A Unified Tooling Framework for Pedestrian Simulation

Mario Imber

Abstract

Mathematical models of walking behaviour of pedestrians find application in numerous areas including architecture, panic analysis, traffic control, urban design, retail industry or entertainment. Pedestrian modelling evolved into a broad field of research with approaches following various different base concepts. Commercial and prototypical software tools exist to enable analysis and simulation of pedestrians in a given environment. In the context of IT security, pedestrian simulation can support the evaluation of environmental design with respect to physical security and crime prevention.

Existing software solutions provide highly integrated components used to design simulation scenarios, run simulations or calculations and visualize or analyse the results. Though this application structure supports a consistent user experience, it prohibits the reuse of scenario definitions with different simulation models due to proprietary interfaces or tightly coupled implementations. This thesis follows an alternative approach of decoupled scenario creation, attempting to develop a more flexible tooling framework for unified definition of pedestrian scenarios.

The framework consists of two parts, first the specification of a generic interface used to describe the parameterization of scenarios expected by a simulation model and to specify scenario instances following that parameterization, and second a software tool supporting the creation of simulation scenarios conforming to that interface. The interface specification is developed based on a generic model of scenario input data, which is derived from an analysis of existing models. A categorization of modelling approaches is given, followed by reviews of selected proposals in literature providing a broad coverage of the problem field. The technical realization of the interface in XML is discussed using UML class diagrams. A Java implementation, the *Scenario Builder*, is presented as the second framework part, a software tool realizing user interface concepts that enable practical adoption of the proposed interface.

Kurzfassung

Die mathematische Modellierung des Bewegungsverhaltens von Fußgängern findet vielfältige Anwendung, unter anderem in den Bereichen Architektur, Panikanalyse, Verkehrskontrolle, Stadtplanung, Einzelhandel und Unterhaltung. Fußgängermodellierung entwickelte sich zu einem breiten Forschungsfeld mit einer Vielzahl unterschiedlicher Ansätze und Grundkonzepte. Softwaretools zur Analyse und Simulation von Fußgängern in einer gegebenen örtlichen Umgebung sind kommerziell und prototypisch verfügbar. Im Zusammenhang mit IT-Sicherheit kann Fußgängersimulation die Planung von Infrastruktur und Umfeld hinsichtlich physischer Sicherheit und Kriminalitätsvorbeugung unterstützen.

Bestehende Softwareprodukte bieten ganzheitliche und hochintegrierte Lösungen, die den Entwurf von Simulationsszenarien, die Durchführung von Simulationen und Berechnungen, sowie die Visualisierung und Analyse der Ergebnisse umfassen. Während diese Anwendungsstruktur den Vorteil von konsistenter Benutzbarkeit einzelner Programme bringt, verhindert sie die Wiederverwendung von Szenariodaten mit anderen Simulationsmodellen aufgrund von proprietären Schnittstellen und eng gekoppelten Implementierungsteilen. Die vorliegende Arbeit verfolgt einen alternativen Ansatz, bei dem die Erzeugung von Eingabedaten entkoppelt erfolgt, um ein flexibleres Framework für eine vereinheitlichte Definition von Fußgängerszenarien zu entwickeln.

Das Framework besteht aus zwei Teilen, erstens der Spezifikation einer generischen Schnittstelle zur Beschreibung der von einem Simulationsmodell geforderten Parametrisierung sowie zur Definition von Szenario-Instanzen die dieser Parametrisierung entsprechen, und zweitens einem Softwarewerkzeug zur Erstellung von Simulationsszenarien gemäß dieser Schnittstelle. Die Schnittstellenspezifikation basiert auf einem verallgemeinerten Modell von Szenario-Eingabedaten, das aus einer Analyse bestehender Modelle abgeleitet wird. Nach der Entwicklung einer Kategorisierung von Modellierungsansätzen in der Literatur werden ausgewählte Modelle vorgestellt und mit Schwerpunkt auf die Struktur der Ein- und Ausgabedaten verglichen. Die Umsetzung der Schnittstelle in XML wird mit Hilfe von UML-Diagrammen erklärt. Als zweiter Teil des Frameworks wird der *Scenario Builder* vorgestellt, eine Java-Implementierung von Benutzerbarkeits-Konzepten für den praktischen Einsatz der entwickelten Schnittstelle.

Contents

1 Introduction **9**
1.1 Software supported Pedestrian Simulation . . . 9
1.2 Development of Pedestrian Simulation Software . . . 9
1.3 Related Work - currently available Software Solutions . . . 10
1.4 Problem Statement - Unified Tooling . . . 11
1.5 Methodology . . . 14
1.5.1 Interface Specification Methodology . . . 14
1.5.2 Tool Development and Validation Methodology . . . 15

2 Pedestrian Modelling **16**
2.1 Application of Pedestrian Models . . . 16
2.2 Classification of Pedestrian Models . . . 17
2.2.1 Classification Criterions . . . 18
2.3 Model Analysis . . . 20
2.3.1 Macroscopic Models . . . 21
2.3.2 Microscopic Models . . . 23
2.3.3 Mesoscopic Models . . . 30
2.3.4 Overview . . . 34
2.4 Input Model Summary . . . 36
2.4.1 Environment Geometry . . . 36
2.4.2 Environment Semanctics . . . 36
2.4.3 Population . . . 36
2.4.4 Model Calibration . . . 37
2.4.5 Reporting Configuration . . . 37

3 Generic Pedestrian Simulation Interface (GPSI) **38**
3.1 Requirements . . . 38
3.2 System Metamodel . . . 39
3.3 Model Architecture . . . 40
3.3.1 Top-level Layering . . . 40
3.3.2 Core Concepts . . . 41
3.3.3 Static Structure . . . 44
3.4 Interface Implementation . . . 45
3.4.1 Technology Choice - XML and XML Schema . . . 45
3.4.2 Data Formats . . . 46
3.4.3 UML for the Documentation of XML Schema Definitions . . . 47
3.4.4 Interface Specification . . . 48

4 Unified Scenario Definition Tool: Scenario Builder **64**
4.1 Requirements . . . 64
4.1.1 Data Organization . . . 64
4.1.2 Data Input and Output . . . 65
4.1.3 Geometry Prototyping . . . 65
4.1.4 Look and Feel . . . 65
4.2 Project Model . . . 66
4.3 User Interface Model . . . 69
4.3.1 Overview . . . 69
4.3.2 Explorer Views . . . 70
4.3.3 Element Inspector . . . 76
4.3.4 Toolbar Actions . . . 82
4.3.5 Layout Editors . . . 82
4.3.6 Infrastructure Editor . . . 87
4.3.7 OD-Map Editor . . . 90
4.3.8 Problems View . . . 93
4.3.9 Problem Categories . . . 93
4.4 Base Technologies and Libraries . . . 95
4.4.1 Eclipse RCP . . . 95
4.4.2 SWT . . . 96
4.4.3 JFace . . . 96
4.4.4 GEF and Draw2d . . . 97
4.4.5 JAXB . . . 98
4.4.6 Commons Math . . . 99
4.4.7 opencsv . . . 99
4.4.8 KTable . . . 99
4.4.9 log4j . . . 99
4.5 Implementation Architecture . . . 100
4.5.1 The root package . . . 100
4.5.2 The `domain` package . . . 100
4.5.3 The `control` package . . . 106
4.5.4 The `view` package . . . 110
4.6 Tool Validation . . . 115
4.6.1 Workflow Example 1: Creating a Simulation Scenario from CAD data . . . 116
4.6.2 Workflow Example 2: Comparing Microscopic, Mesoscopic and Macroscopic Simulation of a Scenario . . . 116
4.6.3 Workflow Example 3: Evaluating Variations of an existing Scenario . . . 117
4.6.4 Workflow Example 4: Migrating existing Scenario data to a changed Simulation Model . . . 117

5 Conclusions and Future Work **119**

List of Figures

1.1 General functional structure of software-supported pedestrian simulation 9
1.2 Development process of simulation software 10
1.3 Integrated architecture of simulation software 12
1.4 Simulation software architecture with unified tooling 13
1.5 Effects of unified scenario definition on the development process of simulation software 13

2.1 Metamodel of pedestrian simulation 20

3.1 Context of the generic input model 39
3.2 Package diagram of the generic input model 41
3.3 Overview class diagram of the generic input model 42
3.4 Deployment diagram of unified tooling 47
3.5 Versioning of GPSI 50
3.6 Root structure of pedestrianScenario.xsd 51
3.7 Items in pedestrianScenario.xsd 52
3.8 Sections in pedestrianScenario.xsd 53
3.9 Vector layouts in pedestrianScenario.xsd 53
3.10 Grid-based layouts in pedestrianScenario.xsd 54
3.11 Graph-based layouts in pedestrianScenario.xsd 55
3.12 Pedestrian types and OD-matrices in pedestrianScenario.xsd 55
3.13 Parameterization in pedestrianScenario.xsd 57
3.14 Parameterization in pedestrianScenario.xsd 58
3.15 Root structure of pedestrianScenarioMetadata.xsd 60
3.16 Abstract parameters in pedestrianScenarioMetadata.xsd 62
3.17 Element types in pedestrianScenarioMetadata.xsd 63

4.1 Class diagram of the Scenario Project Model 67
4.2 Scenario overrides in the Scenario Project Model 68
4.3 Project Explorer (left) and Scenario Explorer (right) 72
4.4 Element Inspector 77
4.5 Element Inspector, text field for string and numerical values 77
4.6 Element Inspector, selection box for enumeration values 78
4.7 Element Inspector, scenario scope 79
4.8 Element Inspector, problem indication 79
4.9 Element Inspector, custom distribution 80
4.10 Element Inspector, ordered discrete, ordered continuous and unordered uniform distributions 80
4.11 Element Inspector, substitute a parameter value by a probability distribution . . 81
4.12 Element Inspector, customize a probability distribution 81
4.13 Element Inspector, revert to the original distribution 81

4.14 Layout editor using a background image 84
4.15 Vector Layout Editor 85
4.16 Grid Layout Editor 87
4.17 Grid Layout Editor, rotated grid 88
4.18 Graph Layout Editor 89
4.19 Graph Layout Editor, manual link distances 89
4.20 Infrastructure Editor, 3 sections on two floors 91
4.21 Infrastructure Editor, rotated section 91
4.22 OD-Map Editor, basins selection 92
4.23 OD-Map Editor, od-matrices definition 92
4.24 Problems View 93
4.25 High-level structure of source packages 101
4.26 Domain base classes 102
4.27 Domain object duplication 104
4.28 Parameter bindings and problem annotations (1) 105
4.29 Parameter bindings and problem annotations (2) 105
4.30 Actions base hierarchy 107
4.31 Operations 108

List of Tables

1.1 Functional structure of available software solutions 12

2.1 Comparison of classifications of pedestrian modelling in literature 18
2.2 Notation and equations of [BSB07] 32
2.3 Closed loop control interpretation of the showcase of [BSB07] 33
2.4 Classification of reviewed models 35

3.1 Examples of model concepts 40
3.2 XSD-to-UML mapping 49
3.3 Built-in distribution types 58

4.1 GEF model implementation by the vector layout editor 112
4.2 GEF model implementation by the grid layout editor 112
4.3 GEF model implementation by the graph layout editor 113
4.4 GEF model implementation by the graph layout editor 114

1 Introduction

1.1 Software supported Pedestrian Simulation

Mathematical formalization of behavioural aspects of pedestrians has been object of research since the 1960s, originally emanating from knowledge about vehicular traffic flow. Different approaches have been taken to develop models of pedestrian dynamics with the intention to enable analysis and simulation of walking behavior in given spatial configurations.

This work focuses on pedestrian simulation models that are dedicated to be implemented as computer programs, in contrast to purely analytical "hand-calculation" methods, see Section 2.2 for an overview of existing modelling approaches.

Software-supported application of a simulation model to real-world situations can generally be structured into three subsystems, used in consecutive steps in the application workflow, as indicated by Figure 1.1.

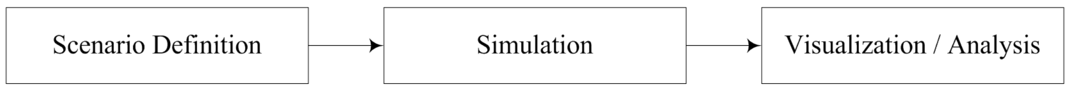


Figure 1.1: General functional structure of software-supported pedestrian simulation

First, the simulation scenario needs to be defined by the user. More precisely, the real-world problem needs to be mapped to a software representation, inherently reducing information to a well-defined finite data set. The simulation step addresses the actual execution of algorithms specified by the simulation model. The result of the simulation is information about pedestrian behaviour, which is in turn presented to the user in the visualization/analysis step. This structure presents a very abstract view, covering various possibilities for concrete implementations. For example, visualization and analysis could be realized using detailed 3-dimensional animation as well as 2-dimensional density plots or numerical or graphical reports, among many other possibilities. Also, with *online* or *real-time simulation* implementations, simulation and visualization/analysis happen in parallel or looped in short time steps rather than strictly after one another.

1.2 Development of Pedestrian Simulation Software

Figure 1.2 summarizes and refines the descriptions of the circular development process of pedestrian simulation software of [Kre07], [Klü03] and [Daa04].

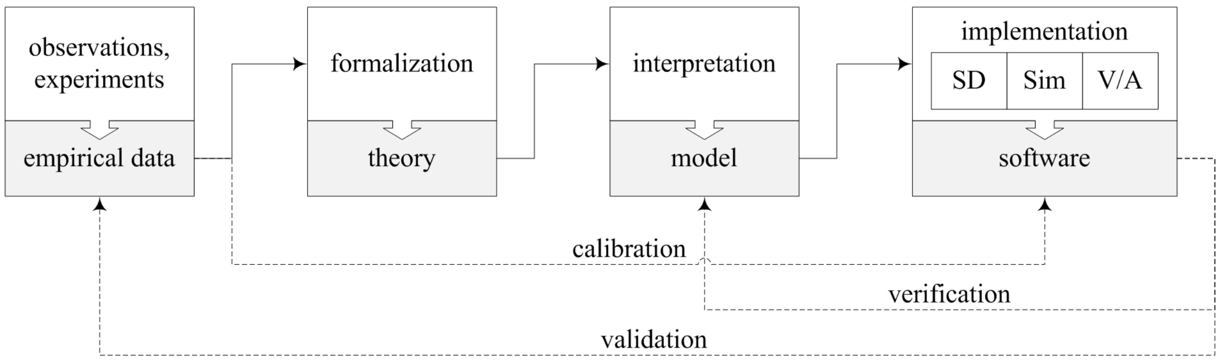


Figure 1.2: Development process of simulation software

Empirical data is gathered in controlled experiments or observations in practice using manual or infrared-detector counting, GPS measurements, questionnaires or video analysis [Daa04]. Based on that data a theory is formulated that identifies parameters and expresses relationships between parameters and various aspects of pedestrian behaviour. In the next step, a model defining a mathematical representation of the simulation system is developed. Finally, the simulation model, together with appropriate input, visualization and analysis functionality according to Figure 1.1 are implemented as a computer program. There are two paths constituting feedback loops in the development process, making it a circular process. Verification means checking whether the software is correctly implementing the model, i.e. whether the results output by the program are consistent with theoretical results of the model. Validation compares the simulation output to empirical data and therefore tests the realism of the simulation. Negative verification requires implementation fixes, whereas unsatisfactory validation results might lead to refinements at earlier stages of the development process, affecting all subsequent steps. Calibration addresses the need of a simulation implementation to have a certain set of fundamental parameter values adjusted to a given problem scope, usually based on empirical data.

1.3 Related Work - currently available Software Solutions

There are numerous software packages available that address several use cases of pedestrian simulation. Comparisons and evaluations can be found in [Klü03], [SHST07], [Rai04] and [RSK07]. Examples are

- ASERI, based on research of Volker Schneider, by Integrierte Sicherheits-Technik GmbH (Germany) [IST]
- CAST Terminal, an airport-centered, and CAST Pedestrian, a general-purpose simulation environment, joint-developed by the Eurocontrol Experimental Centre (France) [EEC] and

the Airport Research Center GmbH (Germany) [ARC]

- Legion, various software solutions and consulting services offered by Legion International Ltd. (UK) [Leg]
- the product line of Massive Software (US) [Mas]
- Myriad II, based on research of Keith Still [Sti00], by Crowd Dynamics Ltd. (UK) [Cro]
- NOMAD and SimPed, based on research of Winnie Daamen, Serge Hoogendoorn and Piet Bovy [Daa02] [HB04], developed at the Delft University of Technology (Netherlands) [DH03]
- PedGo, based on research of Hubert Klüpfel and Tim Meyer-König [KSMK05], by TraffGo HT GmbH (Germany) [Tra]
- PEDROUTE, developed out of PAXPORT for airport terminals but no longer marketed, by the Halcrow Group (UK) [Hal]
- SIMULEX, based on research of Peter Thompson [TM95a], part of the VE-PRO application suite by Integrated Environmental Solutions Limited (UK) [IES]
- ShopSim [Sho], dedicated to shopping street and retail area scenarios, and SimWalk [Simb], a general purpose simulation software, both developed by Savannah Simulations (Switzerland) [Sav]
- STEPS by the Mott McDonald Group (UK) [Mac]
- Urban Analytics Framework (UAF) [UAF] by Quadstone Paramics (UK) [Qua]
- VISSIM [VIS], a multi-modal simulation software based on Dirk Helbing's social forces model [HM95], developed by PTV Planung Transport Verkehr AG, (Germany) [PTV]

Further examination of related work regarding mathematical models is given by Chapter 2.

1.4 Problem Statement - Unified Tooling

Table 1.1 relates the products listed in Section 1.3 to the functional structure of Figure 1.1 according to available vendor information and literature.

It shows that all three subsystems are implemented by all of the solutions listed, either by a single integrated application, or by a set of collaborating applications. Data exchange between subsystems happens directly in runtime memory in the case of integrated programs, and via file system using proprietary formats in the case of multiple tools. Some of the examples provide additional flexibility for visualization and analysis by enabling exports of video-, 3d-rendering or spreadsheet-data. However, scenario definition and simulation are tightly coupled in every

Product	Scenario Definition	Simulation	Visualization/ Analysis
ASERI	fully integrated		
CAST	fully integrated		
Legion	Legion Model Builder	Legion Simulator	Legion Analyzer Legion Viewer Legion 3D
Massive Jet	fully integrated		
Myriad	fully integrated		
SimPed	SimInput, SimControl	SimPed	SimArchive SimAnimation SimAnalysis
PedGo	PedGo Editor	PedGo Sim	PedGo Viewer
PEDROUTE	Network Builder	Simulation Module	Graphics Module Analysis Module
SIMULEX (VE-PRO)	ModellT module	SIMULEX module	
SimWalk	fully integrated		
ShopSim	fully integrated		
STEPS	fully integrated		
UAF	fully integrated		
VISSIM	fully integrated + standalone Viewer		

Table 1.1: Functional structure of available software solutions

example. A given simulation implementation requires its input data to be generated by its corresponding scenario definition implementation. This system architecture, as depicted in Figure 1.3, proves adequate for end-user oriented software, as the user expects and feels more comfortable with a homogeneous solution optimized for the provided functional range.

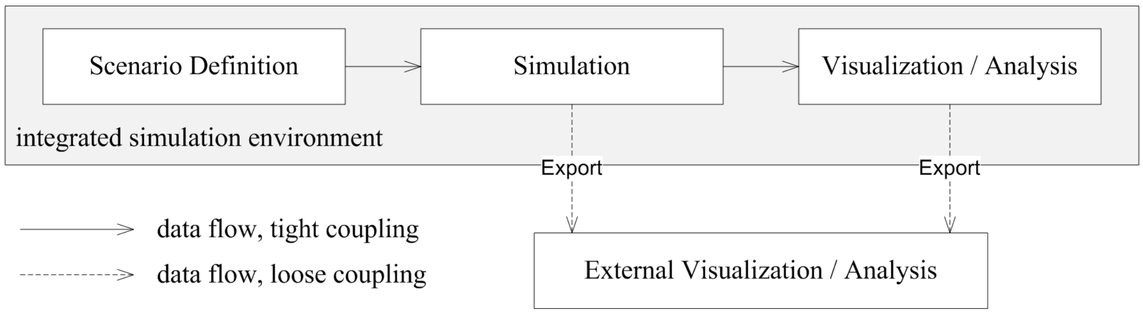

Scenario Definition
Simulation
Visualization / Analysis
integrated simulation environment
Export
Export
data flow, tight coupling
data flow, loose coupling
External Visualization / Analysis

Figure 1.3: Integrated architecture of simulation software

For this thesis, a different approach is taken. The central idea is to decouple scenario definition from simulation as shown in Figure 1.4.

In contrast to the integrated architecture, scenario definition is not part of individual implementations, but instead addressed by an external dedicated component. That component can be configured to be applicable for varying simulation implementations, potentially using completely different underlying models, wherefore it is called *unified.* Data exchange between scenario definition and simulation corresponds to some *unified interface.*

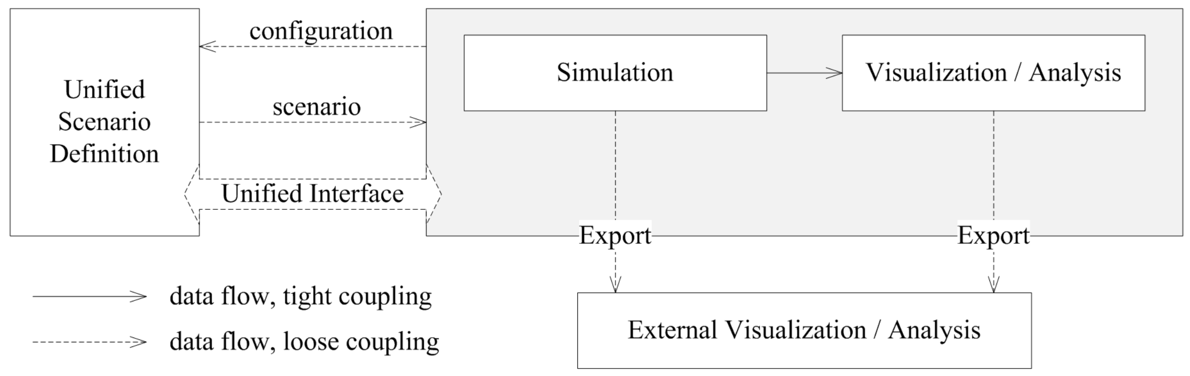

Unified Scenario Definition
configuration
scenario
Unified Interface
Simulation
Visualization / Analysis
Export
Export
External Visualization / Analysis
data flow, tight coupling
data flow, loose coupling

Figure 1.4: Simulation software architecture with unified tooling

This architecture clearly comes with the cost of additional complexity related to the configurability of the scenario definition and the uniformity of the interface, but for benefits from three perspectives. First and as the main motivation, this approach helps modellers and simulation programmers to focus on their main concerns, the design and implementation of simulation models. The unified scenario definition tool is a means to create input data already at an early experimental and prototyping stage, and can easily be adopted to simulation model improvements via configuration. Figure 1.5 shows how simulation tool development is simplified by replacing the implementation of the scenario definition component by much less laborious configuration of the unified implementation in the development process of Figure 1.2.

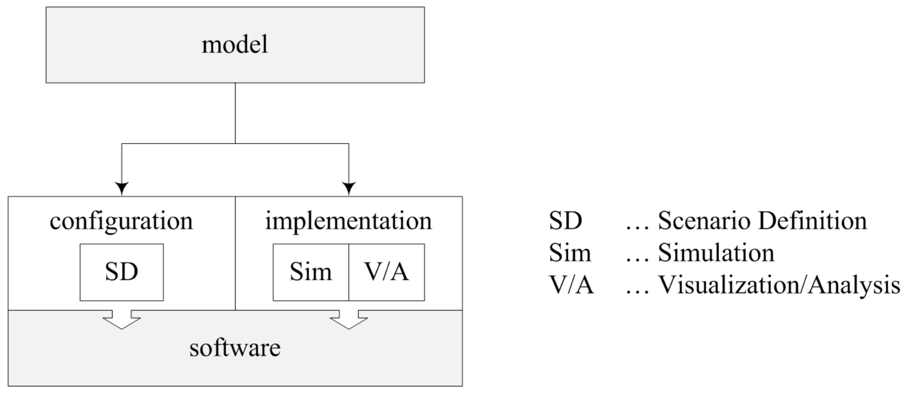


Figure 1.5: Effects of unified scenario definition on the development process of simulation software

Second, from the researcher's perspective, the possibility to define test cases and key scenarios once and let them be processed by any compatible simulation implementation is a means to efficiently perform comparisons and evaluations.

Third, from the point of view of end users, unified scenario definition allows for easier changing between simulators, and given compatible configurations even simultaneaus use of multiple simulators with a single scenario definition. The user is therefore designing simulation scenarios in a more reusable way.

To the knowledge of the author, at the time of writing no implementation of this approach, specialized on pedestian simulation, is available. This thesis is a study of the development of

1. a specification of a unified interface, the *Generic Pedestrian Simulation Interface (GPSI)* and

2. a unified scenario definition software tool to create data corresponding to that interface, called the *Scenario Builder*.

In the following, the conjunction of the interface and the tool is referenced to as the *unified tooling framework*, or shortly the *framework*.

1.5 Methodology

As stated above, this work is broken down into the development of the interface specification and of the software tool.

1.5.1 Interface Specification Methodology

The development of the GPSI is based on a literature study on pedestrian behaviour modelling and simulation. The approach is to use existing findings in pedestrian simulation as a basis for the concepts modelled by the generic interface, and achieve support for future and novel simulation models by integrating some degrees of freedom. The result of the literature study is a collection of input data requirements of the examined simulation models.

In the next step, a generic domain model for pedestrian simulation scenarios is constructed as an abstraction of an appropiate range of input data structures provided by literature, and noted using UML class diagrams. The critical requirement for unified scenario definition to meet the motivation of Section 1.4 is adaptability to concrete modelling approaches. The range of possible simulation models supported by the framework is directly related to the degree of abstraction of the generic domain model. A higher abstraction level in turn increases the complexity of the configuration task (Figure 1.5) and likely decreases usability of the scenario definition tool. Balancing these attributes and finding an appropriate compromise is one important aspect of this step.

Finally, an interface definition technology is chosen and the interface specification is formalized as an implementation of the generic domain model.

1.5.2 Tool Development and Validation Methodology

The Scenario Builder tool is developed following an incremental, object-oriented software development process. Regular analysis, review and evaluation sessions were held with a researchers team at the AIT[1]. Project documentation is based on UML, textual constraints and rationals, and generated code documentation based on source annotations. The implementation consideres recent open-source frameworks and libraries. Validation of the tooling framework is given by considerations of its adoption to subsequent projects and workflow examples.

[1]AIT Austrian Institute of Technology GmbH, TECHbase Vienna, 1210 Wien

2 Pedestrian Modelling

2.1 Application of Pedestrian Models

Pedestrian modelling has a wide area of application, including the following fields.

- Architecture, Facility Planning and Physical Security
  Computer-aided simulation is a tool to estimate the effects of design decisions for buildings and open space. It can be used to examine how a given architecture affects typical situations and how it meets certain expectations and requirements. As thousands of visitors per day were expected, planners and designers of the World Trade Center Memorial site utilized simulation software to visualize and measure functionality of the site from a pedestrian's perspective and to evaluate the site's physical design [MVCA08].

  The "Physical Security" domain of the *Certified Information Systems Security Professional (CISSP)* certification [CIS] addresses the importance of environmental design to information security. The concepts of *Crime Prevention Through Environmental Design (CPTED)* provide guidelines to reduce crime by effects of the physical environment on social behaviour [Har07], for example by avoiding optical barriers to doors of secured areas to make intruders feel less comfortable in attempting to break in. Especially the *Natural Access Control* strategy is capable of being supported by pedestrian simulation tools.

  Another class of use cases that draws particular attention to pedestrian simulation in spatial design is emergency analysis [HFMV02]. Evaluations in this field are typically based on factors like crowd density and egress times, which are appropriate to be measured in simulated environments, as experimental investigations would usually be very costly and sometimes dangerous for human participants and numerical calculation models are less pratical for complex scenarios. [TM95a] presents a simulation model for evacuation analysis which is compared to standard calculation methods of evacuation performance by [TM95b], and to real world evacuation data by [OR01]. Commercial Pedestrian simulation software was used to improve renovating plans and the coordination of serving and security staff of the main venue of the Beijing Olympic Games in 2008, the Beijing National Stadium, with respect to security issues of pedestrian assembling and evacuation [ZWS08]. [Klü03] compares simulation results to evacuation exercises of passenger ships, a movie theater and a primary school.

- Traffic Control and Urban Design
  In urban infrastructure design, multimodal mobility is a crucial issue [Nob07]. Simulation

of the interaction of pedestrian and vehicular traffic provides a means to improve efficiency and security of public transport services. The authors of [BSB07] used pedestrian simulation to develop and evaluate decision rules for an entry restriction system of a Viennese surface subway station attached to a soccer stadium. [HD04] shows how pedestrian simulation is used as an assessment tool supporting the decision for a new access gate system for three railway stations in Lisbon. [Daa04] provides case studies of the Rotterdam Central Station and the ferry-terminals in Vlissingen and Breskens. [KR07] describes the simulation and comparison of two layout variants of the SBB railway station in Bern and makes a prediction of the situation in 2030 based on assumptions of increasing passenger amount, an additional track and another timetable.

- Retail Industry and Retail Policy
  [Tim04] states three fields of business-related interest for pedestrian modelling.
  - Applied to individual retail stores, spatial analysis can support marketing considerations.
  - At the level of shopping centers or shopping areas, behavioral models of shopping pedestrians address the optimization of customer flows and the assessment of market potentials of retail locations. [BT86] tests a model for multi-purpose shopping trips against empirical data of the city center of Maastricht. [SLR07] presents simulations of grocery shopping activities of the region of Umeå (Sweden) and validates the underlying model using gathered population and store data.
  - Pedestrian simulation proves useful in the development of governmental retail policies, for example to vitalize downtown areas.
- Entertainment
  Concerning cinematic productions and computer games, pedestrian modelling occurs especially in terms of crowd simulation. Examples are the battlefield animations of the "Lord of the Rings" movie trilogy [ASDB08] and virtual characters in the game "Splinter Cell" [Zam07].

2.2 Classification of Pedestrian Models

[Klü03], [Daa04], [Kre07] and [SKK$^+$08] identify similar classification categories of pedestrian models as summarized in Table 2.1. Accordant categories are noted in the same row respectively.

[Daa04] lists the modelling approach as another classification category, which is not included in the table as its characteristic is derived from combinations of other criterions. [TBM08] describe a primary categorization of modelling density versus modelling individuals, which corresponds to the traffic/population representation criterion to distinguish macroscopic and microscopic models in Table 2.1, and a secondary categorization for models of individual behaviour concerning the interaction of pedestrians. Regarding the latter, also a distinction of macroscopic,

[Klü03]	[Daa04]	[Kre07]	[SKK$^+$08]
discrete ↔ continuous	scale of independent variables		discrete ↔ continuous
stochastic ↔ deterministic	uncertainty in the process	stochastic ↔ deterministic	stochastic ↔ deterministic
macroscopic ↔ microscopic	traffic representation	population representation	macroscopic ↔ microscopic
estimation ↔ first principles	type of behavioural rules	population behaviour generation	rule based ↔ force based
numerical ↔ analytical	operationalization		
specific ↔ general	area of application	purpose	
quantitative ↔ qualitative			
		space representation	
			hight ↔ low fidelity

Table 2.1: Comparison of classifications of pedestrian modelling in literature

microscopic and mesoscopic models is made, referring to the level of detail at which the influence of individuals to other individuals is modelled.

In the following, these classifications are combined to a generic classification used for the analysis of input data structures in subsequent sections.

2.2.1 Classification Criterions

Granularity

A basic characterization of simlation models is given by the concept of the modelled objects. Microscopic models define representations of individual pedestrians, each with its own separate state and set of properties. Macroscopic models do not consider single individuals but are based on crowd dynamics, using concepts like flows, densities and averaged velocities. Microscopic models provide more accurate results as they process more detailed information, but with the requirement of higher computational effort. The aggregation of individual parameters into macroscopic concepts reduces numerical load for the downside of information loss. As a compromise in between both classes mesoscopic models have been proposed, that deal with groups of persons rather than single individuals ([FMT01]).

Interaction detail

A sub-classification of microscopic, macroscopic and mesoscopic models is possible for models of microscopic granularity regaring how interaction between pedestrians is modelled, see [TBM08, TM07]. Macroscopic interaction modelling is based on fundamental diagrams, i.e. flow-density relationships. Modelling interaction microscopically incorporates detailled information about the relative position, sensing capabilities and individual properties of pedestrians. Mesoscopic models use some combination of the above concepts to model movement.

Scale types

Variables describing a simulated system are either discrete or continuous. The most important variables for the classification of pedestrian models are time and location, their scale types are fundamentally characterizing a model. A discrete time scale means model calculations and updates are performed at certain time steps, models with continuous time scales are typically purely analytical or use event-based or randomized state updates. Discrete space models simplify location information using grids or graphs, modelling continuous space considers more geometrical detail of the modelled scenario. Other possible model variables are velocity, acceleration, density, attraction, etc.

Determinateness

This category refers to the distinction between deterministic and stochastic models, as some models consider uncertainty of pedestrian dynamics using probabilities and random variables, whereas others are defining only fixed relationships.

Behavioural concepts

As [SKK$^+$08] distinguishes between rule-based and force-based interaction implementations, [Kre07] enumerates artificial-intelligence-based, functional, implicit and rule-based behaviour generation and [Daa04] states individual and collective rules to be classes of behavioural rules, in general a classification category addressing the behavioural concepts of model algorithms can be identified, with a wide range of different characteristics proposed.

Application scope

A pedestrian model is either applicable to only a certain specific problem field (typically evacuation) or it is a general model designed for a wider application area.

Operationalization

The application of pedestrian models happens either analytically by finding solutions of sets of equations or by simulation runs. As previously noted, this work concentrates on simulation models.

Behavioural level

[HB04] introduce a hierarchy of three layers at which pedestrian behaviour can be investigated:

- The *strategic level* addresses activity scheduling, i.e. pertains to which, in which order and where pedestrians perform activities.
- The *tactical level* is concerned with higher-level navigation of pedestrians in a spatial configuration, i.e. algorithms and data structures for choosing and describing paths, commonly referred to as *route choice* models.
- The *operational level* is where walking and interaction bahaviour of pedestrians is actually modelled.

This criterion can be used to observe which levels are covered by a particular model.

2.3 Model Analysis

At an abstract level, simulation models are structured as shown in Figure 2.1.

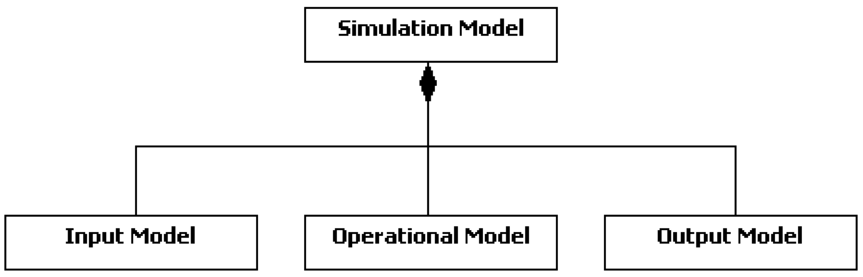


Figure 2.1: Metamodel of pedestrian simulation

The way real-world situations are presented within the model is defined by the *Input Model*, likewise the *Output Model* is concerned with the structure of output data produced by the simulation. The actual interpretation and processing of input and the computation of respective output is defined by the *Operational Model*. Model definitions usually do not explicitly identify their realizations of these three model parts, but rather describe the model as one formal system, still implicitly defining input, operation and output. Therefore this structuring is applicable to all investigated models, and for the purpose of a unification of simulation scenario definition, basically the input model part is relevant.

In this section, representative simulation models of all classes for each classification criterion of Subsection 2.2.1 are reviewed, with particular regard to the structure of the required input data. Thereby, a collection of input model characteristics is derived from the classification of simulation models. Granularity is used for primary ordering as the characterization as macroscopic, microscopic or mesoscopic model is apparently the most prominent classification attribute. The selection of models is not intended to be exhaustive, but to enable an examination of the essential distinctive characteristics of input models.

2.3.1 Macroscopic Models

Macroscopic models have mainly been developed in early pedestrian studies and emerged from traffic flow theory [HCM85]. They are typically based on analogies with gas and fluid dynamics and therefore formulated as partial differential equations, characterising them as rather analytical theories requiring difficult numerical solutions. The continuum model of [Hug02] is discussed as one representative of this kind of models, as an efficient algorithmical application method for it is given by [HWZ$^+$09]. Other examples are the studies of [Hel92].

Dynamic Continuum Model for Pedestian Flow

[Hug02] presents a theory describing the motion of large, goal-directed crowds based on differential equations, and applies it to an example situation of pilgrims moving over the Jamarat Bridge near Mecca. Crowds are treated as a continuum, pedestrian flow is described by its density and velocity. Space and time are modelled by continuous independent variables.

Model review. The basic relationship of flow, density and velocity is given analogously to fluid mechanics and other physical systems by interpreting the number of pedestrians in a crowd as a conserved quantity requiring the continuity equation to hold,

$$\frac{\partial \rho(x,y,t)}{\partial t} + \nabla \cdot \rho(x,y,t)\vec{\nu}(x,y,t) = 0, \tag{2.1}$$

where ρ is the pedestrian density and ν is the pedestrian flow velocity, each at location (x, y) and time t, i.e. the temporal change of pedestrian density equals the spatial change of the time rate of pedestrians flowing through a unit area.

Further characteristics of pedestrian motion are modelled based on three hypotheses, each expressed as an additional equation. In the following, vector notation is used instead of the original component notation and the walking direction unit vector is therefore noted explicit as $\hat{\delta}$ instead of its components (the partial derivatives of the potential function).

1. Walking speed is a function of density, i.e. given the unit vector of the walking direction $\hat{\delta}(x,y,t)$,
$$\vec{\nu}(x,y,t) = f(\rho(x,y,t))\hat{\delta}(x,y,t). \tag{2.2}$$
$f(\rho)$ corresponds to the relationship that is frequently referred to as the *fundamental diagram* in literature [Daa04, DHB05]. There is no widely accepted agreement on the form of the function, apparently it depends on factors like the psychological and physiological state of pedestrians and local conditions. In his numerical examples, [Hug02] uses an alternative form of a function for vehicular traffic proposed by [Gre35].

2. The movement directions of all pedestrians in a crowd are governed by a potential field $\phi(x, y, t)$. At a given location, pedestrians move in the direction of decreasing potential, perpendicular to the isopotential curve of ϕ at that location, thus the unit vector of the walking direction introduced in hypothesis 1 is given as

$$\hat{\delta}(x, y, t) = \frac{-\nabla\phi(x, y, t)}{||\nabla\phi(x, y, t)||}. \tag{2.3}$$

3. Pedestrians avoid extremely high densities, modelled by a tempering function $g(\rho)$, such that

$$\frac{1}{||\nabla\phi(x, y, t)||} = g(\rho(x, y, t)) \cdot ||\vec{\nu}(x, y, t)||. \tag{2.4}$$

Combining Equations 2.1 - 2.4 in component notation yields a partial differential equation system of Equations 2.5 and 2.6 ([Hug02]), of which numerical solutions for $\phi(x, y, t)$ and $\rho(x, y, t)$ describe pedestrian flow in particular situations:

$$-\frac{\partial\rho}{\partial t} + \frac{\partial}{\partial x}\left(\rho g(\rho) f^2(\rho)\frac{\partial\phi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\rho g(\rho) f^2(\rho)\frac{\partial\phi}{\partial y}\right) = 0 \tag{2.5}$$

$$g(\rho) f(\rho) = \frac{1}{\sqrt{(\frac{\partial\phi}{\partial x})^2 + (\frac{\partial\phi}{\partial y})^2}}, \tag{2.6}$$

Subsequently, the model is extended to support multiple pedestrian types, differing in their respective destinations (and therefore potential fields ϕ_i) and velocity functions $f_i(\rho)$.

