Project number:	288577
Project acronym:	UrbanAPI
Project title:	Interactive Analysis, Simulation and Visualisation Tools for Urban Agile Policy
Instrument: **Call identifier:** **Activity code:**	STREP FP7-ICT-2011-7 ICT-2011.5.6 ICT Solutions for governance and policy modelling

Start date of Project:	2011-09-01
Duration:	36 month

Deliverable reference number and title (as in Annex 1):	**D3.3 Rule User Interface Documentation**
Due date of deliverable (as in Annex 1):	**15**
Actual submission date: **Revision:**	*see "History" Table below*

Organisation name of lead contractor for this deliverable:
Fraunhofer IGD

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Title:
Rule User User Interface Elements Documentation
Author(s)/Organisation(s):
Michel Krämer, Andreas Stein, Robert Gregor / Fraunhofer IGD
Working Group:
WP3
References:
D3.5 Data integration components

Short Description:
This report describes the implemented user interface for rule editing. Rule-based systems are used for various purposes within the project. The report discusses existing solutions (state of the art) and how they can be adapted for the project. It also describes the motivation behind using rule-based systems. It also gives an outlook on what will be developed in the future. This is the first version of this report. It will be superseded by an improved version after the second cycle (planned for PM30).
Keywords:
Rule-based system, production rules, domain-specific languages, human-machine interaction

History:

Version	Author(s)	Status	Comment	Date
001	Michel Krämer	New	Created document with initial layout	2012-11-06
002	Andreas Stein	rfc	Added description of graphical rule editor in CityServer3D AdminTool and rule execution example	2012-11-13
003	Michel Krämer	rfc	Added sections 1, 2, 3, 4, 6, 7, 8	2012-12-14
004	Robert Gregor	rfc	Revised section 4.1 + minor fixes	2012-12-19
005	Michel Krämer	Final	Revised document based on comments from internal review	2013-01-14
006	Michel Krämer	Final	Revised document based on review comments: • the state of the art in rule-based languages and rule editors: see sections 5.3 and 5.4 • identification of the entities to be used to build the rules: see	2013-09-08

			ontologies in sections 8.1.3 and 8.1.4 as well as summary in section 8.4 • identification of the operators to be applied to a single entity as well as between the entities: see section 8.4 • examples of rules for data integration, data processing, visual preparation and policy modelling: see example DSLs in sections 8.2 and 8.3 as well as data integration example in section 6.2.1.1	

Review:			
Version	Reviewer	Comment	Date
004	Helmut Augustin	See quality assurance form	2013-01-07
004	Wolfgang Loibl	See quality assurance form	2013-01-09
006	Wolfgang Loibl, Helmut Augustin	Comments in track changes	2013-24-09

Table of contents

About this document

Rule-based systems are used for various purposes within the urbanAPI project: to resolve conflicts during data integration, to prepare and edit data, to change visual attributes prior to presentation and to enforce policy rules during policy modelling processes. Domain-specific languages (DSLs) can help the user to easily declare his own rules.

This document contains an overview over existing rule-based systems (state of the art) and how they can be adapted for the project. It will also describe the motivation behind using domain-specific languages for rule-based systems. Furthermore, it describes the rule editing user interface developed for the UrbanAPI project. This document finally gives an outlook on what will be developed in the future in the UrbanAPI project.

The document will have two versions. A second, revised version of this document is planned for PM 30. The final version of this document will contain all user interface elements, a description of the new language IDE for textual DSLs as well as the scientific work conducted.

