.

Besides, the specification of FieldOfIndividualities shall be extended to enable an instance of FieldOfIndividualities to have its *member* pointed to an implemented SpatialIndividuality, as well to allow adding a CharacteristicVariation that maps to a function for systematically generating parameters of this SpatialIndividuality. It does not break the structure of implemented SpatialIndividuality-s but saves users' work to configure their instances one by one in the new simulation.

Finer-grained reuse, e.g., reuse of implemented actions, is only recommended to be exposed to component developers. It requires further de-coupling of the action units by transformations and organizing the implemented actions into a code library. In addition, model editing supports at the design phase could be provided, allowing developers to pick an implemented action of a specific stereotype at a legible location in the design model, which can be further transformed to invocation code in the new application.

References

[1] J. Banks et al., Discrete-Event System Simulation. Pearson, 2005.
[2] J. A. Sokolowski and C. M. Banks, Principles of Modeling and Simulation: A Multidisciplinary Approach. John Wiley & Sons, 2011.
[3] F. Klügl, M. Fehler, and R. Herrler, "About the role of the environment in multi-agent simulations", in 1st International Workshop on Environments for Multi-Agent Systems (E4MAS04), 2004, pp. 127–149.
[4] C. Läsche, V. Gollücke, and A. Hahn, "Using an HLA simulation environment for safety concept verification of offshore operations", in 27th European Conference on Modelling and Simulation, 2013, pp. 156–162.
[5] M. Berger, W. Helmers and K. Terheyden, Handbuch Nautik 1: Navigatorische Schiffsführung. Springer, 2013.
[6] C. Atkinson and T. Kühne, "Model-driven development: a metamodeling foundation", Software, vol. 20, no. 5, pp. 36–41, 2003.
[7] R. France and B. Rumpe, "Model-driven development of complex software: a research roadmap", in FOSE '07: 2007 Future of Software Engineering, 2007, pp. 37–54.
[8] Object Management Group, "Model Driven Architecture (MDA) - MDA Guide rev. 2.0", 2014, [Online]. Available: http://www.omg.org/mda/.
[9] Object Management Group, "OMG Meta Object Facility (MOF) Core Specification, Version 2.5", 2016, [Online]. Available: https://www.omg.org/spec/MOF/2.5.
[10] Object Management Group, "OMG Unified Modeling Language, Version 2.5.1", 2017, [Online]. Available: https://www.omg.org/spec/UML/2.5.1.
[11] Object Management Group, "Unified Modeling Language: Infrastructure, Version 2.0", 2005, [Online]. Available: https://www.omg.org/spec/UML/2.0.
[12] Object Management Group, "Object Constraint Language, Version 2.4", 2014, [Online]. Available: https://www.omg.org/spec/OCL/2.4.
[13] Object Management Group, "XML Metadata Interchange (XMI) Specification, Version 2.5.1", 2015, [Online]. Available: https://www.omg.org/spec/XMI/2.5.1.
[14] World Wide Web Consortium, "Extensible Markup Language (XML)", 2016, [Online]. Available: https://www.w3.org/XML/.
[15] Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3", 2016, [Online]. Available: https://www.omg.org/spec/QVT/1.3.
[16] C. Atkinson, M. Gutheil, and B. Kennel, "A flexible infrastructure for multilevel language engineering", IEEE Transactions on Software Engineering, vol. 35, no. 6, pp. 742–755, 2009.
[17] C. Atkinson, "Supporting and Applying the UML Conceptual Framework", in The Unified Modeling Language «UML»'98: Beyond the Notation, 1999, pp. 21–36.
[18] C. Atkinson and T. Kühne, "Profiles in a strict metamodeling framework", Science of Computer Programming, vol. 44, no. 1, pp. 5–22, 2002.
[19] O. Eriksson, B. Henderson-Sellers, and P. J. Ågerfalk, "Ontological and linguistic metamodelling revisited: A language use approach", Information and Software Technology, vol.55, no.12, pp. 2099–2124, 2013.
[20] J. C. Mitchell, Foundations for Programming Languages, Vol. 1. Cambridge: MIT press, 1996.
[21] C. Ghezzi and M. Jazayeri, Programming Language Concepts. John Wiley & Sons, 2008.
[22] T. Parr, Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. Pragmatic Bookshelf, 2009.
[23] R. Harper, Practical Foundations for Programming Languages. Cambridge University Press, 2016.
[24] M. Völter, DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace Independent Publishing Platform, 2013.
[25] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography", Sigplan Notices, vol. 35, no. 6, pp. 26–36, 2000.
[26] J. E. Hopcroft, R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd ed. Pearson Education India, 2006.

[27] M. Mernik, J. Heering, and A. Sloane, “When and how to develop domain-specific languages”, ACM Computing Surveys (CSUR), vol. 37, no. 4, pp. 316–344, 2005.
[28] M. Fowler, Domain-Specific Languages, 1st ed. Addison-Wesley Professional, 2010.
[29] M. Fowler, “Language Workbenches: The Killer-App for Domain Specific Languages?”, 2005, [Online]. Available: http://www.martinfowler.com/articles/languageWorkbench.html.
[30] M. Erwig, R.F.Paige and E.Van Wyk, eds, “The state of the art in language workbenches: conclusions from the Language Workbench Challenge”, in Proceedings of the International Conference on Software Language Engineering (SLE2013), 2013, pp. 197–217.
[31] S. Erdweg et al., “Evaluating and comparing Language Workbenches: existing results and benchmarks for the Future”, Computer Languages, Systems and Structures, vol. 44, pp. 24–47, 2015.
[32] M. Pfeiffer and J. Pichler, “A comparison of tool support for textual domain specific languages”, in 8th OOPSLA Workshop on Domain-Specific Modeling (DSM’08), 2008, pp. 1–7.
[33] B. Merkle, “Textual modeling tools: overview and comparison of language workbenches”, in Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA 2010), 2010, pp. 139–148.
[34] M. Völter and V. Pech, “Language modularity with the MPS language workbench”, in 34th International Conference on Software Engineering (ICSE), 2012, pp.1449-1450.
[35] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, EMF: Eclipse Modeling Framework, 2nd ed. Addison-Wesley Professional, 2008.
[36] M. Eysholdt and H. Behrens, “Xtext: implement your language faster than the quick and dirty way”, in Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA 2010), 2010, pp. 307–309.
[37] F. Heidenreich et al., “Derivation and refinement of textual syntax for models”, in Model Driven Architecture - Foundations and Applications, 2009, pp. 114–129.
[38] Lennart C.L. Kats and E. Visser, “The Spoofax language workbench: rules for declarative specification of languages and IDEs”, in ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2010, pp. 444–463.
[39] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev, “ATL: A model transformation tool”, Science of Computer Programming, vol. 72, no. 1–2, pp. 31–39, 2008.
[40] J.-N. Mazón, J. Pardillo, and J. Trujillo, “A model-driven goal-oriented requirement engineering approach for data warehouses”, in Advances in Conceptual Modeling – Foundations and Applications: ER 2007 Workshops CMLSA, FP-UML, ONISW, QoIS, RIGiM,SeCoGIS, 2007, pp. 255–264.
[41] E. S. K. Yu, “Towards modelling and reasoning support for early-phase requirements engineering”, in ISRE'97: 3rd IEEE International Symposium on Requirements Engineering, 1997, pp. 226–235.
[42] N. Koch and A. Kraus, “The expressive power of uml-based web engineering”, in Second International Workshop on Web-oriented Software Technology (IWWOST02), 2002, vol. 16, CYTED.
[43] A. Kraus, A. Knapp, and N. Koch, “Model-driven generation of web applications in UWE”, in International Workshop on Model-Driven Web Engineering (MDWE), vol. 261, 2007.
[44] N. Koch, G. Zhang, and M. J. Escalona, “Model transformations from requirements to web system design,” in 6th International Conference on Web Engineering, 2006, pp. 281–288.
[45] M. J. Escalona and N. Koch, “Metamodeling the Requirements of Web Systems”, in Web Information Systems and Technologies, 2007, pp. 267–280.
[46] F. Jouault and I. Kurtev, “Transforming models with ATL”, in International Conference on Model Driven Engineering Languages and Systems, 2005, pp. 128–138.
[47] A. Fatolahi, S. S. Somé, and T. C. Lethbridge, “Towards A Semi-Automated Model-Driven Method for the Generation of Web-based Applications from Use Cases”, in 4th Model Driven Web Engineering Workshop @ MoDELS, 2008, pp. 31-39.
[48] Use Case Editor (UCEd), [Online]. Available: https://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCEd.html.
[49] Witold Suryn, S. Kherraf, and É. Lefebvre, “Transformation from CIM to PIM using patterns and archetypes”, In 19th Australian Software Engineering Conference, 2008, vol. 00, pp. 338–346.
[50] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd ed. Addison-Wesley Professional, 2004.
[51] M. Kardos and M. Drozdova, “Analytical method of CIM to PIM transformation in Model Driven Architecture (MDA)”, Journal of Information and Organizational Sciences, vol.34, no. 1, pp. 89-99, 2010.