Input model analysis. Numerical solutions of Equations 2.5 and 2.6 require boundary conditions, which are actually representing geometry and pedestrian appearance of a given scenario. [Hug02] proposes any particular situation to be defined by its closed boundaries, entries and exits in continuous space. Closed boundaries are specified by defining the normal derivative of the potential, ϕ, to be zero at boundary locations, entrances are modelled by specifying pedestrian density, ρ, at entrance locations. At exits, ϕ or ϕ_i for multiple pedestrian types respectively has to be set to zero.

[HWZ$^+$09] describes an efficient solution algorithm for the model equations that is based on discretization of space and time. A uniformly spaced mesh is used to encode the geometry of a scenario, aligned with the boundary of the domain such that walls, entrances and exits are located at midpoints between two grid points. Regarding the input model, the arrangement and the grid size, hence the resolution, of that mesh are required as explicit input information.

2.3.2 Microscopic Models

Social Forces Model

[HM95] suggest the concept of "social forces" governing the motion of pedestrians, i.e. abstract vectorial quantities in continuous space representing the effect of the environment on pedestrians. Acceleration and deceleration of pedestrian movement are modelled as reactions to that forces based on utility maximization. The [VIS] software tool is based on this model.

Model review. Four effects are considered in the formulation of [HM95]:

- A given pedestrian wants to reach a destination area on the shortest possible path walking at a certain desired speed. The path is modelled as a polygon with the current location as starting point and the nearest point of the destination area as end point. A deviation of the current walking speed from the desired speed (in direction or magnitude) causes an acceleration to reduce the deviation.
- Pedestrians want to keep a certain velocity-dependent distance from each other. This effect is modelled by monotonic decreasing repulsive potentials originated by each pedestrian. The potential is assumed to have elliptical equipotential lines directed into the walking direction to reflect the consideration of space required for the next step by other pedestrians.
- Pedestrians want to keep a certain distance from borders of walls and obstacles, modelled by monotonic decreasing repulsive potentials originated by the nearest points of each border.
- Objects or other persons can attract pedestrians, possibly with decreasing interest for the attraction with time. These attractive effects are modelled by monotonic increasing attractive potentials, but not taken into account in consequent studies for simplicity.
- The above described attractive and repulsive effects are weakened by a constant factor if they are originated outside an assumed angle of sight to model directed perception.

[HFV00] and [HFMV02] adapt the model to better reflect evacuation and panic situations by redefining the repulsion forces of pedestrians and obstacles, and establish consistency to Newton's 2nd law of motion by considering the mass of pedestrians. The acceleration of a pedestrian i is given by ([HFV00])

$$m_i \frac{d\vec{v}_i}{dt} = m_i \frac{v_i^0(t)\vec{e}_i^0(t) - \vec{v}_i(t)}{\tau_i} + \sum_{j(\neq i)} \vec{f}_{ij} + \sum_W \vec{f}_{iW}, \tag{2.7}$$

where m_i is the mass, $\vec{v_i}$ the actual velocity, v_i^0 the desired speed and $\vec{e_i^0}$ the desired direction of pedestrian i. The relaxation time τ_i denotes how fast the actual velocity is adapted to the desired velocity ([HM95]). $\vec{f_{ij}}$ and $\vec{f_{iW}}$ are the repulsive forces emanating from other pedestrians and obstructions respectively. Without refinement for this study, f_{ij} is the vector sum of three components, f_{social}, describing the psychological tendency of two pedestrians to keep a distance between each other, $f_{pushing}$, a physical force describing body compression, and $f_{sliding}$, describing sliding friction impeding relative tangential motion.

[LKF05] propose further modifications to avoid spatial "overlapping" of simulated pedestrians and to enable a recalibration of parameters to more intuitive values. The authors account $||f_{social}||$ to be dependent on the density of the crowd and the orientation of pedestrians (face-to-face or face-to-back). Also, a software implementation of the model is presented, using a relatively time-efficient explicit numerical integration algorithm. [PGM09] also approach the issue of "overlapping".

[HFMV02] compare simulation results of different situations to empirically observed collective phenomena and spatio-temporal patterns like lane formation, oscillations at bottlenecks, crossing behaviour, "freezing by heating", clogging or the "faster is slower" effect. The authors claim that the proposed model, despite its relative simplicity, provides high realism and performs well in reproducing the empirical findings. [PGM09] provide further comparisons to published experimental data and state their modifications to make the model valid for the description of pedestrian dynamics in normal conditions. The simulated effects are not results of explicit model rules or configuration, but actually consequences of dynamic interaction between simulated pedestrians. [HBJW05] report on practical applications of the model and simulation experiments used to develop suggestions for improvements of pedestrian facility designs.

Input model analysis. It is conspicuous that [LKF05] list and suggests values for a total of 24 parameters used to calibrate the model. Some values, namely the maximum attraction to an exit A_{exit} and the "unsqueezing force" f_{OE} are derived from other parameter values. The face-to-face social repulsive magnitude F, the high-density correlation factor K_1 and the preferred isolated speed v_0 are varied in experimental sets of simulation runs.

The model of [HM95] and its variations define equations with vector variables describing locations of pedestrians, potentials and path points in continuous space, therefore vector-based geometry information is adequate input.

Presented numerical examples are kept simple to focus on isolated collective effects mentioned above. Notably the model lacks path planning capabilities, as it does not describe the calculation of path polygons defining the local targets and therefore the desired walking direction $\vec{e}_i^0$ of a pedestrian i. This means that simulations of scenarios with more complex geometry require an additional route choice layer in turn possibly requiring navigation information as input. [VIS] for example uses static routes defined by the user additionally to the environment plan.

Cellular Automata Models

A large number of model proposals are based on a mathematical theory for modelling dynamical systems called *cellular automata.* Cellular automata were introduced by [Neu66] and comprehensively studied by [Wol94]. In general, a cellular automaton consists of discrete sites, organized as *cells* in a regular *grid* of some dimension, i.e. in a row, a plain lattice or a cube. Each cell holds a discrete state value. Evolution of the system happens in discrete time steps and is determined by local transition rules applied on each cell synchronously. At each update step, the new state of a given cell depends exclusively on its current state and the current states of the cells in its *neighborhood*, whereas different definitions of the neighborhood function are possible and used depending on the modelled system.

[NS92, Nag96] developed a cellular automaton model for road traffic interpreting vehicles as particles moving, i.e. hopping from cell to cell, in a 1-dimensional discrete road string. Applied to pedestrian simulation, cellular automata are usually designed as 2-dimensional grids representing the scenario space. Pedestrians are modelled by moving actors, frequently the formalization of the model uses the agent-based simulation paradigm. [DT02] illustrate how cellular automata and distributed artificial intelligence concepts are combined as a framework for pedestrian simulation. An abstract framework for the development of multi-agent based cellular crowd simulation models is proposed by [BFV07].

Floor Field Model

Model review. In the model proposal of [BKSZ01], interactions of pedestrians with other pedestrians and scenario geometry over long distances (i.e. not within the local neighborhood) are simulated using state variables of cells called *floor fields* in a 2-dimensional cellular automaton. Pedestrians are modelled by particles with simple local transition rules, complex collective effects are reproduced by self-organization. Each cell is occupied by at maximum one particle at each time step. The movement velocity of all particles is fixed to one cell per turn. For each particle, a preferred walking direction is assumed to be known, from which a 3×3 *matrix of preferences* representing the possibilities of a particle to move in the respective direction is calculated based on average velocities and their fluctuations. At an update step, a desired target cell is calculated for all particles in parallel, determined by the preference matrix and the quantities of the floor fields in the neighborhood cells. Conflicts caused by identical target cells of different particles are resolved such that all involved particles except one are required to remain at their current location. Likewise, desired movement to occupied cells results in waiting at the current cell.

Two location-dependent quantities are influencing the desired walking direction as originally given by the matrix of preferences, the *static floor field* and the *dynamic floor field.* The static floor field is time-independent and allows for the specification of the attractiveness of regions of the scenario geometry. The dynamic floor field is time-dependent and constructed in a way to achieve an effect similar to chemotaxis of bacteria or ants following pheromone trails left by other ants ([RFRA07, NSK$^+$06]). Pedestrians are assumed to follow virtual trails of other pedestrians walking some distance ahead, stored by the dynamic floor field, and diffusing and decaying with time. This models empirically observed behaviour to optimize walking efficiency in crowds, see [KS02, SKK$^+$08] for overviews of collective phenomena of pedestrian crowds. [BKSZ01] present a discrete and a continuous alternative for the formulation of the floor fields.

The realism of simulation results is furthermore improved by switching between two different matrices of preferences for each pedestrian, enabling *happy* or *unhappy mood.* Particles in unhappy mode move less directed, i.e. the matrix of preferences is constructed using greater standard deviations and a reduced velocity. A particle switches to unhappy mode when movement to the desired target cell has been blocked too many times in series, i.e. in situations of high density, and switches back to happy mode when a certain number of unblocked timesteps has been possible. By this means unrealistic clogging in front of obstacles is avoided.

[KS02] examine the characteristics of the model for a simple evacuation scenario and the relation between model parameters and evacuation time. It is shown that the model is capable of reproducing regular as well as panic behaviour by choosing appropriate parameter values. Further Model applications are reported in [BKSZ01]. The authors deem the floor field model to correctly capture the most important collective phenomena of pedestrian dynamics. [KS06] compare simulation results to empirical data gathered at evacuation exercises in a primary school and also find good agreement between both.

Input model analysis. The floor field model is dedicated to the operational layer as it expects the desired walking direction of each particle to be known to calculate the matrix of preferences, therefore an additional navigation or path specification layer is necessary with its respective requirements on input data. The term *species* is used to describe subsets of the simulated crowd with common desired walking directions, multiple species are simlpy possible by assigning different preference matrices. As a possible extension, spatial structures could be modelled by assigning preference matrices to grid cells instead of [or in addition to; note from the author] matrices of particles. The model operates on a space representation as a regular grid which can optionally be periodically closed in one or both paraxial directions. The cell size is expected to correspond to the typical area occupied by one pedestrian. Each cell is either *allowed* (at free space) or *forbidden* (at walls and obstructions), yielding the *obstacle number* used by [KS02]. The static floor field can be used to specify attractive regions of space like emergency exits or shop windows, it is either precalculated and provided to the simulation or deduced from geometry using methods like those of [TM07, TBM08].

Amanda Model

Model review. *Amanda* is a cellular automaton model prototypically presented by [DJT01] and [DT02]. Pedestrians are represented by agents moving in 2- or 3-dimensional space modelled by a lattice of cells. The state of each cell consists of a static portion, indicating the type of the cell as *empty*, *wall* or *decision*, and a dynamic portion given by an occupation flag and the *density* of the cell. *Empty* and *decision* cells can be occupied by at most one agent, whereas *wall* cells represent obstructions and boundaries. Though route choice algorithms are not included in the model proposals, navigation is described to be based on a network of nodes and links overlying the cellular grid with *decision* cells constituting the nodes of that graph-based representation of the simulated environment. Each cell is either occupied or free. The density of a cell is defined as the number of occupied cells in the neighborhood of some fixed radius in relation to the total number of cells in that neighborhood.

In contrast to previously presented models, agents of the Amanda model can occupy more than one cell, thus a smaller cell size and therefore finer spatial resolution are possible. Regarding the temporal dimension it is distinguished between shorter time intervals for model updates, called *time-slice-steps* and longer time intervals for visualization, called *time-steps*. The velocity of agents is variable and defined as the number of cells crossed during a time-step. The time-slice-step is chosen appropriate to the assumed maximum velocity, so that the maximum velocity equals one cell per time-slice-step. Lower velocities are simulated by intermediate waiting steps decreasing the speed average. The rule set applied to each agent at each time-slice-step is kept very simple, though not fully explained. Essentially the walking direction is changed at decision points, if the agent intends to move to an occupied cell, based on its current walking velocity and movement direction, the cells targeted by a sidestep to the left or to the right are checked, if they are also occupied, the agent waits for the next round.

Although not explicitly mentioned, the model seems to be restricted to paraxial navigation, since it is not explained how the desired target cell for the next update step is determined in general given the next target decision point and current walking speed, and the example of [DT02] uses a navigation network graph containing only paraxial links. Also, the description of the rule set does not refine the choice of the target cells for sidestepping. Other aspects left at a rather conceptual level are the influence of density and agent behaviour on walking speed, the treatment of conflicts in parallel updates and the discretization of walking speed. To the knowledge of the author, no validation results based on comparisons to empirical data have been reported.

Subsequent research based on the *Amanda* model concentrated on the strategic level (as explained in Section 2.2.1). [DTdV09, DTdV07, DTdV05] studied activity scheduling and route choice behaviour in shopping environments. A mathematical formalization of decision processes in shopping trips is given and its parameters are estimated using collected data. The scheduling model assigns each pedestrian a time-dependent activity agenda, i.e. a set of planned activities with respective priorities. The agenda is adapted whenever activities are completed, due to time pressure (considering e.g. opening hours of shops) or as a result of unplanned, impulse behaviour, typically induced by advertising. Whether or not an agent schedules the visit of a shop is decided by heuristics based on *perceptual fields*, distinguishing between leisure-

and goal-oriented trip behaviour and taking into account the attractiveness and distance of the shop, the experiences already made with that shop and if the view to the shop is interrupted.

Input model analysis. Though the model is described to support 3-dimensional cellular automaton representations of the scenario, only the 2-dimensional case is further discussed and only 2-dimensional examples are provided. Environment geometry is specified by *wall* cells of the regular lattice. Additionally, the navigation network and the coupling of its nodes to grid cells need to be defined. Concerning activity scheduling and route choice, some decision points in the navigation network need to define parameters like *signalling intensity* and flags indicating the possibility to conduct activities ([DTdV05]). Also, for each individual simulated pedestrian an activity agenda and other behavioural parameters need to be specified.

Route Choice Self-Organization Model

Model review. The proposal of [TBM08] places emphasis on the support of dynamic environments, i.e. spatial configurations with dynamic elements like doors that can be opened or closed or elevators that might be temporarily out of order. The model is based on a gereralization of the cellular automata interpretation of previous examples. Pedestrians are modelled by individual agents moving in a regular lattice. A cell of the lattice can be occupied by any number of agents up to some maximum density. The movement rules of agents generate a more sophisticated path finding behaviour, referred to as *route-choice self organization* (RCSO), than that of simple static or shortest route algorithms.

Infrastructure geometry information is encoded into a *binary matrix*, that is an assignment of a value representing *free space* or *obstruction* to every cell of the cellular automaton lattice. Navigation information is provided by a *navigation matrix*, generated using the concept of *sink propagation value (SPV)* explained below. The navigation matrix enables agents to determine the direction to their target by examining only their local neighborhood. Dynamic environments are supported by precomputing a static (*initial*) navigation matrix ($Q^{initial}$) containing "default" knowledge about the environment, and interferring it with a dynamically created *current* navigation matrix ($Q^{current}$), using a smoothing parameter γ with $0 \leq \gamma \leq 1$ to represent the degree of information of agents about the current situation:

$$Q = (1 - \gamma)Q^{initial} + \gamma Q^{current} \tag{2.8}$$

The time required by an agent to walk across a cell is determined macroscopically by a speed-density relationship function. Simulation proceeds in discrete time steps, at which another matrix is calculated to represent the probability of each cell to be entered by an agent as a function of the cell density, reflecting the preference of lower densities in the walking path. The next cell to enter is decided by calculating the maximum entry in the Moore neighborhood of the normalized entrywise product of the binary matrix, the navigation matrix and the interaction matrix.

The model is extended by [TBM08], where space representation is generalized to network graphs, understanding the regular lattice of [TM07] as a special case of an undirected graph. Also, different functions for interaction and navigation (i.e. the SPV) are used. The term *basin* is defined as a set of (not necessarily connected) vertices composing the origin (*source basin*), the destination (*sink basin*), or an intermediate location of particular activity (*saddle basin*) of the movement of agents.

The sink propagation value of a cell in [TM07] or a vertex in [TBM08] is basically a measure of the distance to a given agent's sink basin. [TBM08] defines it to be a positive function that is zero at vertices of the sink basin, infinite at vertices from which the sink basin is unreachable and strictly monotonically increasing with increasing shortest distance to the sink basin. The SPV is calculated using smoothing relaxation ([Win92]) by [TM07] and using Bellman flooding ([Bel58]) by [TBM08].

Simulation results of [TM07] show realistic patterns for simple dynamic environments, [TBM08] illustrates how route choice of individuals is an output of the simulation trading off distance minimization against congestion avoidance, and provides an egress time analysis of a simple example using the generalized graph-based space representation of the model.

Input model analysis. The model as explained by [TBM08] is basically calibrated using 6 parameters of 3 cumulative distribution functions of the beta-distribution used for the speed-density relationship, the interaction matrix and the sink propagation value. Spatial data is provided using a regular grid of cells that are either free or impassable or by a network graph of linked regions. For the generation of the navigation matrix, the set of cells or vertices constituting the target basin of the simulation population (in the given simple examples a homogeneous crowd heading towards a common target is assumed) needs to be specified.

Queueing Network Model

Model review. [Løv94, Løv95] propose a pedestrian model based on queueing network theory. Simulated space is modelled by a network of nodes and undirected links. Nodes represent rooms (or decision points in building sections in general) and links represent doors (or walkways). Three node types are distinguished - *source nodes*, *transportation nodes* and *destination nodes*. Nodes are assigned a maximum capacity and links an effective width. Pedestrians in the scenario are modelled by seperate entities moving from source nodes to destination nodes, wherefore the model is characterized to be of microscopic granularity.

Time is modelled continuous, the number of persons residing in each node of the network as a function of time is formulated as a stochastic process. The model proposal includes an activity scheduling layer similar to [DTdV09] presented above. Route choice is mentioned to be a necessary step in the movement process and usually based on perceived shortest paths but not specified in detail.

An analytical solution method is presented to calculate performance measures as expected values of stochastic variables. Each pedestrian is interpreted as a seperate customer of a queueing system (see [GSTH08] on queueing theory). Also, the simulation method implemented by the *EVACSIM* software of [DWS$^+$92] is summarized. EVACSIM is dedicated to evacuation scenarios. Time continuity is simulated using event-based updates of individual movement processes of persons. Performance measures are determined using monte carlo analysis.

Input model analysis. Essentially, this model requires the queuing network graph representing the simulation scenario as input, and additional information needed by the applied routing rules about the targets of movement of individual pedestrians. The simulated population is either predefinied by an assignment of pedestrians present at starting time to network nodes or a specification of an external arrival process [or a combination of both, note from the author].

2.3.3 Mesoscopic Models

Mesoscopic pedestrian models have been developed inspired by the mesoscopic approach of road traffic simulation ([FMT01]). Instead of describing behaviour of single pedestrians or the flow of a crowd as a whole, the simulated entities are groups of pedestrians. This paradigm appears to be especially practicable for real-time (also referred to as *online*) simulation for the purpose of predictions of critical states, because modelled pedestrian groups usually correspond to numbers of persons in particular areas such as rooms or train platforms, which in turn are quantities suitable to be injected into simulations based on measurements obtained in real-time by counters, visual estimation or image data analysis.

Models of [HTRS03] and [TA04]

Model review. [HTRS03] developed a mesoscopic discrete time and space modelling concept. The spatial environment is simplified to a network of connected areas, formally represented as a directed graph. Each node of the graph is of type *source*, *sink* or *storage*. Every pedestrian group has its own behavioural rules, is generated at a scource node and leaves the network at a sink node. Storage nodes represent resources in the environment like staircases and doors where groups can be delayed, split and merged. One special type of storage nodes are *stations*, representing "server processes" like check-in or ticket purchase. For each storage node a throughput capacity is specified.

The simulation calculates the quantity of pedestrians in every storage at discrete time steps. Movement time between nodes is calculated based on a distribution function. Route choice is not explained as a part of the model but seems to be addressed by the design of behavioural rules of pedestrian groups, which in turn are not described in detail. This simulation model is reported to be implemented as part of a prototypical real-time decision support system that processes measured data and scheduled events to proactively forecast exceedings of thresholds like maximum pedestrian counts in critical areas.

[TA04] refined this approach. Network nodes are referred to as *components* of type *source*, *sink*, *service station* or *passageway.* Grouping of pedestrians is modelled in terms of two dimensions, logical and physical. *Physical groups* represent person quantities in particular areas as calculated by the model of [HTRS03], whereas *logical groups* correspond to the actual groups of [HTRS03]. In general, at each discrete time step, parts of logical groups are allocated to multiple physical groups and physical groups are composed of parts of logical groups.

Logical groups aggregate individual persons with common intentions and route choice behaviour. The formation of logical groups is driven by an event schedule. Service station components cause simple delays. Movement is essentially modelled by passageway components based on *dynamic route segmentation*, that is a discrete approximation of velocity distributions achieved by a segmentation of passageways into *virtual tracks* of different specific velocity. [TA04] also present a prototypical implementation of an early-warning system based on the proposed mesoscopic model, yet simplifying logical group modelling by using static routing.

Input model analysis. [TA04] provide example input data for a simple scenario using XML, specifying

- named elements, i.e. components of the infrastructure network
- arriving pedestrian quantities for source elements
- length and track speeds for passageways
- links between elements with their weight
- starttime, endtime and stepsize of the simulation
- units and reporting information

Model of [BSB07]

[BSB07] present a pedestrian model and its application to the design of an access control system of a subway station. Macroscopic interaction simulation is used to assess decision rules implemented by a door controller with the intention to avoid overcrowding on the train platform.

Model review. The proposed model uses a representation of space as weighted directed graph. Regions of the scenario space are modelled as vertices R_i. If a region R_j is accessible from a region R_i, there is a link $L_{i,j}$ from R_i to R_j, with an associated weight $w_{i,j}$ and a distance $d_{i,j}$. $w_{i,j}$ equals to the length of the walkable part of the common border of R_i and R_j, $d_{i,j}$ is the shortest walking distance between the centers of gravity of the two regions. Some regions are marked to be *sink regions*, where pedestrians leave the scenario. Sink regions are also the possible targets of the movement of pedestrians. *Source regions* are used to model pedestrians arriving at the scenario.

The model defines equations yielding the crowd density in every region at discrete time steps ΔT. Given a region R_i, the number of direct successors K_i, the set of direct successors $O_i = \{R_{j_k} | k = 1, \dots, K_i\}$ and the set of direct predecessors I_i, at time t, the quantities and equations used by the model can be summarized as in Table 2.2.

P	space covered by one pedestrian, constant, usually $0.1 \leq P \leq 0.26$
$P_{t,i}$	crowd density in R_i as ratio of area covered by pedestrians to total area of R_i
D_i	total distance of R_i to the aimed sink region
$v(P_{t,i}, R_i)$	walking velocity in R_i depending on density and type of the region, given by a fundamental diagram
β	(im)patience factor (arbitrarily set by visual inspection for the case study)
$In_{t,i}$, $Out_{t,i}$	number of persons entering / leaving R_i per time interval, nonzero only at source / sink regions.
$o_{t,i}$	indicator for entry restriction of R_i with $0 \leq o_{t,i} \leq 1$, $o_{t,i} = 0$ if entry to R_i is forbidden, $o_{t,i} = 1$ if entry is fully permitted, $0 < o_{t,i} < 1$ for partial opening of doors
p_{t,i,j_k}	attractiveness of R_{j_k} to be walked to from R_i $$p_{t,i,j_k} = exp(D_{j_k} - D_i - \beta P_{t,j_k}) \quad (2.9)$$
$\tilde{f}_{t,i,j_k}$	one-directional pedestrian flow from R_i to R_{j_k} $$\tilde{f}_{t,i,j_k} = v(P_{t,i}, R_i) w_{i,j_k} P_{t,i} \Delta T \quad (2.10)$$
f_{t,i,j_k}	total pedestrian flow from R_i to R_{j_k} $$f_{t,i,j_k} = \tilde{f}_{t,i,j_k} \frac{p_{t,i,j_k}}{\sum_{m=1}^{K_i} p_{t,i,j_m}} \quad (2.11)$$

Table 2.2: Notation and equations of [BSB07]

The time evolution of crowd density in R_i is given as

$$\tilde{P}_{t+1,i} = P_{t,i} + o_{t,i} \sum_{j \in I_i} f_{t,j,i} - \sum_{j \in O_i} o_{t,j} f_{t,i,j} + (In_{t,i} - Out_{t,i}) P / A_i \tag{2.12}$$

[BSB07] emphasize the strong dependence of simulation results on the choice of the graph representation of the infrastructure. Manually defining the graph is claimed to be too laborious even for simple examples, wherefore software automization is proposed. With the presented approach, the nodes of the graph accord to cells of a regular grid overlaying a vector-based representation of the infrastructure. Various issues arise with the mapping of vector-based plans to regular grids, concerning the choice of grid orientation, placement and size and the treatment of particular regions such as doors or bottlenecks. Therefore a fully automated grid mapping without any user interaction seems not to be reasonable, but instead a semi-automatic workflow is proposed:

1. Import a CAD plan of the infrastructure into a software tool.
2. Postprocess the infrastructure layout to get an adequate level of detail and to add missing information relevant for simulation.
3. Define a coarse partitioning of the infrastructure.
4. For each part of the precedent step define an appropriate grid.
5. Manually refine boundary regions, possibly by including smaller cells.
6. Let the links of the graph be created automatically as given by cell neighborhood.
7. Remove links between neighboring cells "devided" by obstructions represented as lines between them.
8. Add additional links not already created by step 6, such as connections between different floors by elevators or staircases.
9. Identify sink cells.

The showcase scenario is interpreted as a closed-loop control (see [DFT92] on feedback control theory) with the following analogies:

control theory entity	showcase instance
system under control	subway station
system output	number of people on the platform, measured, considering errors
reference value	capacity of the next arriving train
controller	implementation of rules when to open or close entrance doors

Table 2.3: Closed loop control interpretation of the showcase of [BSB07]

The authors explain the integration of the closed-loop control into the graph-based model and present results of simulation experiments using SIMULINK and MATLAB [Sima].

Input model analysis. Spatial information of the proposed model is structured as a directed graph. The links of the graph are assigned a weight (width) and a distance. For each node the type of the corresponding region (as for example *open space*, *upwards* or *downwards leading stair* or *bottleneck*) needs to be specified to enable the calculation of the walking velocity $v(P_{t,i}, R_i)$. Sink regions used for route-choice are identified as a set of nodes within the graph. Furthermore, the inflow and outflow of pedestrians is arranged by specifying $In_{t,i}$ and $Out_{t,i}$, and the function $o_{t,i}$ can be used for temporary entry restrictions. The main calibration parameter is the (im)patience factor β.

2.3.4 Overview

Table 2.4 classifies the models reviewed in this section according to Subsection 2.2.1, illustrating a broad coverage of modelling approaches considered as the basis for the subsequent development of a generalized input data model.

	Dynamic Continuum	Social Forces	Cellular Automata	RCSO	Queuing Network	[HTRS03]	[BSB07]
Granularity	macroscopic	microscopic	microscopic	microscopic	microscopic	mesoscopic	mesoscopic
Interaction	macroscopic	microscopic	microscopic	mesoscopic	mesoscopic	macroscopic	macroscopic
Scales	continuous	continuous	discrete	discrete	discrete	discrete	discrete
Determinate-ness	deterministic	deterministic	stochastic[1]	deterministic	stochastic	deterministic	deterministic
Concepts	fluid dynamics	physical forces	local rules	local rules	queueing theory	density per region	density per region
App. Scope	dedicated	common	common	common	common	common	common
Level	operational	operational	operational	operational and tactical	operational	operational	operational and tactical

Table 2.4: Classification of reviewed models

2.4 Input Model Summary

To conclude the analysis of pedestrian simulation models, the following basic aspects of scenario definitions can be identified:

2.4.1 Environment Geometry

As representation of the simulated space, essentially three alternatives and combinations of these are used by the inspected models:

- Vector-based
  ([Hug02], [HFMV02], [HD04], [Hel92])
- Grid-based
  ([BKSZ01], [DT02], [HWZ$^+$09], [BSB07], [TM07], [BA00])
- Graph-based
  ([TM07], [TBM08], [TA04], [BSB07], path planning)

2.4.2 Environment Semanctics

Any model requires the spatial data to be enhanced by some semantic information. In general, parts of the space represent regions of particular interest, for the reviewed models the following demarcations are necessary:
Dynamic continuum ([Hug02]): entrances and exits need to be identified to define boundary conditions
Social forces ([HM95]): destination areas for path polygons
Floor field ([BKSZ01]): special regions for static floor field, navigation
Amanda ([DT02]): decision cells for navigation and scheduling
RCSO ([TM07, TBM08]): basins for the generation of the SPV
Queueing network ([Løv94]): source nodes and destination nodes
Mesoscopic model of [HTRS03] and [TA04]: node types and attributes
Mesoscopic model of [BSB07]: source and sink regions, region types

2.4.3 Population

The specification of the crowd populating a simulation scenario in general defines

- where pedestrians occur, where they leave the scenario and which intentions are directing their movement,
- how many pedestrians are inserted into the simulation,
- when pedestrians are inserted into the simulation, and

- how pedestrians are characterized.

Most example implementations use fixed numbers of pedestrians that are placed once in the scenario or by strict schedules. It is also obvious to describe the occurence of people using insertion rates and relative frequencies of fractions of the crowd. Some models support inhomogeneous crowds in terms of considering individual behavioural configurations of pedestrians, usually at least to simulate different origins and targets of subsets of the pedestrians within a simulation. For example [BKSZ01] distinguish different *species*, [DT02] use individual *activity agendas and behavioural parameters* and the mesoscopic model of [TA04] organizes pedestrians in logical groups. Most model reviews show a fundamental characterization of movement by predefined origins and destinations, the choice of intermediate targets is the subject of tactical modelling or static routing.

2.4.4 Model Calibration

Depending on the complexity of the relationships defined by a model, some number of constant values or characteristic curves are assumed to calibrate the model. The most common subject of calibration is the fundamental diagram, describing the basic dependence of walking speed on crowd density. Further examples are the (im)patience factor β of [BSB07] or the 24 model parameters of [LKF05].

2.4.5 Reporting Configuration

There are various possible kinds of results generated by simulation models, like animated movie clips, curves of characteristic values, density or trajectory plots and statistical data. In some cases, simulation results are related to particular parts of the scenario, such as the density progression within a given area or the egress time of a given group of pedestrians. Therefore simulation models in general require information about the desired output calculation.

3 Generic Pedestrian Simulation Interface (GPSI)

In this chapter, the insight into the various input models gathered by the foregoing analysis is utilized to construct the first part of the framework, the generic input model, that is an abstract model of pedestrian scenarios. As stated in the methodology section (Subsection 1.5.1), a key criterion here is to find an appropriate abstraction level to cover a broad scope of simulation models and at the same time provide for efficient configuration and tooling support. Subsequently the generic input model is implemented as an XML schema definition.

3.1 Requirements

The attributes deciding the degree of adaptability of the interface as motivated in Section 1.4 can be identified as follows:

- Generality,
  the interface enables data exchange suitable for a wide range of simulation models.
- Stability,
  the interface is intended to remain structurally unchanged despite changes of simulation models and implementations.
- Accessibility,
  the implementation of interface adaption takes a minor effort within the implementation of a simulation model.
- Platform independence,
  the choice of programming environments and operating systems is free to the simulation implementation.

Generality and stability are mainly concerns of the model underlying the interface, whereas accessibility and platform independence are related to technological choices and implementation specifics.

3.2 System Metamodel

Figure 3.1 shows how the generic input model explained in this section is situated in the context of unified tooling and pedestrian simulation.

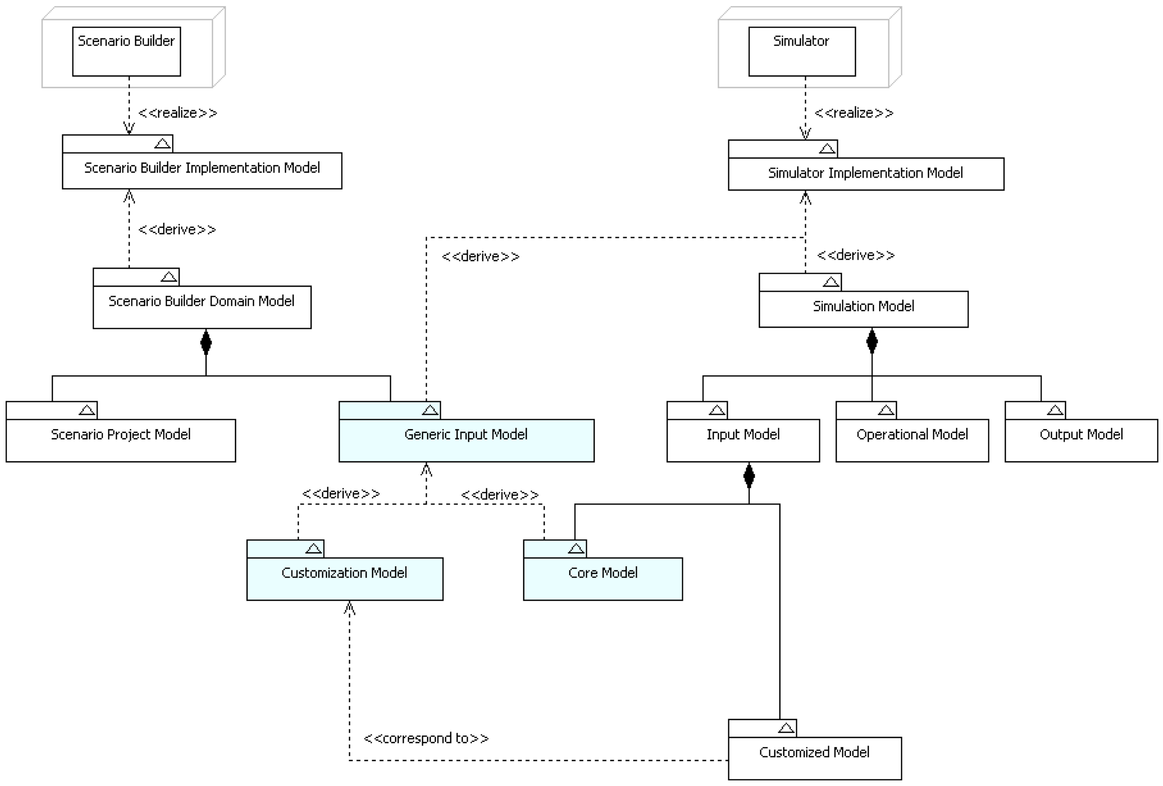


Figure 3.1: Context of the generic input model

The *Simulation Model* on the right side depicts any concrete pedestrian model, the structuring of *Input Model*, *Operational Model* and *Output Model* is explained in Section 2.3. The *Simulation Model* represents a theory, provided either by mathematical formalization or possibly in a more informal way, whereas the *Simulator Implementation Model* is a representation of that theory using concepts of a particular programming paradigm, such as object-orientation. The *Simulator* constitutes the executable implementation of the *Simulator Implementation Model*, i.e. the simulator program accepting input corresponding to the *Input Model* and producing output corresponding to the *Output Model*.

Scenario Builder stands for the unified scenario definition software tool introduced in Section 1.4, specified by its own implementation model (*Scenario Builder Implementation Model*), which in turn is based on some understanding of the problem field, referred to as the *Scenario Builder Domain Model*. That problem field is given by the purpose of the tool to create simulation scenarios using the *Unified Interface* of Figure 1.4, wherefore the domain model consists of the *Generic Input Model* and a model describing the organization of scenario data within the tool, referred to as *Scenario Project Model*.

The emphasis of this presentation lies on the structure of the *Generic Input Model*, namely the distinction of a *Core Model* and a *Customization Model* part. Abstractly spoken, in a given generic input model, there are some characteristics of scenarios that are commonly assumed by all supported simulation models, like for example the fact that "there are one or more types of pedestrians" or "a regular grid has a defined number of columns and rows". All these common concepts constitute the *Core Model*. (Note that *Core Model* and *Customization Model* are terms for abstractions, which exist as concepts but not necessarily with a simple equivalence to parts of the GPSI specification.) In contrast there are peculiarities of several models that are not unified but rather enabled to be expressed using interface customizations. For this purpose, the *Customization Model* part is provided as suitable abstraction of a customizable portion of the input model of supported simulation models (the *Customized Model*).

As mentioned above, a higher degree of abstraction extends the range of supported simulation models but complicates tool support. That abstraction level is determined by the portion of the *Core Model* and the depth of the *Customization Model*. Consider for example the specification of a fixed set of possible types of "service stations" as used by [TA04] with fixed parameters versus the abstract specification "special regions" with customizable parameter lists.

Figure 3.1 points out that the *Customization Model* decides the degree of freedom of concrete models. Variations of input models are possible without any change of the interface and the tool implementation as long as they concern the customizable portion. However, the design of the *Core Model* is critical to the stability of the interface.

Some example concepts of the discussed models are listed in Table 3.1 to support understanding:

model	example concept
Core Model	pedestrian type, infrastructure graph, polygon
Customization Model	parameter type, region type
Customized Model	age of pedestrian, simulation time interval
Simulation Model	potential field, force, attractiveness

Table 3.1: Examples of model concepts

3.3 Model Architecture

3.3.1 Top-level Layering

The generic input model is discussed top-down, starting with the package diagram of Figure 3.2 that shows the base layers of the model.

The *item* package contains entities that represent logical or physical parts of the simulated environment with some semantics for the simulation, and that are typically related to regions in space. The *layout* package addresses the representation of simulation space and is subdivided

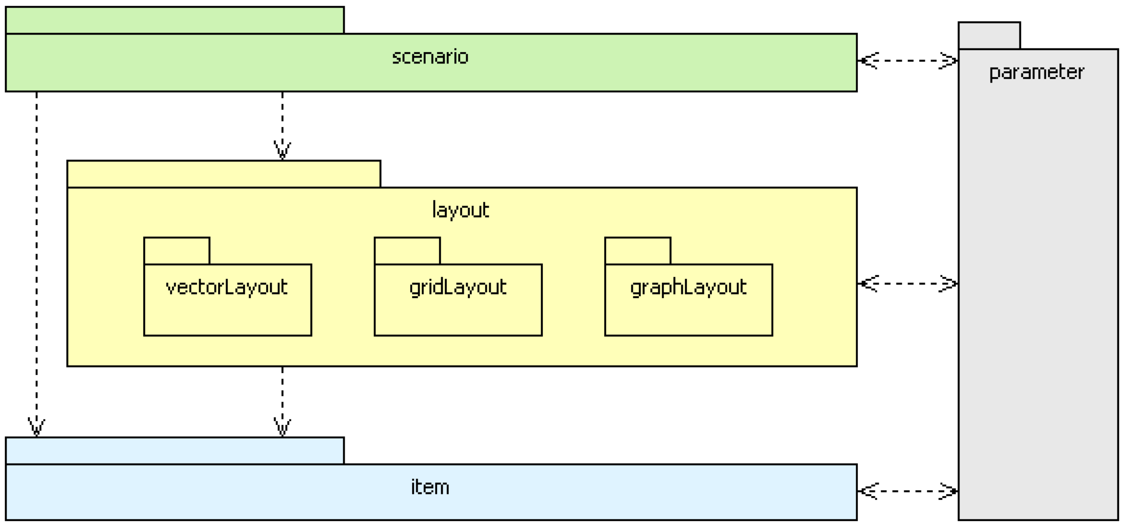


Figure 3.2: Package diagram of the generic input model

into *vectorLayout*, *gridLayout* and *graphLayout*, each subpackage corresponds to the possible types of spatial modelling listed in Subsection 2.4.1. The term "layout" relates to a plan or spatial design in general. Instances of the *item* package are referenced by instances of the *layout* package, which complies with the need for semantic enhancement of spatial information explained in Subsection 2.4.2. Global scenario information such as about population (Subsection 2.4.3), and calibration and configuration data (Subsections 2.4.4 and 2.4.5) is contained in the *scenario* package. As such data can be related to elements of the scenario, for example pedestrian occurrences are defined based on entries and exits, a direct dependency of *scenario* on *item* exists in addition to that on *layout*. The *parameter* package provides means to enhance elements of the model dynamically (at instance level) with attributes and relationships to other model elements. This approach allows for flexible adaptions to data requirements of individual simulation models without changing the static structure of the generic model. The main use of the *parameter* package will be the realization of the *Customization Model* of Section 3.2. Dynamic parameterization is a cross-cutting issue wherefore all other top-level packages depend on it and vice versa. The colors used in Figure 3.2 introduce a color encoding that will susequently be used for member elements of the respective packages.