1 Introduction and motivation

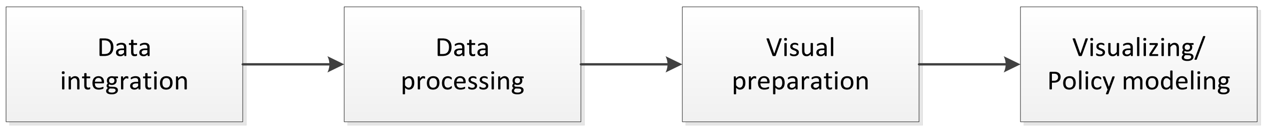
Urban planning aims to manage the territory in order to address the key political concerns of European citizens, including climate change, greenhouse gas emissions, uncontrolled urban sprawl, urban health, traffic, noise, energy consumption and biodiversity loss etc. However, this presents major challenges for urban planners and politicians as cities are extremely complex systems, and the various drivers of change, impacts and responses are strongly interrelated, support, alter or compete with each other.

Indeed the effective governance of the cities and city regions of Europe towards smart cities today is fundamentally undermined by this urban complexity, whereby the high degree of interconnectedness and multiple interactions between socio-economic and environmental factors in a territorial context create major barriers to the effective implementation of sustainable urban development.

ICT enabled governance of cities offers substantial opportunity for the application of enhanced intelligence in urban management, to overcome barriers to sustainable development. This can be achieved by enhanced assessment of urban complexity, improved decision-making support, all facilitating the delivery of more sustainable compact cities. Moreover the wider potential of ICT enabled urban governance is evident in the ability to simultaneously achieve effective management of the complexity of the city, and engage citizens in defining their urban futures.

The UrbanAPI tools will provide advanced ICT-based intelligence in three urban planning contexts. First, directly addressing the issue of stakeholder engagement in the planning process by the development of enhanced virtual reality visualisation of neighbourhood development proposals. Second, at the city-wide scale, developing mobile-device location data based applications that permit the analysis and visual representation of spatial activity across the territory of the city and in relation to the various land-use elements of the city. Finally, developing simulation tool applications in the city-region context addressing multiple challenges in responding to the simultaneous demands of both expanding city populations for certain European cities, and declining and frequently ageing populations elsewhere.

These three contexts have common requirements (see project deliverable D2.1) which lead to a general processing chain. This chain consists of four steps: data integration, data processing, visual preparation, and visualization for policy modelling. All scenarios from deliverable D2.1 deal with different heterogeneous data sources that have to be integrated into one data set. For example, for the city of Vitoria-Gasteiz building footprints, LIDAR data, zoning maps and the municipality’s tree cadastre have to be merged into a single scene. In most of the scenarios the data has then to be processed. For example, in Vitoria-Gasteiz the building footprints have to be extruded to 3D objects based on metadata and information derived from the LIDAR data set. After these first two steps, the information has to be visualized. This often includes a pre-processing step where the visual attributes of the 3D objects in the scene will be altered. In Vitoria-Gasteiz, for example, the different areas from the zoning maps have to be displayed in specific colours. Finally, the scene will be visualized in order to conduct policy modelling.



The four steps in the processing chain highly depend on the specific scenario—i.e. user requirements—and the actual data being processed. Therefore the ICT tools developed in UrbanAPI need to be highly configurable. There are various ways to achieve this:

- Special tools could be developed that are customized to the specific scenarios identified in this project. If done appropriately, these tools will of course fulfil all user requirements. However, they will be specialized for the UrbanAPI use cases only, and it will not be possible to use them for other scenarios in the future as well.

- The tools could be made configurable to a certain degree. This would, for example, include the colours for areas in zoning maps, and other parameters. This solution would be much more flexible than the first one, but only parameters identified in this project could be included. As described above, geo data processing highly depends on the user's needs, the scenario and the data to be processed. A more generic solution which is able to process arbitrary data and to fulfil user requirements that have not been identified in this project yet is hence needed.

- The third solution is based on rule-based systems. These systems allow the user to declaratively specify how the data should be processed. This is the most flexible alternative. The user defines rules which consist of a condition (also known as "left-hand side") and a conclusion (sometimes called action or "right-hand side"). A rule's left-hand side is used to select objects with certain attributes from the data set—e.g. all areas from the zoning map that represent streets. The right hand-side contains actions that operate on the selected objects.