[52] W. Zhang, H. Mei, H. Zhao, and J. Yang, “Transformation from CIM to PIM: a feature-oriented component-based approach”, in Model Driven Engineering Languages and Systems: 8th International Conference (MoDELS 2005), 2005, pp. 248–263.
[53] A. Rodríguez, E. Fernández-Medina, and M. Piattini, “Towards CIM to PIM transformation: from secure business processes defined in BPMN to Use-Cases”, in Business Process Management: 5th International Conference (BPM 2007), 2007, pp. 408–415.
[54] A. Rodríguez, E. Fernández-Medina, and M. Piattini, “CIM to PIM Transformation: a reality”, in Research and Practical Issues of Enterprise Information Systems II, 2008, pp. 1239–1249.
[55] A. Rodríguez, E. Fernández-Medina, and M. Piattini, “Towards Obtaining Analysis-Level Class and Use Case Diagrams from Business Process Models”, in Advances in Conceptual Modeling – Challenges and Opportunities: ER 2008 Workshops CMLSA, ECDM, FP-UML, M2AS, RIGiM, SeCoGIS, WISM, 2008, pp. 103–112.
[56] A. Rodríguez, I. G.-R. de Guzmán, E. Fernández-Medina, and M. Piattini, “Semi-formal transformation of secure business processes into analysis class and use case models: An MDA approach”, Information and Software Technology, vol. 52, no. 9, pp. 945–971, 2010.
[57] A. Rodríguez, E. Fernández-Medina, J. Trujillo, and M. Piattini, “Secure business process model specification through a UML 2.0 activity diagram profile”, Decision Support Systems, vol. 51, no. 3, pp. 446–465, 2011.
[58] J. J. Gutiérrez et al., “Visualization of use cases through automatically generated activity diagrams”, in Model Driven Engineering Languages and Systems: 11th International Conference (MoDELS2008), 2008, pp. 83–96.
[59] C. Hahn, D. Panfilenko, and K. Fischer, “A model-driven approach to close the gap between business requirements and agent-based execution,” in Proceedings of the 4th Workshop on Agent-based Technologies and Applications for Enterprise Interoperability, 2010, pp. 13–24.
[60] Object Management Group, “Service Oriented Architecture Modeling Language, Version 1.0.1”, 2012, [Online]. Available: http://www.omg.org/spec/SoaML.
[61] C. Hahn, C. Madrigal-Mora, and K. Fischer, “A platform-independent metamodel for multiagent systems”, Autonomous Agents and Multi-Agent Systems, vol. 18, no. 2, pp. 239–266, 2009.
[62] V. De Castro, E. Marcos, and J. M. Vara, “Applying CIM-to-PIM model transformations for the service-oriented development of information systems”, Information and Software Technology, vol. 53, no. 1, pp. 87–105, 2011.
[63] J. Gordijn and J. M. Akkermans, “Value-based requirements engineering: exploring innovative e-commerce ideas”, Requirements Engineering, vol. 8, no. 2, pp. 114–134, 2003.
[64] M. D. D. Fabro, J. Bézivin, and P. Valduriez, “Weaving Models with the Eclipse AMW plugin”, In Eclipse Modeling Symposium, Eclipse Summit Europe, 2006, pp. 37-44.
[65] B. Bousetta, O. El Beggar, and T. Gadi, “A methodology for CIM modelling and its transformation to PIM”, Journal of Information Engineering and Applications, vol. 3, no. 2, 2013.
[66] A. Kriouile, N. Addamssiri, T. Gadi, and Y. Balouki, “Getting the static model of PIM from the CIM”, in 3rd IEEE International Colloquium in Information Science and Technology (CIST), 2014, pp. 168–173.
[67] A. Kriouile, T. Gadi, N. Addamssiri, and A. E. Khadimi, “Obtaining behavioral model of PIM from the CIM”, in 4th International Conference on Multimedia Computing and Systems (ICMCS), 2014, pp. 949–954.
[68] A. Kriouile, A. Addamssiri, and T. Gadi, “An MDA method for automatic transformation of models from CIM to PIM”, American Journal of Software Engineering and Applications, vol. 4, no. 1, pp. 1–14, 2015.
[69] Y. Rhazali, Y. Hadi, and A. Mouloudi, “Disciplined approach for transformation CIM to PIM in MDA”, in 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2015, pp. 312–320.
[70] Y. Rhazali, Y. Hadi, and A. Mouloudi, “A methodology for transforming CIM to PIM through UML: From business view to information system view”, in 3rd World Conference on Complex Systems (WCCS), 2015, pp. 1–6.
[71] Y. Rhazali, Y. Hadi, and A. Mouloudi, “Transformation approach CIM to PIM: from business processes models to state machine and package models”, in 1st International Conference on Open Source Software Computing (OSSCOM), 2015, pp. 1–6.