3.3.2 Core Concepts

Literature analysis (Subsection 2.4.2) shows, that a broadly required type of semantic enhancement of space data is related to navigation, simply put to a specification of entrances, exits and in some cases intermediate locations. For the generic input model, the generalized concept of *basins*, as introduced by [TM07, TBM08] is adopted to represent reference entities for pedestrian movement.

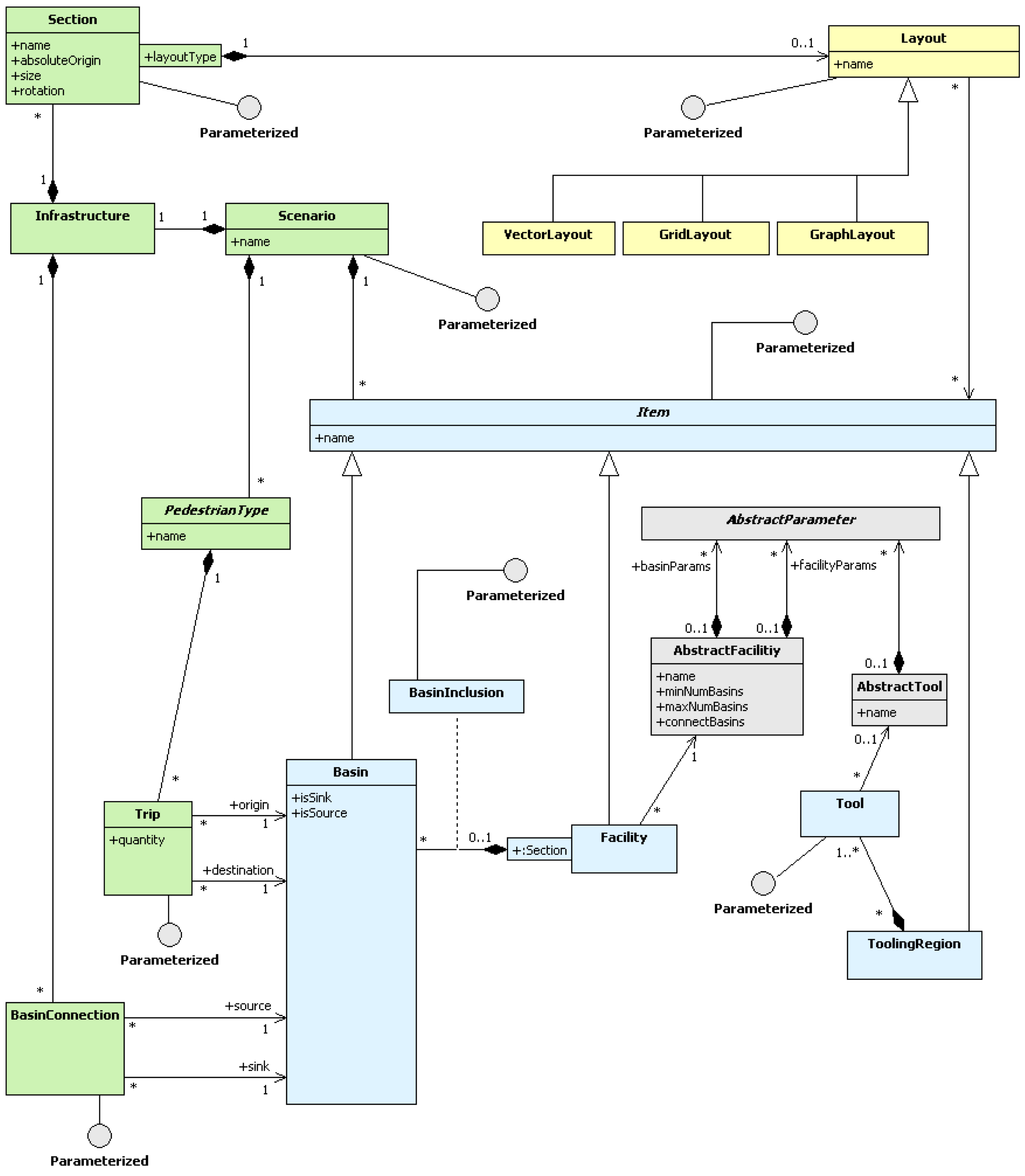


Figure 3.3: Overview class diagram of the generic input model

Another type of demarcation is needed for regions of some particular meaning for the movement behaviour of pedestrians, like special regions of the floor field of [BKSZ01], region types of [BSB07], shops in Amanda [DT02] or service stations of [TA04]. Entities encapsulating the semantics of this kind of simulated objects are subsequently called *facilities*.

Finally, configuration data for reporting purposes is in general related to regions in space, for example if density progressions of selected rooms or transition counts of a certain passageway are requested. Therefore a third type of spatial marking is dedicated to reporting and analysis in the following more generally referred to as *tooling region*.

The main commonality of basins, facilities and tooling regions, therefore generalized as *items*, is that they represent physical or logical entities that can have spatial occurrences in layouts. Note that the assignment to spatial concepts is decoupled from the identity of the respective item, similar to classes that exist within a UML model and are optionally represented in one or more diagrams. This reflects the consistency of semantics of multiple occurrences of a single item.

The proposed generic model considers simulated space to be partitioned into interconnected *sections*. This approach is not directly resulting from the model analysis of Section 2.3 but a feature to increase flexibility and improve support of more complex scenarios, and does not impose any limitations as yet simple scenarios with a single section are also possible instances. Actually, the section concept might match the idea of *submodels* of [BSB07].

Space within sections is specified by *layouts*. Different types of layouts exist for the respective spatial models. An important point in the design of spatial representation is the decision to enable multiple layouts of different types to be specified for a section, which is based on two reasons. First, as stated in Section 1.4, efficient comparison of different models is one of the motivations for the unified tooling approach. Allowing for space definition using multiple layout models enables a single scenario definition to be compatible to a wider range of simulation models. Second, "hybrid" models operating on multiple space representations are not yet strongly represented in literature, but still make an interesting and promising attempt. [GSN04] propose such a model that combines the social forces model with the floor field model and a graph-based model, and accordingly uses multiple types of spatial data in parallel.

All of the reviewed models except the graph-based ones are specified and demonstrated using 2-dimensional space representations, some of them with possible enhancements to the third dimension. For this thesis, a "pseudo-3d" approach is taken. Sections represent parts of the investigated space, like floors of a building or separate areas of a park etc. The spatial arrangement of sections is given by absolute (*world*) coordinates defining the origin and expansion in three dimensions, and the rotation around the z-axis for each section. Layouts of sections use 2-dimensional spatial models. The xy-projection of a section defines a rectangular area that bounds any layouts for that section. Layouts in turn use coordinates that are relative to the containing section's absolute origin. Therfore sections can easily be rearranged without changing any layouts.

Connections between sections as established for example by staircases, elevators or simply open space need endpoint locations within sections to be specified. As the geometry of sections is defined by means of the contained layouts, endpoint locations need to be marked within layouts. Furthermore, as introduced above, multible layouts can be defined for one section, in which case a given connection endpoint needs to be consistently represented in each involved layout. Connection endpoints therefore have the same characteristics as items, and since they are structurally related to navigation and path planning, it is natural to use *basin* entities as endpoints of connections between sections. This in turn requires the spatial representations of a basin to be restricted to a single section, otherwise put every basin is part of one section.

The basins of a scenario and the connections between them constitude a network graph referred to as *infrastructure*. The infrastructure therefore is a graph-based spatial representation of the scenario superimposing the layouts of its sections. In contrast to graph-based layouts of sections, the infrastructure graph is 3-dimensional, as sections are arranged by 3-dimensional, absolute coordinates.

In traffic research, commonly used structures for the specification or estimation of traffic volume are *trip matrices*, or *origin/destination matrices*, shortly *OD-matrices*, see [MSCCJ10, LC07]. OD-matrices define quantities for origin-destination pairs. This concept is adopted for pedestrians by the proposed model. As introduced above, entrances and exits, therefore origins and destinations of pedestrian movement are represented by basins. Basins are nodes in the infrastructure graph, which therefore can be used by navigation models for path planning or by 3-dimensional graph-based models for high-level simulation.

3.3.3 Static Structure

The architecture of Figure 3.2 is refined by the class diagram of Figure 3.3, to formalize the preceding concepts by an object-oriented model.

Scenario is the root container element of the model, i.e. all scenario data is directly or indirectly aggregated by a *Scenario* instance. The scenario's *Infrastructure* contains its spatial information and consists of *Sections* and *BasinConnections*. The population of the scenario is specified by its *PedestrianTypes* and the respective *Trips*. The latter are equivalent to entries of the OD-matrix, that is the total of all *Trips* of a *PedestrianType* builds the OD-matrix of that pedestrian type. Simulations are intended to insert pedestrians of the respective types, interpreting the *quantity* attribute of *Trip* either as absolute or relative value, depending on the model and additional parameterization. A *Layout* corresponds to one of the three spatial models stated in Subsection 2.4.1, therefore it is of one of the types *VectorLayout*, *GridLayout* and *GraphLayout*. Each *Section* can be assigned at maximum one *Layout* per layout type, denoted by the quantifying attribute *layoutType*. The sets of *Items* referenced by each *Layout* of one *Section* can be checked to be identical in order to assert consistent data for the involved simulation models. *BasinConnections* are connections between basins of different sections, i.e. links of the infrastructure graph of the scenario.

Items exist in the context of a *Scenario*. They may be referenced by *Layouts* of different *Sections* and types with the restriction that a given *Basin* may only be referenced by *Layouts* of one and the same *Section*. There are three types of *Items*: *Basins*, *Facilities* and *ToolingRegions*. Each *Basin* is either *source*, *sink* or *both* to specify whether it can be the origin or destination of pedestrian trips. *Facilities* are generic items corresponding to an *AbstractFacility*. An *AbstractFacility* defines the appearance and the structure of a kind of facility. *Facilities* may contain *Basins*. A facility can appear in layouts of different sections (e.g. an elevator). The *BasinInclusion* composition is qualified by a *Section* attribute, to model the restriction of basins to a single section, and that facilities contain different basins per section. *minNumBasins* and *maxNumBasins* define the range of the number of basins contained per section. *connectBasins* indicates whether basins of different sections contained by the facility are connected per default or not. *facilityParams* is the set of *AbstractParameters* (discussed below in Section 3.4.4) that can be bound to the *Facility*, *basinParams* is the set of *AbstractParameters* that can be bound to each placement of a *Basin* within the *Facility*. *ToolingRegions* are assigned a set of *Tools*, each corresponding to one *AbstractTool*, which in turn defines the *AbstractParameters* for the *Tool*. A *Tool* represents a configurable element of report generation, like counters or density meters.

Customization aspects are addressed by members of the *parameter* package, i.e. elements of gray fill color. *AbstractFacilities* and *AbstractTools* are used to dynamically define types of facilities and tools. The *Parameterized* interface implemented by most of the model classes (note, also by *Basin*, *Facility* and *ToolingRegion* via their common parent class *Item*) enables the dynamic association of parameters with model instances, see Section 3.4.4.

3.4 Interface Implementation

In this section, a data format that implements the proposed model, the *Generic Pedestrian Simulation Interface (GPSI)*, is presented. First, the choice of XML and XSD as interface technologies and the involved data formats are explained. Next, the way UML is used for the documentation of XML Schema Definitions in the following is introduced. Finally, the interface is specified in detail.

3.4.1 Technology Choice - XML and XML Schema

The GPSI is an XML-based interface, as XML ([BPSM$^+$08]) is a de-facto standard for electronic data representation and exchange. The strength of XML related to the specification of software interfaces lies in the adaptability to domain-specific needs. Although there is also wide use of XML without associated structural definitions (as summarized by [NM09]), a schema language is required to define document types which associated instance documents can be validated against. Apart from various historical alternatives [LC00, CEM03], the most important standards for XML metadata are the W3C recommendations *Document Type Definition (DTD)*, included in the original XML specification [BPSM+08], and the younger *XML Schema* standard of [FW04, TBMM04, BM04, CEM03]. [BNdB04, BMNS05] show that at an abstract level, i.e. considering expressiveness, XML Schema is actually an extension of DTD, wherefore XML Schema is preferred for the specification of the GPSI. While XML Schema definitions in many existing cases do not exceed the expressive scope of DTDs [BMNS05], the GPSI specification makes extensive use of the distinctive features, especially those related to typing and inheritance due to the object-oriented modelling approach, and to the enhancements of keys and foreign keys.

The interface specification satisfies the implementation-related requirements of Section 3.1, as it is

- accessible, because rich framework or library support is available for any important programming environment to enable efficient implementation of XML-based data input and output, especially for modern object-oriented languages, and
- platform independent, as interoperability is an inherent quality of XML.

3.4.2 Data Formats

Section 1.4 explains the unified interface specified in the following to support data exchange in two directions. Customization and configuration data is provided as input to the unified scenario definition tool and scenario data is generated as output, both conforming to respective parts of the interface specification.

Figure 3.4 shows the software components and data artifacts involved by the concrete realization of the GPSI using XML and XML Schema. The highlighted parts constitute the actual interface specification and realize the equally colored model parts of Figure 3.1.

The interface is implemented by two XML Schema definitions (XSDs). Scenario data, i.e. output of the Scenario Builder and input of simulators, is contained in XML documents conforming to `pedestrianScenario.xsd`. These XML documents, called *scenario files* in the following, are per convention named with the extension `.scen`. Input data of the Scenario Builder concerns model customization and tooling configuration, and defines abstract concepts of scenario descriptions, wherefore that data is provided by XML documents called *metadata files* using the extension `.scen-meta` and the corresponding schema file is called `pedestrianScenarioMetadata.xsd`.

The file `metadataInstance.scen-meta` is one exemplary metadata file, that needs to be provided to the Scenario Builder application in order to design corresponding pedestrian scenarios, `scenarioInstance.scen` is one exemplary scenario file, created as output by the Scenario Builder. `projectInstance.sbp` is a *project file* (more thoroughly discussed in Chapter 4), created and used by the tool to persist working data in an internal (non-XML) format.

Note that there are two dimensions of abstraction. First, XSDs are meta-definitions of XML instance documents, and second, customizable aspects of scenario files are defined on an ab-

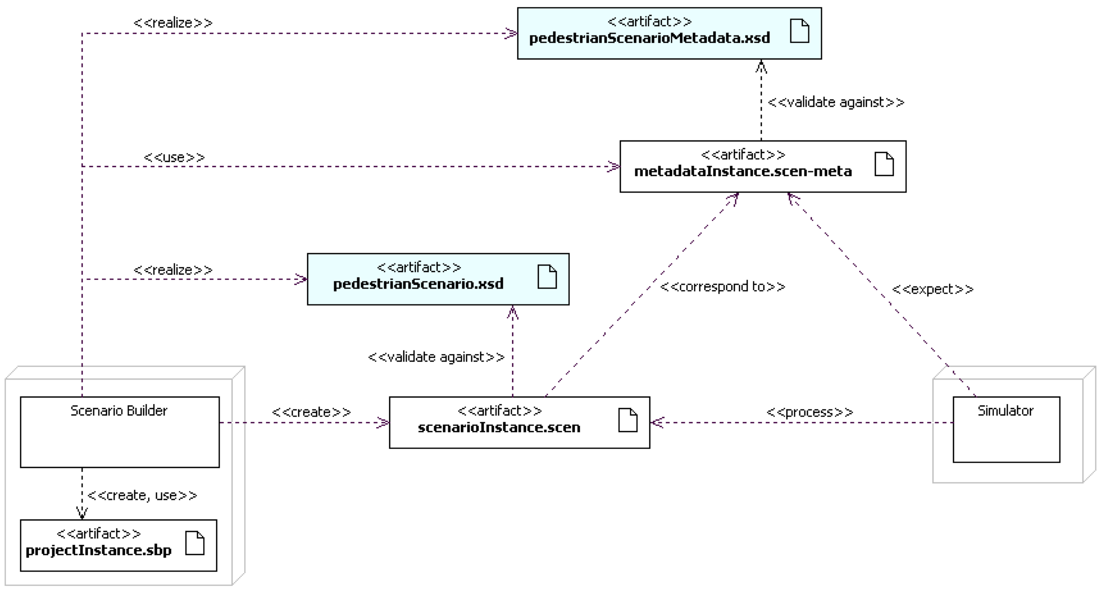


Figure 3.4: Deployment diagram of unified tooling

stract level by metadata files (that are in turn instance documents of an XSD). Because the scenario definition tool statically implements the structure and semantics of the schema definitions, `pedestrianScenario.xsd` and `pedestrianScenarioMetadata.xsd` are hard-wired with the Scenario Builder implementation. Therefore a given version of Scenario Builder supports only the associated version and backwards compatible later versions of the XSDs. At the same time, a wide range of different input requirements of simulation models is supported without changing the tool implementation by providing appropriate metadata files.

3.4.3 UML for the Documentation of XML Schema Definitions

In Subsection 3.4.4, the XSDs of the interface are presented in detail. One disadvantage of XML Schema compared to DTD is that because XSDs are themselves XML documents, they contain a lot of syntactic overhead that makes them rather hard to read for humans (as XML is designed for machine-to-machine communication). For this reason, a graphical, UML-based notation is preferred for the documentation of the interface.

For the mapping of XSD constructs to UML diagram fragments, decisions about the level of detail need to be made. Complete conservation of schema information in UML representations enables exact reverse engineering, but results in verbose and complicated diagrams. The approach of [SPKI04] is to waive such roundtrip support and instead focus on the clarity of UML diagrams and use existing UML constructs for simplification and better understanding. These principles also apply for this chapter, as the emphasis lies on clear documentation and additional fully detailed specification is provided by the XSD listings in the appendix (?), for which in turn cross-linked HTML-based documentation can easily be auto-generated by various available tools.

[SPKI04] define a set of XSD-to-UML transformation rules, which inspire the method used for this thesis, but are not exactly applied. Table 3.2 shows the rules applied to the interface XSDs for the UML documentation of this chapter. These are the main differences to [SPKI04]:

- Local element definitions with complex types are mapped to classes instead of attributes.
- Identity constraints are noted by attribute stereotypes instead of association qualifiers.
- [SPKI04] does not define a mapping for `unique` elements.
- Attribute defaults are not noted.
- Constructs not occurring in the interface implementation are neglected.

Classes that represent anonymous types need artificial names that are generated by capitalizing the corresponding element name. Deviance from this rule is possible to avoid naming conflicts. Additionally, that classes get assigned the stereotype `<<anonymous>>`.

3.4.4 Interface Specification

Section 3.3 explains the top-level architecture and the core concepts of the generic input model. The refinement of that concepts is illustrated accompanying the documentation of the GPSI implementation. In other words, the details of the model are discussed by means of UML diagrams, which at the same time reflect the XML-based implementation by following the rules of Subsection 3.4.3.

Each of the two XSDs introduced in Subsection 3.4.2 uses its own namespace. As XSD namespaces are mapped to UML packages, this results in two packages, `pedestrianScenario` for scenario definitions and `pedestrianScenarioMetadata` for parameterization definitions. The conceptual package structure used for the architecture layering of Section 3.3 is preserved by the coloring of classes in the following diagrams.

Versioning

Given the multidimensional abstraction structure of Subsection 3.4.2, versioning is an important and non-trivial issue. Two possible application scenarios need to be considered regarding the version mechanism of schemas and instance documents. First, each scenario file is constructed based on the parameterization definition of one particular metadata file. As for the generic design, multiple metadata files might exist and as simulation models evolve, also multiple versions of one metadata definition are likely. For a given scenario file, it needs to be possible to identify the corresponding metadata file and its version. Second the framework itself, that is the generic input model might evolve, resulting in new versions of the schema definitions. Therefore both scenario files and metadata files should contain information about the corresponding XSD versions.

XML Schema construct		UML constructs
namespace		package, named after the last path segment of URL-style namespaces
element declaration	global element	class, `<<root>>` stereotype for the root element of the schema
	local element	class, related via composition to the class of the containing element or type
	occurrence constraints (`minOccurs`, `minOccurs`)	multiplicity of composition ends
attribute declaration	local attribute	attribute
	`optional` constraint	multipicity `[0..1]`
	`required` constraint	multipicity `[1]`
simple type definition	global `restriction` simple type	class, generalized by a `built-in` stereotyped class for the base type
	local `list` simple type	attribute with multiplicity
	local `enumeration` simple type	enumeration, drawn overlapping the class of the containing element or type
complex type definition	sequence compositor	composition relationship to classes representing the contained elements
	derived by extension	class generalized by the class for the base type
identity constraint definition	stereotype of the attribute selected by the selector and field elements, the scope given by the containing element is not noted	
	`key` element	`<<key>>` stereotype
	`keyref` element	`<<ref>>` stereotype and dependency relationship
	`unique` element	`<<unique>>` stereotype

Table 3.2: XSD-to-UML mapping

Both cases are addressed by the solution shown by Figure 3.5. XML Schema defines an optional `string`-typed `version` attribute of the root `schema` element. This attribute is used by the XSD implementations of the GPSI and is referred to by `schemaVersion` attributes of the instance document roots (which are in turn specified by the corresponding XSDs). The association of scenario files to metadata files is supported by obligate `name` and `version` attributes of metadata root elements and respective `metadataName` and `metadataVersion` attributes of scenario root elements, that can be matched with the available metadata definitions. All of these attributes are typed as `string`, so that any user-defined naming and versioning policy is supported.

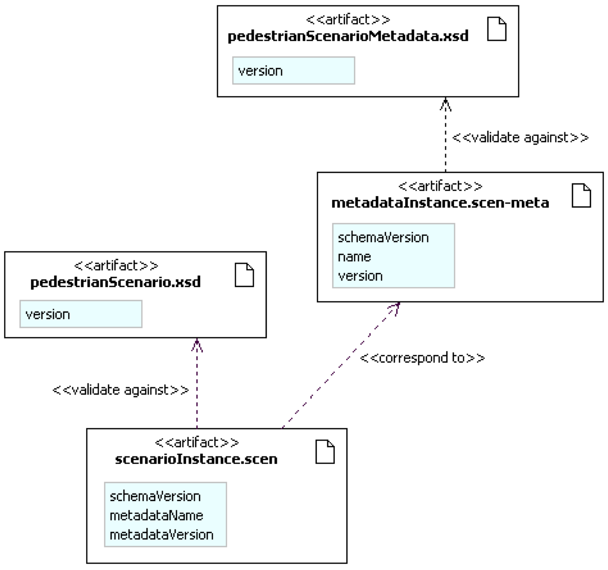


Figure 3.5: Versioning of GPSI

Scenario Definition Schema

This section presents the detailled realization of the design of Figure 3.3 and documents the schema of `pedestrianScenario.xsd`.

Dynamic parameterization. The *Parameterizable* interface of Section 3.3 is implemented by an abstract XSD type called `ParameterHolder`. This is mentioned beforehand though the parameterization part of the schema is discussed in detail in a later paragraph (Section 3.4.4), because many types of the scenario schema use `ParameterHolder` as base type to implement the ability to have parameter values assigned corresponding to the abstract definition of a metadata file.

Root structure. Figure 3.6 is the UML representation of the root `scenario` element. At minimum, a scenario needs to contain `sections`, optional elements are `items`, `basinConnections` and `pedestrianTypes`. `scenario` contains the version attributes explained in Section 3.4.4.

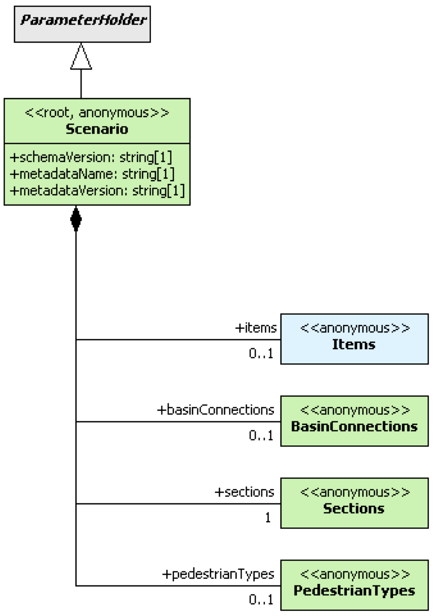


Figure 3.6: Root structure of pedestrianScenario.xsd

Items. Figure 3.7 shows the first two components of the scenario definition, the `items` and the `basinConnections` sets.

Each `item` is a `ParameterHolder` and has an integer identifier as entity key and a unique name. According to Section 3.3, there are three types of items: `Basin`, `Facility` and `ToolingRegion`. Basins refer to the `Section` they belong to by the section's `id` and are typed as `source`, `sink` or `source-sink`, that is both. The `type` of a basin determines its parameterization by the metadata definition. The possible types of facilities are not statically known, but also part of metadata definitions. A `facility` instance refers to its abstract type by the `type` attribute, which holds the unique `name` of a `facility` element in the `pedestrianScenarioMetadata` package (i.e. in the metadata file). The assignment of a `basin` to a `facility` is possible by an intermediate `basinInclusion` element. This indirection enables the basin and its association with a facility to be parameterized independently from each other, as `basinInclusion` is also a subtype of `ParameterHolder`.

Inter-section-connections are realized by directed connections of basins of different sections, which are implemented by `basinConnections`. A `BasinConnection` contains a `sourceRef` attribute referring to the `id` of the source `basin` and a `sinkRef` attribute for the sink `basin` of the connection. .

Sections. The partitioning of simulation space into sections is explained in Section 3.3. The `sections` element of a `scenario` contains one `section` element with a key identifier and a unique

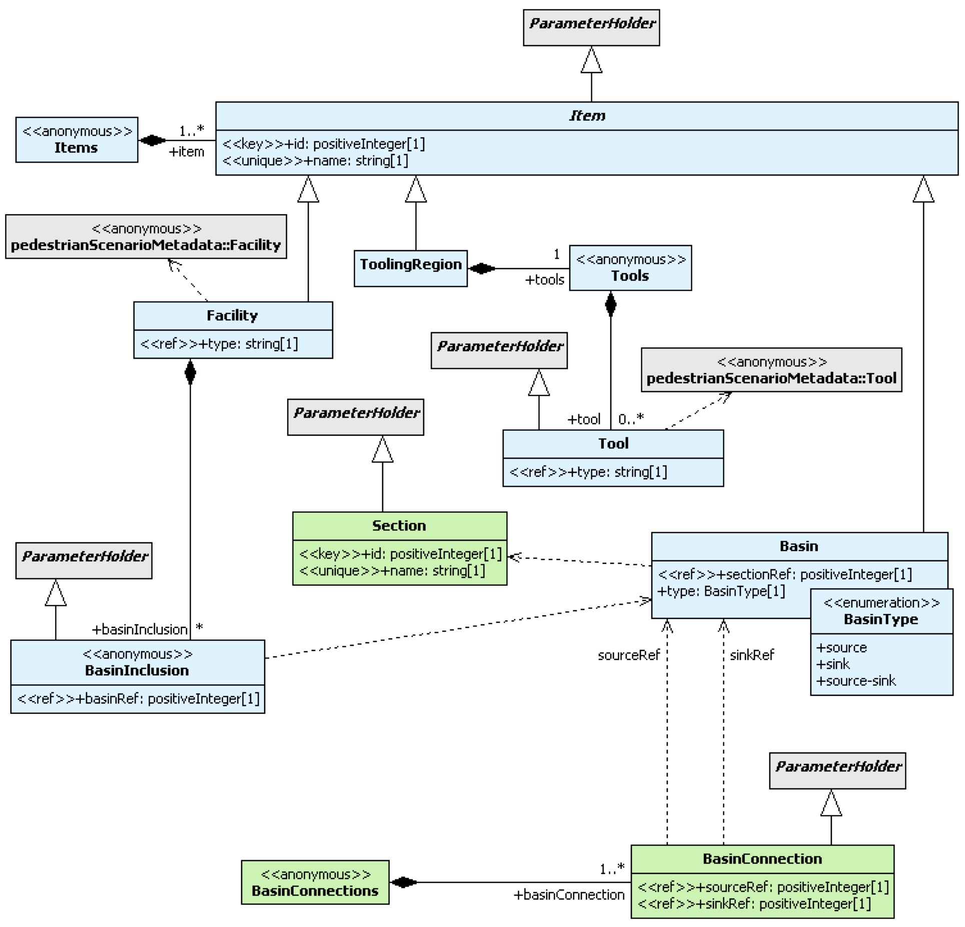

ParameterHolder
Item
<<key>>+id: positiveInteger[1]
<<unique>>+name: string[1]
<<anonymous>> Items
1..* +item
<<anonymous>> pedestrianScenarioMetadata::Facility
ToolingRegion
1 +tools
<<anonymous>> Tools
Facility
<<ref>>+type: string[1]
ParameterHolder
<<anonymous>> pedestrianScenarioMetadata::Tool
ParameterHolder
+tool 0..*
Tool
<<ref>>+type: string[1]
Section
<<key>>+id: positiveInteger[1]
<<unique>>+name: string[1]
ParameterHolder
Basin
<<ref>>+sectionRef: positiveInteger[1]
+type: BasinType[1]
<<enumeration>> BasinType
+source
+sink
+source-sink
+basinInclusion *
<<anonymous>> BasinInclusion
<<ref>>+basinRef: positiveInteger[1]
sourceRef
sinkRef
ParameterHolder
<<anonymous>> BasinConnections
1..* +basinConnection
BasinConnection
<<ref>>+sourceRef: positiveInteger[1]
<<ref>>+sinkRef: positiveInteger[1]

Figure 3.7: Items in pedestrianScenario.xsd

name for each section of the scenario, see Figure 3.8. The absolute positioning of sections within the simulated space is implemented by `absoluteOrigin`, `size` and `rotation`. The geometry of sections is specified by layouts, a maximum of one layout of each model type can be contained by a section, implemented by `vectorLayout`, `gridLayout` and `graphLayout`. .

A `VectorLayout` (Figure 3.9) consists of a `walkableRegion` defining the boundary of the space reachable for pedestrians and lists of `obstructionRegions` and `itemRegions`. Obstruction regions define areas of impassable space, item regions implement the assignment of items to spatial regions. All of these region types are specified by `VectorShapes`, which is the abstract base type of `Polygons` and `PolyLines`, both 2-dimensional shapes given by lists of `Points`.

A `GridLayout` (Figure 3.10) is defined by the alignment and meshing of the grid and sets of cells marked as obstruction or having items assigned. The former is given by `gridOrigin`, `rotation` and `cellSize`, specifying the origin relative to the section's origin, the dimension of the grid cells and the clockwise xy-rotation of the grid around its origin in degree, relative to the section's rotation. The latter are defined by `itemRegions`, listing references to `items` with corresponding sets of `cells`, and a list of impassable `obstruction`'s `cells`.

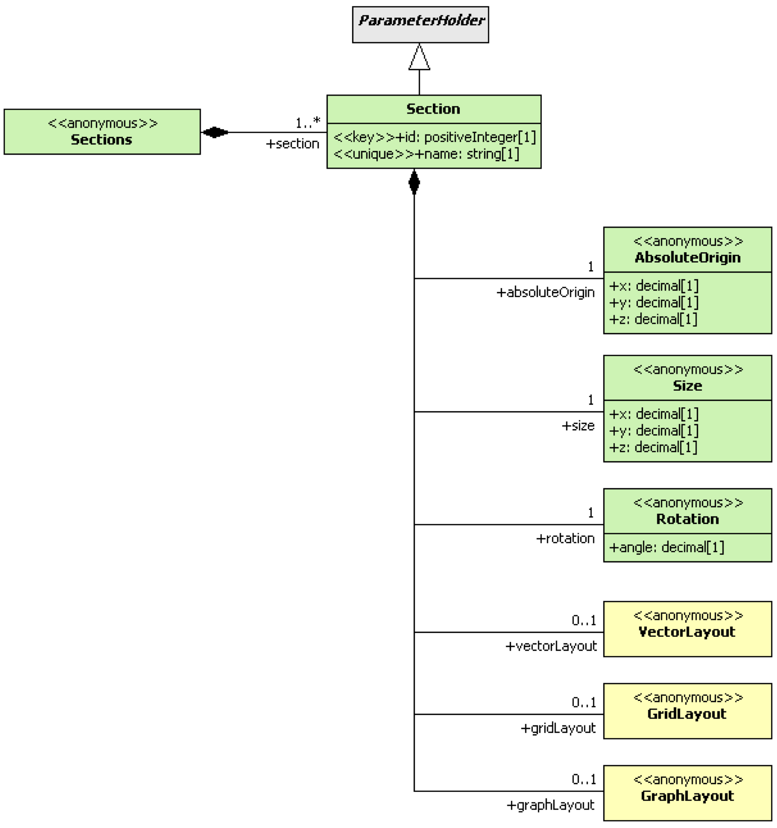
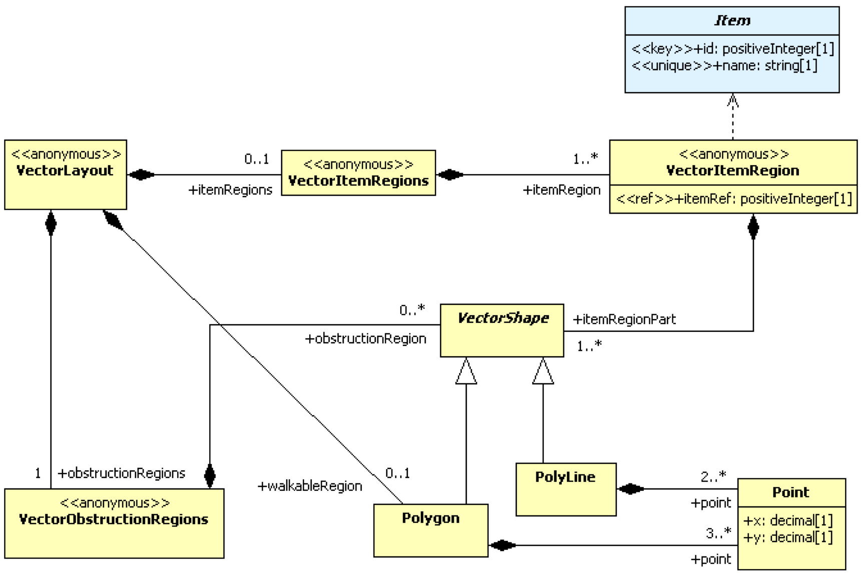


Figure 3.8: Sections in pedestrianScenario.xsd



Item
<<key>>+id: positiveInteger[1]
<<unique>>+name: string[1]
<<anonymous>> VectorLayout
0..1
+itemRegions
<<anonymous>> VectorItemRegions
1..*
+itemRegion
<<anonymous>> VectorItemRegion
<<ref>>+itemRef: positiveInteger[1]
0..*
+obstructionRegion
VectorShape
+itemRegionPart
1..*
1
+obstructionRegions
<<anonymous>> VectorObstructionRegions
+walkableRegion
0..1
Polygon
PolyLine
2..*
+point
3..*
+point
Point
+x: decimal[1]
+y: decimal[1]

Figure 3.9: Vector layouts in pedestrianScenario.xsd

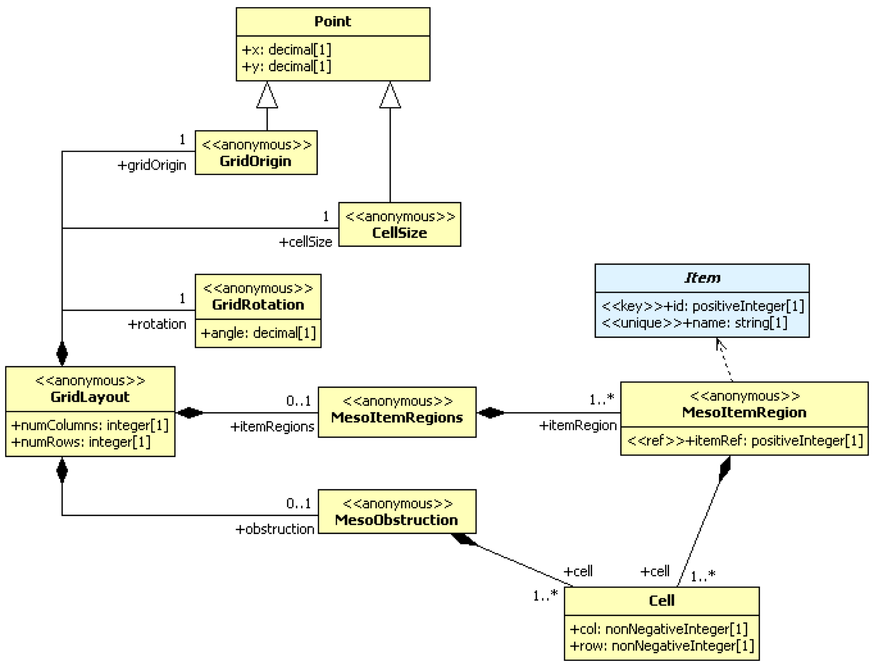


Figure 3.10: Grid-based layouts in pedestrianScenario.xsd

`GraphLayouts` (Figure 3.11) define networks of `nodes` and `links`, with each `link` referencing by `id` its adjacent nodes. A `node` has a location given by its `x` and `y` coordinates and can optionally have `items` assigned. `Links` contain a `distance` value and can additionally be dynamically parameterized as `Link` is subtyping `ParameterHolder`.

Pedestrian types and OD-matrices. The simulated types of pedestrians and where they enter and leave the simulation is modelled by the `pedestrianTypes` element. `PedestrianType` is abstract with three concrete subtypes, every `PedestrianType` has a name and an identifier attribute. An `AtomPedestrianType` defines the type of a simple person, the remaining two subtypes are composite elements. `PedestrianDistributionTypes` compose inhomogeneous populations by defining child types and their relative occurrence frequencies and `PedestrianGroupType` represents groups of pedestrians somehow belonging to each other like families or tourist parties . The number of occurrences of group members is modelled by a `ParameterValue` rather than a simple attribute to enable optional specification by probability distributions instead of numerical values, `numOccurrences` is constrained to the types `IntegerParameterValue` and `DiscreteDistributedParameterValue` with `IntegerParameterValues` as support.

The `name` of pedestrian types is constrained to uniqueness within their composition level, i.e. within the root types and the direct members of any `PedestrianGroupType` and `PedestrianDistributionType`. The `id` attribute is a global key and unique within all pedestrian types. As `PedestrianType` is a subtype of `ParameterHolder`, all concrete types can be parameterized by the user.