With a rule-based system the dynamic parts of the software can be described with production rules that are on the one hand highly adaptable to the application domain and on the other hand can be changed during runtime. So, they do not need the software developer to recompile the whole system, especially when just a small part of the process has changed. Rule-based systems have shown to be powerful enough for processes such as data integration (Thum & Krämer, 2011), (Reitz, Krämer, & Thum, 2009) or even data validation and quality evaluation (Coors & Krämer, 2011), (van Oosterom, 2009). Since rule-based systems have been used for quite some time in the area of artificial intelligence and machine learning, they are already quite mature (Forgy, 1982), (Miranker, 1987), (Doorenbos, 1995). The following paragraphs describe how the four processing steps can benefit from rule-based systems.

- **Data integration.** Geospatial data sources are typically quite heterogeneous. There are a number of data exchange formats and different schemas. Data sources often have been created with different tools and been processed in various ways. This makes the data integration process rather complex. It greatly depends on the data to be processed. In order to support a wide range of data sources, a quite flexible system is needed that can be customized by the user without requiring the software vendor to change or extend the GIS. Rule-based systems can be of a great help here, because rules can be specified by the end-users directly in the GIS. The declarative style of production rules allows for a much better readability than a general purpose scripting language would do (see discussion below).

- **Data processing.** This step is about processing the data that has been integrated in the first one in order to derive new data. This step depends on the user's requirements which may be different in each project or application the user is working on. With a GIS that uses a rule-based system and that is therefore highly configurable, the user can define automated processes that meet these requirements. For example, in Vitora-Gasteiz building footprints have to be combined with the terrain model and LIDAR data in order to derive 3D building models. With the system developed in this project the user can define an automated process that calculates a building's height at a certain position using the LIDAR data and then extrudes the footprint to the correct block model (see example in section 6.2.1.1).

- **Visual preparation.** Rules can also be very helpful in this step. For example, the following rule could be used to colorize all roofs red:

```
when
    there is a roof r
then
    set color of r to red
```

This step can also make use of a production rule system's forward chaining (see discussion below):

```
when
    there is a plane e and
    the normal vector of e points upwards
then
    e is a roof

when
    there is a roof e
then
    set color of e to red
```

These two rules will be executed one after another. To be precise: firing the first rule which infers that a plane whose normal vector points upwards is a roof will automatically make the second rule fire as well. This process is called forward chaining. The language used here is a domain-specific language (see section 5.2), that means a language that is tailored towards a very specific application domain. Such a language could contain terms like "points upwards" which have to be defined elsewhere. For example, "points upwards" could mean that the angle between the plane and the coordinate system's X-Y-plane is lower or equal than 45°.

- **Policy modelling.** Rules can also be used during interactive policy modelling. For example:

  ```
  when
      there is a waste bin w and
      there is an circular area c with a radius of 2 kilometres around w and
      the number of waste bins within c is less than 20
  then
      alert user
  ```

  For a more complete discussion on policy modelling rules see section 8.

2 Motivation for domain-specific rule languages

In the urbanAPI project, domain-specific languages (DSLs) will be used to allow domain experts without a background in computer science to define their own rules. Krämer motivates the use of DSLs for rule-based systems as follows (Krämer M. , 2013): “Rule-based systems in their current form can be quite useful in the applications described above, but they can also be rather hard to understand for non-IT personnel. Rule definition languages are currently rather complex. The user should have some experience in programming. He or she is also often expected to have a deep understanding of internal matters—i.e. data models the rules will operate on, algorithms, etc. Handling all these details keeps him or her from focusing on the actual problem.