[72] Y. Rhazali, Y. Hadi, and A. Mouloudi, "A new methodology CIM to PIM transformation resulting from an analytical survey", in 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2016, pp. 266–273.
[73] Y. Rhazali, Y. Hadi, and A. Mouloudi, "Model transformation with ATL into MDA from CIM to PIM structured through MVC", Procedia Computer Science, vol. 83, no. Supplement C, pp. 1096–1101, 2016.
[74] Y. Rhazali, Y. Hadi, and A. Mouloudi, "A model transformation in MDA from CIM to PIM represented by web models through SoaML and IFML", in 4th IEEE International Colloquium on Information Science and Technology (CiSt), 2016, pp. 116–121.
[75] Y. Rhazali, Y. Hadi, and A. Mouloudi, "CIM to PIM transformation in MDA from service-oriented business models to web-based design models", International Journal of Software Engineering and Its Applications, vol. 10, no. 4, pp. 125–142, 2016.
[76] M. Brambilla and P. Fraternali, Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML. Morgan Kaufmann, 2014.
[77] T. Yue, L. C. Briand, and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models", Requirements engineering, vol. 16, no. 2, pp. 75–99, 2011.
[78] H. R. Sharifi, M. Mohsenzadeh, and S. M. Hashemi, "CIM to PIM transformation: An analytical survey", International Journal of Computer Technology and Applications, vol 3, no. 2, pp.791-796, 2012.
[79] IEEE Standard Glossary of Software Engineering Terminology.IEEE, 1990.
[80] A. Kriouile, T. Gadi, and Y. Balouki, "CIM to PIM transformation: a criteria based evaluation", International Journal of Computer Technology and Applications, vol. 4, no. 4, pp. 616–625, 2013.
[81] H. Couclelis, "People manipulate objects (but cultivate fields): beyond the raster-vector debate in GIS", in Theories and Methods of Spatio-temporal Reasoning in Geographic Space, 1992, pp. 65–77.
[82] A. Galton, "A formal theory of objects and fields", in International Conference on Spatial Information Theory, 2001, pp. 458–473.
[83] M. Worboys and M. Duckham, GIS: A Computing Perspective. London: Taylor & Francis, 1995.
[84] M. Yuan, "Representing complex phenomena with both object- and field-like properties", Cartography and Geographic Information Science, vol. 28, no. 2, pp. 83–96, 2001.
[85] T. J. Cova and M. F. Goodchild, "Extending geographical representation to include fields of spatial objects", International Journal of Geographical Information Science, vol. 16, no. 6, pp. 509–532, 2002.
[86] R. H. Güting et al., "A foundation for representing and querying moving objects," ACM Transactions on Database Systems, vol. 25, no. 1, pp. 1–42, 2000.
[87] R. H. Güting and M. Schneider, Moving Objects Databases. Morgan Kaufmann, 2005.
[88] J. Xu and R. H. Güting, "A generic data model for moving objects", GeoInformatica, vol. 17, no. 1, pp. 125–172, 2013.
[89] G. Camara, M. J. Egenhofer, K. Ferreira, and P. Andrade, "Fields as a generic data type for big spatial data", in 8th International Conference on Geographic Information Science, 2014, pp. 159-172.
[90] W. Kuhn and A. Ballatore, "Designing a language for spatial computing", in Proceedings of AGILE 2015, Geographic Information Science as an Enabler of Smarter Cities and Communities, 2015, pp. 309–326.
[91] International Organization for Standardization, "ISO19101-1 Geographic information — Reference model — Part 1: Fundamentals", 2014.
[92] International Organization for Standardization, "ISO19103 Geographic information — Conceptual schema language", 2015.
[93] International Organization for Standardization, "ISO19107 Geographic information — Spatial schema", 2003.
[94] International Organization for Standardization, "ISO 19137 Geographic information — Core profile of the spatial schema", 2007.
[95] International Organization for Standardization, "ISO19108 Geographic information — Temporal schema", 2002.
[96] International Organization for Standardization, "ISO19141 Geographic information — Schema for moving features", 2008.
[97] International Organization for Standardization, "ISO19123 Geographic information — Schema for coverage geometry and functions", 2005.

[98] International Organization for Standardization, “ISO 19109 Geographic information — Rules for application schema”, 2015.
[99] Open Geospatial Consortium, “OGC Reference Model, Version 2.1”, 2011, [Online]. Available: https://www.ogc.org/standards/orm.
[100] Open Geospatial Consortium, “The OGC Abstract Specification Topic 0: Abstract Specification Overview, Version 5”, 2005, [Online]. Available: https://portal.opengeospatial.org/files/?artifact_id=7560.
[101] International Organization for Standardization, “Co-operative agreement between ISO/TC 211 Geographic information/Geomatics and the Open GIS Consortium, Inc. (OGC)”, 1999.
[102] Open Geospatial Consortium, “The OpenGIS Abstract Specification Topic 5: Features, Version 5.0”, 2009, [Online]. Available: https://portal.opengeospatial.org/files/?artifact_id=29536.
[103] D. O’Sullivan and G. L. Perry, Spatial Simulation: Exploring Pattern and Process. John Wiley & Sons, 2013.
[104] S. Tisue and U. Wilensky, “NetLogo: A simple environment for modeling complexity”, in International Conference on Complex Systems, 2004, vol. 21, pp. 16–21.
[105] K. Sullivan, M. Coletti, and S. Luke, “GeoMason: geospatial support for MASON”, Department of Computer Science, George Mason University, Technical Report Series, 2010.
[106] J. B. Dabney and T. L. Harman, Mastering Simulink. Pearson, 2004.
[107] W. J. Pierson Jr and L. Moskowitz, “A proposed spectral form for fully developed wind seas based on the similarity theory of SA Kitaigorodskii”, Journal of Geophysical Research, vol. 69, no. 24, pp. 5181–5190, 1964.
[108] C. D. Mobley, “Modeling sea surfaces: a tutorial on Fourier transform techniques, Version 2.0”, 2016, [Online]. Available: https://oceanopticsbook.info/view/references/publications/#mobley-2016.
[109] D. V. Widder, The Heat Equation. Academic Press, 1976.
[110] J. D. Logan, Applied Partial Differential Equations. Springer, 2014.
[111] T. Toffoli and N. Margolus, Cellular Automata Machines: A New Environment for Modeling. Cambridge: MIT Press, 1987.
[112] D. O’Sullivan, “Graph cellular automata: a generalised discrete urban and regional model”, Environment and Planning B: Planning and Design, vol. 28, no. 5, pp. 687–705, 2001.
[113] XS. Yang and Y. Young, “Cellular automata, PDEs, and pattern formation,” in Handbook of Bioinspired Algorithms and Applications, pp.271-282, 2005.
[114] J. Rudnick and G. Gaspari, Elements of the Random Walk: An Introduction for Advanced Students and Researchers. Cambridge University Press, 2004.
[115] L. M. Marsh and R. E. Jones, “The form and consequences of random walk movement models”, Journal of Theoretical Biology, vol. 133, no. 1, pp. 113–131, 1988.
[116] D. Ben-Avraham and S. Havlin, Diffusion and Reactions in Fractals and Disordered Systems. Cambridge University Press, 2000.
[117] M. E. Newman and R. M. Ziff, “Fast Monte Carlo algorithm for site or bond percolation”, Physical Review E, vol. 64, no. 1, 016706, 2001.
[118] J. Adler and U. Lev, “Bootstrap percolation: visualizations and applications”, Brazilian Journal of Physics, vol. 33, no. 3, pp. 641–644, 2003.
[119] K. Malarz and S. Galam, “Square-lattice site percolation at increasing ranges of neighbor bonds”, Physical Review E, vol. 71, no. 1, p. 016125, 2005.
[120] M. Eden, “A two-dimensional growth process”, Dynamics of fractal surfaces, vol. 4, pp. 223–239, 1961.
[121] J.-F. Gouyet, Physics and Fractal Structures. Springer, 1996.
[122] D. Stauffer and A. Aharony, Introduction to Percolation Theory, 2nd ed. London: Taylor & Francis, 2018.
[123] T. Clark, P. Sammut, and J. Willans, Applied Metamodelling: A Foundation for Language Driven Development, 3rd ed. arXiv, 2015.
[124] J. Guttag, “Abstract data types and the development of data structures”, Communications of the ACM, vol. 20, no. 6, pp. 396–404, 2002.
[125] J. L. Sierra, “Language-driven software development (invited talk)”, in OpenAccess Series in Informatics (OASIcs), 2014, vol. 38, pp. 3–12.
[126] M. M. Tanik and R. T. Yeh, “Rapid prototyping in software development”, Computer, vol. 5, pp. 9–11, 1989.