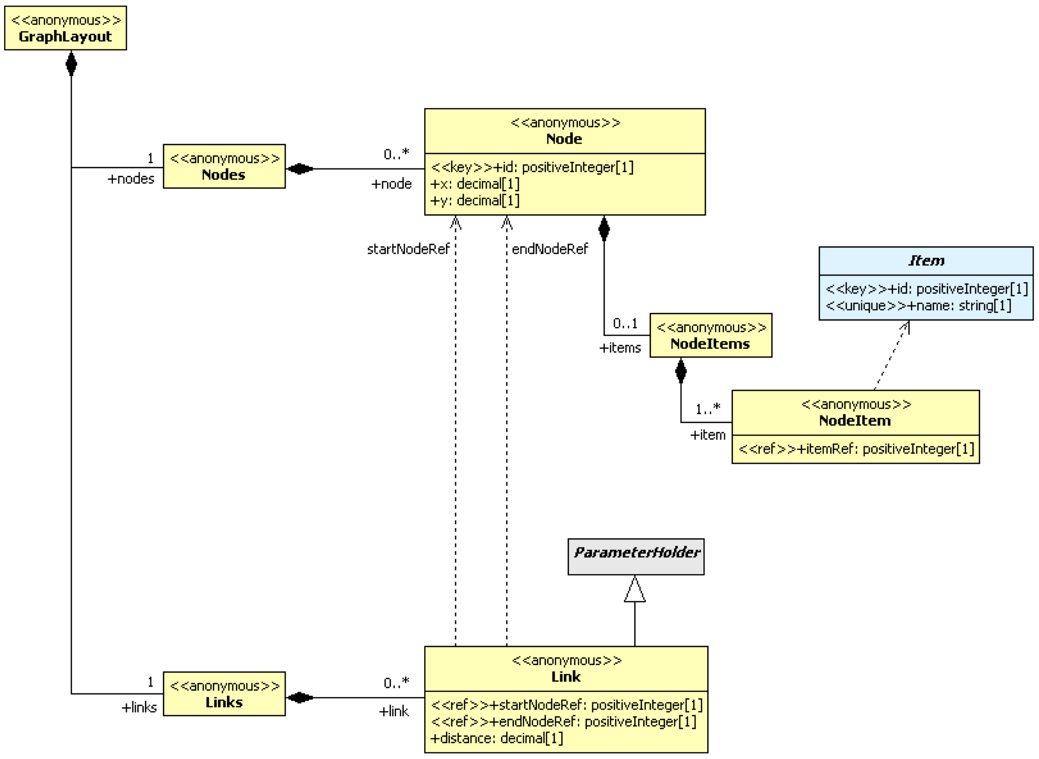
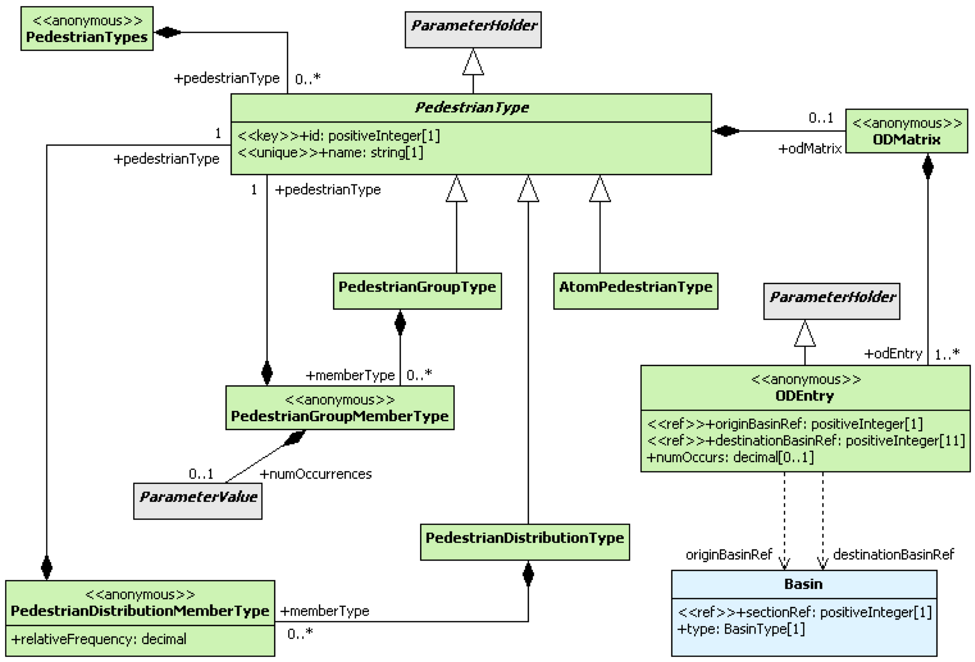


Figure 3.11: Graph-based layouts in pedestrianScenario.xsd



<<anonymous>> PedestrianTypes
+pedestrianType
0..*
ParameterHolder
PedestrianType
<<key>>+id: positiveInteger[1]
<<unique>>+name: string[1]
1
+pedestrianType
1
+pedestrianType
0..1
+odMatrix
<<anonymous>> ODMatrix
PedestrianGroupType
AtomPedestrianType
ParameterHolder
+odEntry
1..*
+memberType
0..*
<<anonymous>> PedestrianGroupMemberType
<<anonymous>> ODEntry
<<ref>>+originBasinRef: positiveInteger[1]
<<ref>>+destinationBasinRef: positiveInteger[1]
+numOccurs: decimal[0..1]
0..1
+numOccurrences
ParameterValue
PedestrianDistributionType
originBasinRef
destinationBasinRef
<<anonymous>> PedestrianDistributionMemberType
+relativeFrequency: decimal
+memberType
0..*
Basin
<<ref>>+sectionRef: positiveInteger[1]
+type: BasinType[1]

Figure 3.12: Pedestrian types and OD-matrices in pedestrianScenario.xsd

Pedestrian types are instantiated at simulation runtime. Pedestrians usually enter the scenario at source basins and ultimately leave it at sink basins. More general behaviour is also supported by the framework due to the generic parameterization of pedestrian type structures, but the core concept of OD-matrices is dedicated to support that pattern. An `ODMatrix` can be assigned to a `PedestrianType` to define the trip structure of that type. An `ODMatrix` consists of `ODEntrie`s, which implement *Trips* of Subsection 3.3.3. The attributes `originBasinRef` and `destinationBasinRef` reference source and sink basins and `numOccurs` defines the corresponding relative or absolute amount. Additional trip behaviour information, like for example intended intermediate tasks, can be added using the `ParameterHolder` base type.

Parameterization. This paragraph refines the abstract `ParameterHolder` type. Parameter holders are able to carry values for simulation parameters as specified at an abstract level by a metadata definition. That abstract definition is presented below in Section 3.4.4, basically a parameter is specified by its name, type and multiplicity. The association of a value reflecting the abstract parameter specification at instance level is realized by a concept referred to as *parameter binding*.

Figure 3.13 shows the implementation of parameter bindings. A `ParameterBinding` refers to its abstract definition by the `name` attribute. Depending on the multiplicity of the parameter, the parameter binding contains one or more `parameterValue`s. The type hierarchy of parameter values reflects that of abstract parameter definitions (Section 3.4.4). Types of numerical parameter values are `IntegerParameterValue` and `RealParameterValue`, boolean values are assigned by `BooleanParameterValue`s and strings by `StringParameterValue`s. Choices of enumerations are represented by `EnumerationParameterValue`s. Other scenario entities can be referenced by `ReferenceParameterValue`s, which specify the `id` of the referenced instance. Unspecified values are modelled as `EmptyParameterValue`s.

A special role is fulfilled by `CompositeParameterValue`, as it enables the instantiation of complex structured types. A `CompositeParameterValue` is a container of nested parameter bindings (`childBinding`s).

A substantial part of the parameterization model is dedicated to a feature allowing parameter values to be substituted by probability distributions, which provides comprehensive flexibility for the integration of stochastics in simulations. Random variables may be inherent to a model or used for input data generation. An example for the first case is a randomized resolution of conflicting cell occupations in a cellular automaton model. A scenario of the second case might define its population using statistical data about physics and intentions of pedestrians. Both cases are addressed by the generic approach of enabling probability distributions to be used instead of fixed parameter values.

As the the type of a parameter determines the type of applicable probability distributions, an additional type hierarchy of `DistributedParameterValue` is specified, see Figure 3.14. A basic distinction is made between distributions of ordered (like integer) and unordered (like string) support types.

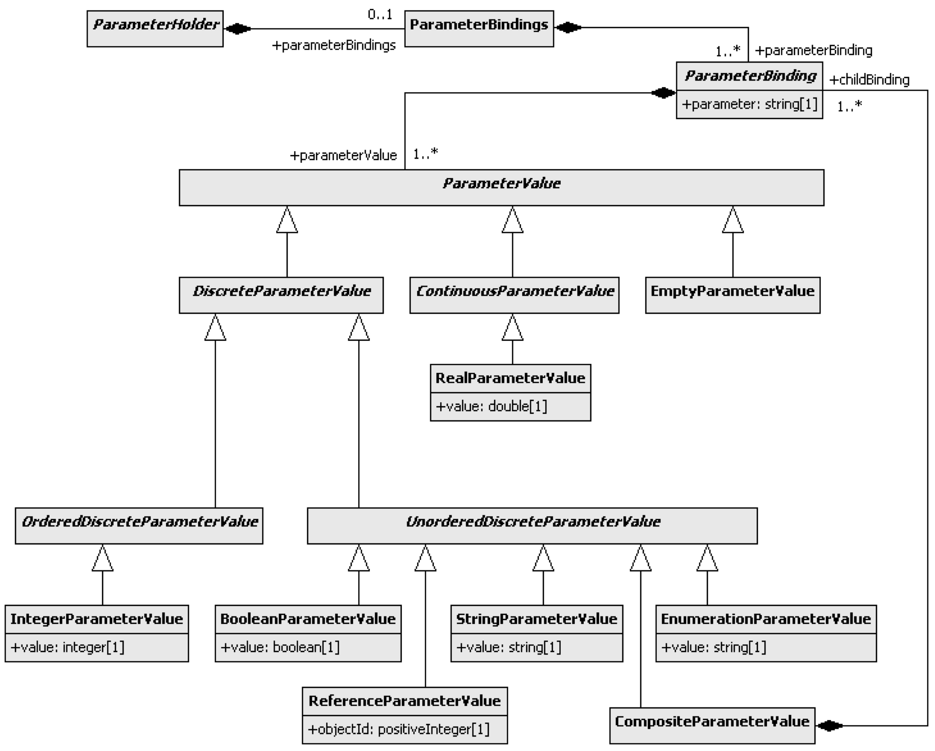


Figure 3.13: Parameterization in pedestrianScenario.xsd

Discrete uniform distributions with unordered support are modelled by `UnorderedUniform-DistributedParameterValue` that contains a list of values of equal possibility. An `Unordered-DistributedParameterValue` assigns relative frequencies to value categories.

Distributions for ordered support are by themselves subject to abstract parameterization, that is they are modelled generically as defining a support range and a parameter set. A set of well-known distributions is built into the specification, see Table 3.3, but due to the generic structure, additional user-defined distributions can be added by metadata definition. The `type` attribute of `OrderedDistributedParameterValue` identifies either one of the built-in types or references an abstractly defined distribution of a metadata file.

Metadata Definition Schema

In the following, the implementation of the (grey-colored) *parameter* package of Figure 3.2 is explained and the schema of `pedestrianScenarioMetadata.xsd` is documented. Instance documents of this schema, i.e. metadata files, basically contain information about

- named types of facilities and tools,
- parameterization and
- custom distributions.

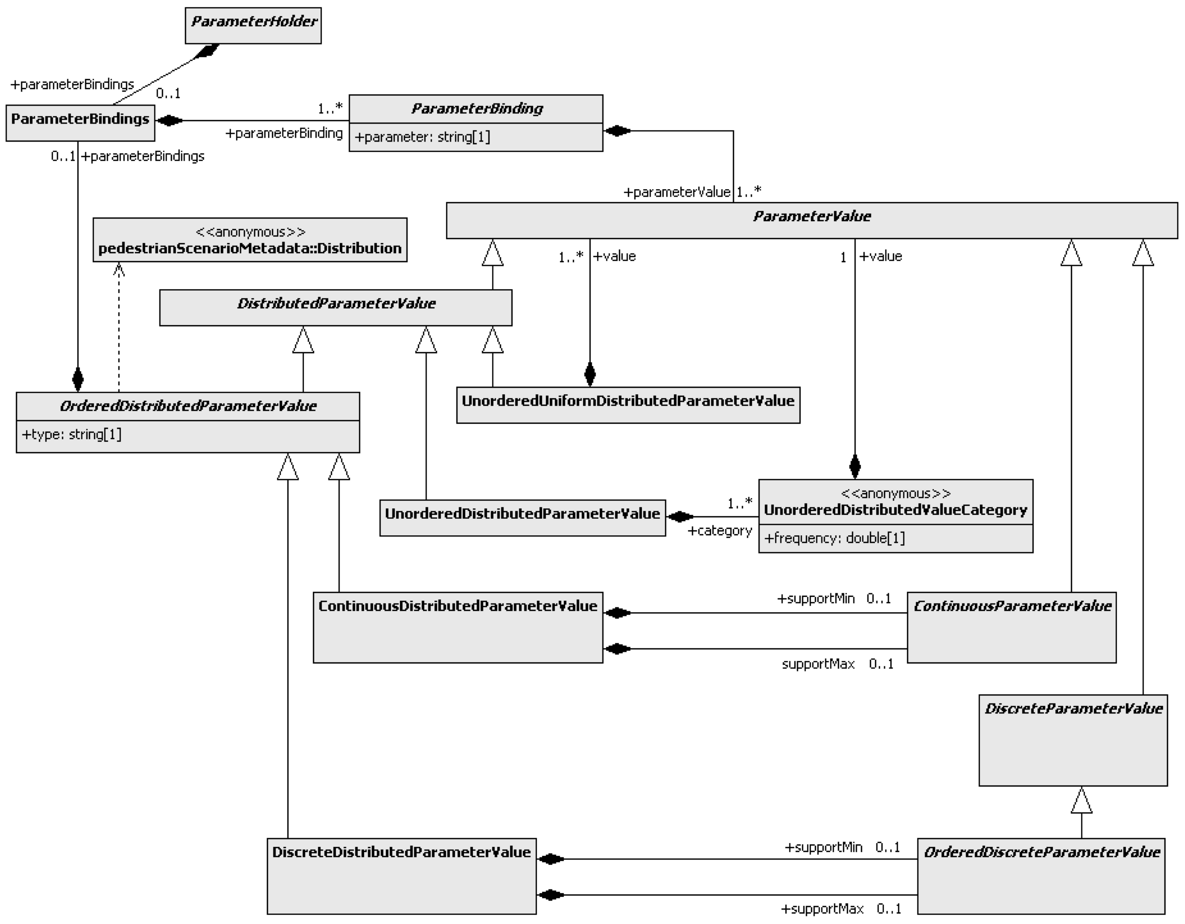


Figure 3.14: Parameterization in pedestrianScenario.xsd

distribution type	parameter name	parameter type
`Ordered Uniform Distribution`	-	
`Binomial Distribution`	`n` `p`	integer, `n` ≥ 2 real, $0 \leq$ `p`
`Poisson Distribution`	`gamma`	real, `gamma` > 0
`Geometric Distribution`	`p`	real, $0 \leq$ `p` ≤ 1

`Hypergeometric Distribution`	`m`
	`nPopulation`
	`nSample`
		integer, $1 \leq$ `m`
		integer, $1 \leq$ `nPopulation`
		integer, $1 \leq$ `nSample`

`Cauchy Distribution`	`s` `t`	real, $0 <$ `s` real
`Gaussian Distribution`	`sigma` `gamma`	real, $0 <$ `sigma` real

Table 3.3: Built-in distribution types

Multiplicity bounds. Two global simple types are defined by the schema to be used for multiplicity bounds. `LowerBound` is simply a XSD built-in `nonNegativeInteger`, `UpperBound` is an `integer` with an inclusive minimum of `-1` to indicate "unlimited".

Root structure. The root structure of `pedestrianScenarioMetadata.xsd` is shown in Figure 3.15. For intuitive understanding of the schema it is important to have in mind that the naming of elements is related to the abstract context of the metadata definition. For example, `facility` elements of `pedestrianScenarioMetadata.xsd` abstractly define named types of `Facility` elements of `pedestrianScenarioMetadata.xsd`.

The root `scenarioMetadata` element contains the version attributes explained in Section 3.4.4. Its first child element `facilities` contains the abovementioned facility type definitions. The `facility` element, together with `parameterSets` explained below, implements the *AbstractFacility* concept of Subsection 3.3.2. An abstract facility is identified by its `name` and defines the possible range of the number of associated basins by `minNumBasins` and `maxNumBasins`. The attribute `autoConnectBasin` is a flag indicating whether infrastructural connections should be automatically generated between basins of a facility instance, i.e. if the facility in general connects sections.

The second child element `tools` is a list of `tool` elements, corresponding to *AbstractTool* of Subsection 3.3.2. Each tool has a unique `name` used to select it as part of a tooling region.

Dynamic parameterization of scenario entities is specified by the `elementParameters` list, that contains `parameterSets`. A `parameterSet` is an assignment of parameters to subject entity types. Parameters are specified by `AbstractParameter` elements, entity types are identified by `ElementType` elements. (Note that the term *entity* is used here for objects of a scenario instance to avoid confusion with XML *elements*. Though within `pedestrianScenarioMetadata.xsd`, `element[..]` is used for names of XML elements representing that notion of entities.) A `parameterSet` is an m:n association of `ElementTypes` and `AbstractParameters`, i.e. any number of subjects can be assigned any number of parameters by one `parameterSet`, and the metadata definition contains any number of `parameterSets`. This allows for flexible reuse of parameters and sets of related parameters. Actually, `parameters` elements can be understood as type definintions and the assignment to element types as subtyping by multi-inheritance.

Finally, `distributions` specifies custom types of probability distributions. A `distribution` has a unique `name` attribute and a `type` attribute indicating the distribution to be either `continuous` or `descrete`, and contains a list of `AbstractParameters`.

Parameterization. The above descriptions refer the `AbstractParameter` element type for the purpose of parameterization of scenario entities and distributions. As explained for the scenario definition schema, all elements with `ParameterHolder` as base type can contain `Parameter-Bindings`, which in turn define values for named parameters. The set of applicable parameters and their names, structures and constraints in turn are specified by the metadata definition currently examined, see Figure 3.16. An `AbstractParameter` defines such a parameter by name,

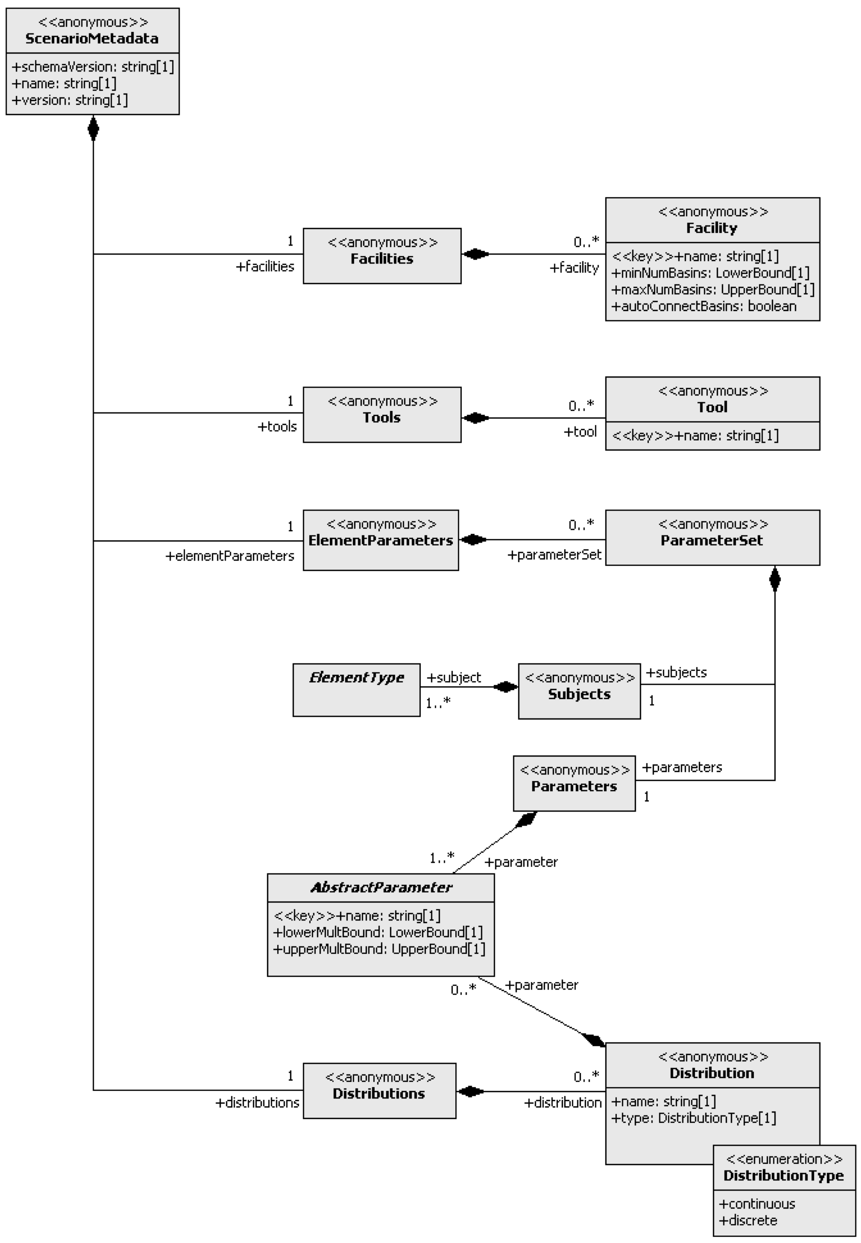


Figure 3.15: Root structure of pedestrianScenarioMetadata.xsd

multiplicity and type. The name is unique within composition levels, precisely within the root parameters of all parameterSets, the root parameters of any distribution and the direct childs of any CompositeParameterType. Multiplicity bounds are realized by lowerMultBound and upperMultBound. The parameterType child element has the complex type ParameterType with the type hierarchy of Figure 3.16. Inner structures of parameters, that is CompositeParameterValues at instance level are possible using CompositeParameterTypes which contain their child AbstractParameters in the childParameters list. RealParameterType and IntegerParameterType are numerical types with optional range bounds. An EnumerationParameterType defines a list of strings as possible literal values for a parameter. StringParameterTypes can be constrained by a length range and a regular expression pattern (see [Fri06] on regular expressions). ReferenceParameterTypes enable parameters to hold references to scenario entities, usually to express some semantical relationship. The list of elementTypes child elements constrains the possible types of entities referenced by the parameter.

Element types. The schema of the metadata definition reflects on entity types of scenarios, i.e. element types of scenario definitions. It therefore requires to use that types as first-class objects, concretely to define subjects in parameter sets and type restrictions of reference parameter types. The type hierarchy is implemented based on the abstract ElementType type as documented by Figure 3.17.

Most of the subtypes are self-explanatory. The subtypeRestriction elements of ToolElementType and FacilityElementType are used to reference named types of tools or facilities respectively to provide parameters of (or references to) specific user-defined types. The same applies for subtypeRestriction of BasinInclusionElementType, where only pedestrianScenario:BasinInclusions contained by facilities of the specified type are selected. BasinElementTypes can be refined by including the required basin type of source, sink or source-sink. If no type is defined for a BasinElementType, any pedestrianScenario:Basin, regardless of its type is selected.

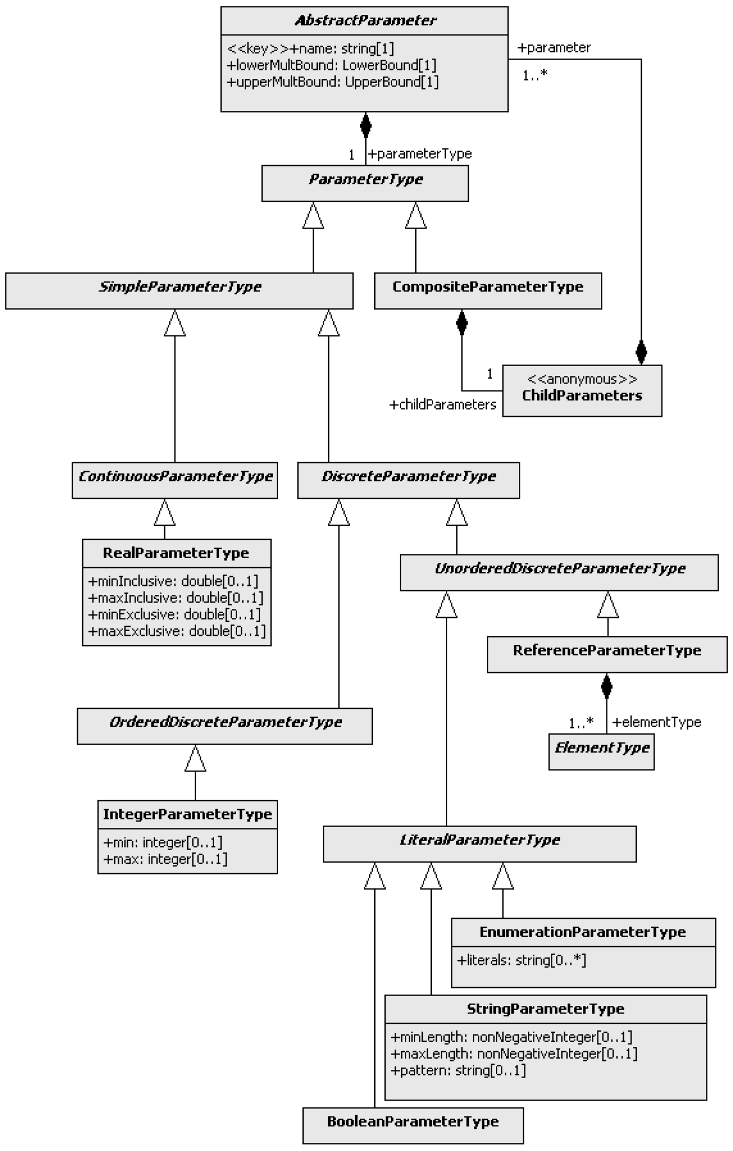


Figure 3.16: Abstract parameters in pedestrianScenarioMetadata.xsd

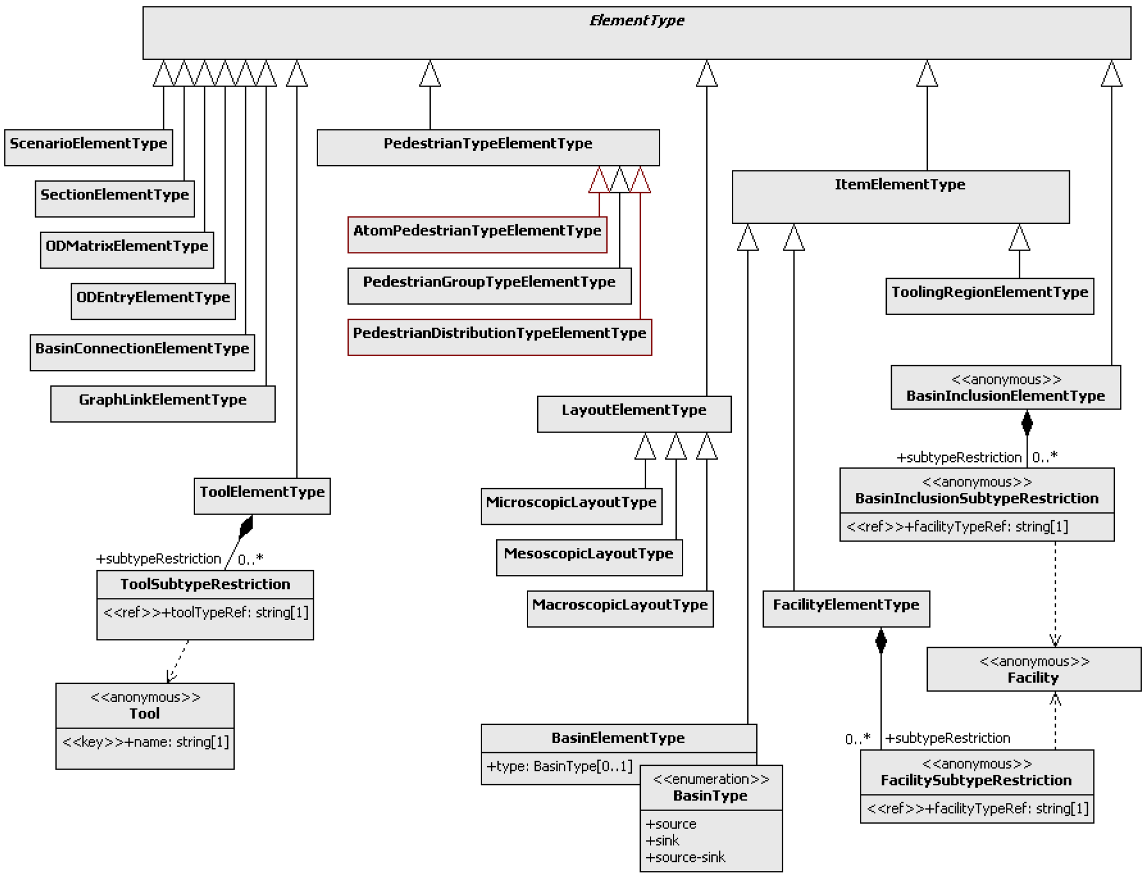


Figure 3.17: Element types in pedestrianScenarioMetadata.xsd

4 Unified Scenario Definition Tool: Scenario Builder

The second part of the realization of the generic tooling approach concerns the development of the *Scenario Builder*, a software tool that enables the creation and maintenance of simulation scenarios conforming to the generic interface. This chapter surveys requirements analysis, design and implementation of the Scenario Builder, and closes with examinations of workflow examples and pratical adoption.

4.1 Requirements

Essentially, the concepts and structures modelled by the data interface need to be reflected by the user interface, and appropriate means need to be provided by the tool for the creation and manipulation of the model elements. A substantial part of the requirements analysis for the Scenario Builder is therefore covered by the design of the generic interface (Chapter 3). On that structural basis, additional workflow-related functional requirements are identified. The following subsections explain the requirement fields at a coarse level, refinement is given in Section 4.2 and Section 4.3 by means of more detailed specifications of data structures and user interface functionality.

4.1.1 Data Organization

A feature not addressed by the interface specification is the support of efficient comparison of different, but somehow related simulation scenarios. Obvious use cases reasoning this requirement are the evaluation of design alternatives or changes, e.g. the replacement of a swinging door by a revolving door, or the estimation of the effects of changing exterior conditions, like overcrowding or panic versus normal conditions.

GPSI defines data structures aggregated by the simulation scenario as the topmost concept. To support multi-scenario comparisons, the *project* concept is proposed to be superimposed by the Scenario Builder, with multiple scenarios being manageable within a single project. The scenarios of a project should be enabled to share common resources but define individual variations of spatial and logical configuration.

4.1.2 Data Input and Output

Naturally, scenario data processed and produced by the tool needs to correctly conform to the interface specification of Chapter 3. This applies for input and output; as shown by Figure 1.3 configuration data, i.e. metadata definitions need to be accepted as specified to generate GPSI-conform scenario definitions as output. Given the XML-based implementation of the interface this requirement demands the correct validation of input and output XML data against the schema definitions of `pedestrianScenario.xsd` and `pedestrianScenarioMetadata.xsd`.

Besides XML-based scenario data specified by the GPSI, the tool needs to persist user data storing the projects that are created and maintained by the application. That data is strongly implementation-specific and constrained by only few functional user requirements. Application data should be persisted on a per-project basis, in a way that enables easy import and export of single projects into different working environments, and easy backups of project states.

4.1.3 Geometry Prototyping

The construction of geometry (vector-, grid- or graph-based) is a laborious part of the scenario definition workflow. Scenario Builder can improve efficiency and aid the user by offering the following features:

- CAD-Import. Practical simulation projects usually examine environments with geometry data available beforehand in the form of CAD plans, which provide a proper basis for vector-based scenario layouts. Scenario Builder should therefore be able to automatically generate vector layouts from selected layers of files of the popular and industry-standard DXF-format (Drawing Interchange File Format [DXF]).
- Layout transformation. Designing scenarios for simulations using different layout types involves the construction of multiple layouts for single sections. This suggests special tool support to derive new layouts from existing layouts. Especially, the assumably most frequent case of a grid-based layout being based on an existing vector-based layout needs to be addressed by features that allow to map parts of or complete vector layouts into grid representations.
- Templating. In cases where neither CAD-import nor layout transformation are applicable, the user should be enabled to use arbitrary graphics, e.g. scanned sketches or photographs as templates for layout creation. When editing vector-, grid- or graph-based layouts, a background image of a common graphics format should be selectable.

4.1.4 Look and Feel

Section 1.4 explains benefits of the unified tooling approach for both scientific professionals and end users. The Scenario Builder application is developed focusing on the former, expert user type. It is therefore designed as a tool mainly handled by power users [Rei03], allowing to waive some comfort functionality in favour of more flexibility and less development effort respectively.

Nonetheless, Scenario Builder is required to offer the usage experience of modern rich client applications, using standard features such as customizable multi-view interfaces, tabbed editors, multi-monitor support, tree views for structured data, context-sensitive property sheets, undo- and redo-support, drag and drop, copy/cut and paste, and keyboard shortcuts. Graphical editors should provide commonly expected functionality like zooming, snapping and graphical feedback.

4.2 Project Model

The Scenario Builder Domain Model has been introduced by Figure 3.1 in Section 3.2. It is defined as the conceptual structure of the domain-related data used by the application. The part of the domain model that is related to output data generation is the Generic Input Model explained in Chapter 3. The other part of the domain model, the Scenario Project Model, is treated in this section. The Scenario Project Model describes how scenario data is structured by the tool implementation. This organization differs in some aspects from the GPSI model to enable user-oriented tool support and to provide the additional functionality explained in Section 4.1, but the same core concepts apply. Note that the Scenario Project Model also differs from the implementation data structure, as the implementation model is *derived* by (not containing) the domain model, see Figure 3.1, that is the concept presented in this section is subsequently *implemented* using a programming language, refining the structure and adding implementation specific features.

Figures 4.1 and 4.2 show class diagrams of the Project Model. The main difference to the interface model (Figure 3.3) is the *Project* being the root container of scenario data. Instead of a simple set of *Scenarios* containing their own data the *Project* is a repository of elements which are referenced by any number of *Scenarios*. This way of data structuring, together with the concept of *scenario overrides* discussed below, allows for a flexible mapping of similarities and variations between scenarios claimed in Subsection 4.1.1.

A *Project* contains *Scenarios*, *Items*, *PedestrianTypes*, *ODMaps* and one *Infrastructure*.

The *Infrastructure* of a *Project* defines the high-level spatial configuration for all *Scenarios* of that *Project*. It consists of *Sections* and *BasinConnections* that are organized analagous to Subsection 3.3.2, that is *Sections* are 3-dimensionally located and sized, with *Layouts* defining the 2-dimensional geometry of the xy-projection, and *BasinConnections* describing ways for pedestrians to move from one section to another by referencing endpoint *Basins* (which in turn are spacially associated within *Layouts*). In contrast to the interface model, any number of *Layouts* of each type may exist within one *Section*. Still, the limitation to a maximum of one layout per type and section is applied to the context of a single scenario by the *ScenarioLayout* - every *Scenario* owns a *ScenarioLayout*, a *ScenarioSectionLayout* references zero or one *Layout*

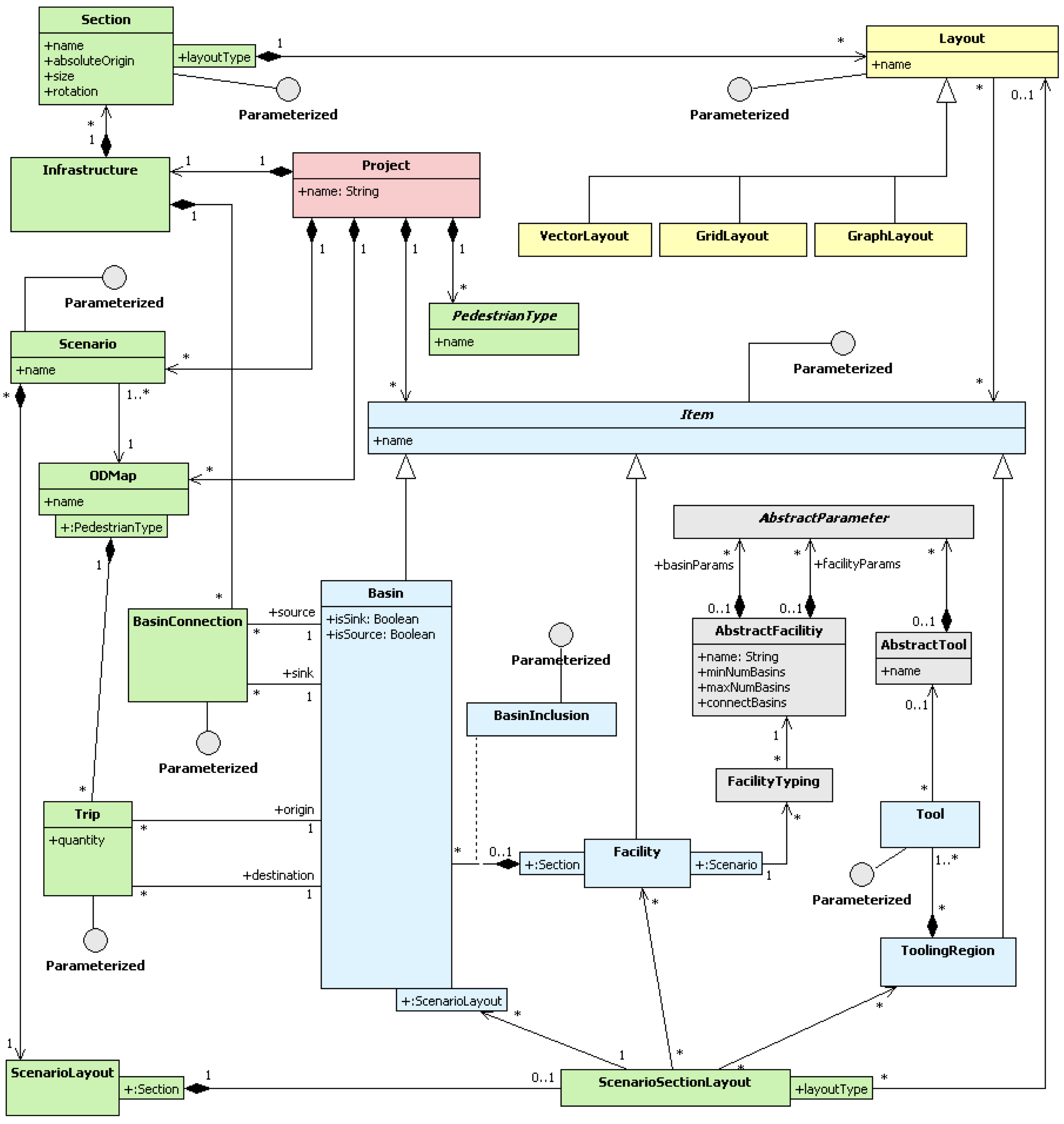


Figure 4.1: Class diagram of the Scenario Project Model

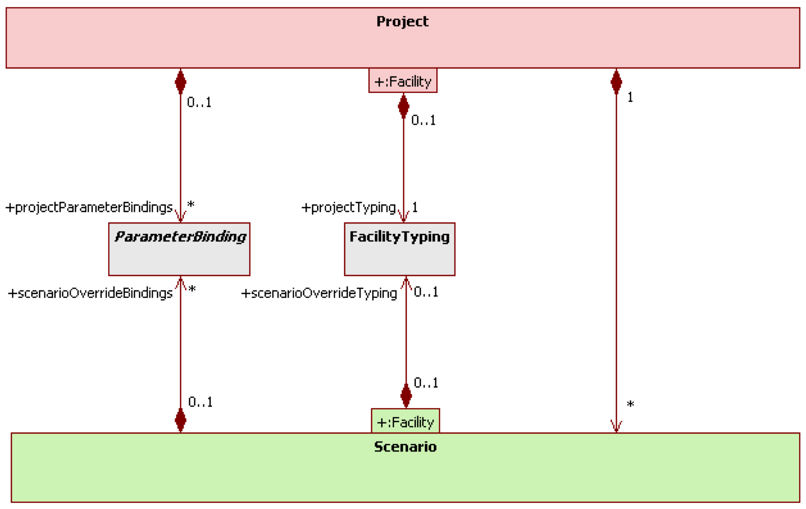


Figure 4.2: Scenario overrides in the Scenario Project Model

for each layout type, noted by the *layoutType* quantifier. The sets of *Items* referenced by each *Layout* of one *ScenarioSectionLayout* can be checked to be the same in order to assert consistent data for the involved simulation models. This set of *Items* therefore defines the set of *Items* referenced by the *ScenarioSectionLayout* itself, explicitly depicted as associations between *ScenarioSectionLayout* and *Basin*, *Facility* and *ToolingRegion* respectively. Different *Scenarios* may reference different subsets of the *Project's Basins*, while only a single *Infrastructure* with *BasinConnections* exists within a *Project*. The Project Model's *Infrastructure* is mapped to the GPSI model's *Infrastructure* by considering only *BasinConnections* of *Basins* occuring in *Layouts* referenced by the exported *Scenario*. An alternative approach of multiple named infrastructure elements similar to *ODMaps* has been considered but discarded in favour of simplicity as related use cases are assumably rare.

Items exist in the context of a *Project*. They may be referenced by *Layouts* of different *Sections* and types with the restriction that a given *Basin* may only be referenced by *Layouts* of one and the same *Section*. Therefore within one *ScenarioLayout* a *Basin* can only be referenced by *Layouts* of a single *ScenarioSectionLayout*. Like with the GPSI model, *Facilities* are defined by *AbstractFacilities*, but the Project Model uses an additional indirection by *FacilityTypings*. In addition to the *Project's* typing of a *Facility*, each *Scenario* may contain its own *FacilityTyping* for each *Facility* of the *Project*, see the *projectTyping* and *scenarioOverrideTyping* associations in Figure 4.2. This enables the variation of the type of a Facility within different simulation scenarios.