This is somewhat contradictory to the purpose of rule-based systems. Since production rules are described in a declarative way, they should allow the user to focus on *what* should be done and not *how*. However, rule-based systems are not tailored to one specific application. They rather try to be generic and flexible. This flexibility leads to several issues. For example, rule-based systems allow for a technique called *rule chaining*: knowledge inferred from executing one or more rules may lead to other rules being fired. This technique can lead to a network of dependent rules which can quickly become overly complex even for use cases that appear to be simple at first glance. Apart from that, improperly defined rule conditions can make pattern matching very slow when the system is forced to search the knowledge base linearly or to even build cross products of facts.

Current production rule systems such as Drools[1] or JESS[2], for example, also do not make use of application-dependent specifics. For example, in the geospatial domain spatial indexes (R-trees, quad-trees, etc.) can be used to search the knowledge base a lot faster than it would be possible

[1] http://www.drools.org
[2] http://www.jessrules.com

with a general purpose inference algorithm such as RETE. Furthermore, rule-based systems do not provide a common set of functions for specific applications—they rather leave it up to the user to define functions on his own using a general purpose programming language.

Rule languages often try to reflect the flexibility and generic nature of the underlying rule-based system. Hence, it's common to use general purpose languages for rule declaration as well. For example, JESS and Drools use LISP and Java respectively. Drools attempts to make the declaration of rule conditions a little bit easier by providing a Java dialect called MVEL which is specifically designed for conditional expressions. However, regardless of how simple rule languages seem to be structured at first glance, as soon as they reflect the full flexibility of the underlying system they become so-called *turing tarpits*, which means they allow for everything—hence they are turing complete—but are hard to learn and too complicated for even simple tasks (Perlis, 1982). A famous example for a turing tarpit is XSLT which has been designed for model transformation but is also turing complete. The use of general purpose languages for rule-based systems like it has been done in JESS or Drools does not make the situation much better. Although performing simple tasks is easier with a general purpose language, their genericness is usually not very conducive, especially when the rule-based system is used in a single application domain for very specific use cases.

The language used to define rules is the main interface between the user and the rule-based system. In order to hide the system's complexity, the rule language should be designed for a single application domain instead of being too generic. It should on the one hand have a reduced vocabulary, but should also be powerful enough to solve problems within this single application domain very easily. Such a language is usually called *Domain-Specific Language* or *DSL*. There are two forms of DSLs: internal and external ones. In this project, we focus on external DSLs which can be designed independently of any implications of a host language. In the following, the term *DSL* is hence used synonymously with *external DSL*."

3 Scientific problems

Before DSLs can fully be used in UrbanAPI some problems have yet to be solved. First of all, methods have to be developed which allow domain-specific languages to be created with the following properties:

- The languages have to be based on user requirements and it has to be made sure they are targeted to the respective application domain.

- A DSL has to be expressive enough, so problems from this application domain can be solved in an easy manner
- The language must not be too flexible, so domain experts can still handle it. Turing-tarpits should also be avoided.

One of the most important aspects here is decoupling the technical details from the actual problem description. Furthermore, while designing a DSL, one has to find a balance between the language's expressiveness and its flexibility. If single expressions (tokens) are too expressive and the vocabulary is too limited, the language will be rather inflexible. This means it can only be used in a very specific application and probably only in one very specific use case. On the other hand, the more flexible the language is, the more complicated it becomes for the domain expert to use.

4 User benefits

Domain-specific languages in urbanAPI make it a lot easier for domain experts without a background in computer science to create automated processes. Krämer summarizes the benefits of domain-specific languages for the user as follows (Krämer M. , 2013): "Compared to general purpose languages, DSLs have a very limited vocabulary. Instead, they are tailored to a specific application which makes them very easy to understand for the domain expert. The GIS user will be able to write custom scripts (i.e. rules) without being required to have a strong background in computer science. If use cases become too complex, the domain expert can still engage an external service provider to write the rules. However, the communication between the domain expert and external service provider will be a lot easier, since rules written in the DSL are more readable and thus easier to understand. Hence, the risk to get rules from the external service provider which do not meet the user's requirements will be tremendously reduced.