[127] W. R. Bischofberger and G. Pomberger, Prototyping-Oriented Software Development: Concepts and Tools. Springer Science & Business Media, 2012.
[128] Object Management Group, “Business Process Model and Notation (BPMN), Version 2.0”, 2011, [Online]. Available: https://www.omg.org/spec/BPMN/2.0/.
[129] J. V. Guttag and J. J. Horning, “The algeraic specification of abstract data types”, Acta Informatica, vol. 10, no. 1, pp. 27–52, 1978.
[130] International Organization for Standardization, “ISO19111 Geographic information -- Spatial referencing by coordinates” 2007.
[131] R. Rew and G. Davis, “NetCDF: an interface for scientific data access”, IEEE Computer Graphics and Applications, vol. 10, no. 4, pp. 76–82, 1990.
[132] D. A. Griffith, Spatial Autocorrelation and Spatial Filtering: Gaining Understanding Through Theory and Scientific Visualization. Springer Science & Business Media, 2003.
[133] A. Okabe et al., Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, 2nd ed. John Wiley & Sons, 2000.
[134] P. A. Longley et al., Geographic Information Systems and Science. John Wiley & Sons, 2005.
[135] N. Wirth, “What can we do about the unnecessary diversity of notation for syntactic definitions?”, Communications of the ACM, vol. 20, no. 11, pp. 822–823, 1977.
[136] Object Management Group, “MOF Model to Text Transformation Language, Version 1.0”, 2008, [Online]. Available: https://www.eclipse.org/acceleo/.
[137] M. J. Egenhofer and R. D. Franzosa, “Point-set topological spatial relations”, International Journal of Geographical Information System, vol. 5, no. 2, pp. 161–174, 1991.
[138] E. Clementini, J. Sharma, and M. J. Egenhofer, “Modelling topological spatial relations: Strategies for query processing”, Computers & Graphics, vol. 18, no. 6, pp. 815–822, 1994.
[139] M. J. Egenhofer, “Spatial SQL: a query and presentation language,” IEEE Transactions on Knowledge and Data Engineering, vol. 6, no. 1, pp. 86–95, 1994.

Appendix A: CIM-PIM Transformations

This appendix documents details of transformations from SEDL descriptions to PIM-layer models, which are introduced in Section 5.3. Elements in the listings of this appendix are denoted by their names. Since the output model elements may get the name from the input SEDL elements, output model elements are written in *italic* to avoid confusion. For a clear representation, some steps are presented in separate listings as support functions and are invoked in other listings. This appendix does not fix an exact way of transforming the free-text description of a DescriptionItem (i.e., its "*description*" attribute) but leaves it as an implementation choice. In general, it is recommended to transform this piece of text as the comment that is referenced to and/or placed with the generated artifacts from the DescriptionItem.

A.1 Description2Config

Listing A.1-a presents the overall transformation logic from an SEDL description to a configuration schema expressed by Configuration Schema Description Profile specified in Subsection 5.2.1.

	Require: a valid SEDL description s with entry SimulatedEnvironment Envi
1	**Create** *ConfigSchema Envi*;
2	
3	**For each** SpatialIndividuality SI of Envi
4	Boolean configSI ← false;
5	Perform CreateConfigSI(SI, configSI);
6	**If** configSI = true **Then**
7	Add the generated component *SI* to *Envi*;
8	**End if;**
9	**End for;**
10	
11	**For each** FieldOfIndividualities FoI of Envi with member M
12	Boolean configFoI ← false;
13	Boolean configM ← false;
14	Perform CreateConfigSI(M, configM);
15	**If** configM = true **Then**
16	Perform CreateConfigComponent(FoI, configFoI);
17	Add a *SubComponent link* between the generated component *M* (as the *sub* end) and
18	the generated component *FoI*;
19	**End if;**
20	**For each** CharacteristicVariation CV of FoI
21	**If** CV has any parameter **Then**
22	Perform CreateSimpleConfig(CV);
23	Perform CreateConfigComponent(FoI, configFoI);
24	Add a *SubComponent link* between the generated component *CV* (as the *sub* end)
25	and the generated component *FoI*;
26	**End if;**
27	**End for;**
28	**If** configFoI = false **Then**
29	**If** FoI has any parameter, **Then**
30	Perform CreateSimpleConfig(FoI);
31	configFoI ← true;
32	**End if;**
33	**End if;**
34	**If** configFoI = true **Then**
35	Add the generated component *FoI* to Envi;
36	**End if;**
37	**End for;**

Listing A.1-a: Description2Config.

The function CreateConfigSI (SpatialIndividuality SI, Boolean configSI) used in Listing A.1-a is presented in Listing A.1-b. This function transforms a SpatialIndividuality to necessary structures in a configuration schema.

```
1   For each ThematicProperty P of SI
2       Boolean configP ← false;
3       For each Variation IV_P of P
4           If IV_P has options Then
5               Perform CreateAlternativeConfig(IV_P);
6               Perform CreateConfigComponent (P, configP);
7               Add a SubComponent link between the generated component IV_P (as the sub end)
8   and the generated component P;
9               Perform CreateConfigComponent(SI, configSI);
10              Add a SubComponent link between the generated component P (as the sub end)
11  and the generated component SI;
12          End if;
13          If IV_P has any parameter Then
14              Perform CreateSimpleConfig(IV_P);
15              Perform CreateConfigComponent (P, configP);
16              Add a SubComponent link between the generated component IV_P (as the sub end)
17  and the generated component P;
18              Perform CreateConfigComponent(SI, configSI);
19              Add a SubComponent link between the generated component P (as the sub end)
20  and the generated component SI;
21          End if;
22      End for;
23  End for;
24  For each individuality Variation IV_SI of SI
25      If IV_SI has options Then
26          Perform CreateAlternativeConfig(IV_SI);
27          Perform CreateConfigComponent(SI, configSI);
28          Add a SubComponent link between the generated component IV_SI (as the sub end) and
29  the generated component SI;
30      End if;
31      If IV_SI has any parameter Then
32           Perform CreateSimpleConfig(IV_SI);
33           Perform CreateConfigConponent(SI, configSI);
34           Add a SubComponent link between the generated component IV_SI (as the sub end)
35  and the generated component SI;
36       End if;
37  End for;
38  For each CharacteristicVariation CV_SI of SI
39      If CV_SI has any parameter Then
40          Perform CreateSimpleConfig(CV_SI);
41          Perform CreateConfigConponent(SI, configSI);
42          Add a SubComponent link between the generated component CV_SI (as the sub end)
43  and the generated component SI;
44       End if;
45  End for;
46  If configSI = false Then
47      If SI has any parameter Then
48          Perform CreateSimpleConfig(SI);
49          configSI ← true;
50      End if;
51  End if;
```

Listing A.1-b: CreateConfigSI (SpatialIndividuality SI, Boolean configSI).

The function CreateSimpleConfig(Configurable Con) used in previous listings are presented in Listing A.1-c. It transforms a Configurable to a *SimpleConfig*.

1	Create *SimpleConfig Con*;
2	Perform AddConfigItems(Con, *Con*);

Listing A.1-c: CreateSimpleConfig(Configurable Con).

Listing A.1-d presents the function CreateAlternativeConfig(Configurable Con). It transforms a Configurable with options to an *AlternativeConfig*.

1	Create *AlternativeConfig Con*;
2	**For each** option Option of Con
3	Perform CreateSimpleConfig(Option);
4	Add a *ConfigOption* link between the generated component *Option* (as the *option* end) and
5	the generated component *Con*;
6	**End for**;

Listing A.1-d: CreateAlternativeConfig(Configurable Con).

The function CreateConfigComponent(Configurable Con, Boolean configCon) is presented in Listing A.1-e. It transforms a Configurable to a *ConfigComponent* with nested *ConfigComponent-s*, if the current transformation process has not done it.

1	**If** configCon = false **Then**
2	**If** Con has any parameter **Then**
3	Create *ConfigComponent Con*;
4	Perform AddConfigItems(Con, *Con*);
5	configCon ← true;
6	**Else**
7	Create GroupComponent Con;
8	configP ← true;
9	**End if;**
10	**End if**;

Listing A.1-e: CreateConfigComponent(Configurable Con, Boolean configCon).