A *Project's* collection of *PedestrianTypes* specifies the types of pedestrians available in any *Scenario* of that *Project*. It provides the building blocks of the simulation population configurable per scenario by means of *ODMaps*. *ODMaps* do not exist in the GPSI specification, as at scenario level the OD-matrices of pedestrian types are directly defined by a collection of trips for each pedestrian type. With the Project Model, *ODMaps* as an intermediate element of indirection are conceptually interchangable named collections of OD-matrices for a given set of pedestrian types. An *ODMap* assigns *Trips* to *PedestrianTypes*, again the *quantity* attribute of *Trip* is intended to be interpreted by simulation implementations in a proper way, either as absolute or relative value. A *Project* may hold any number of *ODMaps*, but exactly one *ODMap* is required to be referenced by every *Scenario* to define its population.

Elements that realize *Parameterized* can be customized using *AbstractParameters* and *ParameterBindings*, see Subsection 3.4.4. The complete set of *ParameterBindings* for any element of a *Project* is associated with that *Project* (Figure 4.2, *projectParameterBindings*). An additional set of *ParameterBindings* can be associated with each *Scenario* of the *Project* (Figure 4.2, *scenarioOverrideBindings*), meaning that these *ParameterBindings* overrule *ParameterBindings* to the same *AbstractParameter* for the same subject element. This concept enables the use of project-level default configurations, combined with per-scenario variations (e.g. the speed of an escalator might usually be x, but for a certain scenario it shall be y to examine the effects of that variation). The features of *Scenario*-based *FacilityTypings* and *ParameterBindings* are summarized as *scenario overrides.*

Further descriptions of the detailled structure of *Layouts*, *PedestrianTypes*, *AbstractParameters* and *ParameterBindings* can be given identical to the GPSI model, wherefore with reference to Chapter 3 they are not repeated here.

4.3 User Interface Model

4.3.1 Overview

The user interface of the Scenario Builder basically enables the application user to create, manipulate and maintain data corresponding to the Project Model (Section 4.2). The complexity of the Project Model, considering e.g. multiple layout types, custom data types, scenario overrides, consistency constraints etc. suggests a rich client application concept based on a set of flexibly arrangable view panes each providing a dedicated part of the overall functionality.

As structured data is commonly presented and manipulated using collapsible and expandable trees, a tree view pane (the *Project Explorer*, Section 4.3.2) is used to expose the repositories of elements provided by projects. The multi-scenario capability of projects requires further support by the user interface, as with increasing complexity of projects the user will likely prefer to concentrate on a single scenario at a time instead of being overburdened by a growing total number of project elements. Therefore a second type of tree view panes (the *Scenario Explorer*, Section 4.3.2) is used to provide a filtered view on those parts of the domain that are referenced by given scenarios respectively.

Different graphical editing capabilities are needed for each of the three layout types, therefore three types of layout editor panes are required. The analogy of project elements associated with layout elements to UML classes represented in class diagrams has already been mentioned in Subsection 3.3.2, concerning the user interface the inspiration by UML tools also leads to a combination of tree- or list-based logical views and graphical editors. According to the layout types of the Project Model, this section introduces the *Vector Layout Editor* (Section 4.3.5), the *Grid Layout Editor* (Section 4.3.5) and the *Graph Layout Editor* (Section 4.3.5).

The high-level spatial configuration of a project is given by its infrastructure as defined in Section 4.2. The arrangement of sections and the configuration of basin connections between sections can intuitively be done using graphical editing, addressed by the *Infrastructure Editor* (Subsection 4.3.6).

OD-maps are project elements with an inner structure that cannot be mapped by a simple property sheet or a dialog form. They are therefore subject to another, non-graphical dedicated editor (the *OD-Map Editor*, Subsection 4.3.7), used to select origin and destination basins of the contained OD-matrices and to fill the OD-matrices of pedestrian types with values.

The above user interface parts visualize project elements and some of their properties. Beyond that, project elements generally have characteristics that are not visible or editable in the respective views. A commonly accepted solution are context-sensitive *property sheets*, presenting attributes of currently selected objects in a table of names and values. The *Element Inspector* (Subsection 4.3.3) is accompanying all other views as a feature-added, flexible property sheet view enabling comprehensive manipulation of project elements.

The Project Model imposes several kinds of constraints for project instances to be valid, amongst others multiplicity bounds of abstract parameters or layout consistency within scenarios. The scenario definition tool must not block the user's work by forcing her to resolve constraint violations instantaneous and assert every project to be in a valid state at any time, but instead bundle information about existing problems for subsequent handling. This is a solution similar to modern IDEs that allow compile-time errors and warnings to exist but mark them within program code and collect them in lists linked with the related source. This concept is realized by the *Problems View* (Subsection 4.3.8).

4.3.2 Explorer Views

There are two types of explorer views, the *Project Explorer* and the *Scenario Explorer*, providing tree-structured presentations of project elements.

As explained in Section 4.2, a project may contain multiple scenarios, each of which using a subset of the elements the project contains. The Project Explorer enables the user to browse all project elements without any scenario-related information. It is therefore an overview of the repository of project elements available for any scenario of the project. The Scenario Explorer allows browsing scenario-related information.

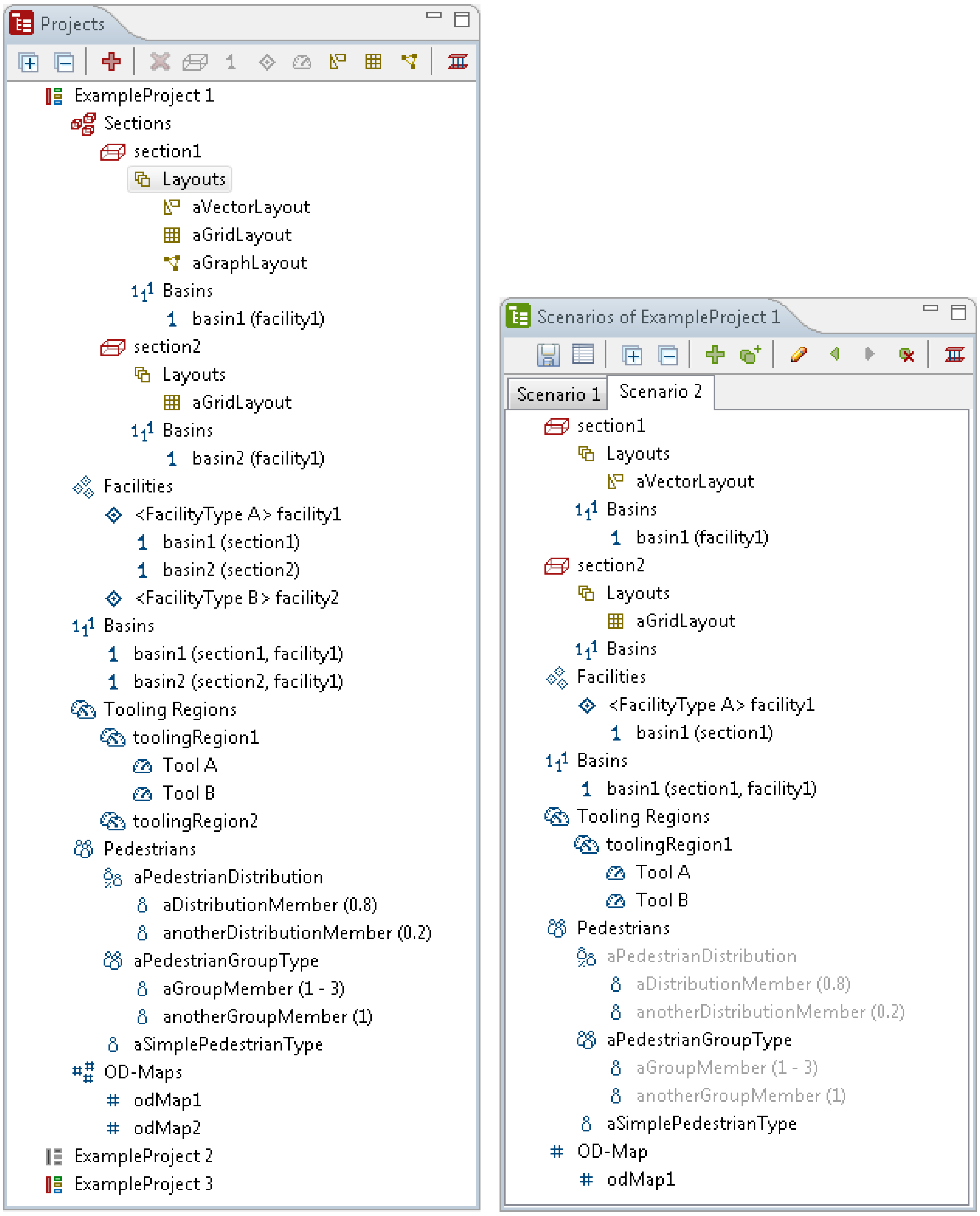
In the following, the term *element nodes* is used for nodes of an explorer tree that represent project elements. Element nodes can more specifically be referenced by prefixing the type of the associated element, e.g. a *layout node* is a tree node representing a Layout in terms of the Project Model (or a subtype of Layout). For a clearer structure of the explorer views, intermediate tree levels are used to separate child nodes by their type or semantic category. These intermediate levels contain so-called *category nodes* that act as simple container nodes, wherefore they don''t have project-related properties by themselves. Category nodes can more specifically be referenced by prefixing the name of the category – e.g. *sections category node*.

Each node is labeled using a small icon marking its type. Thus for example a vector layout node can easily be distinguished from a grid layout node. Element nodes are labeled using the name attribute of the associated project element. Child nodes are in general ordered alphabetically, section nodes are ordered by the index of the associated sections within the project. The explorer trees support multiple selection of tree nodes, right-clicking a node displays a context menu with actions applicable to the selected nodes. Double-clicking a layout node or an od-map node opens an appropriate editor for the associated project element.

Project Explorer

Multiple Projects are accessible simultaneously within a Scenario Builder program instance. The project explorer presents every project of the workspace as a single tree as shown in Figure 4.3, using the following node types:

- Project nodes
  Project nodes are the root nodes of project trees. Projects are either *open* or *closed*. The open/closed-state is depicted by the node icon. Project nodes for closed projects do not contain child nodes. Selecting a project node or one of its direct or indirect child nodes focuses the Scenario Explorer on the associated project.
  The accessible actions for project nodes within the Project Explorer are:
  - Open project
    Opening a project means exposing its content to user interface, i.e. the project node contains child nodes to constitute the project tree, the scenario explorer can show scenarios of the project, editors for elements of the project can be opened and the Problems View (Subsection 4.3.8) lists problems and errors associated with the project.
  - Close project
    Discards parts of the user interface corresponding to the project, i.e. closes editors of elements contained by the project and removes the child nodes of the the project node. Also, the Problems View does not contain entries associated with closed projects.
  - Remove project
    Removes a project from the Scenario Builder and moves its contents to a garbage directory in the file system.
- Sections category nodes, contain child nodes for the sections of a project and provide an action to create a new section.



```
ExampleProject 1
    Sections
        section1
            Layouts
                aVectorLayout
                aGridLayout
                aGraphLayout
            Basins
                basin1 (facility1)
        section2
            Layouts
                aGridLayout
            Basins
                basin2 (facility1)
    Facilities
        <FacilityType A> facility1
            basin1 (section1)
            basin2 (section2)
        <FacilityType B> facility2
    Basins
        basin1 (section1, facility1)
        basin2 (section2, facility1)
    Tooling Regions
        toolingRegion1
            Tool A
            Tool B
        toolingRegion2
    Pedestrians
        aPedestrianDistribution
            aDistributionMember (0.8)
            anotherDistributionMember (0.2)
        aPedestrianGroupType
            aGroupMember (1 - 3)
            anotherGroupMember (1)
        aSimplePedestrianType
    OD-Maps
        odMap1
        odMap2
ExampleProject 2
ExampleProject 3
```

```
section1
    Layouts
        aVectorLayout
    Basins
        basin1 (facility1)
section2
    Layouts
        aGridLayout
    Basins
Facilities
    <FacilityType A> facility1
        basin1 (section1)
Basins
    basin1 (section1, facility1)
Tooling Regions
    toolingRegion1
        Tool A
        Tool B
Pedestrians
    aPedestrianDistribution
        aDistributionMember (0.8)
        anotherDistributionMember (0.2)
    aPedestrianGroupType
        aGroupMember (1 - 3)
        anotherGroupMember (1)
    aSimplePedestrianType
OD-Map
    odMap1
```

Figure 4.3: Project Explorer (left) and Scenario Explorer (right)

- Section nodes, represent sections, contain category nodes for layouts and basins of a section and provide actions to delete, rename and reorder a section and to create new layouts and basins.

- Layouts category nodes, contain layout nodes for all layouts in the containing section and provide actions to create new vector-, grid- and graph-based layouts.

- Layout nodes, allow to open a layout editor and provide actions to delete, rename and duplicate the associated layout.

- Basins category nodes,
  occur as a child nodes of section nodes and directly within project nodes. They contain basin nodes for all basins placed in the containing section in the first case or for all basins of the project in the second case. Basins that are associated with facilities are also included. Basins category nodes provide an action to create a new basin.

- Basin nodes, represent basins and occur as child nodes of facility nodes for basins that are part of a facility, and of basin category nodes for all basins of a project. Context information, i.e. the containing section and optionally facility, which is not given by the parent node is contained in the node labels. A basin node provides actions to delete and rename the basin, and to select the node of the associated facility in the project explorer as a navigation shortcut.

- Facilities category nodes, contain child nodes for the facilities of a project and provide an action to create a new facility.

- Facility nodes to represent facilities. A facility node contains child nodes for basins associated with the facility and provides actions to delete and rename the facility, and to create a new basin associated with the facility.

- Tooling regions category nodes, contain child nodes for the tooling regions of a project and provide an action to create a new tooling region.

- Tooling region nodes, represent tooling regions, contain child nodes for associated tools and provide actions to delete and rename the represented tooling region, and to add a tool to the tooling region.

- Tool nodes, represent tools within a tooling region and provide an action to remove a tool from the containing tooling region.

- Pedestrian types category nodes, contain child nodes for the pedestrian types of a project, provide actions to create new atom, group or distribution pedestrian types.

- Atom pedestrian type, pedestrian group type and pedestrian distribution type nodes, represent pedestrian types and provide actions to rename and remove the pedestrian type

from the type structure.
If the presented pedestrian type is part of a pedestrian group type, the number of occurrences or the minimum and maximum of its distribution support respectively are added in brackets to the node label. If it is part of a pedestrian distribution, the relative occurrence frequency is added in brackets. Group type and distribution type nodes contain child nodes for the contained pedestrian types and provide an action to add new child types.

- OD-maps category nodes, contain child nodes for the od-maps of a project and provide an action to create a new od-map.
- OD-map nodes, represent od-maps, allow to open an od-map editor and provide actions to delete, rename and duplicate the associated od-map.

The project explorer furthermore contains buttons triggering actions to

- open an infrastructure editor for the selected project,
- create a new project,
- create new project elements of respective types, depending on the current selection in the project tree, and
- collapse or expand all tree nodes.

Scenario Explorer

Projects can contain multiple scenarios. The scenario explorer consists of a tabbed panel with every scenario of one selected project being presented within a tab as a *scenario tree* as illustrated in Figure 4.3. The content of the scenario explorer is swapped whenever the project focused by the project explorer (Section 4.3.2) changes, i.e. a node of a different project tree gets selected. The scenario explorer uses the types of element and category nodes similar to the Project Explorer. Element nodes in general provide actions to change the selection of the Project Explorer to the related element nodes in a project tree.

- Section nodes, represent sections of the project. As sections are defined per project, every scenario node contains section nodes for the same set of sections. Section nodes provide an action to select layouts, that is let the user select the vector layout, grid layout and graph layout to be used for the section in the selected scenario. At most one layout per type can be chosen from the complete list of project layouts.
- Layout nodes for vector layouts, grid layouts and graph layouts used by the selected scenario in the containing section. Layout nodes allow to open appropriate layout editors and also provide an action to change the scenario's layout set.

- Facilities category nodes, contain child nodes for the facilities referenced by any layout of the containing scenario.
- Facility nodes, represent facilities that are referenced by one or more layouts of the scenario.
- Basin nodes, represent basins and occur as child nodes of facility nodes for basins that are part of a facility, and of basin category nodes for all basins of a scenario. Context information, i.e. the containing section and optionally facility, which is not given by the parent node is contained in the node labels.
- Tooling regions category nodes contain child nodes for the tooling regions referenced by any layout of the containing scenario.
- Tooling region nodes, represent tooling regions that are referenced by one or more layouts of the containing scenario and have child nodes for the tools of the tooling region.
- Tool nodes for the tools of a tooling region.
- Basins category nodes, contain child nodes for basins referenced by any layout of the containing scenario.
- Pedestrian types category nodes, contain child nodes for all pedestrian types of the project. Pedestrian types that do not have an od-matrix associated by the od-map of the scenario are displayed faded to indicate that they are currently not relevant for the scenario. Still they are displayed, because od-matrices could be inserted for nested pedestrian types.
- Pedestrian type nodes, represent pedestrian types (atom, group or distribution) of the project.
- OD-map category nodes, contain at most one child node for the od-map used in the scenario and provide an action to select the scenario's od-map from the list of od-maps of the project.
- OD-map nodes. At most one od-map node is contained in a scenario tree to represent the od-map selected for the scenario. OD-map nodes allow to open an od-map editor and provide actions to select another od-map for the scenario and to change the od-map to a newly created duplicate of the current one.

The scenario explorer also contains buttons for the following actions related to the scenario represented by the currently active scenario tab:

- Export scenario
  Generates XML-data corresponding to the GPSI specification for the scenario and saves it in a file specified by the user.

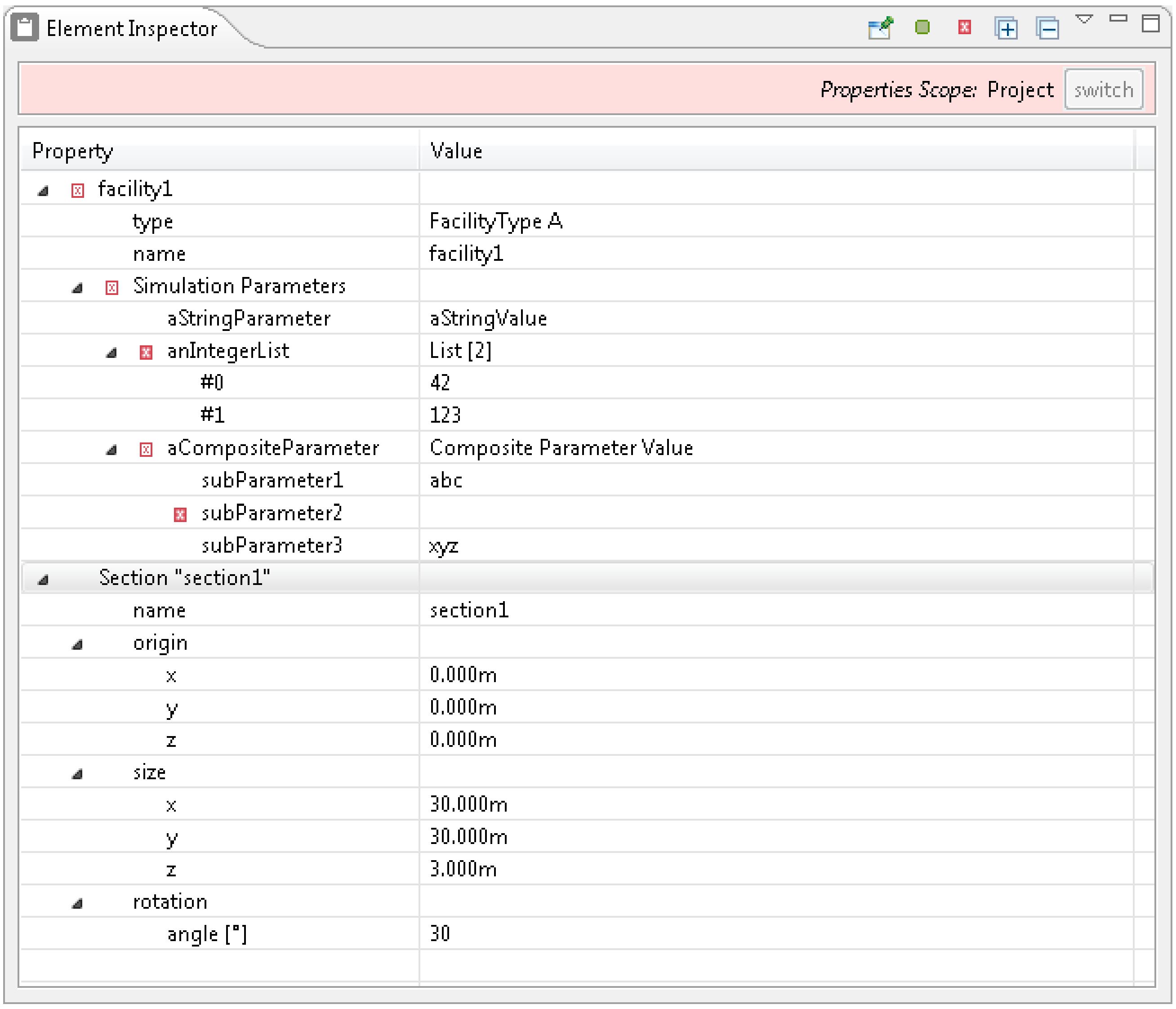
- Focus Element Inspector on scenario
  As scenarios are not represented by nodes but by tabs containing scenario trees, this action is used to let the Element Inspector (Subsection 4.3.3) focus on the active scenario.
- Collapse / expand all tree nodes
- Create new scenario
- Duplicate scenario
  Creates a new scenario as a duplicate of the active one as a starting point for alternative simulation configurations.
- Rename scenario
- Reorder scenario
  Scenarios are presented in a user-defined ordering within the scenario explorer. There are actions to move the active scenario to the left and to the right.
- Delete scenario
- Open infrastructure editor
  Opens an Infrastructure Editor (Subsection 4.3.6) that shows only basins and basin connections that are relevant for the active scenario.

4.3.3 Element Inspector

The element inspector is a generic view that switches its content to support the currently active part of the user interface. It enables the user to view and edit properties and inner structures of project elements selected in the active view. The property sheets of the Element Inspector are not plain name-value tables, but structured as trees of nested elements and properties. In general, the Element Inspector supports multi-selection, i.e. every selected element of the active view is displayed as the root of a property tree.

Figure 4.4 shows a screenshot of the Element Inspector with a facility and a section selected (for instance in the Project Explorer as currently active view). In the example, all nodes of the properties sheet are expanded. The structure of the property trees depends on the type of the inspected element and the parameterization applied by the metadata definition used for the project. The nesting of tree nodes is visualized in the left column of a 2-column table. The right column shows property values and content descriptions for some nodes that are roots of a subtree (e.g. `List[2]` for a value list with 2 entries or `Composite Parameter Value` for the container node of a parameter binding with a composite parameter value).

When a changable value entry in the right column is clicked, an appropriate input control gets activated to enable the user to edit the value. String values and numerical values are input using a textfield (see Figure 4.5). Input validation is performed in real-time while the value is typed, based on the required data type and value constraints (e.g. `min`, `max` of `IntegerParameterType`

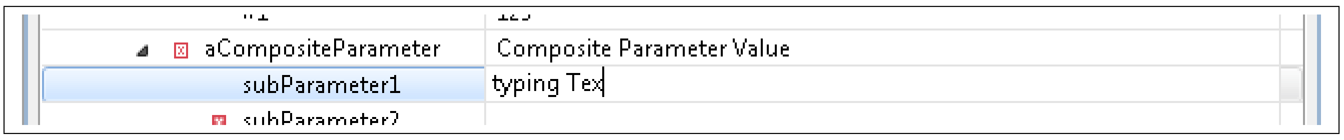

Element Inspector

Properties Scope: Project switch

Property	Value
facility1	
type	FacilityType A
name	facility1
Simulation Parameters	
aStringParameter	aStringValue
anIntegerList	List [2]
#0	42
#1	123
aCompositeParameter	Composite Parameter Value
subParameter1	abc
subParameter2	
subParameter3	xyz
Section "section1"	
name	section1
origin	
x	0.000m
y	0.000m
z	0.000m
size	
x	30.000m
y	30.000m
z	3.000m
rotation	
angle [°]	30

Figure 4.4: Element Inspector

or `minLength`, `maxLength` and `pattern` of `StringParameterType`, see Section 3.4.4). Boolean and enumeration values are specified using a selection box (see Figure 4.6). Hovering over entries of metadata-defined parameters shows the related type information. The Element inspector also makes comprehensive use of a right-click context menu to enable further manipulation of properties, especially values of non-primitive types like of composite parameters, reference parameters, value lists and probability distributions, see Section 4.3.3.



aCompositeParameter	Composite Parameter Value
subParameter1	typing Tex

Figure 4.5: Element Inspector, text field for string and numerical values

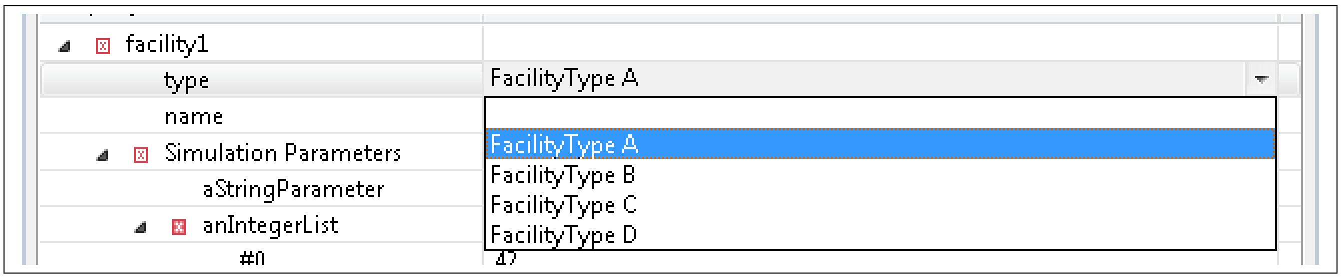


Figure 4.6: Element Inspector, selection box for enumeration values

View Scopes

Section 4.2 explains how element parameter values can be overridden in scenarios. The means to view and edit parameter values is the Element Inspector. As a project element can have multiple values assigned for the same parameter – one base value defined by the project and at most one override value for each scenario – Scenario Builder uses the concept of *view scopes* to define the context the Element Inspector displays parameterization for. The Element Inspector is either set to *project scope* to display project-defined parameter values , or to *scenario scope* of one certain scenario to display scenario-defined variations. The colored bar above the property sheet (Figure 4.4) shows the current scope - light red means project scope, green means scenario scope with the name of the scenario also displayed. The *switch* button can be pushed to change the view scope. When an editor view is active, a selection box in a global toolbar enables the choice of either project scope or a scenario to set scenario scope for the editor. The scope button of the Element Inspector is only enabled if the selection for the Element Inspector belongs to the Scenario Explorer (and therefore a certain scenario), or to a *scenario-scoped* editor.

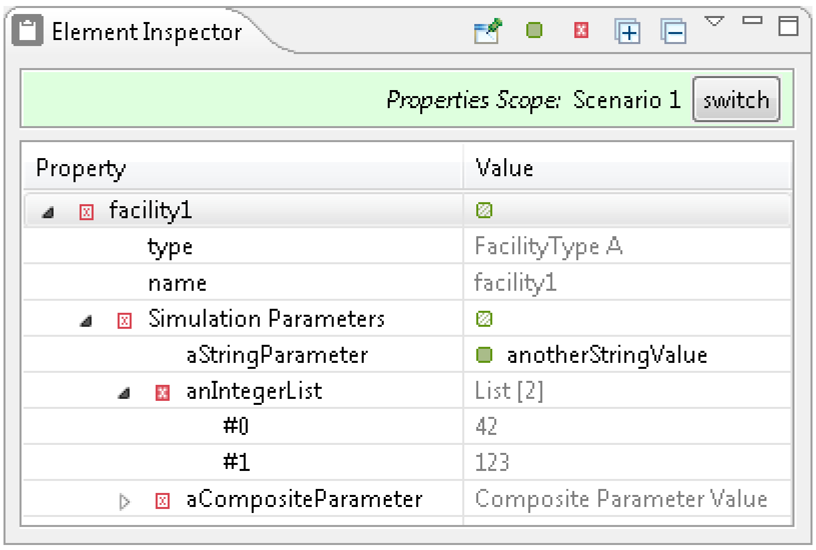
Figure 4.7 shows an example of the Element Inspector switched to scenario scope. Overridden values are marked with a small green square in the right column. Tree nodes with overridden values for any child in their subtrees are marked with small green striped squares with white background.

Problem Indication

Problems (see also Subsection 4.3.8 associated with elements and values displayed by the Element Inspector are indicated by red problem markers in the left column. Tree nodes with nested problems, i.e. problem entries in their subtrees are marked with problem markers with white background. Hovering over entries with problem markers shows information about the indicated problems, see Figure 4.8.

Probability Distributions

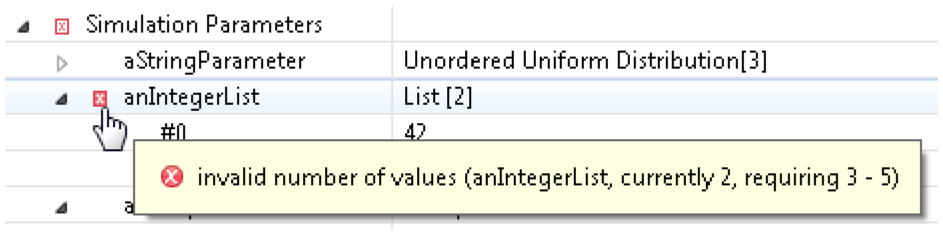
The GPSI specification allows values of metadata-defined parameters to be substituted by appropriate probability distributions (Section 3.4.4). The user interface support for that feature



Properties Scope: Scenario 1 switch

Property	Value
facility1	
type	FacilityType A
name	facility1
Simulation Parameters	
aStringParameter	anotherStringValue
anIntegerList	List [2]
#0	42
#1	123
aCompositeParameter	Composite Parameter Value

Figure 4.7: Element Inspector, scenario scope



Simulation Parameters	
aStringParameter	Unordered Uniform Distribution[3]
anIntegerList	List [2]
#0	42

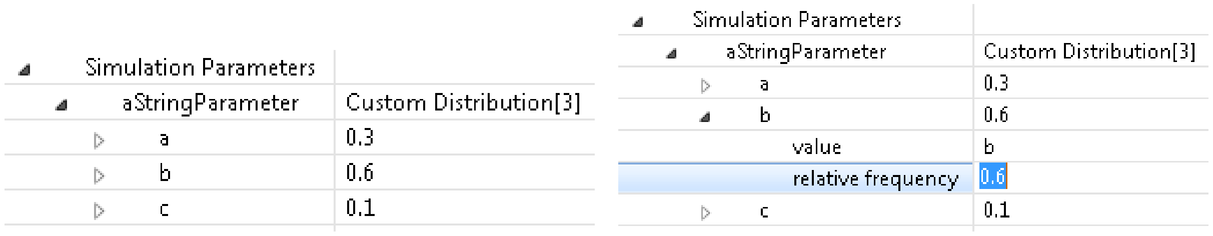
invalid number of values (anIntegerList, currently 2, requiring 3 - 5)

Figure 4.8: Element Inspector, problem indication

is provided by the Element Inspector. The context menu (Section 4.3.3) of a distributable value offers a list of applicable distribution types to use as replacement for the value, see Figure 4.11. Distributed values are displayed in the Element Inspector using subentries to define

- the lower and upper bounds of the distribution support and the distribution parameters for distributions with ordered support (Figure 4.10, `anIntegerParameter` and `aReal-Parameter`),
- the list of support values for uniform distributions of unordered support (Figure 4.10, `aStringParameter`) , and
- the list of support values and the respective relative frequencies for unordered custom distributions (Figure 4.9)

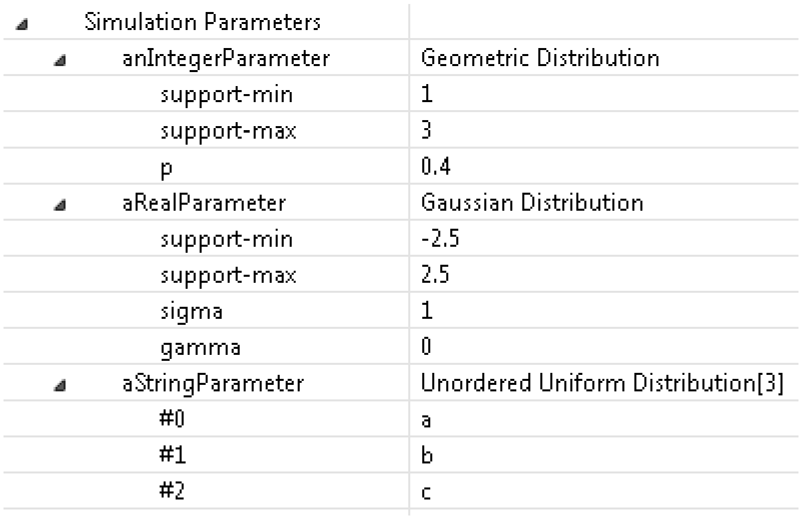
The Element Inspector includes a feature to manually fine-tune distributions based on *well-known distributions* specified by the GPSI, see Table 3.3. An example workflow is illustrated by Figure 4.11, Figure 4.12 and Figure 4.13, where an integer list entry (`123`) is replaced by a binomial distribution. Subsequently, the distributed value is transformed to a custom distribution, i.e. the relative frequencies are calculated in the range of the specified distribution



Simulation Parameters	
aStringParameter	Custom Distribution[3]
a	0.3
b	0.6
c	0.1

Simulation Parameters	
aStringParameter	Custom Distribution[3]
a	0.3
b	0.6
value	b
relative frequency	0.6
c	0.1

Figure 4.9: Element Inspector, custom distribution



Simulation Parameters	
anIntegerParameter	Geometric Distribution
support-min	1
support-max	3
p	0.4
aRealParameter	Gaussian Distribution
support-min	-2.5
support-max	2.5
sigma	1
gamma	0
aStringParameter	Unordered Uniform Distribution[3]
#0	a
#1	b
#2	c

Figure 4.10: Element Inspector, ordered discrete, ordered continuous and unordered uniform distributions

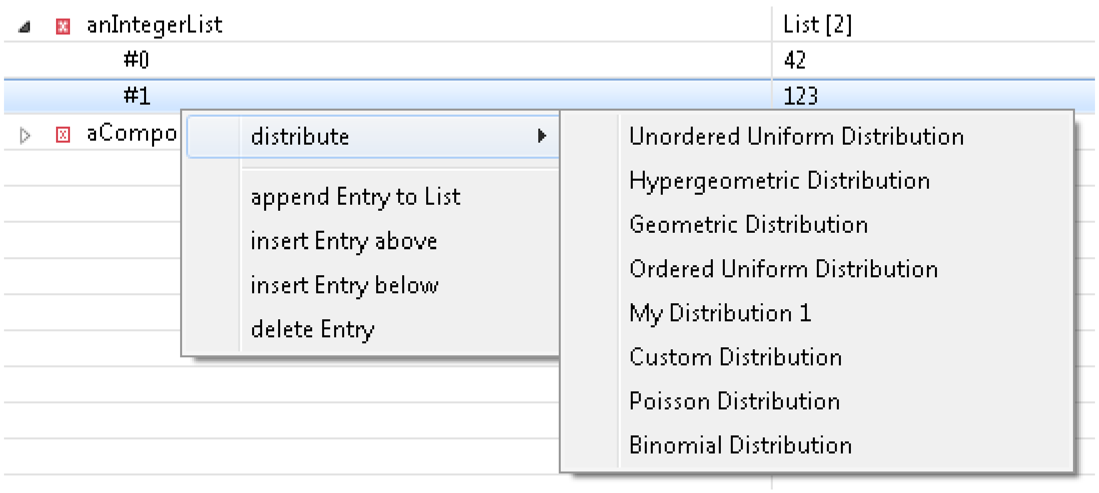
support. This is only possible for discrete distributions and some maximum support range, see Section 4.3.3. Finally, the value of the entry is "reverted" to the originally applied distribution (which is remembered when a distribution is "customized").

Context Menu Actions

The context menu of the Element Inspector provides actions related to the following functional categories.

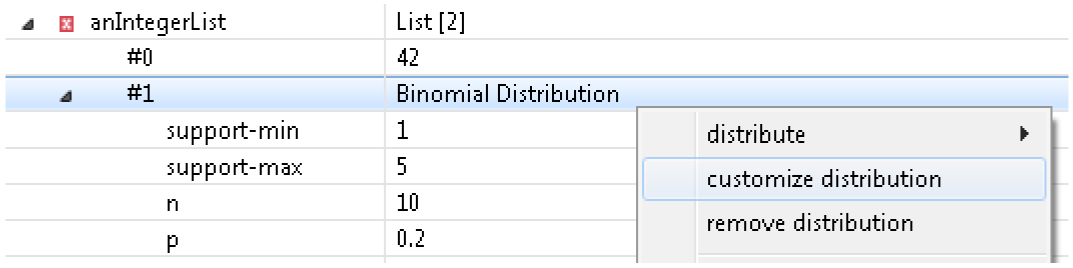
Navigation. Select a project element in the Project Explorer, center the currently active Layout Editor on a project element, open appropriate editors for project elements.

Layout-Item assignment. Assign/unassign an item to/from layout elements currently selected in an the active layout editor (see Subsection 4.3.5), unassign a certain item or all items of a type from a layout element.



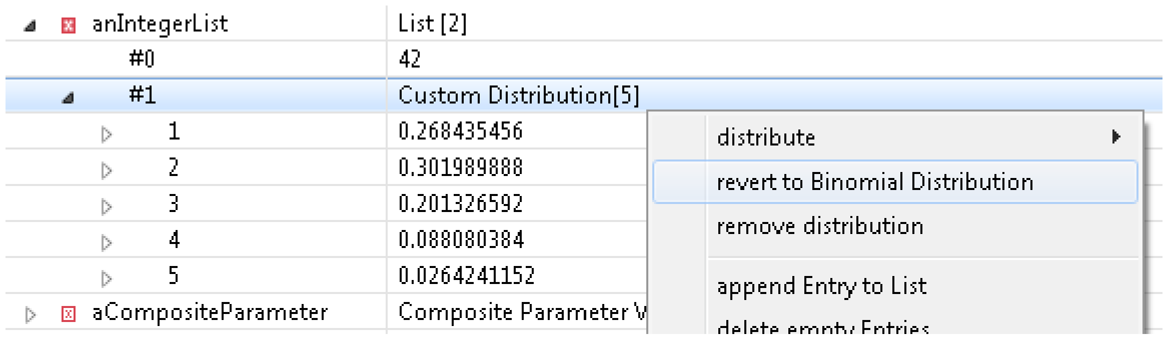
anIntegerList	List [2]
#0	42
#1	123
aCompo	

Figure 4.11: Element Inspector, substitute a parameter value by a probability distribution



anIntegerList	List [2]
#0	42
#1	Binomial Distribution
support-min	1
support-max	5
n	10
p	0.2

Figure 4.12: Element Inspector, customize a probability distribution



anIntegerList	List [2]
#0	42
#1	Custom Distribution[5]
1	0.268435456
2	0.301989888
3	0.201326592
4	0.088080384
5	0.0264241152
aCompositeParameter	Composite Parameter V

Figure 4.13: Element Inspector, revert to the original distribution

Layout geometry manipulation. Scale, rotate, convert and reorder shapes; delete vertices or shapes. Layout manipulation using the Element Inspector can also efficiently be done by editing vertex coordinates in the property sheet.

Probability distributions. Distribute values and remove distributions; customize and revert distributions (see Section 4.3.3).

List manipulation. Append, insert and delete list entries; delete all / all empty list entries.

Scenario overrides. Override a value in scenario scope; remove an override value.

Special functionality for certain entry types. Assign reference values; select background layouts and background images of layouts (Subsection 4.3.5).

4.3.4 Toolbar Actions

The Element Inspector also contains some common view-related toolbar actions:

- Pin view, prevents the current property sheet from being changed by a new selection.
- Expand scenario overrides, expands property trees to expose all overridden values in scenario scope.
- Expand problems, expands property trees to expose all problem entries.
- Expand all, collapse all nodes.