Domain-specific languages typically have a small vocabulary, but single expressions can be very elaborate. Defining processes for data integration, visual preparation, etc. will be much easier since the domain expert does not have to think about *how* to do something, but more about *what* should be done. So, he or she will work on a very abstract level without being required to cope with technical details. This has the following advantages:

- Scripts (i.e. rules) will be more readable and understandable. The actual problem can be much better described in this abstract way. Implementation details which would make the script a lot more complex can be omitted.
- The system can be updated or even replaced (e.g. if new algorithms should be added or if the general environment has changed) without requiring the scripts to be changed.

- Decoupling technical details from the problem may allow the same scripts to be applied to a different application (or use case within the same domain). This increases reusability.”

5 State of the art

5.1 Rule-based systems

In this section we will compare different approaches that allow rule-based expert systems to be created. We will compare the production rule system Drools with two semantic web technology frameworks—in particular we choose Sesame and Apache Jena. All three frameworks are available under an Open Source License and for all of them there are several offerings for commercial support.

Besides these three frameworks, there are also other alternatives—for example CLIPS, JESS or even the logic programming language Prolog. It is also possible to replace or supplement certain parts in a semantic web technology stack by other libraries such as different Triple Store back-ends, entire Triple Store implementations, Query Engines, OWL implementations or RDF Schema and OWL reasoners, as well as additional forward, backward or hybrid chaining reasoners[3]. However, Drools, Sesame and Jena are the ones which are among the most mature, are regularly updated provide reasonable documentation and have a quite strong community.

Drools is a Java centric production rule system. Its rules are defined in a text based form and are evaluated and compiled to Java Bytecode at runtime. While Drools has initially been developed with a focus on forward chaining, there is growing support for backward chaining as well. Compared to forward chaining, backward chaining is a process where the user describes a certain condition and the expert system uses rules to find facts that led to this condition (for example, a doctor describes a patient’s symptoms and then lets the expert system search for possible diseases).

Drools uses its own rule definition language. For the right-hand side of rules, general purpose languages such as Java, Groovy, Python, etc. can be used. Also, the framework also already contains a mechanism to create DSLs. However, this mechanism simply works with ‘search’ and ‘replace’ in order to map DSL expressions to expressions in the Drools rule language. This is extremely inflexible and only allows a very limited set of DSLs since you’re always bound to the constraints of the host language. A complete compiler/interpreter for an external DSL allows for much more, easier to understand languages.

[3] e.g. there are several pluggable reasoners for Apache Jena Jena Framework

Since Drools rules allow direct access to arbitrary Java objects as well as method invocation. A Drools rule-based system can thus read and write its state (i.e. facts) directly to an arbitrary Java Model[4], without the need for custom integration or wrapping code. Besides Drools, there is no alternative implementation that can be used in conjunction with an existing set of Drools rules. While this might be a drawback in terms of interoperability for some use cases, it also offers large benefits in terms of development productivity and ease of use in scenarios where there is no requirement to directly access or evaluate the production rules by other systems[5]. Production rules in Drools are triggered by events. This allows forward chaining to be implemented quite easily. In UrbanAPI events can also be of great use for interactive policy modelling. Whenever the domain expert changes a parameter, an event can be triggered which will make the production rule system evaluate the scene again (for an example use case, please have a look at section 8). Event-based forward chaining and interactive modelling hence complement each other quite well.