The function AddConfigItems(Configurable Con, *ConfigComponent Con*) presented in Listing A.1-f transforms ConfigurableParameter-s to *ConfigItem-s* in a configuration schema. For a ConfigurableParameter whose type is Options, the transformation adds an *Enumeration* to the generated schema as the type of the generated *ConfigItem*. This *Enumeration* should contain all possible options of this *ConfigItem*, Since the current SEDL does not provide formal terms to catch configuration options separately, they should be added by component developers.

1	**For each** ConfigurableParameter P of Con
2	Add *ConfigItem p* to the generated *ConfigComponent Con* from P
3	paratype ← type of P
4	**Switch**(paratype)
5	**Case** FreeText: set *p* to the *String* type;
6	**Case** DataSource: set *p* to the *SourceString* type;
7	**Case** Spatial: set *p* to the *SpatialString* type;
8	**Case** Time: set *p* to the *TimeString* type;
9	**Case** Options: add an *Enumeration P* to the generated schema, set the p's to it;
10	**Case** Switch: set p to the *Boolean* type;
11	**Case** Number: set p to the *Real* type;
12	**End if;**

Listing A.1-f: AddConfigItems(Configurable Con, *ConfigComponent Con*).

A.2 Descrption2Structure

Listing A.2-a provides the details to transform an SEDL description to a data structure model applying Simulated Environment Structure Profile specified in Subsection 5.2.2 in a two-dimensional context.

	Require: a valid SEDL description s with entry SimulatedEnvironment Envi
1	**For each** SpatialIndividuality SI of Envi
2	Create *Class SIData*;
3	**If** dimNum of SI equals to 2 **Then**
4	Apply the *LocalFeature* stereotype to *SIData*;
5	Set *geometry* of *SIData* to *Polygon*;
6	**Else if** SI has change involving geometry **Then**
7	Apply the *LocalFeature* stereotype to *SIData*;
8	Set *geometry* of *SIData* to *Polygon*;
9	**Else if** dimNum of SI equals to 0 **Then**
10	Apply the *LocalFeature* stereotype to *SIData*;
11	Set *geometry* of *SIData* to *Point*;
12	**Else if** SI has RigidBodyMovement **or** LocationThemeDependency
13	Apply the *LocalFeature* stereotype to *SIData*;
14	Set *geometry* of *SIData* to Point;
15	**Else**
16	Apply the *GlobalFeature* stereotype to *SIData*;
17	**End if;**
18	**End if;**
19	**End if;**
20	**End if;**
21	**For each** ThematicProperty P of SI
22	Add *Attribute p* to *SIData*;
23	**If** P has ThematicValueDistribution **Then**
24	Perform CreateSpatialFunction(P);
25	Set type of *p* to *SIP_Dist* which is generated from the previous line;
26	**Else**
27	Set *p* to a default value type;
28	**End if;**
29	**End for;**
30	**End for;**
31	
32	**For each** FieldOfIndividualities FoI of Envi with member M
33	Create *Class FoIData*;
34	Create *Class MData* applying the *CollevtiveFeatureUnit* stereotype as its *unit*;
35	**If** M has change involving geometry **Then**
36	Apply the *PolygonSetFeature* stereotype to *FoIData;*
37	Set g*eometry* of MData to *Polygon*;
38	**Else if** dimNum of M equals to 2 **Then**
39	Set *geometry* of *MData* to *Polygon*;
40	**If** M has change involving location **Then**
41	Apply the *PolygonSetFeature* stereotype to *FoIData*;
42	**Else**
43	Apply the *TesserlatedFeatureType* stereotype to *FoIData*;
44	**End if;**
45	**Else**
46	Set *geometry* of *MData* to *Point*;
47	**If** M has change involving location **Then**
48	Apply the *PointSetFeature* stereotype to *FoIDara*;
49	**Else**
50	Apply the *PointSitesFeature* stereotype to *FoIData*;
51	**End if;**
52	**End if;**
53	**End if**
54	**For each** CharacteristicVariation CV_M of M
55	**For each** of its indexName "idx_m"

56	Add *Attribute idx_m* to *MData*;
57	**End for;**
58	**End for;**
59	**For each** ThematicProperty P of M
60	Add *Attribute p* to *MData*;
61	**End for;**
62	**End for;**

Listing A.2-a: Descrption2Structure.

The function CreateSpatialFunction(ThematicProperty SIP) used in Listing A.2-a is presented in Listing A.2-b. It transforms a ThematicProperty with some ThematicValueDistribution-s to a class applying the *SpatialFunction* stereotype, which holds the distribution functions representing the spatially heterogeneous property. The generated *SpatialFunction* class is set as the type of the generated attribute from SIP. After initializing an instance of this class, the values of its attributes that control the distribution functions could be altered during executions. This essentially updates the distribution forms of its represented property.

In the following listing, *SIP* is the configuration component generated from SIP by the transformation documented in Appendix A.1. If no such a component is generated from SIP, this parameter remains empty. The same applies to other generated operations in the following steps.

1	Create a *Class SIP_Dist* applying the *SpatialFunction* stereotype;
2	**For each** ThematicValueDistribution Dist of SIP
3	Add *Operation dist()* to *SIP_Dist*;
4	**For each** of its ConfigurableParameter CP
5	Add *Attribute cp* with the declared type to *SIP_Dist*;
6	**End for;**
7	**If** Dist has options **Then**
8	**Add** *Attribute dist_op* to *SIP_Dist*; // to mark the active option
9	**For each** option Option of Dist
10	Add *Operation option()* to *SIP_Dist*;
11	**For each** ConfigurableParameter CP_O of Option
13	Add *Attribute cp_o* with the declared type to *SIP_Dist*;
14	**End for;**
15	**End for;**
16	Add a *conditional brunch*[34] to *dist()* which: 1) checks the value of *dist_op*, 2) invokes
17	the generated *Operation* from the corresponding option denoted by the value of *dist_op* as the
18	behavior of *dist()*;
19	**End if;**
20	Add *Constructor SIP_Dist (SIP sip)* to *SIP_Dist* and *statements* in it to initialize all attributes
21	generated from ConfigurableParameter-s using *sip*;
22	**End for;**

Listing A.2-b: CreateSpatialFunction(ThematicProperty SIP).

A.3 Mapping from SEDL Items to Computation Units and Supporting Structures

In Listing A.3, *SIData* and *SI* are the outputs from SpatialIndividuality SI by the transformations documented in Appendix A.1 and Appendix A.2, respectively. Similarly, *FoI* and *FoIData* with *MData* as its unit type are the outputs from FieldOfIndividualities FoI, respectively. If no SI or FoI is generated from A.1, generation pieces in this listing corresponding to them remain empty.

	Require: a valid SEDL description s with entry SimulatedEnvironment Envi **Require:** output of s from Description2Config in A.1 **Require:** output of s from Description2Structure in A.2
1	**For each** SpatialIndividuality SI
2	Create *Class ComputeSI*;

[34] The Java-based implementation in Chapter 6 maps it to a switch statement at the PSM layer.