4.3.5 Layout Editors

Layout editors are used to design the geometry of layouts within sections. There are three types of layout editors, addressing vector-, grid- and graph-based layouts respectively. Though they mainly implement functionality dedicated to the respective layout type, the different types of layout editors share some commonalities.

All types of layouts contain layout elements – shapes, cells or nodes – that are presented graphically in 2-dimensional space. Layouts are located in sections by defining an offset relative to the section origin and the dimension of a bounding rectangle, referred to as the *layout bounds*. The coordinate system of a layout is constituted by the section bounds, i.e. layout definition happens relative to sections. Layout editors display the layout's coordinate system aligned paraxial to the screen. Therefore, rotating a section does not affect the visualization of layout editors (section rotations are visible only in the Infrastructure Editor, Subsection 4.3.6).

Layout elements basically define parts of the simulation space, that are given their relevance for simulations by being marked as *obstruction* or linked to facilities, basins or tooling regions. These options can generally be combined. Linking and unlinking items and layout elements is referred to as *assigning* and *unassigning items* and is done using the Project Explorer or the Scenario Explorer, via drag and drop or context menu. Layout editors are aware of selections within other views and show feedback in terms of highlighting layout parts that correspond to selected project elements. Also, layout editors can be centered on layout elements that have selected project elements assigned. All types of layout editors allow zooming

- in and out,
- to fit the layout bounds,
- to fit the section bounds,
- to fit the current selection, and
- to a default zoom level.

Layout editors use two supportive views that can be arranged the same way as the other view parts explained in this section, the *Tools Palette* and the *Editor Properties* view. The specific content of the supportive views depends on the type of the currently active layout editor. The *Tools Palette* enables the user to select the tool to be used for layout editing as it is common for graphical editors. The *Editor Properties* view presents information and options related to the active editor like absolute and relative position coordinates and snapping options.

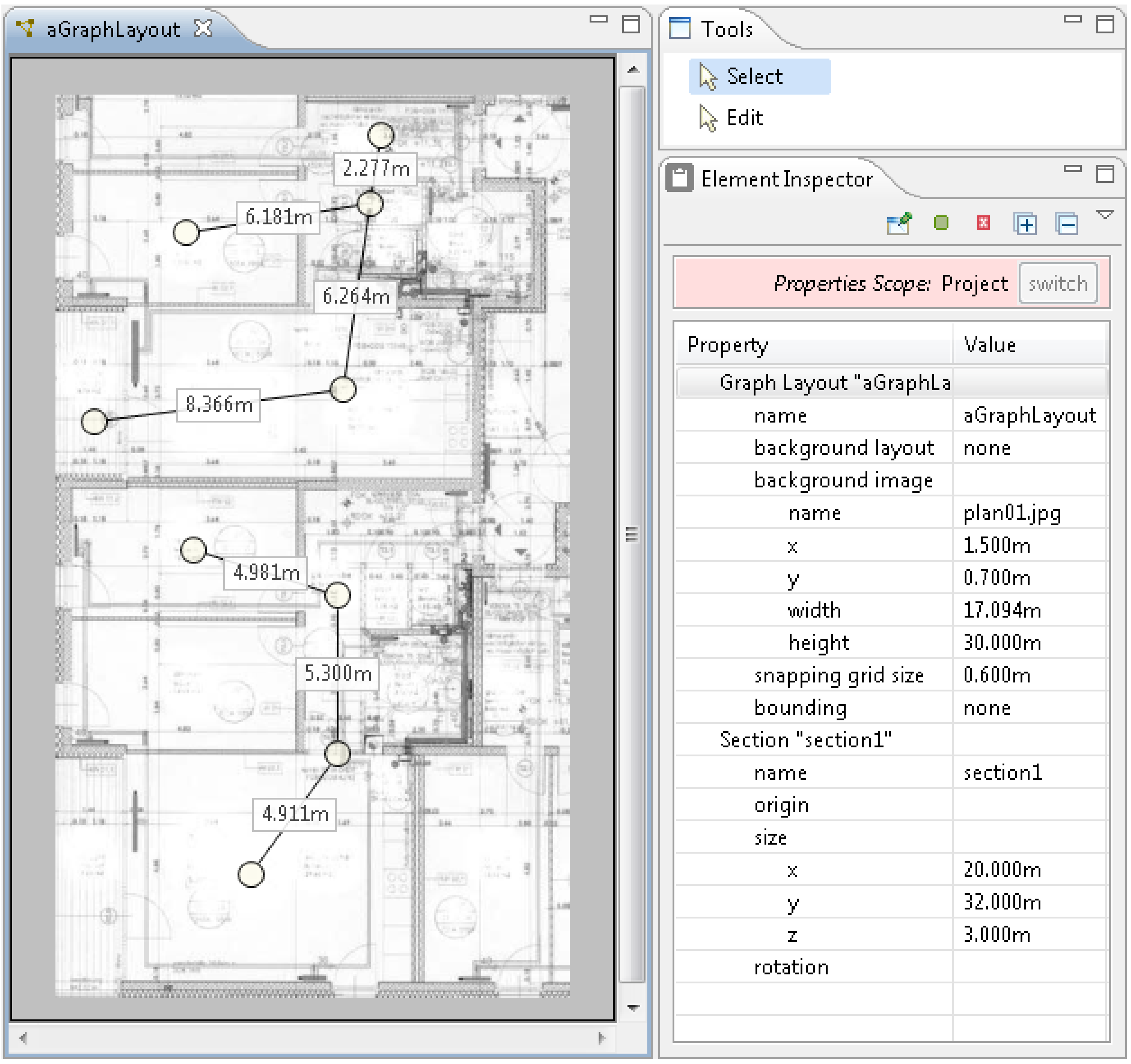
The requirement of geometry prototyping was stated in Subsection 4.1.3. It is met by the capability of all layout editors to optionally display the edited layout against a selectable background image or another layout of the same project (of any type). The background image or background layout are selectable in the Element Inspector. The background image can furthermore be located and scaled using the Element Inspector, see Figure 4.14. A background layout is always located using the project's world coordinate system.

Vector Layout Editor

The layout elements of vector layouts are called *regions*. Possible shape types of regions are rectangles, polygons and polylines, for each of which a tool exists in the tools palette together with a *selection tool*. A small example layout is shown in Figure 4.15.

A Vector Layout defines one *bounding region*, that is a polygon or a rectangle defining the bounds of the walkable space of the layout. The bounding region cannot be deleted and is visualized as a white shape below the layout elements.

Facilities and tooling regions assigned to regions occupy the whole filling area of a polygonal or rectangular shape or all line segments of a polyline shape respectively, whereas basins are assigned to a set of line segments of the bounding of polygons or rectangles or parts of a polyline. The obstruction state of a region can be changed using the context menu of the selection tool or by using the Element Inspector.



Property	Value
Graph Layout "aGraphLa	
name	aGraphLayout
background layout	none
background image	
name	plan01.jpg
x	1.500m
y	0.700m
width	17.094m
height	30.000m
snapping grid size	0.600m
bounding	none
Section "section1"	
name	section1
origin	
size	
x	20.000m
y	32.000m
z	3.000m
rotation	

Figure 4.14: Layout editor using a background image

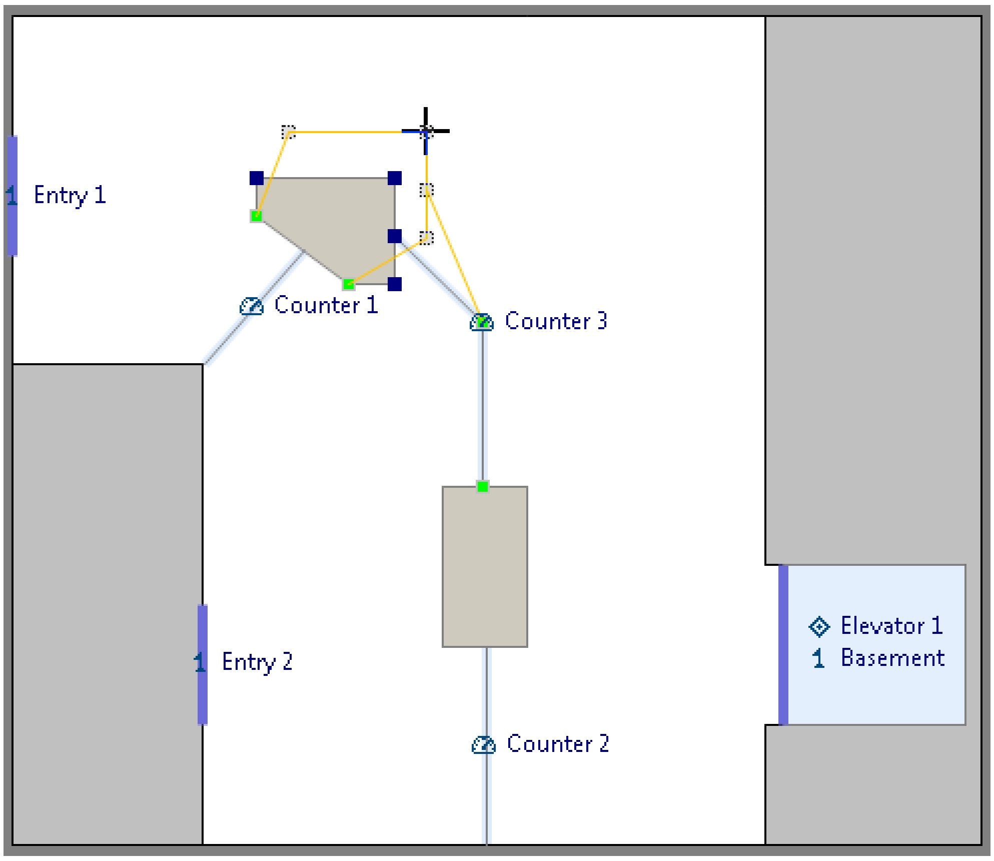


Figure 4.15: Vector Layout Editor

A newly created region is filled yellow, meaning has not assigned any items nor is it an obstruction. Regions with items assigned are colored blue, obstruction regions are colored gray or grayblue if they have items assigned. Line segments with basins assigned are bold and dark blue. The filling of a region uses transparency to indicate overlapping parts of shapes by darker fill areas. Item assignments are also indicated by labels of shapes showing a type icon and the item name.

The section bounds (the xy-projection of the section containing the edited layout) are visualized as a rectangle in lighter gray below the bounding region, in a fresh layout the bounding region exactly covers the section bounds.

The Element Inspector of a vector layout contains entries to define a background grid to support layout construction (not to be confused with the grid of a grid layout, Section 4.3.5). The shape creation tools support snapping of the pointer to that grid as well as to line segments and vertices of shapes, and movement constraining to multiples of 45°.

The selection tool is used to

- manipulate existing layout regions, i.e. move shapes and parts of shapes; add or remove vertices to polygons and polylines; resize rectangles; delete, cut, copy and paste regions; change item assignments; change obstruction state; change the z-order of shapes; scale and rotate shapes; and to

- change the viewport's location and zoom factor, using marquee zoom and panning.

Grid Layout Editor

A grid-based layout is basically created by defining for each cell of the grid

- whether it represents an obstruction, and
- the set of assigned items.

The placement of the grid is defined relative to the containing section's bounds specifying the origin location, the cell size, the number of cells in horizontal and vertical direction and the grid rotation. The section bounds are visualized as a rectangle in lighter gray below the grid. The grid is always displayed paraxial to the screen, so if a non-zero grid rotation is set, the inverse rotation is applied to the section bound's rectangle in the Grid Layout Editor.

The Grid Layout Editor uses the same color encoding as the Vector Layout Editor to visualize obstruction state and item assignments but item labels are not used. Basin cells are bordered dark blue.

The tools palette of the Grid Layout Editor contains a *selection tool* and four drawing tools: *free draw*, *line draw*, *filled rectangle* and *rectangular outline*. The drawing tools allow to change cells in the grid. The Editor Properties view is used to choose between drawing obstruction or items currently selected in the Project Explorer. The selection tool is used to select and manipulate cells in the grid. Rectangular selections are possible by clicking and dragging, *flood-select* is available via context-menu to recursively select similar neighbor cells (exactly, a neighbor cell is added to the selection, if the starting cell is an obstruction cell and the neighbor cell is an obstruction, or if the neighbor cell references at least one item that is also referenced by the starting cell). Multiple selections can be combined additive and subtractive, configurable in the Editor Properties View. Selected cells can be manipulated using the context menu.

Layout transformation. If a vector layout is set as background layout for the edited grid layout, actions are available in the Grid Layout Editor to perform layout transformation (introduced in Subsection 4.1.3). Full transformation is possible to completely recreate the cell states based on the background layout and the relative grid location and orientation, see Figure 4.17 Partial transformation is available for selected cells to apply the transformation algorithm only on the selected cells.

Graph Layout Editor

Graph layouts are linked graphs with optionally items assigned to its nodes and a distance property and arbitrary user-defined properties of its edges, called *links* as they represent links for pedestrian movement between locations defined by nodes. The default distance property

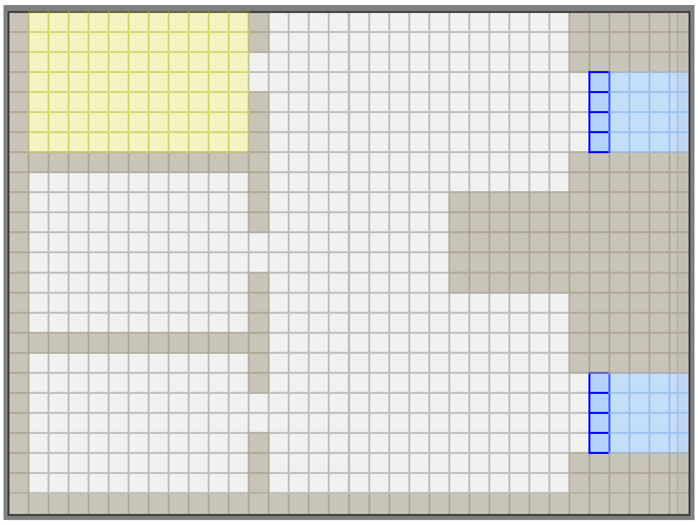


Figure 4.16: Grid Layout Editor

of a link is calculated by the Graph Layout Editor, but it is possible to define a manual value using the Element Inspector to be used within scenario export data instead (see Figure 4.19). Nodes can have facilities, tooling regions and basins assigned using the Project Explorer or the Scenario Explorer via drag and drop or context menu. Nodes are visualized using the same encoding of the fill color as the Vector Layout Editor (Section 4.3.5).

The Tools Palette of the Graph Layout Editor contains a *selection tool* to manipulate parts of the layout's graph and an *edit tool* to create new nodes and links.

The graph of a graph layout does not define spatial boundaries of the layout, instead these can optionally be added by selecting a *bounding region* of a vector layout in the Element Inspector to be used for the visualization of the layout editor.

4.3.6 Infrastructure Editor

The Infrastructure Editor provides a top-level view on the simulation's space. An Infrastructure Editor instance is opened with a selection of layouts, with at most one layout per section. The selected layouts provide the spatial information for the Infrastructure Editor's view on the project's sections. It is not necessary to include layouts of all sections, selecting only layouts of a subset of the project's sections possibly provides a clearer view on the infrastructure. Opening an Infrastructure Editor from within the Scenario Explorer pre-selects layouts assigned to the scenario.

The Infrastructure Editor uses a two-dimensional presentation of the sections. As sections are arranged in three dimensions (by absolute origin and dimension), the z-coordinate is encoded using so-called *z-panes*. Z-panes are rectangular two-dimensional areas containing layout visualizations of sections with equal origin z-coordinates. All z-panes of one Infrastructure Editor instance are of equal size, resulting from merging the bounds of the xy-projection of all

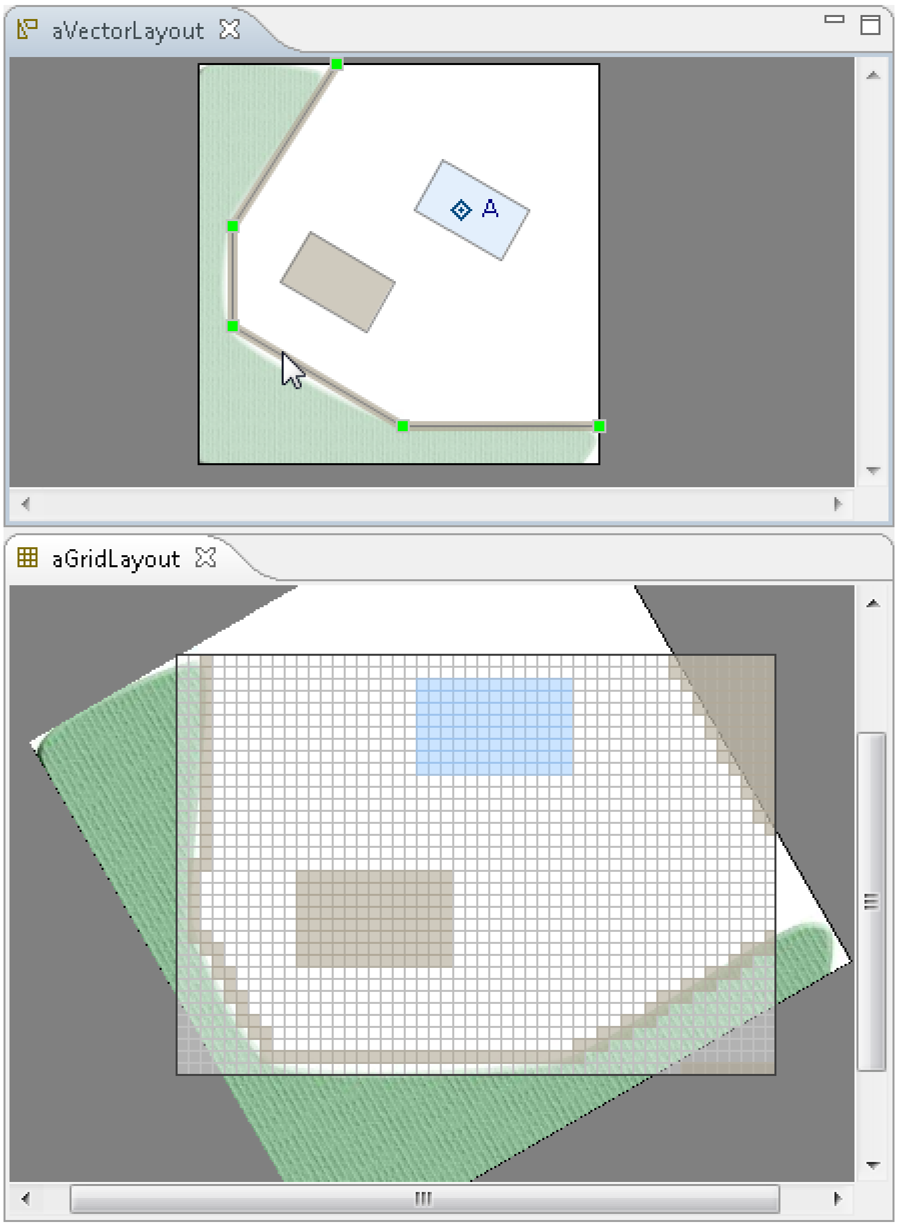


Figure 4.17: Grid Layout Editor, rotated grid

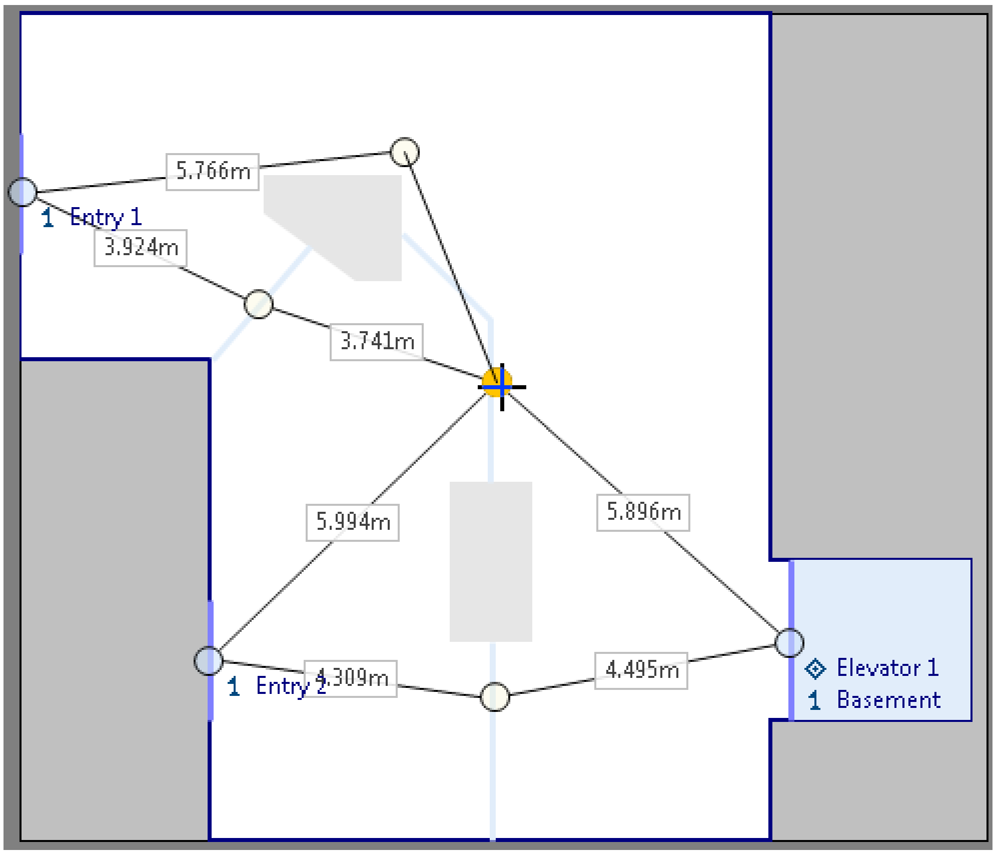


Figure 4.18: Graph Layout Editor

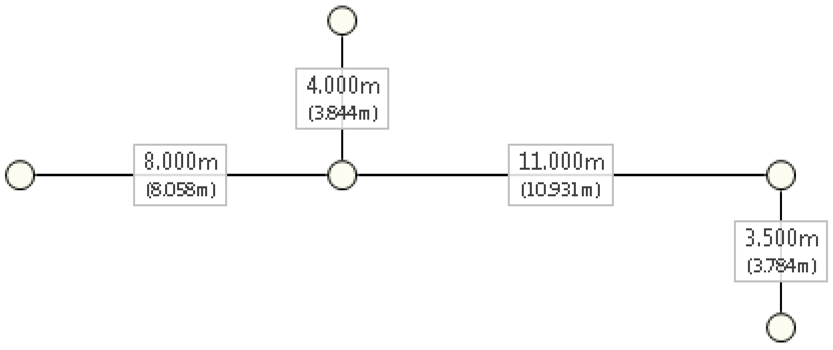


Figure 4.19: Graph Layout Editor, manual link distances

presented Sections (i.e. the bounds of the z-panes are the bounds of the part of the simulation space defined by the presented sections). The z-panes are arranged vertically or horizontally (selectable in the Editor Properties view), so that the z-dimension is visualized by the assignment of each Layout to the z-pane corresponding to its section's z-coordinate. Figure 4.20 shows the Infrastructure Editor with three sections in two z-panes (presenting two floors). Figure 4.21 shows the Infrastructure Editor with a corridor between two rooms modelled as its own section and rotated within the simulation space.

Within z-panes, section bounds are presented as dark-gray, possibly rotated (if the Section has a rotation angle set) rectangles. Bounding regions and grid bounds are visualized as shapes with white filling. Layout content is displayed using the same coloring as within Layout Editors. Basins referenced by layout elements are indicated by green nodes. If multiple basins are referenced by a layout using identical spatial configuration (the same line segments in vector layouts, the same cells in grid layouts, the same node in graph layouts), they are consolidated within one single node in Infrastructure Editors. The display of basin names next to nodes can be toggled in the Editor Properties view.

Connections between nodes represent inter-section-connections of basins. As one node can represent multiple Basins and as basin connections are directed, one connection can represent multiple basin connections. Double-clicking a connection or using the context menu opens a dialog enabling the configuration of basin connections possible for the connection. Also, selecting a connection in the Infrastructure Editor lets the Element Inspector list the represented basin connections.

If nodes are selected, unselected nodes already connected to one or more of the selected nodes are highlighted light blue and unselected nodes not connected but connectable to any of the selected nodes (i.e. there are basins represented by a node that are connectable to basins of the selected nodes), are highlighted light red. If two nodes are selected, the context menu offers an action to configure basin connections, bringing up the above mentioned dialog.

4.3.7 OD-Map Editor

The OD-Map Editor is used to

- define the dimensions of od-matrices by selecting origin and destination basins,
- assign od-matrices to pedestrian types, and
- select od-matrices to edit their properties in the Element Inspector.

The OD-Map Editor is organized using two pages, selectable by tabs below the editor area.

The basins page of the OD-Map Editor (Figure 4.22) lists available source- and sink-basins on the left side and basins already chosen to be part of the od-map's matrices on the right side. Buttons between the "available"- and "included"-tables are used to add, remove and reorder Basins. Multiple selections are possible in each of the tables. The od-matrices page of the

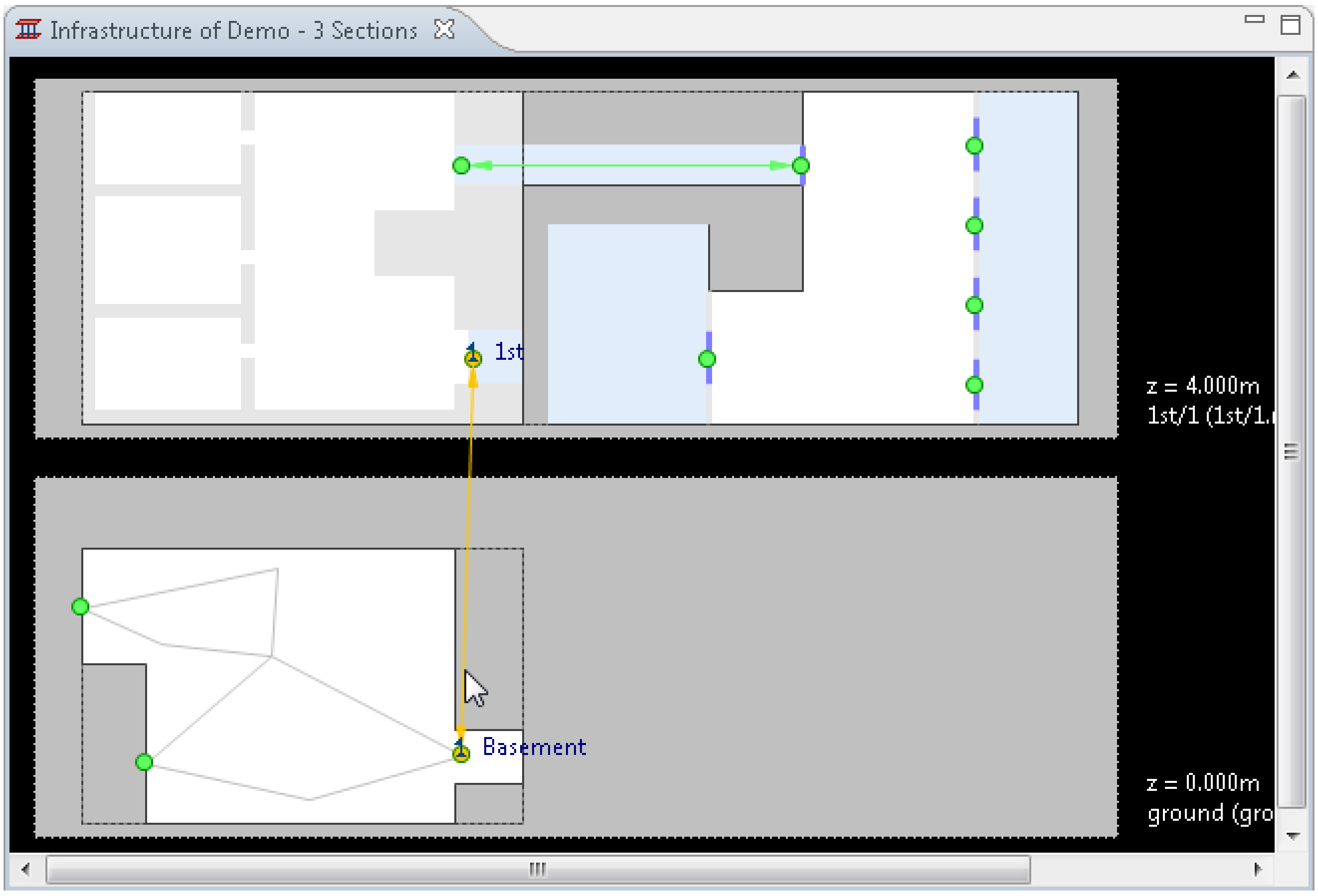


Figure 4.20: Infrastructure Editor, 3 sections on two floors

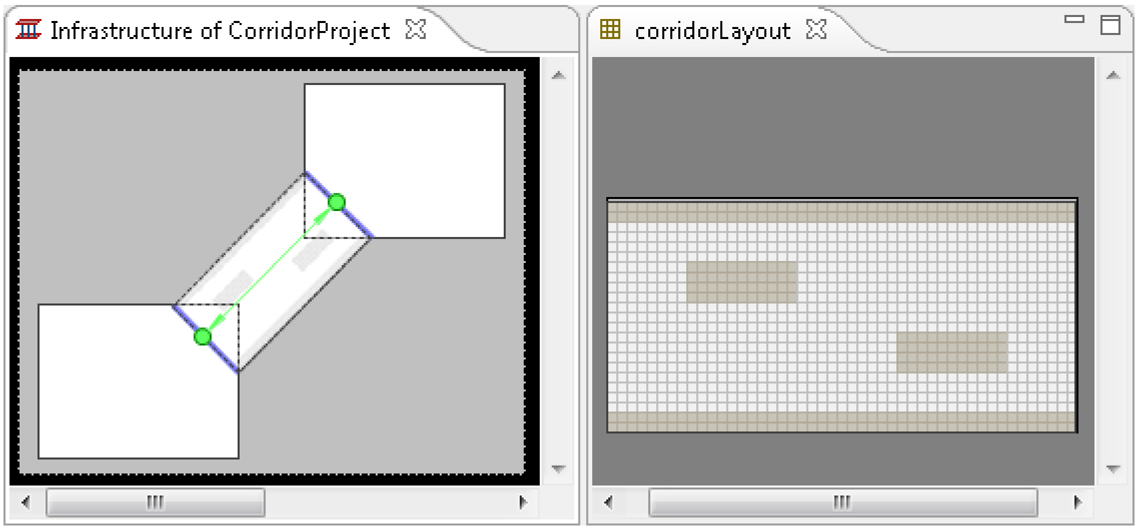
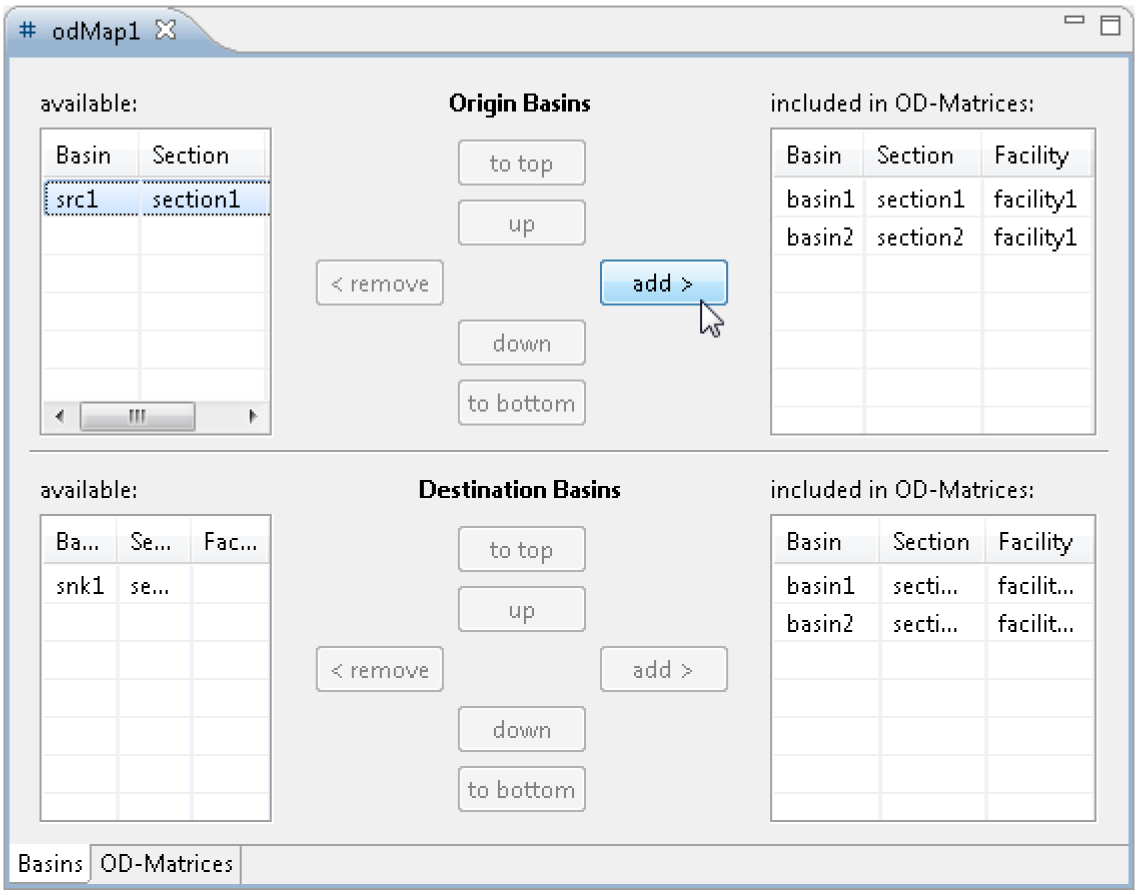


Figure 4.21: Infrastructure Editor, rotated section



Origin Basins

available:

Basin	Section
src1	section1

included in OD-Matrices:

Basin	Section	Facility
basin1	section1	facility1
basin2	section2	facility1

Destination Basins

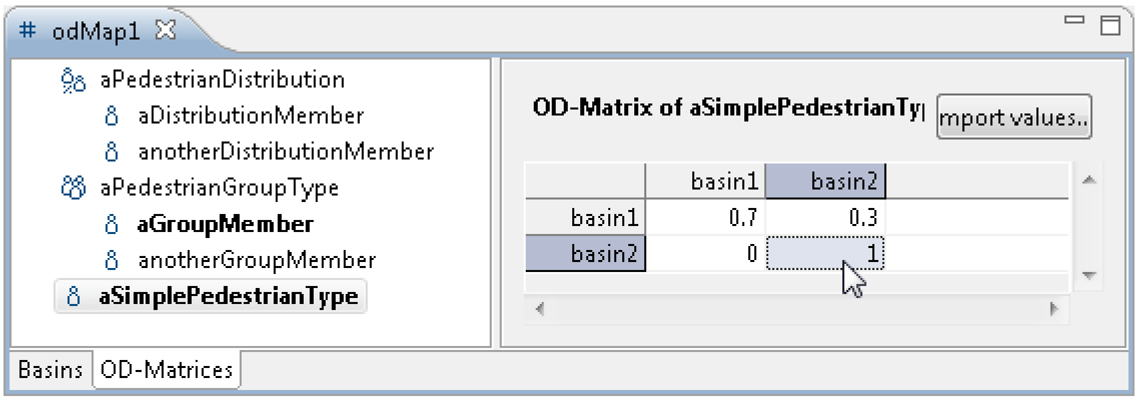
available:

Ba...	Se...	Fac...
snk1	se...	

included in OD-Matrices:

Basin	Section	Facility
basin1	secti...	facilit...
basin2	secti...	facilit...

Figure 4.22: OD-Map Editor, basins selection



OD-Matrix of aSimplePedestrianTy|

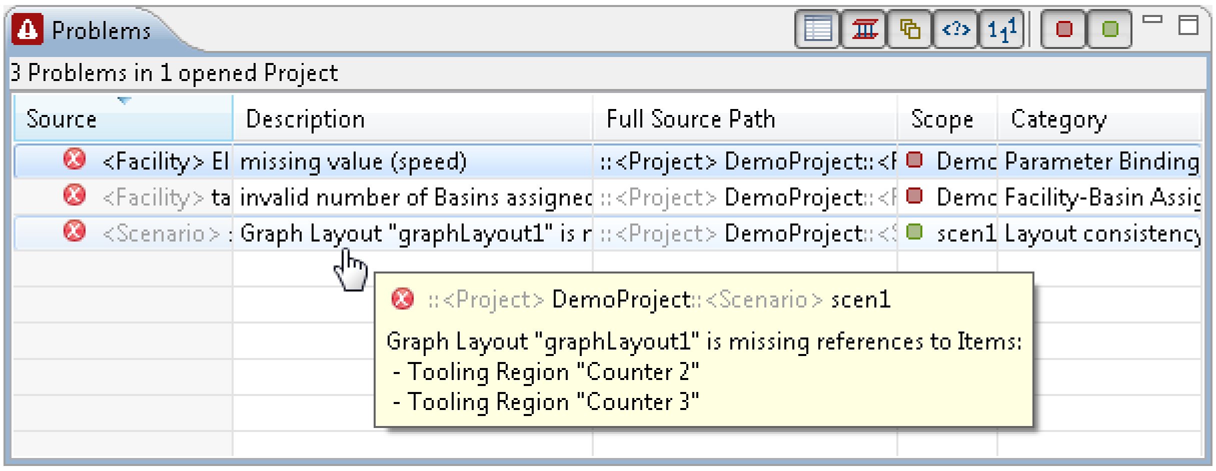
	basin1	basin2
basin1	0.7	0.3
basin2	0	1

Figure 4.23: OD-Map Editor, od-matrices definition

OD-Map Editor (Figure 4.23 displays the hierarchy of pedestrian types using bold entries for types with assigned od-matrices. New od-matrices can be created and imported from CSV-files for selected entries in the type hierarchy. Selecting a pedestrian type focuses the Element Inspector on the assigned od-matrix to edit the matrice's metadata-defined properties, e.g. timing information.

4.3.8 Problems View

The Problems View lists all validation problems detected in projects that are not closed. The columns of the problems table can be rearranged and selected for sorting. Double-clicking a table row focuses the Project Explorer or the Scenario Explorer, depending on the problem's scope, on the problem source. Hovering on a problem entry shows detailed information about the problem in a tooltip.



3 Problems in 1 opened Project

Source	Description	Full Source Path	Scope	Category
<Facility> El	missing value (speed)	::<Project> DemoProject::<F	🟥 Demc	Parameter Binding
<Facility> ta	invalid number of Basins assignec	::<Project> DemoProject::<F	🟥 Demc	Facility-Basin Assig
<Scenario> :	Graph Layout "graphLayout1" is r	::<Project> DemoProject::<S	🟩 scen1	Layout consistency

::<Project> DemoProject::<Scenario> scen1
Graph Layout "graphLayout1" is missing references to Items:
- Tooling Region "Counter 2"
- Tooling Region "Counter 3"

Figure 4.24: Problems View

4.3.9 Problem Categories

Problems within Scenario Builder projects are categorized as follows.

- Parameter problems
  Missing parameter values or values violating constraints defined in metadata currently used by the project.

- Basin assignment problems
  Invalid number of basin assignments per section for a facility (as defined by the `minNumBasins` and `maxNumBasins` attributes of `facility` in metadata files)

- OD-map / infrastructure problems
  A scenario has no od-map assigned or the od-map cannot be satisfied by the scenario's infrastructure.

- Layout consistency problems
  The layouts of a acenario do not reference a consistent set of items.

- Type problems
  The project's metadata definition does not define the type of a facility or tool. This type of problem can occur when metadata has been switched for a project.

The problems table contains the following columns.

- Source, the project element causing the problem
- Description, a description of the problem
- Full source path, the fully qualified path to the problem source, useful e.g. for nested parameter values or pedestrian types in deep hierarchies
- Scope, the scope of the problem (project or scenario)
- Category, the problem category

The buttons in the Problems View's toolbar are used to filter the content of the problems table. Problems can be filtered by category and by scope.