Sesame is an extendable RDF Triple Store and RDF Schema inferencing system that can be queried using SPARQL. Both facts and rules are stored inside an RDF Graph. These facts can be read and written[6] through direct use of a Java API or through SPARQL queries which can themselves be passed to the Java API as query strings or by transmitting SPARQL queries over an HTTP connection. RDF can be serialized to various text based formats and is not dependent on any specifics of a programming language, runtime environment or even specific API. This leads to easy integration and interoperability among heterogeneous systems. Facts, rules and queries can be easily shared or reused among components developed on entirely different technology stacks and platforms. A rule-based system implemented on top of Sesame can also distribute its knowledge base across multiple nodes in a network which can be especially useful for large datasets that do not fit into main memory or for seamless integration of third party RDF data that is hosted on external systems. However such a system cannot read and write its state transparently to an existing Java Business Model without the need to develop additional integration or wrapping code since the existing model has to be mapped to a RDF triple based representation. Sesame provides a built in backward-chaining reasoner that can inference new facts by relying on RDFS rules and facts stated in RDF. RDFS does allow the expression of common vocabularies and taxonomies that consist of classes, object- and data properties. However there is no support for value and cardinality restrictions or even rules involving first order logic. Thus, RDFS backward chaining on its own doesn’t fit our requirements within UrbanAPI. However there are several other alternative ways to support more complex rules and reasoning on top of Sesame. The first would be to extend Sesame by custom functions that are implemented in Java. These functions can then be referenced inside FILTER clauses of SPARQL queries and used as a simple yet flexible and

[4] such as the Common Data Model of the CityServer 3D, see D3.5 Data integration components
[5] i.e. systems not running inside a Java Runtime Environment or running on an outdated JRE
[6] Write support is introduced by SPARQL 1.1 and implemented by recent Sesame releases

extensible form of backward-chaining[7]. However it shares some of the drawbacks of a Drools based approach since these functions are implemented in Java and specific to Sesame. So while their syntax and semantics can be formally specified[8] and they can be used by standard SPARQL clients without additional effort they have to be implemented as individual extensions for each SPARQL Query Engine they should be used with.

Sesame in comparison to Drools does not offer built-in facilities for custom DSLs. Though SPARQL is itself already a DSL, it is obviously designed as a query language for Triple stores, similar to SQL for RDBMS.

There are some efforts going on to build user interfaces for semantic queries that are a lot easier to understand than mere SPARQL. For example, in the FP7 project TELEIOS[9] a Visual Query Builder UI is created which translates graphical elements to SPARQL queries. However, even though this system is rather easy to understand for users having some background in the referenced domain, it largely reflects the WHERE clause of a SPARQL query and is intended for information retrieval and not for policy modelling.

Jena is a more complex semantic web framework than Sesame. Besides offering roughly all the features of Sesame, it provides built in support for backward-chaining OWL lite reasoning[10]. OWL lite is a standardized format that can be used in a similar fashion as RDFS to declare rules for an RDF dataset. Besides the definition of classes as well as data and object properties there is support for value and cardinality constraints on properties. But in contrast to OWL full or OWL/DL there is no support for first order logic constraints. Within the UrbanAPI project, it is very likely that this wouldn’t fit the requirements for an efficient implementation of a concise high-level DSL.

The first solution to circumvent this issue is to combine the built-in OWL lite reasoner with SPARQL queries as described above.

The second solution would be to integrate a more powerful reasoner such as the Pellet OWL 2 reasoner that supports OWL DL backward-chaining reasoning. There are even solutions that provide forward-chaining on top of Jena and Pellet such as DLEJena[11], but after a brief evaluation their adoption, code maturity, backing community and documentation render their use at least questionable within the scope of this project.

[7] e.g. this is done in the FP7 TELEIOS project to implement stSPARQL and GeoSPARQL support, which both extend SPARQL with spatial / topological backward-chaining reasoning support on top of Sesame. See http://www.strabon.di.uoa.gr/
[8] as done for stSPARQL and GeoSPARQL
[9] http://www.earthobservatory.eu/
[10] At the time of writing there is no stable direct support for the current OWL 2 standard in Jena
[11] http://lpis.csd.auth.gr/systems/DLEJena/

A third solution would be to use the general purpose rule engine which is also provided by the Jena Framework. This engine can work on RDF datasets and rules which are expressed in a custom, Jena specific syntax. Backward-chaining, Forward-Chaining and combinations of both are supported. However these custom rules share the same drawbacks as a Drools based approach, as they are specific to Jena without any alternate implementations.