3	Add *Object si* of the *SI* type to *ComputeSI*;
4	Add *Constructor ComputeSI(SI si_con)* to *ComputeSI*;
5	Add *statements* in the *Constructor* to initialize *si* with *si_con*[35];
6	**For each** ConfigurableParameter CP within the scope of SI
7	Add *Attribute cp* of the declared type to *ComputeSI*;
8	**If** CP does not belong to an AlternativeMode
9	Add *statements* to the *Constructor* to initialize *cp* with the value in *si_con*;
10	**End if**;
11	**End for;**
12	**For each** CharacteristicVariation CV of SI
13	Add *Operation cV()* to *ComputeSI*;
14	**For each** of its indexName “idx”
15	Add *Attribute named “idx”* to *ComputeSI*;
16	**End for;**
17	**For each** of its ConfigurableParameter CP_CV
18	Add a *Parameter cp_cv* of the declared type to *cV()*;
19	**End for;**
20	Add *statements* to the *Constructor* to use *cV()* to initialize the generated *Attributes*
21	from CV’s indexes, using corresponding *Attributes* generated from CP_CV-s as actual
22	parameters;
23	**End for;**
24	**For each** individual Variation IV of SI or of its ThematicProperty
25	**If** IV is not a ThematicValueDistribution
26	Add *Operation iV()* to *ComputeSI*;
27	Add *Parameter var* to *iV()*;
28	Set the type of *var* to the mapped attribute type in *SIData* from the Variable of
29	IV, and *iV()*’ return type to the type from the Variant of IV;
30	**If** IV has options **Then**
31	**Add** *Attribute iV_op* to *ComputeSI*; // to mark the active option
32	**For each** option Option of IV
33	Add *Operation option()* to *ComputeSI*;
34	Add *Parameter var_o* to *option()*;
35	Set the type of *var_o* to the mapped attribute type in *SIData* from the
36	Variable of IV, and *option()*’s return type to the type from the Variant of IV;
37	**End for;**
38	Add a *conditional brunch* to the *Constructor* which: 1) checks which option
39	of *iV()* is configured in *si_con*, 2) initializes *iV_op* based on the configured option and *Attributes*
40	in *ComputeSI* relevant to this option using values in *si_con*.
41	Add a *conditional brunch* to *iV()* which: 1) checks the value of *iV_op* to
42	determine the active option, 2) invokes the generated *Operation* corresponding to active option
43	as the behavior of *iV();*
44	**End if;**
45	**End if;**
46	**End for;**
47	**End for;**
48	
49	**For each** FieldOfIndividualities FoI with member M
50	Create *Class ComputeFoI*;
51	Add *Object foiData* of the *FoIData* type to *ComputeFoI*;
52	Add *Object foi* of the *FoI* type to *ComputeFoI*;
53	Add *Constructor ComputeFoI (FoI foi_con)* to *ComputeFoI*;
54	Add *statements* in the *Constructor* to initialize *foi* with *foi_con*;
55	**For each** ConfigurableParameter CP within the scope of FoI
56	Add *Attribute cp* of the declared type to *ComputeFoI*;

[35] e.g.,“this.si = si_con” in the Java-like form

57	**If** CP does not belong to an AlternativeMode
58	Add *statements* to *Constructor* to initialize cp with the value in *foi_con*;
59	**End if**;
60	**End for;**
61	**For each** CharacteristicVariation CV of FoI
62	Add *Operation cV()* to ComputeFoI;
63	**For each** of its indexName “idx”
64	Add *Attribute named “idx”* to *ComputeFoI*;
65	**End for;**
66	**For each** of its ConfigurableParameter CP_CV
67	Add *Parameter cp_cv* of the declared type to *cV()*;
68	**End for;**
69	Add *statements* in *Constructor* to use *cV()* to initialize the generated Attributes from
71	CV’s indexes, using the corresponding *Attributes* generated from CP_CV-s as actual
72	parameters;
73	**End for;**
74	**For each** CharacteristicVariation CV_M of M
75	Add *Operation cV_M()* to *ComputeFoI*;
76	**For each** of its ConfigurableParameter CP_CVM
77	Add Parameter cp_cvm of the declared type to cV_M();
78	**End for;**
79	Add *statements* in the *Constructor* which: 1) get the units iterator of *foiData*, 2) loop
80	over this iterator to execute cV_M() whose body part should implement the function that
81	initializes the generated *Attributes* from CV_M’ indexes for each unit, using the corresponding
82	*Attributes* generated from CP_CVM-s as actual parameters;
83	**End for;**
84	**For each** individual Variation IV_M of M or of its ThematicProperty
85	Add *Operation iV_M()* to *ComputeFoI*;
86	Add *Parameter var* to *iV_M();*
87	Set the type of *var* to the mapped attribute type in *MData* from the Variable of IV_M,
88	and *iV_M()*’s return type to the type from the Variant of IV_M;
89	**If** IV_M has options **Then**
90	**Add** Attribute *iV_M_op* to *ComputeFoI*; // to mark the active option
91	**For each** option Option of IV_M
92	Add *Operation option()* to *ComputeFoI*;
93	Add *Parameter var* to option();
94	Set the type of *var* to the mapped attribute type in *MData* from the Variable
95	of IV_M, and iV()’s return type to the type from the Variant of IV_M;
96	**End for;**
97	Add a *conditional brunch* to the *Constructor* which: 1) checks which option of
98	*iV()* is configured in *foi_con*, 2) initializes *iV_M_op* based on the configured option and
99	*Attributes* in *ComputeSI* relevant to this option using values in *foi_con*.
100	Add a conditional brunch to the iV_M() which: 1) checks *iV_M_op* to determine
101	the active option, 2) invokes the generated *Operation* from the active option as the behavior of
102	*iV_M()*;
103	**End if;**
104	**End for;**
105	**End for;**

Listing A.3: Generate Computation Units and Support Structures.

A.4 Generate the Dependency Graph for EnvironemtnalPhenomenon Computation

Listing A.4 generates a directed graph for an EnvironmentalPhenomenon Ep based on its Variation-s. This graph is used to derive the execution order of the generated computation units from Ep by the transformation documented in Appendix A.3 at a simulation step, as introduced in Subsection 5.3.4.

	Require: a valid EnvironmentalPhenomenon Ep
	Require: *Class ComputeEp* generated from Ep by the transformation in A.3

```
1   //Initial construction of the graph
2   If Ep is a FieldOfIndividualities
3       Use its member SpatialIndividuality as Ep in the following steps;
4   End if;
5   Create a directed graph G with
6       Each edge has an attribute that stores references to Operations of computation units
7   generated from Variation-s of Ep by the transformation documented in A.3;
8       Node's attribute that stores a reference to a subgraph;
9   Add nodes t, l, g to G; //represents time, location, geometry
10  For each ThematicProperty p of Ep
11      Add a node p to G;
12  End for;
13  If RigidBodyMovement m of Ep exists Then
14      Add an edge t → l to G;
15      Store references to Operations generated from every m with this edge;
16  End if;
17  If Deformation d of Ep exists Then
18      Add an edge t → g to G;
19      Store references to Operation generated from every d with this edge;
20  End if;
21  If GeometryLocationDependency gl_d of Ep exists Then
22      Add an edge between g and l, pointed to the Variant of gl_d;
23      Store references to the Operations generated from every gl_d with this edge;
24  End if;
25  For each GeometryThemeDependency gc_d of Ep
26      If no following edge exists Then
27          Add an edge between g and the involved p node, pointed to the Variant of gc_d;
28      End if;
29      Store a reference to the Operation generated from gt_d with this edge;
30  End for;
31  For each LocationThemeDependency lt_d of Ep
32      If no following edge exists Then
33          Add an edge between l and the involved property node, pointed to the Variant of lt_d;
34      End if;
35      Store a reference to the Operation generated from lt_d with this edge;
36  End for;
37  For each ThemeDependency tt_d within the scope of Ep
38      If no following edge exists Then
39          Add an edge between the two involved property nodes, pointed to the Variant of tt_d;
40      End if;
41      Store a reference to the Operation generated from tt_depend with this edge;
42  End for;
43  For each ThematicProperty p
44      If ThemeDynamics dyn_p of p exists Then
45          Add an edge t → p to G;
46          Store references to Operation generated from every dyn_p with this edge;
47      End if;
48  End for;
49   //Loops elimination
50  Search for cycles in G;
51  For each found cycle
52      Add a node cyc to G;
53      Add Operation cyc() to ComputeEp;
54      For each incoming edge e of a node in this cycle from a node n outside this cycle
55          If n → cyc does not exist in G Then
56              Add an edge n → cyc to G;
```

57	**End if;**
58	Add the referenced *Operations* of e to the reference list of n → cyc;
59	Remove e;
60	**End for;**
61	**For each** outgoing edge e' of a node in this cycle to a node n' outside this cycle
62	**If** cyc → n' does not exist in G
63	Add an edge cyc → n' to G;
64	**End if;**
65	Add the referenced *Operations* of e' to the reference list of cyc → n';
66	Remove e';
67	**End for;**
68	Store a reference to the subgraph of the cycle with cyc;
69	Exclude the nodes in the subgraph from the computation order derivation;
70	**End for;**

Listing A.4 Generate Dependency Graph for Computation.