4.4 Base Technologies and Libraries

This section lists the frameworks and third-party libraries the Scenario Builder implementation depends on. The most essential decision is to base the application on the *Eclipse Rich Client Platform* framework [RCP, ML05] due to its well-suited application model, feature-rich and extensible component library, flexible license restrictions and inherent platform independence. Java (JDK Version 6) [Javb] is therefore chosen as programming language.

4.4.1 Eclipse RCP

[Rub06] gives an overview of the issues addressed by the *Eclipse Rich Client Platform*, an extensible open source framework for general-purpose desktop applications. Eclipse RCP provides the infrastructure of the *Eclipse Integrated Development Environment (IDE)*, which is presumably its most popular application example. Other examples besides a large number of further software development tools are the *Maestro* program of NASA used to operate Mars rovers [ML05], the commercial office productivity suite *Lotus Symphony* [Lot] or the open source BitTorrent client *Vuze* [Vuz].

The platform's *Workbench* component provides means for structuring the user interface of an application using the metaphor of a *Workbench* presented in a *Workbench Window*. The Workbench Window contains

- a *Menu Bar* with menu entries that can change depending on the active *Editor* and *View*,
- a *Tool Bar* with tool buttons that can change depending on the active *Editor*,
- an *Editor Area* containing *Editors*,
- several *View Areas* surrounding the *Editor Area* containing any number of *Views* and
- a *Status Line* at the bottom.

Views and *Editors* can be flexibly stacked, tiled and detached by the user. *Perspectives* can be used to persist visual arrangements of *Views* and *Editors* suitable for particular funcionality.

Besides the user interface model, the RCP imposes an architecture of loosely coupled software modules called *plug-ins* defining dependencies by *extension points*. The extensibility of RCP based applications facilitated by plug-ins accounts for one of the most important strengthes of the Eclipse IDE - the wide selection of commerial and non-commercial, community-driven enhancements of the core functionality. As the Scenario Builder is a tool of very special purpose, no higher level plug-ins can be reused to build the application but only plug-ins providing the core funcionality of the platform are incorporated. Also, the conception as stand-alone integrated application does not require the Scenario Builder to define extension points for other plug-ins. Scenario Builder builds on the Eclipse RCP version 3.4.1, bundled with SWT, JFace and GEF (see below), all using the same version number.

4.4.2 SWT

The *Standard Widget Toolkit (SWT)* [SWT, NW04] is a thin layer between native graphical user interface platforms of operating systems and Java software, providing a lightweight GUI toolkit that enables native performance and look and feel accessible via a common API for Java programmers. In contrast to *Swing*, SWT uses native widgets whenever possible and emulates user interface elements only in cases where the underlying operating system lacks an appropriate implementation.

SWT sessions are modelled by a *Display* that provides a connection to the underlying GUI platform. Within a *Display*, *Shell Widgets* reflect windows - *Root Shells* model main application windows, *Secondary Shells* are created as childs of other *Shells* and model windows that exist in the context of other windows, like dialogs. A *Shell* is the root of a tree of *Widgets*, that represents the GUI elements inside the window. All *Widgets* except top-level *Shells* are instantiated with a reference to its parent *Widget*, imposing a strict hierarchical structure following the composite pattern. The arrangement of widgets can be controlled by configurable *Layouts*. User interaction with the widget tree is achieved by running an *Event* dispatching loop handled by the *Display*, i.e. user interface events are queued by the operating system and dispatched to a *Widget* by the *Display*. Applications can connect to event handling by registering *Listeners* for specified *Event* types at *Widgets*.

The user interface framework of the RCP is implemented using SWT, wherefore Scenario Builder like most RCP applications and plug-ins is also based on SWT for a consistent user experience, although Swing or AWT could alternatively be used. Using the Eclipse RCP as a framework providing a base hierarchy of widgets constituting the workbench, application-specific user interface parts are usually hooked in by implementing `IWorkbenchPart.createPartControl(Composite parent)` in views and editors or creating dialog shells in action handlers as childs of `IWorkbenchWindow.getShell()`.

4.4.3 JFace

JFace is a user interface library built on top of SWT. It does not hide the SWT API but adds higher level concepts used by the RCP workbench and most RCP plug-ins.

- *Viewers* provide a higher level of abstraction upon interactive SWT widgets to allow user interfaces to be programmed at a more domain related level. JFace *Viewers* are adapters of data-contained controls, such as lists, tables and trees. They mediate between domain models and SWT controls. Viewers populate and update their content data using *content providers* and render labels as advised by *label providers*, both provided by the application implementation. Certain subtypes of viewers furthermore allow sorting and filtering by means of *viewer sorters* and *viewer filters*.

- *Actions* are application events that are triggered by the user performing some kind of input like selecting a menu entry, clicking a toolbar button or typing a key. One *Action* may be able to be invoked in multiple ways, e.g. a document can be opened in an *Editor* by double-clicking a node in a tree view or by using an entry of a context menu.

- *Dialogs* are windows that are popped up on top of the main application window. Their content can be structured using navigation trees or tabbed panes.
- *Wizards* are *Dialogs* that guide the user through a sequenced set of tasks.
- Registries of images and fonts provide clean management of UI resources of the operating system.

4.4.4 GEF and Draw2d

Originally developed for graphical modelling tools, especially UML-tools, the *Graphical Editing Framework (GEF)* [HSW08, GEFa] is an Eclipse-based framework that supports graphical displaying and editing of user data corresponding to conceptually any model. GEF builds on SWT in that it either paints on an SWT `Control` (usually a `Canvas`), or handles an SWT `Tree` widget.

GEF depends on the *Draw2d* plug-in [GEFb], a toolkit that provides a mapping of "lightweight" (in terms of not being associated with operating system resources), composable *Figure* objects to a graphical representation on an SWT `Canvas`, including

- painting of figures, using SWT's *Graphical Context* (`GC` class), considering z-order and clipping
- dispatching of SWT events (like mouse clicks and drags) to figures,
- deferred graphics updates to avoid flicker caused by displaying intermediate states,
- layout management and
- hit testing.

With Draw2d enabling an efficient figure-based model on top of SWT, GEF adds data mapping and visual editing capabilities on top of Draw2d. Draw2d `IFigure`s or SWT `TreeItem`s, contained in GEF *viewers* constitute the *view* component of an MVC-Pattern implementation by the GEF framework (*Model-View-Controller*, [GHJV95]). The *model* component is not part of the framework and its structure is completely left to the application, i.e. in GEF (domain) model objects occur as plain Java `Object`s. The *controller* of a GEF editor is realized by a hierarchical composition of objects called *editparts*. Editparts associate domain objects (the *model* in MVC) with graphical objects (the *view* in MVC, figures or tree items). The information flow along this association, provided by editparts, is bidirectional:

- figures on the viewer's canvas are initialized and refreshed by editparts to reflect the state of domain objects, and

- domain objects are manipulated by editparts triggered by user interaction with figures on the viewer's canvas

The containment hierarchies of editparts and figures are usually designed to parallel the domain model, though the actual factorization is up to the application programmer. To achieve the updating behaviour of the first mentioned situation but keep the domain model independent of the controller, GEF suggests loose coupling by registering editparts as change listeners with the respective domain objects (see the *Observer Pattern*, [GHJV95]). In the realization of the second case, concerning user interaction, further framework concepts are involved. The user interacts with the editpart viewer by means of editing modes called *tools*, usually visualized as icons in a *tools palette.* Tools are basically state machines that accept SWT events as input. As explained, Draw2d includes dispatching of SWT events to figures. Using GEF, this feature is actually replaced[1] by forwarding events to an *edit domain* instead. An edit domain defines a set of tool palette entries of a graphical editor and knows the currently active tool. SWT events received by an edit domain are passed to the active tool, which in turn translates them into *requests* that represent user interaction at the level of editparts. In general, tools determine *source* and a *target editparts* to be addressed by requests, either based on the viewer's selection or by asking the viewer for the editpart at the current location of the mouse pointer. The targeting mechanism of the latter ultimately involves the hit testing capabilities of Draw2d. Moreover, editparts can decline responsiveness for requests or designate another editpart. Request handling is delegated by editparts to *edit policies* for better localization of behaviour related to different request types. Requests are used to ask editparts to show graphical feedback while an interaction is in progress and to query editparts for appropriate *commands.* Commands encapsulate undo- and redoable domain model operations, i.e. instead of an editpart invoking domain logic directly, commands are executed by a *command stack* held by the edit domain to support an undoable history of operations. Note that in Scenario Builder, GEF commands are actually translated into a similar concept offered by Eclipse RCP with `IUndoableOperation`s to achieve one single global command stack, see Section 4.5.3.

GEF is used and customized for the Scenario Builder's implementation of layout editors and the infrastructure editor as explained in Subsection 4.5.3.

Version 3.4.1

4.4.5 JAXB

Java Architecture for XML Binding (JAXB) [JAXb] is an API that enables object oriented creation of and access to XML documents. This Java Community Process standard is part of the *Java Platform, Enterprise Edition* [Java] since version 5 as a base technology for the platform's support of web services [MP02], and the Standard Edition since version 6. It is

[1]Concretely, Draw2d event dispatching happens in `SWTEventDispatcher`, which is overridden by `DomainEventDispatcher` in GEF, which in turn is set on the `LightweighSystem` in `GraphicalViewerImpl`.

applicable for general purpose applications as a standalone API. Scenario Builder uses the *JAXB Reference Implementation* of the *Glassfish* community [JAXa], version 2.1.8.

The JAXB implementation includes a *binding compiler*, `xjc`, that takes XML schemas as input and generates a set of Java classes used for marshalling and unmarshalling corresponding XML instance documents. Scenario Builder takes instances of `pedestrianScenarioMetadata.xsd` as input and produces instances of `pedestrianScenario.xsd` as output (Subsection 3.4.4), both schemas are compiled into classes of the `domain.meta` and `domain.export` packages.

The XSD schema compilation is integrated into the build process, so that the mapped classes are kept consistent with the `xsd` resources. This ensures compatibility of the Java implementation with the interface specification when either of both evolve, as static incompatibilities due to changed model structures result in compile time errors.

4.4.6 Commons Math

Commons Math is a mathematics and statistics library of the *Apache Software Foundation* [Mat]. It is used (in version 1.2) by the `domain.meta.abstractDistributions` package to calculate probabilities when parameter bindings using well-known distributions are "customized", see Section 4.3.3.

4.4.7 opencsv

opencsv [ope] is a simple open-source library for reading and writing CSV data in Java programs. *opencsv* version 1.8 is used to import od-matrices into the OD-Map Editor (Subsection 4.3.7).

4.4.8 KTable

KTable [KTa] is a custom SWT widget by Friederich Kupzog and Lorenz Maierhofer, providing a grid-structured control with more flexibility than SWT `Tables`. It implements a spreadsheet-like user interface model suitable for the presentation of od-matrices by the OD-Map Editor (Subsection 4.3.7). Scenario Builder uses *KTable* version 2.1.3.

4.4.9 log4j

Logging information of the application is sent to *log4j* [log], version 1.2.15, configured by a properties file to use two *logging appenders*, a rotating file appender writing to a dedicated log directory to support reconstruction of production faults, and a custom appender implemented by the `logging.LogViewAppender` to display logging output in a viewer in the workbench.

4.5 Implementation Architecture

4.5.1 The root package

Figure 4.25 shows the high level package structure of the Scenario Builder source code. The implementation comprises a total of 1245 Java classes (including 92 JAXB-generated classes) and about 70k computed lines of code (including 3k lines of JAXB-generated code), measured by the Eclipse Metrics plugin [Met]. In the following discussion the common root package part is discarded from package paths.

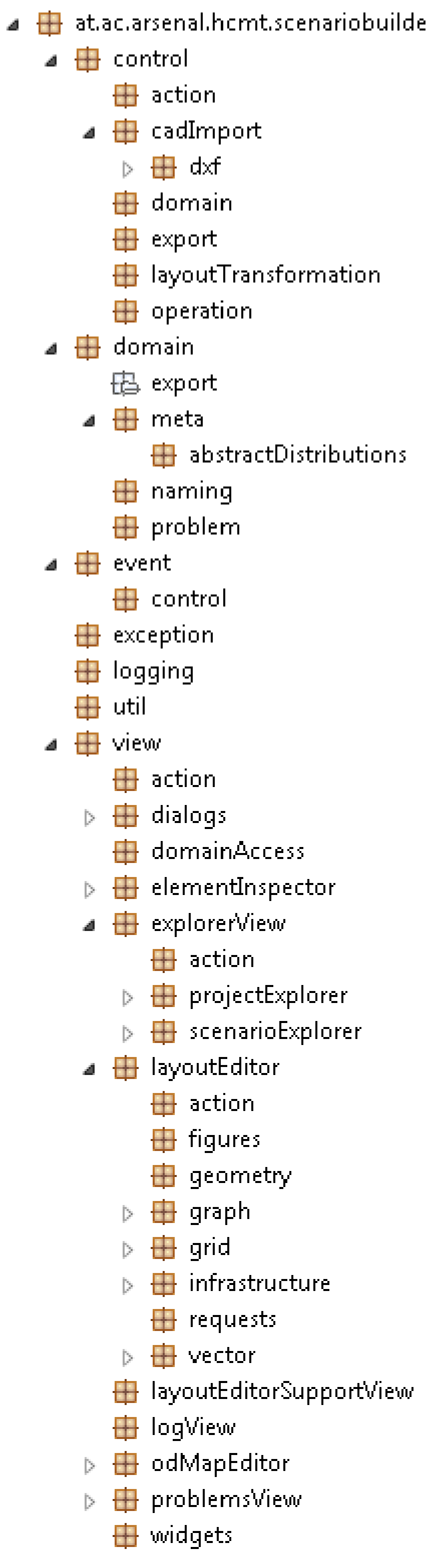
The key top-level packages are `domain`, `view` and `control`. They to some extent reflect the MVC-pattern [GHJV95] at a coarse level, though actually some of the subpackages of `view` have themselves an inner structure following MVC. This applies especially for GEF-based graphical editors with editparts as controller components residing within a `view` package. Still that kind of view-side controller functionality always bridges to the top-level controller mechanism explained in Subsection 4.5.3 by effectively triggering actions of the `control` package.

The remaining top-level packages, `event`, `exception`, `logging` and `util` deal with cross-cutting concerns also discussed below.

Few classes are directly contained by the root package, namely those hooking into the RCP application lifecycle to handle workspace loading and shutdown, and initialize global properties, actions and the default perspective. There is also some static helper funcionality implemented to enable access to application resources like data files and images.

4.5.2 The `domain` package

The classes in the `domain` package model the actual project data the user works on (often referred to as *the* model). Classes of anything that is persisted to outlast program execution are located directly in `model`. That base set of domain classes essentially reflects the Scenario Project Model discussed in Section 4.2. The `export` subpackage exclusively contains classes generated by JAXB to map the scenario export XML schema defined by `pedestrianScenario.xsd` (Subsection 3.4.4). The `meta` subpackage contains the JAXB-generated classes mapping `pedestrianScenarioMetadata.xsd`, together with helper classes[2] that encapsulate initialization and higher-level access to metadata provided by raw XSD element wrappers. It also contains the `BindingValidator` interface and its implementation classes that are used for validation of parameter values based on metadata constraints. Nested inside `meta` is the `abstractDistributions` package with implementations of well-known and user-defined distribution types (see Section 3.4.4). Additional classes and subpackages of `model` support functional requirements like object duplication (for copy & paste), problem annotation and layout transformation.



```
at.ac.arsenal.hcmt.scenariobuilder
    control
        action
        cadImport
            dxf
        domain
        export
        layoutTransformation
        operation
    domain
        export
        meta
            abstractDistributions
        naming
        problem
    event
        control
    exception
    logging
    util
    view
        action
        dialogs
        domainAccess
        elementInspector
        explorerView
            action
            projectExplorer
            scenarioExplorer
        layoutEditor
            action
            figures
            geometry
            graph
            grid
            infrastructure
            requests
            vector
        layoutEditorSupportView
        logView
        odMapEditor
        problemsView
        widgets
```

Figure 4.25: High-level structure of source packages

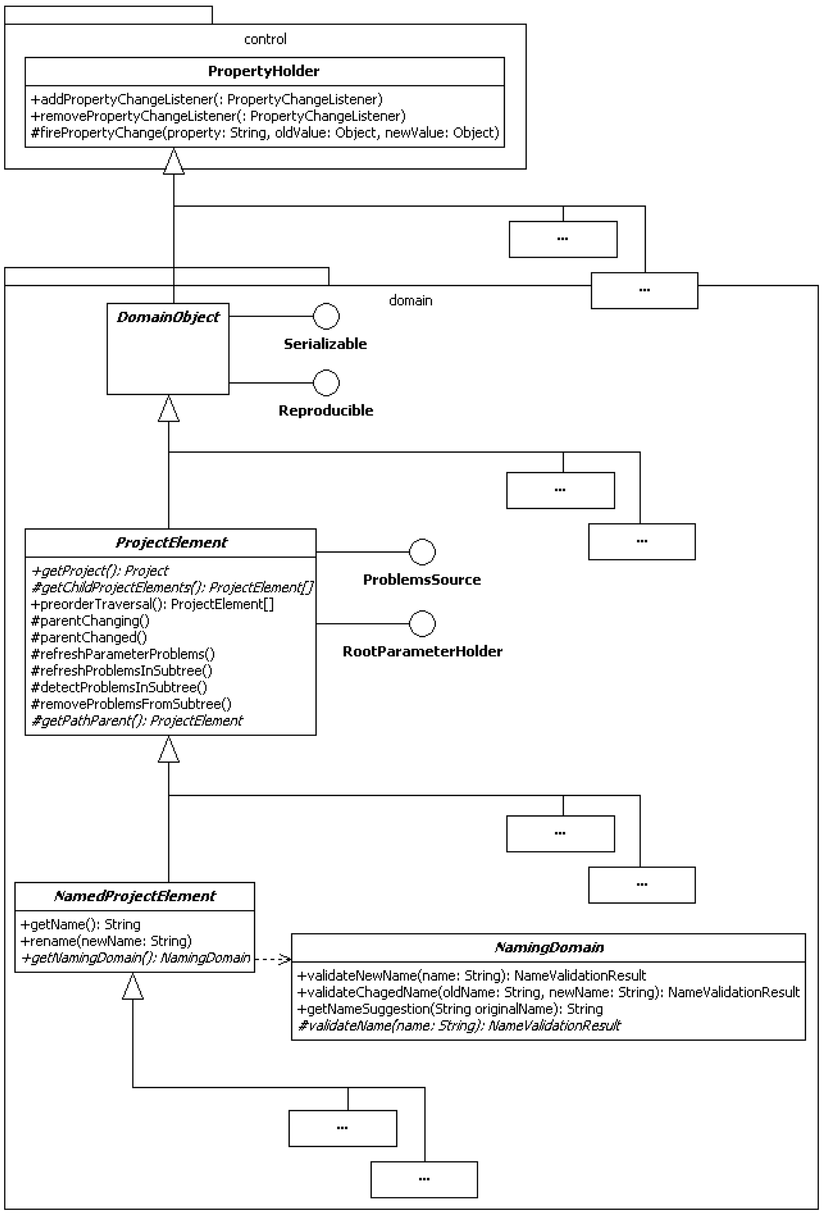

control
PropertyHolder
+addPropertyChangeListener(: PropertyChangeListener)
+removePropertyChangeListener(: PropertyChangeListener)
#firePropertyChange(property: String, oldValue: Object, newValue: Object)
...
domain
DomainObject
Serializable
Reproducible
ProjectElement
+getProject(): Project
#getChildProjectElements(): ProjectElement[]
+preorderTraversal(): ProjectElement[]
#parentChanging()
#parentChanged()
#refreshParameterProblems()
#refreshProblemsInSubtree()
#detectProblemsInSubtree()
#removeProblemsFromSubtree()
#getPathParent(): ProjectElement
ProblemsSource
RootParameterHolder
NamedProjectElement
+getName(): String
+rename(newName: String)
+getNamingDomain(): NamingDomain
NamingDomain
+validateNewName(name: String): NameValidationResult
+validateChagedName(oldName: String, newName: String): NameValidationResult
+getNameSuggestion(String originalName): String
#validateName(name: String): NameValidationResult

Figure 4.26: Domain base classes

Base hierarchy

Figure 4.26 shows the root hierarchy of domain objects. Notably all `domain` classes extend a class of the `control` package, `PropertyHolder`, which realizes the *observer* pattern ([GHJV95]) using the `PropertyChangeSupport` of *Java Beans* to enable controller and view objects to register as listeners with the domain object for relevant properties. The abstract `DomainObject` class adds actual domain characteristics, mainly by implementing `Serializable`, wherefore all subclasses are warned to specify a `serialVersionUID` and need to able to be persisted using *Java Object Serialization* ([JOS]). Note that the `PropertyHolder` superclass holds only transient state. Moreover `DomainObject` implements `Reproducable` that specifies an interface used by cut/copy & paste operations as explained below. `DomainObject` provides a default implementation of `Reproducible` based on `Object.clone()`. Further specialization in the domain type hierarchy is given by `ProjectElement`. `ProjectElements` are `DomainObjects` that are able to hold parameter bindings to specify values for user-defined parameters (as introduced in Section 3.4.4), and can be sources of domain problems (to be reported by the Problems View). These capabilities are exposed by the `RootParameterHolder` and `ProblemsSource` interface respectively and implemented using delegate classes as explained below. A `ProjectElement` belongs to zero or one `Project` which is answered by the `getProject()` method. `ProjectElements` of a `Project` are organized by a containment hierarchy reflected by `getChildProjectElements()` and `preorderTraversal()`. This hierarchy is used for recursive cleanup and refresh of parameter bindings and domain problems. Finally, project elements that are identified by user-definied names subclass `NamedProjectElement`, which provides a name space mechanism. Depending on the domain object type, the name of a `NamedProjectElement` in general is constrained to be unique within some context. To localize this behaviour, `NamingDomains` are used to validate changed and new names and to suggest new names for duplicated project elements. The `domain.naming` package contains all specific `NamingDomain` subclasses.

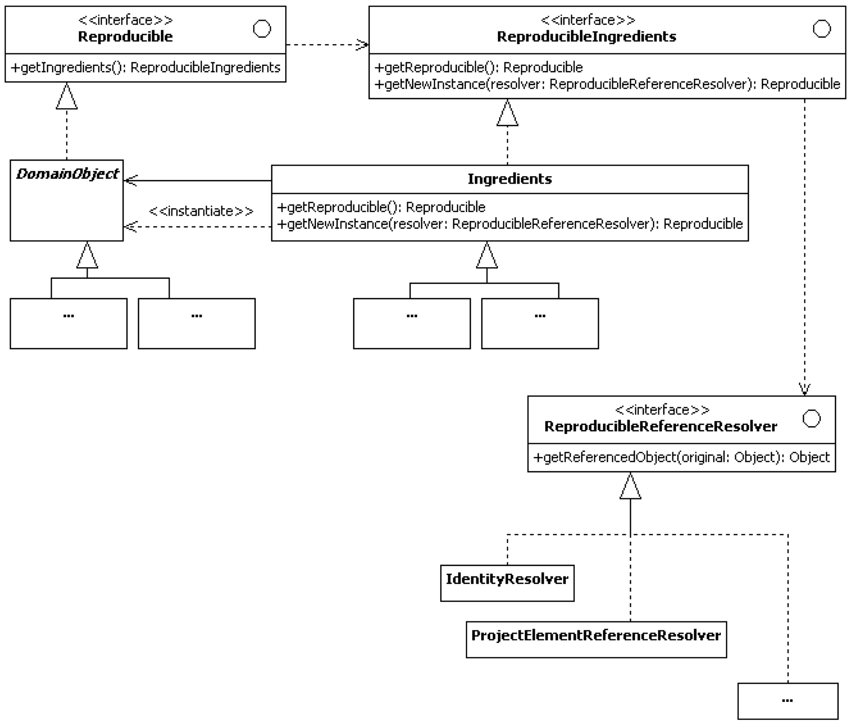
Direct concrete subclasses of all of the explained top level classes exist in the Scenario Builder implementation, i.e. there are no “empty” intermediate abstraction levels.

Domain object duplication

The ability to copy, cut and paste domain objects is a general requirement supported by a set of interfaces and implementations paralleling the domain object hierarchy. When the user invokes a copy or cut action, snapshots of the selected domain objects need to be retained in memory. Their current state needs to be preserved independently from future manipulations of the originals. Associated objects are either recursively duplicated or remembered by reference. When an insertion action is invoked, new domain objects need to be created accordingly and inserted into a project hierarchy. This is a non-trivial process in general, as duplicates can be inserted into different contexts, possibly incompatible regarding the associated domain objects.

[2] In the development configuration, JAXB uses its own dedicated source folder in the project directory to avoid mixing up generated and programmed code, but both contribute to the `domain.meta` package.

The class diagram in Figure 4.27 refines the realization of `Reproducible` by `DomainObject`. `Reproducibles` implement `getIngredients()` to provide their `ReproducibleIngredients`, which hold the state information required to create new instances at a later time. The actual reproduction happens in the `getNewInstance()` method implementation that takes a `ReproducableReferenceResolver` as input parameter. The reference resolver is queried for objects to be associated with the new instance to reflect the original association. `IdentityResolver` is a default implementation answering the same object instance, `ProjectElementReference-Resolver` interposes a check for the associated object to reside in the same project as the new instance. More sophisticated policies are implemented by layout editor actions. It is up to the subclasses of `DomainObject` to implement adequate reproduction behaviour by providing their own `ReproducibleIngredients` implementations, usually as inner classes.



<<interface>>
Reproducible
+getIngredients(): ReproducibleIngredients
<<interface>>
ReproducibleIngredients
+getReproducible(): Reproducible
+getNewInstance(resolver: ReproducibleReferenceResolver): Reproducible
DomainObject
<<instantiate>>
Ingredients
+getReproducible(): Reproducible
+getNewInstance(resolver: ReproducibleReferenceResolver): Reproducible
...
<<interface>>
ReproducibleReferenceResolver
+getReferencedObject(original: Object): Object
IdentityResolver
ProjectElementReferenceResolver

Figure 4.27: Domain object duplication

Parameter bindings and problem annotations

Apart from `ProjectElement` as already outlined, there are further object types able to bind parameter values to abstract parameter definitions, and to be associated with domain problems. Without examining structural details of the domain implementation, these behavioural categories are not exactly aligned. Therefore code reuse is obvious, but not possible via subclassing because of structural differences and Java not supporting multi-inheritance. Figure 4.28 and Figure 4.29 outline the solution using interfaces and delegation.

By means of distribution support values and child values of composite parameter values, nested ParameterHolders create a containment hierarchy that is traversed when project elements are validated to update their problems or check compatibility with abstract parameter definitions, and passed through upwards to notify project elements about value changes. The root objects of parameter holder hierarchies (ProjectElements or instances of the Member association class of composite pedestrian types) implement RootParameterHolder, intermediate levels implement ParameterHolder.

ProblemSources can be queried for their Problems, using a technical categorization provided by ProblemInternalCategorys. Problem instances moreover reference a display category, the ProblemUserCategory, and know a string representation of a path locating them within a project (using ProjectElement.getPathParent(), see Figure 4.26), a problem description and the project element to be focused in the user interface related to the problem. All concrete Problem subclasses are located in the domain.problem package.

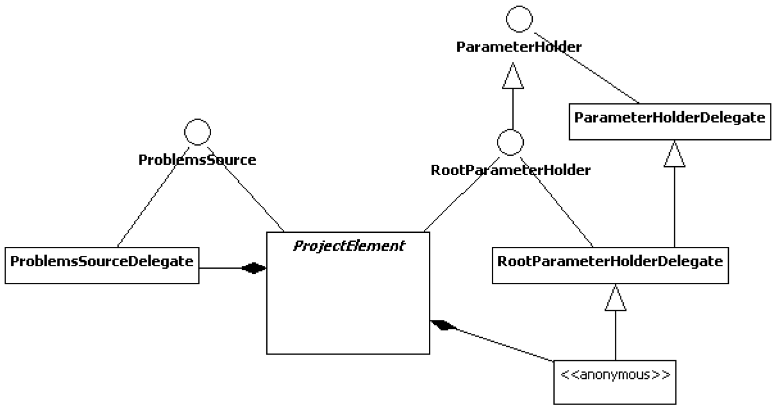


Figure 4.28: Parameter bindings and problem annotations (1)

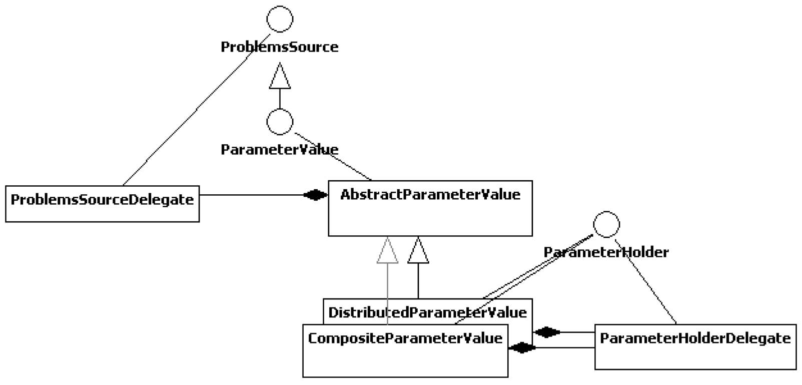


Figure 4.29: Parameter bindings and problem annotations (2)

4.5.3 The `control` package

Actions and operations

Actions are a workbench concept that encapsulates commands issued by the user and decouples them from different possible representations in the user interface. RCP uses `IActions` of JFace to represent the "non-UI side of a command" (from source documentation of `IAction`).

Most of the action classes of Scenario Builder inherit their base functionality from `ContextualAction`, see Figure 4.30. The diagramed class hierarchy starts with the JFace `Action`, extended by GEF with `WorkbenchPartAction` and `SelectionAction` to add knowledge about the action's context and a selection-sensitive update mechanism. `ContextualAction` specializes this behaviour by interpreting selections and deciding about enabling the action based on the type of selected objects. It also provides comfort methods to access selected domain objects. Three *selection modes* are supported - `SelectionMode.SINGLE` enables an action for single selected matched objects, `SelectionMode.MULTIPLE_ALL` enables multi-selection but requires all selected objects to meet the type criterions and `SelectionMode.MULTIPLE_ANY` allows multi-selection with at least one matching selected object. The object type matching is configured by specifying a set of `Class` objects, either to be directly selected, or via an indirection using `SelectableRepresentatives` that are asked for a represented object (such as tree nodes mapping to domain objects).

Actions of the Scenario Builder application are typically instanciated by `WorkbenchParts` when the part control is created. They are enabled and disabled based on the current selection and invoked by the platform when associated user interaction occurs, i.e. a menu entry is selected, a toolbar button is pushed or an accelerator key is pressed. Some actions are purely user interface related, like actions for zooming, list filtering or tree folding, though most actions result in manipulation of domain objects. The latter require an intermediate layer of command handling to support *undo* and *redo* commands. This layer is basically provided by the platform's operation history (`IOperationHistory`) that associates undoable operations (`IUndoableOperations`) with undo contexts (`IUndoContexts`). Undo contexts are used to define application scopes operations belong to. Scenario Builder, see Figure 4.31, manages undoable operations using the global operation history of the workbench, that is all user actions affecting opened projects of an application instance are strictly serialized in one single command stack, still operations are tagged with undo contexts of the respective projects defined to be "matched" by the global context. When a project gets closed or deleted (see Section 4.3.2), the operations of its undo context are disposed from the global undo history. The `control.operation` package contains the `CompositeOperation` class that allows to reuse operations by nesting them inside more complex operations. It also provides the `OperationService` singleton that administers project contexts, accesses to the global operation history and localizes operation error handling. The `OperationService` therefore bridges user interface actions to domain operations. As mentioned in Subsection 4.4.4, GEF contains its own command stack framework, which is integrated into the global history by a wrapper class, `DomainOperationCommand`.

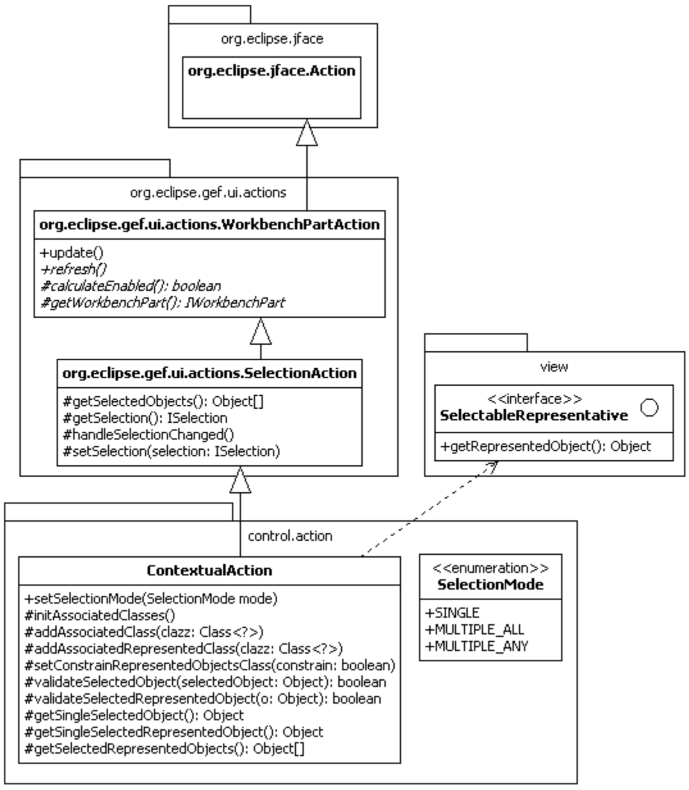

org.eclipse.jface
org.eclipse.jface.Action
org.eclipse.gef.ui.actions
org.eclipse.gef.ui.actions.WorkbenchPartAction
+update()
+refresh()
#calculateEnabled(): boolean
#getWorkbenchPart(): IWorkbenchPart
org.eclipse.gef.ui.actions.SelectionAction
#getSelectedObjects(): Object[]
#getSelection(): ISelection
#handleSelectionChanged()
#setSelection(selection: ISelection)
view
<<interface>>
SelectableRepresentative
+getRepresentedObject(): Object
control.action
ContextualAction
+setSelectionMode(SelectionMode mode)
#initAssociatedClasses()
#addAssociatedClass(clazz: Class<?>)
#addAssociatedRepresentedClass(clazz: Class<?>)
#setConstrainRepresentedObjectsClass(constrain: boolean)
#validateSelectedObject(selectedObject: Object): boolean
#validateSelectedRepresentedObject(o: Object): boolean
#getSingleSelectedObject(): Object
#getSingleSelectedRepresentedObject(): Object
#getSelectedRepresentedObjects(): Object[]
<<enumeration>>
SelectionMode
+SINGLE
+MULTIPLE_ALL
+MULTIPLE_ANY

Figure 4.30: Actions base hierarchy

Metadata definitions

When Scenario Builder is launched, available metadata definitions are loaded from `.scen-meta`-files residing in a predefined directory within the Eclipse workspace into memory. The `javax.xml.validation.Validator` of the Java runtime library is used to assert the conformance with `pedestrianScenarioMetadata.xsd` before the JAXB `Unmarshaller` deserializes the XML data into instances of `domain.meta` classes.

Metadata definitions are identified by their name and version, see Section 3.4.4. A `Project` specifies the metadata definition it is based on using that name and version strings. Projects and metadata definitions are loosely coupled, i.e. no `domain.meta` objects are referenced by (and therefore persisted with, see the next paragraph) `domain` objects. Instead, a single instance of `MetaDataService` in the `control` package provides access to `ScenarioMetadataHelper`s for each available metadata version, which in turn can be asked for abstract definitions of facilities, tools and propability distributions, and for abstract parameters of a given `ParameterHolder`. Parameter bindings of domain objects are implemented by mapping abstract parameter names to parameter values, so that `AbstractParameter`s are referenced by their name string, not directly by an object association.

When projects are loaded at application startup, Scenario Builder checks whether the specified

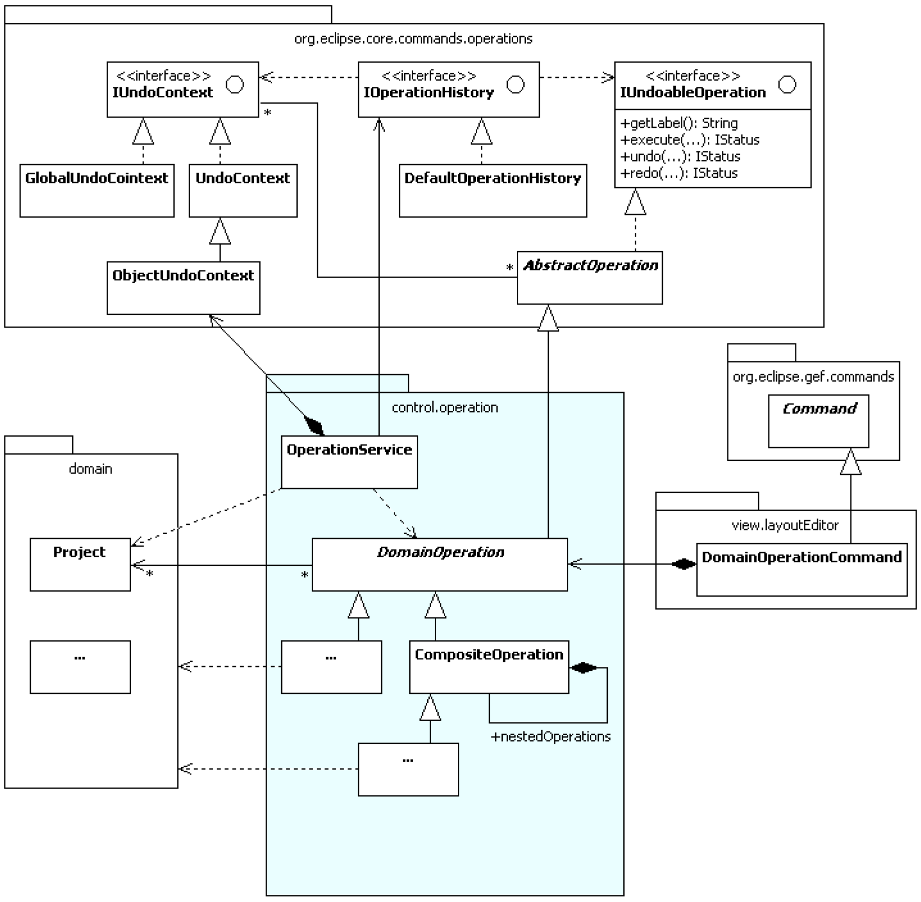


Figure 4.31: Operations

metadata definitions exist in the workspace. If no `.scen-meta` file can be matched by name and version, the project is loaded anyway with a user warning, and an alternative metadata definition can be selected. If a matching metadata definition is found when a project is loaded or when metadata is changed at any time, the parameter bindings of the project are validated against its abstract definitions. Incompatible parameter bindings are reported to the user and not ultimately dropped but kept within an invisible context (actually implemented by a class named `SilentParameterHolderDelegate`). They can be reapplied to the visible context when incompatibilities are resolved at a later time. This allows flexible switching between different (versions of) metadata definitions without information loss and a best-effort policy for partially compatible simulation models.

Domain data persistence

The specification of Subsection 4.1.2 requires domain data to be persisted per project and in a way that enables import and export of projects into and from workspaces of different Scenario Builder installations. As the most straightforward solution, *Java Object Serialization* [JOS] is utilized to persist domain object trees in the file system. The root object of serialization is a `Project` instance. Therefore special care needs to be taken regarding the association graph of directly and recursively referenced classes of `Project`. All project-related information needs to be held by serializable objects, actually `DomainObjects`, ultimately accessible from a `Project`. Fields containing derived data or references to non-domain objects, especially property change listeners, need to be marked as `transient`, and (re-)initialized after deserialization as needed by implementing `readObject(ObjectInputStream ois)`.

The `control.DomainDataService` class handles

- synchronization states of projects, indicating whether the current in-memory state reflects the persisted state, used for a "changed" decoration of the project label in the user interface and enabling the *save* action, aware of the operation history
- saving, renaming and deletion of projects
- opening and closing of projects
- initialization and re-validation of metadata associations

Scenario export

The output of XML data conforming to `pedestrianScenario.xsd` of the GPSI (Subsection 3.4.4) for a selected scenario is done by the `ExportService` class in three steps. First, the internal representation of the scenario made up of objects of the `domain` package (implementing the Scenario Project Model) is mapped to an in-memory instance of the Generic Input Model implemented by the JAXB generated classes of the `domain.export` package. The result of this step is an object tree with a `domain.export.Scenario` as root. Second, the `domain.export.Scenario` is translated to its XML representation and written to the file system by a JAXB `Marshaller`. Finally, the generated XML output is validated using `javax.xml.validation.Validator` against the GPSI XSD.

Layout transformation

Scenario Builder supports layout transformation from vector layouts to grid layouts as proposed in Section 4.3.5, see also the screenshot of Figure 4.17. This feature basically involves discretization of vector shapes. As items can be assigned to fill areas and line segments of polygonal regions, cell representations of both are required.

Line segments are transformed into grid cells using the fast voxel traversal algorithm for ray tracing of [AW87]. Fill area discretization is implemented as follows. For every vector region, the cellular bounding rectangle of its shape is determined by finding the minimum and maximum coordinate values of the shape vertices and translating them into the specified grid coordinate system. Subsequently, all cells of the bounding rectangle are iterated and checked for containment of vector coordinates in the vector shape using `Figure.containsPoint()` of Draw2d.

CAD import

CAD plans in DXF ([DXF]) format can be imported into projects as new vector layouts. After selecting a `.dxf` file, its data is analyzed to determine the names of the contained layers and the overall dimensions of the plan. The user is asked to select a set of drawing layers to be included and to input a scale factor and an origin offset used to interpret measures of the geometric data. The import is performed by classes of the `control.cadImport` package. DXF data is read line by line and interpreted by a state machine algorithm implemented by `DXFLayoutInterpreter` that adds shapes to a newly created vector layout. Its abstract base class, `DXFInterpreter`, manages a set of active *triggers*, defined by group codes and values that are looked for when proceeding through DXF data elements. When trigger conditions are detected, a handler method is called that updates the internal state and adopts the trigger set to the new state

The DXF specification features supported by the underlying version of Scenario Builder are lines, polylines, circles, circular arcs, hatches with boundary path types of the above and block insertions.

4.5.4 The `view` package

This section gives a brief overview of the implementation of the `view` package that implements the user interface concepts illustrated in Section 4.3.

Explorer Views

The Project Explorer and the Scenario Explorer view parts are based on `TreeViewer`s of JFace that provide tree-structured views of a *model* accessed by a *content provider* (`IContentProvider`) and described by a *label provider* (`IBaseLabelProvider`). To decouple the structure of view trees from the structure of domain elements, the latter are wrapped by `ExplorerNodes` to constitute the viewer's model. Subclasses of `ExplorerNode` exist for all types of tree nodes. A single domain element type can be wrapped by multiple node classes as for example a basin is listed as a child of a section node and of a facility node. The tree node registers itself as a listener for changes of the wrapped domain object and translates domain events to notifications of a centralized event service dedicated to explorer views implemented by `control.EventService`. These node events are received by the explorer's content provider (`view.explorerView.ExplorerContentProvider`) to reflect structural changes in the hierarchy of tree nodes. The `EventService` is also used to broadcast `ProjectActivationEvents`, events that signal the change of the project the workbench focuses on, particularly needed by the Scenario Explorer to swap its content to the scenario tabs of the new active project.

The root content object of the Project Explorer is a `WorkspaceNode`. The `WorkspaceNode` does not have a visual representation as a single tree node but provides the root `ProjectNodes` of the viewer using the `DomainDataService` (Section 4.5.3). The tabs of the Scenario Explorer are managed by the `ScenarioExplorerView` by mapping `Projects` to collections of `ScenarioPresentation` objects, that in turn associate scenarios, tree viewers and tab items of an SWT `TabFolder`.

Layout Editors

Layout editors are used to view and edit geometric data of project sections. A dedicated editor type exists for vector, grid and graph layouts respectively, each based on GEF (Subsection 4.4.4). The default base behaviour of GEF is modified at several points:

- The `GraphicalEditor` of GEF is overridden and customized by `view.layoutEditor.GraphicalEditor` that supports editor scopes (see Section 4.3.3), global undo and redo, and modified zooming using `view.layoutEditor.MetricZoomManager` instead of GEF's default `ZoomManager`.
- `FreeformViewport` is extended to forgo automatic contraction of the edit area when drawing tools are used outside existing figure's bounds. Furthermore, a custom `AutoexposeHelper` implementation is used to allow tools to expand the edit area by scrolling out of its current bounds.
- `AbstractLayoutEditorTool` adds to `AbstractTool` zoom-enabled graphical feedback and convenience methods for scale translation between domain coordinates and viewer coordinates.
- `AbstractGraphicalEditPart` is extended to also support zoom-enabled graphical feedback and implement `SelectableRepresentative` (see Section 4.5.3).
- `RoundingScalingRootEditPart` is a root edit part that replaces the scaling pane of the default root edit part's graphical layer stack to incorporate its own `ScaledGraphics` subclass, `RoundingScaledGraphics`, which in principle modifies the display of zoomed figures by applying `Math.round()` instead of `Math.floor()` to scaled coordinates.
- `GraphicalEditDomain` overrides GEF's `DefaultEditDomain`. It reports the current pointer location to `PropertyChangeListeners` of the graphical viewer and allows to temporarily suppress the editor's context menu as needed by some drawing tools.
- `ScopeContributionItem` is a JFace `ControlContribution` that implements a scope selection box as toolbar contribution of an active layout editor.

With this shared basis, the different layout editor types further specialize their behaviour by subclassing and implementing context menu providers, zoom managers, edit part factories and graphical viewers. They realize the editing model imposed by the GEF framework to meet the respective part of the domain and user interface model. The following paragraphs list the concrete implementations of edit parts, edit policies and tools for all layout editor types.

Vector Layout Editor GEF framework model implementation:

edit parts	layout, bounding, region, vertex
connection edit parts	vector line, bounding line
edit policies	layout, reference location, region as component, vertex as component, region item assignment, snap to line feedback, snap to vertex feedback, vector line restructuring, polygonal shape editing, rectangle editing, vertex drag
tools	selection (with marquee, move shape and move vertex drag tracker), create polygon, create polyline, create rectangle

Table 4.1: GEF model implementation by the vector layout editor

Creation and update of figures by edit parts of the vector layout editor is handled by delegate classes, subclasses of `VectorShapeFigureDelegate`, that map domain shapes to Draw2d figures.

Snapping, that is the automatic alignment of the current editing location with well-defined locations in the layout, is supported by GEF by means of `SnapToHelper`s used by tools. This base implementation operates on presentation data, i.e. viewer coordinates. Scenario Builder provides its own `SnapToHelper`s based on domain coordinates to assure exact locations without precision loss due to zooming, namely `SnapToGrid` for snapping to a user-defined drawing grid (specified by Section 4.3.5), `SnapToLine` for line segments of shapes and `SnapToVertex` for shape vertices.

Grid Layout Editor GEF framework model implementation:

edit parts	grid layout
connection edit parts	-
edit policies	-
tools	selection, free draw, line draw, rectangular outline draw, rectangle draw

Table 4.2: GEF model implementation by the grid layout editor

The Grid Layout Editor follows a monolithic design, a single edit part representing the grid layout provides presentation logic and is the target for tools. This layout edit part draws and updates

- a *grid figure*,
- a collection of *cell figures*, organized in a map indexed by cell locations, and
- a collection of *cell feedback figures*, organized in a map indexed by cell locations.

The grid figure paints the background for cell figures and cell feedback figures. It is implemented efficiently using Draw2d's `FigureUtilities.paintGrid()` with parameters reflecting the cell size and the number of columns and rows of the grid layout. Cell figures are filled rectangles of the size of one grid cell that collectively reflect the current state of the layout domain object. A grid cell is overlayed by a cell figure if it is defined to be obstructive and/or has items assigned. The layout edit part is registered as a change listener with the grid layout to keep its grid figure and cell figures synchronized with the domain state. Cell feedback figures are also cell-sized rectangles that are inserted and removed as reaction to requests issued by tools.

Graph Layout Editor GEF framework model implementation:

edit parts	layout, bounding region, node
connection edit parts	link
edit policies	layout, link, link endpoint, node as component, node as connection node, node item assignment, node drag
tools	selection (with move node drag tracker), draw

Table 4.3: GEF model implementation by the graph layout editor

Graph layouts show conceptually higher compatibility with GEF concepts than the other layout types wherefore the implementation of the Graph Layout Editor is very straightforward following GEF examples. Some complexity is added to the selection tool to enable marquee zooming. Moreover, the drawing tool is a dedicated implementation. The same snapping mechanisms are applied as to the Vector Layout Editor.

Infrastructure Editor

The Infrastructure Editor provides a pseudo-3d view on a selection of sections and layouts as explained in Subsection 4.3.6. The geometric information of the selected layouts is schematically displayed within the sections (and therefore translated into world coordinates). The sole editable objects in the editor are inter-section-connections of basins, so the main information to be displayed are basin locations and connections between them. In difference to layout editors, the Infrastructure Editor does not directly present and manipulate objects of the `domain` package, but uses an intermediate model residing in its own `domain` subpackage to encapsulate the treatment of multiple basins at identical locations. *Identical location* has a dedicated interpretation for each layout type. Considering vector layouts, it means the same set of line segments, in grid layouts the same set of cell locations and in graph layouts simply an assignment to one and the same node. The Infrastructure Editor displays multiple occurrences of basins at identical locations as one single node for editing inter-section-connections, called *infrastructure node*, connected by *infrastructure links*. The per-basin configuration of connections between infrastructure nodes is enabled by a dialog with a selectable structured list of possible basin connections, also considering basin types (i.e. allowing connections only from sources to sinks).

The GEF concepts are applied by the Infrastructure editor as listed in Table 4.4.

edit parts	infrastructure, layout, layout content, layout background image, infrastructure node
connection edit parts	infrastructure link
edit policies	selection, infrastructure link
tools	selection

Table 4.4: GEF model implementation by the graph layout editor

Element Inspector

The Element Inspector is basically an Eclipse `PropertySheet`, but uses a customized version of the `PropertySheetPage` to remove unwanted action buttons, support parameter scopes and include the scope bar shown in the screenshot of Figure 4.7.

Domain objects are represented by trees of *property sources* implementing `IElementInspector-PropertySource`, an interface extending `IPropertySource`, adding parameter scope and domain problems support. The base implementation, `ElementInspectorPropertySource` implements `getPropertyDescriptors()` to answer *property descriptors* of tree childs enhanced by parameter scope and problems information. Property descriptors are either `Container-PropertyDescriptors` to describe nested property sources, or leaf descriptors. Customized types of the latter exist for combo boxes and text fields. Property sources exist for all types of project elements, `ParameterHolderPropertySources` contain binding hierarchies to user-defined parameters, including distributions and composite parameters. A `HoverHelper` is hooked at the property sheet control to provide tooltips with domain specific information. The `ElementInspectorPropertySheetEntry` exposes scope and problems information of property descriptors to the user interface by specifying icon images of the name and value fields, in turn assigned to tree items by an overridden `PropertySheetViewer` of the custom `PropertySheetPage`.

The `action` subpackage of `elementInspector` contains action classes used exclusively by the context menu of the Element Inspector. Their common base class is `ElementInspectorAction`. `ElementInspectorAction` manages the enabled state of an action by letting the subclass define the types of supported property sources, similar to the `ContextualAction` (Section 4.5.3).

OD-Map Editor

The OD-Map Editor consists of two parts in a tabbed view, implemented as a `MultiPageEditorPart` with an `ODMapBasinsEditor` and an `ODMapMatricesEditor` as editor parts.

The `ODMapBasinsEditor` (see Figure 4.22) is used to select origin and destination basins included in od-matrices of the od-map. Both parts of the editor page, origin and destination, use a BasinsPageTablesProvider instance to handle the controls and event listeners of the *available list*, the *included list* and the buttons in between. The lists are implemented as SWT Tables.

The ODMapMatricesEditor (see Figure 4.23) lets the user select a pedestrian type to view and edit the od-matrix for. The pedestrian type hierarchy is rendered by a JFace TreeViewer. The matrix control uses KTable [KTa]. The import of matrix values from CSV files uses the OpenCSV [ope] library.

Synchronization between the domain model state and an OD-Map Editor instance is achieved by both editor parts being registered as change listeners with related domain objects.

Problems View

The problems table is a JFace TreeViewer using a content provider, a label provider, a viewer sorter and a viewer filter.

The model of the viewer is provided by a class called ProblemsService that holds a collection of workspace problems, containing all domain problems of currently opened projects in the workspace. The ProblemsService receives events from the DomainService (Section 4.5.3) and the opened Projects to keep the problems collection consistent with the domain state. A ProblemsContentProvider in turn is registered as a change listener with the ProblemsService to mediate between the workspace problems collection and the viewer.

A ProblemsLabelProvider answers the displayed content of each column of the problems table for a given problem. It subclasses StyledCellLabelProvider to be able to use Styled-Strings to display element types paler than element names. Styling policies for labels are encapsulated in the StringPolicy class. Icons are used to indicate whether a problem is related to project scope or a scenario scope, see the screenshot of Figure 4.24.

The sorting algorithm of the problems list is implemented by the view.MultiSorter implementation of ViewerSorter. The MultiSorter contains a ProblemsSorter for every column and manages a priority list of sort columns and directions, controlled by column click events. Filtering is handled by ViewerFilters for each of the categories listed in Subsection 4.3.9.

4.6 Tool Validation

The proposed framework is in use as part of a current research project at the AIT[3], concerned with the development of visualized, multi-model simulation software to support complex large-scale public transport planning.

The Scenario Builder software tool provides adequate flexibility to support various work-flows and problem statements. Some examples are reported in the following to illustrate the improvement in efficiency gained by the tool.

[3]AIT Austrian Institute of Technology GmbH, TECHbase Vienna, 1210 Wien

4.6.1 Workflow Example 1: Creating a Simulation Scenario from CAD data

The CAD import feature allows for easy prototyping of scenario geometry. As CAD plans often contain a large amount of detail-scale information irrelevant for pedestrian simulation, some pre-editing of the available CAD data might be necessary. Still, information filtering based on a selection of the imported plan layers is possible directly within Scenario Builder.

CAD imports create vector layouts available in a Scenario Builder project. The shapes of imported layouts are not marked as obstructions and do not have any items assigned, therefore they are initially "invisible" to the simulation (i.e. they are not included in GPSI output data). The semantics assignment needs to be done in an extra step by selecting shapes and setting them to be obstructive or represent items. Additional item occurrences or layout details can be created by drawing shapes manually.

For example, if appropriate CAD data is available, in the imported plan all shapes can be selected and marked to be walls using the context menu's "make obstruction" action. Afterwards, special regions and objects like waiting queues, ticket machines or elevators can be created in the Project Explorer and dragged into shapes manually drawn in the layout.

Even if no CAD plans but instead scans of sketches or photographies are available, Scenario Builder supports geometry generation by letting the user select any image as a drawing background of the layout editor.

4.6.2 Workflow Example 2: Comparing Microscopic, Mesoscopic and Macroscopic Simulation of a Scenario

Though layouts of any type can be created from scratch and stand alone, a typical workflow involves deriving a grid layout from a vector layout, and creating a graph layout based on any of the two above mentioned. This is for example the case when simulation models of different granularity (see Subsection 2.2.1) should be applied for comparison or hybrid simulation.

Once a vector layout of a scenario exists, either manually drawn or imported from CAD data, a corresponding grid layout can be developed following these steps:

- Create a new grid layout in the Project Explorer.
- Open the Layout Editor for the new grid layout.
- Select the base vector layout as background layout using the Element Inspector.
- Adjust cell size, grid size and grid orientation using the Element Inspector.
- Select all cells or the cells of required areas.
- Perform layout transformation for the selected cells using a toolbar action.
- Manually perform partial corrections or beautifications as needed.

A graph layout based on an existing vector layout or grid layout can be created as follows.

- Create a new graph layout in the Project Explorer.
- Open the Layout Editor for the new graph layout.
- Select the base layout as background layout using the Element Inspector.
- Draw nodes and links superimposing the background layout.

This workflow results in a stack of compatible layouts of different types for one scenario section, representing consistent semantics because of identical items assigned by reference to spatial occurrences. The integrated solution allows for centralized management of layout data instead of inhomogeneos artifacts maintained using different tools that are likely to drift apart related to common simulation parameters.

4.6.3 Workflow Example 3: Evaluating Variations of an existing Scenario

A typical use case of pedestrian simulation is the estimation of effects of environmental design alternatives. Scenario Builder addresses this problem category by means of *scenario overrides*, a concept allowing multiple scenarios of a project to share common data but define particular per-scenario variations. Scenario overrides can be applied for the following object types.

- values of user-defined simulation parameters,
  e.g. to define a different elevator acceleration or maximum walking speed
- facility types, and therefore facility parameterization
  e.g. to use a different elevator type or replace a conventional door by a sliding door
- layouts,
  e.g. to change the width of a corridor or add obstacles
- od-maps, and therefore the simulation population,
  e.g. to examine overcrowded situations or alternative flows

Without explicit tool support, scenario variations would require branched versions, and therefore error-prone redundancy, or frequent reconfiguration of simulation data.

4.6.4 Workflow Example 4: Migrating existing Scenario data to a changed Simulation Model

When new versions of simulation models evolve, their input data models possibly also change. Therefore existing input data needs to be updated to migrate scenarios to changed model versions. Also, existing scenarios might be intended to be simulated using completely different or new models.

The input data model of a simulation model is reflected by a metadata definition file interpreted by Scenario Builder. The tool allows to switch metadata definitions of a project at any time. Metadata switching therefore implements model migration and is not a trivial feature. Project data needs to be revalidated and decisions need to be made about elements and parameter values that become incompatible with the new metadata definition. Scenario Builder provides a best-effort strategy that maintains parameterization wherever possible and remembers elements and values that need to be dropped for any potential future use or stepping back to a previous metadata definition. After a metadata switch, missing values or validation problems are marked and reported by the Problems View to guide the user through the necessary manual steps of model migration.

Without the proposed tool-supported model migration policy, simulation scenario data needs to be manually reworked to meet the requirements of changed simulation models. Stepping back to the original version means losing all intermediate changes to the input data or doing the cumbersome manual metadata switch again backwards.

5 Conclusions and Future Work

In this thesis, a categorization of existing pedestrian modelling approaches is given and representatives of different characteristics are reviewed. Based on an analysis of the their data models, a generic input data model and its implementation in XML schemas is presented. The *Generic Pedestrian Simulation Interface* supports vector-, grid- and graph-based spatial specification of multi-section simulation environments, and a flexible structure of logical concepts used to enhance spatial regions with model-specific data. A rich parametrization layer provides adaptability to a broad range of concrete models. The population of simulation scenarios is modelled by a type hierarchy of individuals, pedestrian groups and frequency distributions, annotated with trip information.

The second part of the framework proposal addresses application support of model input generation conforming to the generic interface. Atop of structural conditions imposed by the data model, a set of workflow-related requirements is identified, covering multi-scenario management and comparison, data formats and geometry prototyping. A user interface concept meeting the requirement analysis is illustrated, a decomposition of the overall functionality into main application parts is given and features are listed at a more detailled level.

The user interface concept has been implemented by a production level desktop application called *Scenario Builder*, which is discussed in the final part of the work. An overview of the base frameworks and libraries and their integration is given, before the implementation architecture is explained. Scenario Builder is an *Eclipse Rich Client Platform* standalone application, using the *Eclipse Graphical Editing Framework* for graphical editors of the simulation infrastructure. Object-to-XML mapping is realized using the *JAXB* architecture. Selected design specifics are discussed for each of the main components - domain model, view and controller.

This work attempts to unify input generation for pedestrian simulation models by providing a generic data interface and software tool support. It focuses on benefits emerging for simulation model developers and researchers by enabling early prototyping, relieving model implementations from common parts required to design scenarios, and therefore alleviating the focus on the actual pedestrian model. Future work should address the development of code libraries implemented in various programming languages that mediate between GPSI data and in-memory representations of concrete simulation models. Taken a step further, that library could be designed as a pluggable runtime framework that provides infrastructure functionality for higher-level model implementations. Finally, the ideas of reuse, decoupling and platform independence also apply to visualization and analysis, see Figure 1.4, wherefore a similar approach or an extension of GPSI should be considered for that component.

Bibliography

[ARC] Airport Research Center GmbH homepage, http://airport-consultants.com, accessed in November 2009.

[ASDB08] Mohamed Adi Azahar, Mohd Shahrizal Sunar, Daut Daman, and Abdullah Bade. Survey on Real-Time Crowds Simulation. pages 573–580, 2008.

[AW87] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*, pages 3–10, 1987.

[BA00] Victor Blue and Jeffrey Adler. Modeling Four-Directional Pedestrian Flows. *Transportation Research Record: Journal of the Transportation Research Board*, 1710:20–27, January 2000.

[Bel58] Richard E. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[BFV07] Stefania Bandini, Mizar Luca Federici, and Giuseppe Vizzari. Situated Cellular Agents Approach to Crowd Modeling and Simulation. *Cybernetics and Systems: An International Journal*, 38:729–753, 2007.

[BKSZ01] Carsten Burstedde, Kai Klauck, Andreas Schadschneider, and Jürgen Zittartz. Simulation of pedestrian dynamics using a two-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295(3-4):507–525, June 2001.

[BM04] Paul Biron and Ashok Malhotra. XML Schema Part 2: Datatypes (Second Edition). *W3C Recommendation*, 2004.

[BMNS05] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *Proceedings of the 14th international conference on World Wide Web*, pages 712–721, Chiba, Japan, 2005. ACM.

[BNdB04] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML schema: a practical study. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pages 79–84, Paris, France, 2004. ACM.

[BPSM+08] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Recommendation*, 2008.

[BSB07] Dietmar Bauer, Stefan Seer, and Norbert Brändle. Macroscopic pedestrian flow simulation for designing crowd control measures in public transport after special events. In *SCSC: Proceedings of the 2007 summer computer simulation conference*, pages 1035–1042, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[BT86] Aloys Borgers and Harry Timmermans. A Model of Pedestrian Route Choice and Demand for Retail Facilities within Inner-City Shopping Areas. *Geographical Analysis*, 18:115–128, 1986.

[CEM03] Charles E. Campbell, Andrew Eisenberg, and Jim Melton. XML schema. *SIGMOD Rec.*, 32(2):96–101, 2003.

[CIS] CISSP on the (ISC)2 homepage, https://www.isc2.org/cissp, accessed in December 2010.

[Cro] Crowd Dynamics Ltd. homepage, http://www.crowddynamics.co.uk, accessed in November 2009.

[Daa02] Winnie Daamen. SimPed: a Pedestrian Simulation Tool for Large Pedestrian Areas. *conference proceedings EuroSIW*, 2002. CDROM. Simulation Interoperability Standards Organization, Orlando, Florida.

[Daa04] Winnie Daamen. *Modelling Passenger Flows in Public Transport Facilities.* PhD thesis, Delft University of Technology, 2004.

[DFT92] John C. Doyle, Bruce A. Francis, and Allen R. Tannenbaum. *Feedback Control Theory.* Macmillan, 1992.

[DH03] Winnie Daamen and Serge Hoogendoorn. Research on pedestrian traffic flow in the Netherlands. *Proceedings Walk*, 21(IV):101–117, 2003.

[DHB05] Winnie Daamen, Serge Hoogendoorn, and Piet Bovy. First-Order Pedestrian Traffic Flow Theory. *Transportation Research Record: Journal of the Transportation Research Board*, 1934(-1):43–52, January 2005.

[DJT01] Jan Dijkstra, Joran Jessurun, and Harry J. P. Timmermans. A Multi-Agent Cellular Automata Model of Pedestrian Movement. *M. Schreckenberg and S.D. Sharma (ed.): Pedestrian and Evacuation Dynamics. Springer-Verlag.*, pages 173–181, 2001.

[DT02] Jan Dijkstra and Harry Timmermans. Towards a multi-agent model for visualizing simulated user behavior to support the assessment of design performance. *Automation in Construction*, 11(2):135–145, February 2002.

[DTdV05] Jan Dijkstra, Harry J. P. Timmermans, and Bauke de Vries. Modelling Behavioural Aspects of Agents in simulating Pedestrian Movement. *Proceedings of CUPUM 05, Computers in Urban Planning and Urban Management*, 2005.

[DTdV07] Jan Dijkstra, Harry J. P. Timmermans, and Bauke de Vries. Empirical Estimation Of Agent Shopping Patterns for simulating Pedestrian Movement. *CUPUM07 Computers in Urban Planning and Urban Management*, 2007.

[DTdV09] Jan Dijkstra, Harry Timmermans, and Bauke de Vries. *Modeling Impulse and Non-Impulse Store Choice Processes in a Multi-Agent Simulation of Pedestrian Activity in Shopping Environments*, chapter 4, pages 63–85. Emerald Group Publishing Limited, 2009.

[DWS+92] K.H. Drager, J. Wicklund, H. Soma, D. Duoung, A. Violas, and V. Lan. EVACSIM: A comprehensive evacuation simulation tool. *Proceedings of the 1992 Emergency Management and Engineering Conference*, pages 101–108, 1992.

[DXF] Autodesk homepage, DXF Reference, http://usa.autodesk.com/adsk/servlet/-item?siteID=123112&id=12272454, accessed in September 2010.

[EEC] Eurocontrol Experimental Centre homepage, http://www.eurocontrol.int, accessed in November 2009.

[FMT01] Michael Florian, Michael Mahut, and Nicolas Tremblay. A Hybrid Optimization-Mesoscopic Simulation Dynamic Traffic Assignment Model. *IEEE Intelligent Transportation Systems Conference Proceedings*, pages 118–121, 2001.

[Fri06] Jeffrey Friedl. *Mastering Regular Expressions, Third Edition.* O'Reilly Media, 2006.

[FW04] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer (Second Edition). *W3C Recommendation*, 2004.

[GEFa] Eclipse GEF, http://www.eclipse.org/gef, accessed in October 2010.

[GEFb] Eclipse Graphical Editing Framework Programmer's Guide, http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.gef.doc.isv/guide.html, accessed in October 2010.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gre35] Bruce D. Greenshields. A study of traffic Capacity. *Proceedings 14th Annual Meeting Highway Research Board*, pages 448–477, 1935.

[GSN04] Christian Gloor, Pascal Stucki, and Kai Nagel. Hybrid techniques for pedestrian simulations. *Lecture Notes in Computer Science, Cellular Automata, 6th International Conference on Cellular Automata for Research and Industry*, 2004.

[GSTH08] Donald Gross, John F. Shortle, James M. Thompson, and Carl M. Harris. *Fundamentals of Queueing Theory.* Wiley (New York), 2008.

[Hal] Halcrow Group homepage, http://www.halcrow.com, accessed in November, 2009.

[Har07] Shon Harris. *CISSP Certification All-in-One Exam Guide, Fourth Edition.* McGraw-Hill Osborne, 2007.

[HB04] Serge P. Hoogendoorn and Piet H.L. Bovy. Pedestrian route-choice and activity scheduling theory and models. *Transportation Research Part B: Methodological*, 38:169–190, 2004.

[HBJW05] Dirk Helbing, Lubos Buzna, Anders Johansson, and Torsten Werner. Self-Organized Pedestrian Crowd Dynamics: Experiments, Simulations, and Design Solutions. *TRANSPORTATION SCIENCE*, 39(1):1–24, 2005.

[HCM85] *Highway Capacity Manual.* Transportation Research Board (Washington, D.C), special report 209 edition, 1985.

[HD04] Serge Hoogendoorn and Winnie Daamen. Design Assessment of Lisbon Transfer Stations using Microscopic Pedestrian Simulation. *Computers in railways*, IX:135–147, 2004.

[Hel92] Dirk Helbing. A Fluid Dynamic Model for the Movement of Pedestrians. *Complex Systems*, 6:391–415, May 1992.

[HFMV02] Dirk Helbing, Illés J. Farkas, Péter Molnár, and Tamás Vicsek. Simulation of Pedestrian Crowds in Normal and Evacuation Situations. *Pedestrian and Evacuation Dynamics*, pages 21–58, 2002.

[HFV00] Dirk Helbing, Illes Farkas, and Tamas Vicsek. Simulating Dynamical Features of Escape Panic. *Nature*, 407:487–490, 2000.

[HM95] Dirk Helbing and Péter Molnár. Social force model for pedestrian dynamics. *Physical Review E*, 51(5):4282–4286, May 1995.

[HSW08] Randy Hudson, Pratik Shah, and Joe Winchester. *The Graphing Editing Framework (GEF).* Addison-Wesley Publishing Company, 2008.

[HTRS03] André Hanisch, Juri Tolujew, Klaus Richter, and Thomas Schulze. Online Simulation of Pedestrian Flow in Public Buildings. *Proceedings of the 2003 Winter Simulation Conference*, pages 1635–1641, 2003.

[Hug02] Roger L. Hughes. A continuum theory for the flow of pedestrians. *Transportation Research Part B: Methodological*, 36(6):507–535, July 2002.

[HWZ$^+$09] Ling Huang, S.C. Wong, Mengping Zhang, Chi-Wang Shu, and William H.K. Lam. Revisiting Hughes' dynamic continuum model for pedestrian flow and the development of an efficient solution algorithm. *Transportation Research Part B: Methodological*, 43:127–141, 2009.

[IES] Integrated Environmental Solutions Limited, Simulex homepage, http://www.iesve.com/Software/VE-Pro/Simulex, accessed in November 2009.

[IST] Integrierte Sicherheits-Technik GmbH homepage, http://www.ist-net.de, accessed in November 2009.

[Java] Java EE 6 Technologies homepage, http://www.oracle.com/technetwork/java/-javaee/tech/index.html, accessed in November 2010.

[Javb] Oracle Java homepage, http://www.oracle.com/java, accessed in October 2010.

[JAXa] JAXB Reference Implementation project homepage, http://jaxb.dev.java.net, accessed in November 2010.

[JAXb] JAXB specification, JSR-222, http://jcp.org/aboutJava/communityprocess/final/-jsr222/index.html, accessed in November 2010.

[JOS] Java Object Serialization, http://download.oracle.com/javase/6/docs/technotes/-guides/serialization/index.html, accessed in November 2010.

[Klü03] Ludwig Hubert Klüpfel. *A Cellular Automaton Model for Crowd Movement and Egress Simulation.* PhD thesis, Universität Duisburg-Essen, 2003.

[KR07] Franziska Klügl and Guido Rindsfüser. Large-Scale Agent-Based Pedestrian Simulation. *Multiagent System Technologies*, pages 145–156, 2007.

[Kre07] Tobias Kretz. *Pedestrian Traffic - Simulation and Experiments.* PhD thesis, Universität Duisburg-Essen, 2007.

[KS02] Ansgar Kirchner and Andreas Schadschneider. Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. *Physica A: Statistical Mechanics and its Applications*, 312(1-2):260–276, September 2002.

[KS06] Tobias Kretz and Michael Schreckenberg. F.A.S.T. - Floor field- and Agent-based Simulation Tool. *International Symposium of Transport Simulation (ISTS06)*, 2006.

[KSMK05] Hubert Klüpfel, Michael Schreckenberg, and Tim Meyer-König. Models for Crowd Movement and Egress Simulation. *Traffic and Granular Flow*, pages 357–372, 2005.

[KTa] KTable homepage, http://sourceforge.net/projects/ktable, accessed in November 2010.

[LC00] Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Rec.*, 29(3):76–87, 2000.

[LC07] Pei-Wei Lin and Gang-Len Chang. A generalized model and solution algorithm for estimation of the dynamic freeway origin-destination matrix. *Transportation Research Part B: Methodological*, 41(5):554–572, June 2007.

[Leg] Legion homepage, http://www.legion.com, accessed in November 2009.

[LKF05] Taras I. Lakoba, D. J. Kaup, and Neal M. Finkelstein. Modifications of the Helbing-Molnár-Farkas-Vicsek Social Force Model for Pedestrian Evolution. *Simulation*, 81(5):339–352, 2005.

[log] Apache log4j homepage, http://logging.apache.org/log4j, accessed in November 2010.

[Lot] IBM Lotus Symphony homepage, http://symphony.lotus.comm accessed in October 2010.

[Løv94] Gunnar G. Løvås. Modeling and simulation of pedestrian traffic flow. *Transportation Research Part B: Methodological*, 28(6):429–443, December 1994.

[Løv95] Gunnar G. Løvås. On performance measures for evacuation systems. *European Journal of Operational Research*, 85(2):352–367, September 1995.

[Mac] STEPS software information on the Mott MacDonald Ltd. homepage, http://www.mottmac.com/skillsandservices/software/stepssoftware, accessed in November 2009.

[Mas] Massive Software homepage, http://www.massivesoftware.com, accessed in November 2009.

[Mat] Apache Commons Math homepage, http://commons.apache.org/math, accessed in November 2010.

[Met] Eclipse Metrics Plugin, http://metrics.sourceforge.net, downloaded in October 2010.

[ML05] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications.* Addison-Wesley Professional, 2005.

[MP02] Rashim Mogha and V. V. Preetham. *Java Web Services Programming.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[MSCCJ10] R. Mínguez, S. Sánchez-Cambronero, E. Castillo, and P. Jiménez. Optimal traffic plate scanning location for OD trip matrix and route estimation in road networks. *Transportation Research Part B: Methodological*, 44(2):282–298, February 2010.

[MVCA08] Michael Monteleone, Niek Veraart, Suany Chough, and Wendy Aviles. Pedestrian Simulation Modeling Study for World Trade Center Memorial. *Transportation Research Record: Journal of the Transportation Research Board*, 2073(-1):49–57, December 2008.

[Nag96] Kai Nagel. Particle hopping models and traffic flow theory. *Phys. Rev. E*, 53(5):4655–, May 1996.

[Neu66] John Von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, 1966.

[NM09] Martin Nečaský and Irena Mlýnková. Discovering XML keys and foreign keys in queries. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 632–638, Honolulu, Hawaii, 2009. ACM.

[Nob07] Claudia Nobis. Multimodality: Facets and Causes of Sustainable Mobility Behavior. *Transportation Research Record: Journal of the Transportation Research Board*, 2010:35–44, 2007.

[NS92] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *J. Phys. I France*, 2(12):2221–2229, December 1992.

[NSK$^+$06] Katsuhiro Nishinari, Ken Sugawara, Toshiya Kazama, Andreas Schadschneider, and Debashish Chowdhury. Modelling of self-driven particles: Foraging ants and pedestrians. *Physica A: Statistical Mechanics and its Applications*, 372(1):132–141, December 2006.

[NW04] Steve Northover and Mike Wilson. *SWT: The Standard WidgetTtoolkit, Volume 1.* Addison-Wesley Professional, 2004.

[ope] opencsv homepage, http://opencsv.sourceforge.net, accessed in November 2010.

[OR01] P. Å. Olsson and M. A. Regan. A comparison between actual and predicted evacuation times. *Safety Science*, 38(2):139–145, July 2001.

[PGM09] Daniel R. Parisi, Marcelo Gilman, and Herman Moldovan. A modification of the Social Force Model can reproduce experimental data of pedestrian flows in normal conditions. *Physica A: Statistical Mechanics and its Applications*, 388(17):3600–3608, September 2009.

[PTV] PTV Planung Transport Verkehr AG homepage, http://www.ptvag.com, accessed in November 2009.

[Qua] Quadstone Paramics homepage, http://www.paramics-online.com, accessed in November 2009.

[Rai04] Management Systems for Large Events and Perturbations at Stations. Research Brief T161-RB, Rail Safety and Standards Board, Evergreen House, 160 Euston Road, London, 2004.

[RCP] Eclipse Rich Client Platform homepage, http://wiki.eclipse.org/Rich_Client_Platform, accessed in October 2010.

[Rei03] Edwin D. Reilly. Power user. In *Encyclopedia of Computer Science*, pages 1419–1419. John Wiley and Sons Ltd., Chichester, UK, 2003.

[RFRA07] Vitorino Ramos, Carlos Fernandes, Agostinho C. Rosa, and Ajith Abraham. Computational Chemotaxis in Ants and Bacteria over Dynamic Environments. *Congress of Evolutionary Computation*, CEC 07:1009–1017, 2007.

[RSK07] C. Rogsch, A. Seyfried, and W. Klingsch. Comparative Investigation of the Dynamic Simulation of Foot Traffic Flow, 2007.

[Rub06] Dan Rubel. The Heart of Eclipse. *Queue*, 4(8):36–44, 2006.

[Sav] Savannah Simulations homepage, savannah-simulations.com, accessed in November 2009.

[Sho] ShopSim homepage, http://www.shopsim.biz, accessed in November 2009.

[SHST07] Siamak Sarmady, Fazilah Haron, M. M. Mohd Salahudin, and Abdullah Zawawi Hj. Talib. Evaluation of Existing Software for Simulating the Crowd at Masjid Al-Haram. *Jurnal Pengurusan JWZH*, 1:137–145, 2007.

[Sima] Simulink homepage, http://www.mathworks.com/products/simulink, accessed in November 2009.

[Simb] SimWalk homepage, http://www.simwalk.ch, accessed in November 2009.

[SKK$^+$08] Andreas Schadschneider, Wolfram Klingsch, Hubert Klüpfel, Tobias Kretz, Christian Rogsch, and Armin Seyfried. Evacuation Dynamics: Empirical Results, Modeling and Applications. In B. Meyers, editor, *Encyclopedia of Complexity and System Science*. Springer-Verlag, 2008.

[SLR07] Tilman A. Schenk, Günter Löffler, and Jürgen Rauh. Agent-based simulation of consumer behavior in grocery shopping on a regional level. *Journal of Business Research*, 60(8):894–903, August 2007.

[SPKI04] Flora Dilys Salim, Rosanne Price, Shonali Krishnaswamy, and Maria Indrawan. UML Documentation Support for XML Schema. *Australian Software Engineering Conference*, 0:211, 2004.

[Sti00] Keith G. Still. *Crowd Dynamics*. PhD thesis, University of Warwick, 2000.

[SWT] SWT homepage, http://www.eclipse.org/swt, accessed in October 2010.

[TA04] Juri Tolujew and Felix Alcalá. A Mesoscopic Approach to Modeling and Simulation of Pedestrian Traffic Flows. *Proceedings 18th European Simulation Multiconference*, 2004.

[TBM08] Kardi Teknomo, Dietmar Bauer, and Thomas Matyus. Pedestrian Route Choice Self-Organization. *3rd International Symposium of Transportation Simulation*, 2008.

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures (Second Edition). *W3C Recommendation*, 2004.

[Tim04] Harry Timmermans. *Retail location and consumer spatial choice behavior*, chapter 7, pages 133–147. Kluwer Academic Publishers, 2004.

[TM95a] Peter A. Thompson and Eric W. Marchant. A computer model for the evacuation of large building populations. *Fire Safety Journal*, 24(2):131–148, 1995.

[TM95b] Peter A. Thompson and Eric W. Marchant. Testing and application of the computer model 'SIMULEX'. *Fire Safety Journal*, 24(2):149–166, 1995.

[TM07] Kardi Teknomo and Alexandra Millonig. A Navigation Algorithm for Pedestrian Simulation in Dynamic Environments. *Proceeding of the 11th World Conference on Transportation Research (WCTR)*, 2007.

[Tra] TraffGo HT GmbH homepage, http://www.traffgo-ht.com, accessed in November 2009.

[UAF] Urban Analytics Framework homepage, http://www.pedestrian-simulation.com, accessed in November 2009.

[VIS] VISSIM homepage, http://www.ptvag.com/software/transportation-planning-traffic-engineering/software-system-solutions/vissim, accessed in November 2009.

[Vuz] Vuze homepage, http://www.vuze.com, accessed in October 2010.

[Win92] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, 3rd edition, 1992.

[Wol94] Stephen Wolfram. *Cellular Automata and Complexity: Collected Papers*. Addison-Wesley, 1994.

[Zam07] Fabio Zambetta. Simulating sensory perception in 3D game characters. In *IE '07: Proceedings of the 4th Australasian conference on Interactive entertainment*, pages 1–3, Melbourne, Australia, Australia, 2007. RMIT University.

[ZWS08] Nana Zhu, Jiangyan Wang, and Jiangang Shi. Application of Pedestrian Simulation in Olympic Games. *Journal of Transportation Systems Engineering and Information Technology*, 8(6):85–90, December 2008.