A.5 Generate the Computation Activity for an EnvironemtnalPhenomenon

Listing A.5-a presents the transformation from a SpatialIndividuality to the computation activity to simulate it. Listing A.5-b presents the transformation to create the same kind of activity from a FieldOfIndividualities. Listing A.5-c presents the similar steps in these two listings as a support function.

For a concise representation, the following elements are denoted using the same reference, e.g., n, if not being explicitly denoted otherwise: a node n in the dependency graph, its represented property n of the EnvironmentalPhenomenon, and the transformed attributes *n* in the SimulatedFeatureType class (or in the class instances) from this property.

	Require: a valid SpatialIndividuality SI **Require:** *Class ComputeSI* generated from SI by the transformation in A.3 **Require:** dependency graph G of SI generated by A.4
1	Add *Object siData* of the *SIData* type to *ComputeSI*;
2	Add *Operation computeSI(Time t)* to *ComputeSI*;
3	Add *Object siData'* of the *SIData* type to *ComputeSI*[36];
4	Append *statements* to *computeSI(Time t)* to assign values of *siData* to *siData'*;
5	Append *statements* to *computeSI(Time t)* to update *timestamp* of *siData*;
6	Perform CreateComputationActivity (G, *ComputeSI*, *siData*, *computeSI(Time t)*);

Listing A.5-a Generate Computation Activity for a SpatialIndividuality.

In Listing A.5-b, the feature data object in *ComputeFoI* generated by the transformation documented in Appendix A.3 is denoted as *FoIData foiD*ata. The type of its units is denoted as *MData*.

	Require: a valid FieldOfIndividualities FoI **Require:** *Class Compute*FoI generated from FoI by the transformation in A.3 **Require:** dependency graph G of FoI generated by A.4
1	Add *Operation computeFoI(Time t)* to *ComputeFoI*;
2	Add *Object foiData*' of the *FoIData* type to *ComputeFoI*[37];
3	Append *statements* to *computeFoI(Time t)* to assign values of *foiData* to *foiData'*;
4	Append *statements* to *computeFoI(Time t)* to update *timestamp* of *foiData*;
5	Add *Operation computeM*(*Time t, MData mData*) to *ComputeFoI*;
6	Perform CreateComputeActivity (G, *ComputFoI*, *mData*, *computeM(Time t, MData mdata)*);
7	Append *statemens* to *computeFoI(Time t)* which: 1) get the units iterator of *foiData*, 2) loop over
8	this iterator to update states of all units of *foiData*, by executing *computeM(Time t, MData*
9	*mData)* with each unit as the value of the *mData*;

Listing A.5-b Generate Computation Activity for a FieldOfIndividualities.

[36] This step and the next step create an object to hold the current state of the data object before updating it at a computation step. It may become unnecessary when being mapped to more specific layers.
[37] similar to the *siData'* in Listing A.5-a.

The parameters of the support function in Listing A.5-c should be the transformation outputs from the same Environmental Phenomenon. The G should be the graph generated from Ep by the function documented in Appendix A.4.

When Ep is a SpatialIndividuality SI, the *Class ComputeEp* should be the computation class *ComputeSI* generated from SI by the transformation documented in A.3. The *epData* should the feature data object *siData* held by the class. The *Operation computeEp()* should be *computeSI(Time t)* of *ComputeSI* as shown in Listing A.5-a.

When Ep is a FieldOfIndividualities FoI, the *Class ComputeEp* should be the computation class *ComputeFoI* generated from FoI by the transformation documented in A.3. The *Operation computeEp()* should be the *computeM(Time t, MData mData)* in *ComputeFoI* and the *epData* should its parameter *mData*, as shown in Listing A.5-b.

1	Traverse G starting with node t to get a topologic sequence of nodes;
2	**For each** node n after t in the sequence
3	Add *Operation computeN()* to *ComputeEp*;
4	**If** n represents a ThematicProperty **Then**
5	Set the return type of *computeN()* to corresponding n's type in *epData*;
6	Add *local Object n_obj* to *computeN()*, which is of its return type;
7	Add *statements* to *computeN()* which: 1) execute the referenced *Operations* of its
8	incoming edges, 2) use the value of an *Attribute a* in *epData* as the corresponding *Parameter a*'s
9	value of the executed Operation, 3) use *n_obj* to hold intermediate values;
10	Add a *return statement* at the end of *computeN()* which returns *n_obj*;
11	Append *statements* to *computeEp()* which update the corresponding n of *epData* by
12	executing *computeN()*;
13	**Else if** n represents geometry or location **Then**
14	Set the return type of *computeN()* to the geometry type *in epData*;
15	Add *local Object geo_obj* to *computeN()*,which is of its return type;
16	Add *statements* to *computeN()* which: 1) execute referenced Operations of its incoming
17	edges, 2) use the value of *Attribute a* in *epData* as the corresponding *Parameter a*'s value of the
18	executed *Operation*, 3) use *geo_obj* to hold intermediate values;
19	Add a *return statement* at the end of *computeN()* which returns *geo_obj*;
20	Append *statements* to *computeEp()* which update the *geometry* value of *epData* by
21	executing *computeN();*
22	**Else if** n represents a cyclic subgraph cyc **Then**
23	Add a *member Datatype Cyc* to ComputeEp[38];
24	**For each** node n_cyc within cyc
25	Add an *Attribute* of corresponding n_cyc' s type in *epData* to *Cyc*;
26	**End for**;
27	Set the return type of *computeN()* to *Cyc*;
28	Add *local Object cyc_obj* of the *Cyc* type to *computeN()*;
29	Add a *return statement* to the end of *computeN()* which returns *cyc_obj*;
30	Append *statements* to *computeEp()* which execute *computeN()* and assign
31	involved attribute values of *epData* with values of computed *cyc_obj*;
32	**End if;**
33	**End if;**
34	**End if**
35	**End for;**

Listing A.5-c CreateComputeActivity
(*Graph G, Class ComputeEp, Object epData, Operation computeEp()*).

[38] Create a compound datatype to return multiple values. The specific form of the multiple return values in an implementation combined with PIM-PSM transformations should be adapted to the chosen platform. The compound datatype is only one strategy.

Appendix B: SEDL Descriptions for Use Cases

This appendix provides the SEDL descriptions for use cases in Chapter 7, written in the demonstrative implementation of SEDL with textual concrete syntax as introduced Chapter 6.

B.1 SEDL Description for the Alternative Path Assessment Use Case

```
Simulated Environment Sea {
"A cargo ship executes a planned path passing an area of interest under
various weather conditions. During the voyage, the ship is informed by
VTS and its own sensors about situation of wind and ships in surroundings.
Its movements is influenced by the force from the ocean wave.

This is a use case which aims at demonstrating the usage of the development
framework. Requirements of the environment phenomena may be simplified to
remain the focus. Some behaviours of environmental phenomena may also not
necessary but are included to cover different aspects that can be expressed
by SEDL for demonstration purpose.The described simulated environment should
not be used for safe-critical simulations which requires high-quality
results, without confirmation of the system modellers in those simulations"

    Spatial Individuality Wind {
    "The ship model should be fed with the average speed and direction of
    the wind in the simulated area at each simulation step."
        type: Global

        Thematic Property Speed {
            Dynamics: SpeedDynamics {
            "The wind speed could change over time either randomly or
            linearly,which can be chosen by users for an execution"

                option Random
                "The wind speed has some random turbulence over time with
                a modifiable mean and a modifiable variance."
                     parameter:mean,Number "mean speed during an execution";
                               variance,Number "variance of the turbulence"

                option Linear
                "The wind speed changes linearly over time at a modifiable
                rate per execution step,start with a modifiable value"
                     parameter:init,Number "initial wind speed";
                               rate,Number "change rate of the speed per step"
            }
        }

        Thematic Property Direction {
            Dynamics: RandomTurbulence {
            "The wind direction roughly stays at a same mean value set by
            users with small random turbulence over time."
                parameter: mean, Number "mean wind direction, 0-360 degree
                                         with north as 0"
            }
        }
    }
    Spatial Individuality Wave {
    "The wave information in the area of interest should be fed to the ship
    model. Patterns of their values should match the expected behaviours
    when the sea is blowed by wind, which changes both in space and time.

    For a short demonstration, only the wave height is requested by the ship
    model in this use case. For each type of change, only one aspect is paid
    attention on. Other properties and variations that can be expressed in a
    similar form in SEDL are ignored in the use case."
```

```
Spatial Individuality Wave {
"The wave information in the area of interest should be fed to the ship
model. Patterns of their values should match the expected behaviours
when the sea is blowed by wind, which changes both in space and time.

For a short demonstration, only the wave height is requested by the ship
model in this use case. For each type of change, only one aspect is paid
attention on. Other properties and variations that can be expressed in a
similar form in SEDL are ignored in the use case."

    type: Regional
    Thematic Property Height {
        Spatial Distribution: BlowByWind{
        "Patterns of wave heigh should match the expected behaviours
        when the sea is blowed by wind, which changes both in space
        and time.

        Refined by developers - the difference of waves over space of
        values can be described by a Phillips wave variance specturm.
        The wave height at a location are then computed from variance
        of that location according to the specturm via a inverse
        fourier transformation."
        }
        Dynamics: FullyDevelopedSea{
        "Refined by developers - the are wave specturm are time dependent.
        At each time, the time is updated which alters the computation of
        values for different locations."
        }
    }
}


Field of Individuality BackgroundTraffic {
"A set of ships moves in the environment to add some marine traffic flows.
The number of ships should be modifiable for different executions to test
the cargo ship's behaviours in environments various traffic density."
     parameter: numberOfShips, Number "the number of ships in the traffic"

     Spatial Individuality Ship {
     "The geometry of the vessels can be neglected at the scale of
     the simulation."

        type: Point

        Characteristic Variation: RandomInitialLocation {
        "The background vessels should be randomly placed in the
        environment. Users should be able set a mean location for the
        generated ship sets in an execution"
            parameter: meanX,Spatial"mean x coordinate of ships'
                                     initial locations";
                       meanY,Spatial"mean y coordinate of ships'
                                     initial location"
            index: initialLocation
        }

        Characteristic Variation: RandomSpeed {
        "The speed of a vessel in the background traffic should be randomly
        picked from a set of possible values that are specific to different
        ship types. The probability that a value is picked should equal to
        the ratio of the corresponding vessel type in the whole traffic flow
        in the real-world counterpart of the simulated world"

             index:speed
        }
```

```
            Movement: RandomMove {
            "The background vessels moves to its current moving direction at
            its initial speed at each step. Different vessels should not
            collide to each other"
            }

            Location Theme Dependency with self.MovingDirection
                                 as variable: ChangingDirection{
            "Derived by developers - the ship moves towards the new moving
            direction"
            }

            Thematic Property MovingDirection {
                Dynamics: CorrelatedWalks {
                "The moving direction of a ship should turn a angle relative to
                the direction of the last step. The angle should be drawn from
                a normal distribution."

                 parameter: mean,Number "mean of the turning angle,
                                           0-50 degree";
                            variance,Number "variance of the turning angle,
                                              0-50 degree"
                }
            }
        }
    }
}
```

B.1 SEDL Description for the Alternative Path Assessment Use Case

```
Simulated Environment StormySea {
"The purpose of this simulation is to evaluate the robustness of several storm
avoidance strategies of a ship during the voyage. A ship model is developed
with embedded logic of the strategies under evaluation.Voyages are simulated
with the ship model being set in the simulated environment with some severe
storm. The ship model is informed by the current statues of the storm and
adjusts its behaviours. The execution starts when the ship receives the warning
first time"

    Spatial Individuality Storm {
    "A storm should be simulated in the environment. The ship is informed with
    the current moving speed and direction of the storm during the voyage. For
    this use case, the environment component should also in charge of computing
    an unavigable area to the ship at each communication step as the geometry of
    the storm, so that the ship can adjust its behavious to avoid entering the
    unavigable area during the voyage."
        parameter: initialLocation, Spatial
                      "User should be able to draw a polygon on the map through
                      the user interface as the unavigable area of the storm at
                      the beginning of an execution.";
                   season, Option
                      "User should be able to choose one of the four seasons for an
                       execution. The storm should follow the normal storms'
                       behaviours in the Atlantic Ocean in the chosen season"
        type: Regional

        Movement: AtlanticPattern{
        "The storm should follow the normal moving pattern of storms in the
         Atlantic Ocean in the season that the current execution simulates."
        }

        Geometry Theme Dependency with self.MaximunWindSpeed
                                 as variable: UnnavigableArea{
        "Derived by developers - the unavigable area to the ship should increase
        with the maximum wind speed by a factor. For this use case, this area
        should be represented by the geometry the storm. It is an area with wind
        speed higher than a certain threshold that can be adjusted by users."
```

```
            parameter: windSpeedThreshold,Number
                "the maximum wind speed with which the ship can safely navigate."
        }

        Location Theme Dependency with self.MovingDirection
                                  as variable: MovingDirectionUpdate{
        "Derived by developers - the storm moves towards the updated direction"
        }
        Location Theme Dependency with self.MovingSpeed
                                  as variable: MovingSpeedUpdate{
        "Derived by developers - the storm moves at the updated moving speed"
        }

        Thematic Property MaximunWindSpeed {
        "The maximum wind speed of the storm"
            Dynamics: AlteredBySeaTemperature {
            "Refined by developers - the maximum wind speed of the storm
            changes during the execution by the temperature of the sea surface
            where is it passing. The temperature is read from a file."
            }
        }

        Thematic Property MovingDirection {
        "The moving direction of the storm"
            Dynamics: AlteredByAtmosphericCirculation {
            "Refined by developers - the moving direction of the storm changes
            during the execution by the atmospheric circulation of the Atlantic
            ocean in the configured season of the current execution. The
            direction and force of the circulation is read from a file."
            }
        }

        Thematic Property MovingSpeed {
        "The moving speed of the storm"
            Dynamics: InfluencedByAtomosphricCirculation  {
            "Refined by developers - moving speed of the storm changes during
            the execution by the atmospheric circulation of the Atlantic ocean
            in the configured season of the current execution. The direction
            and the force of the circulation is read from a file"
            }
        }
    }
}
```

Declaration of Authorship

I, Liqun Wu, born on June 12, 1987 in Anhui, China, hereby declare that I am the sole author of the thesis with the title “A Language-Driven Development Framework for Components to Generate Environmental Data for Simulation Scenarios” and have not used any sources other than those specified. I further declare that I have followed the general principles of scientific work and publications, as defined in the guidelines of good academic practice of the Carl von Ossietzky University of Oldenburg.

Liqun Wu   Date