As mentioned for Sesame, Jena also does not allow direct interaction with a previously existing Java Model (i.e. that of CityServer3D). Given the fact that there are no plans for integrating any third party RDF data within the project, using Jenas general purpose rule engine does not seem to provide any tangible benefit over Drools.

After this high level comparison we are now going to compare two low-level examples of notation specifics of Drools based forward chaining and a hybrid combination of SPARQL queries and OWL DL backward-chaining reasoning as a rather minimalistic approach that is independent of the specifics of a certain SPARQL Query Engine or OWL reasoner implementation.

- Drools production rules that are used for forward-chaining have the form

  ```
  when <condition> then <conclusion or action>
  ```

  For example, the following rule adds all facts from the knowledge base that have the type ‘Person’ and whose attribute ‘age’ is lower than 10 to the global list ‘children’.

  ```
  when p : Person ( age < 10 ) then children.add(p);
  ```

  Since production rules are event-based, one can use forward chaining to define workflows:

  ```
  when p : Person ( age < 10 ) then children.add(p);
  when c : Person ( ) from children then System.out.println(
      c.name + “ is a child”);
  ```

- The following example in OWL DL / SPARQL will essentially do the same as the production rules above but this time using backward chaining (instead of printing out “is a child” to the command line it will simply select all children; actions such as ‘println’ cannot be used in SPARQL queries).

  (OWL rule, Manchester notation):
  ```
  Class: Child
  ```

```
        SubClassOf:
            Person
        EquivalentTo:
            hasAge exactly 1 xsd:integer[>= 0 , < 10]
```

(SPARQL query):

```
SELECT ?c
WHERE {
       ?c rtf:type :Child .
}
```

The following table summarizes the features that are important for the UrbanAPI project and how they are implemented in Drools or could be using a combination of SPARQL and OWL that is independent of the programming language environment and a specific SPARQL Query Engine or OWL Reasoner implementation.

	Drools	**SPARQL Query Engine + OWL Reasoner**
Triggers	Events	Queries
Chaining	Forward (backward possible in latest version)	Backward (simple forward chaining possible through SPARQL INSERT queries)
Data store	In-memory knowledge base	Triple store, either in memory, file-based, backed by DBMS or distributed across network
Data model	POJOs	RDF Triples

To conclude, workflows as they are needed in geospatial data processing can be described very well with production rules and forward chaining. Apart from that, event-driven workflows support the project's requirements. Nevertheless, Drools can only reason about data which is already in memory, whereas a semantic web based approach could potentially access larger data sets that could even be spread across different machines. However, the integration components built into the CityServer3D framework which is used in this project compensate this drawback. In UrbanAPI we therefore decided to use Drools.

5.2 Domain-specific languages

Krämer, Ludlow and Khan summarize the state of the art in domain-specific languages as follows (Krämer, Ludlow, & Khan, 2013): “In computer science domain-specific languages are used for a number of purposes. For example, in the UNIX operating system configuration files have been written in custom languages for quite some time. In agile software development domain-specific languages are used to quickly adapt to changing user requirements. In recent times, DSLs have gained a lot of interest, especially in the scientific community. One of the most actively pursued topics is DSL design.

Mernik et al. differentiate between five phases of DSL development: decision, analysis, design, implementation and deployment (Mernik, Heering, & Sloane, 2003). The analysis phase is one of the most important ones since it includes specifying user requirements. Mernik et al. identify three common ways to develop a new DSL:

- The DSL will be implemented based on an existing language which will be included completely. Internal DSLs are an example for this case.
- An existing language will be limited to the means needed for the application domain.
- Application-specific vocabulary and language constructs will be added to an existing language.

Apart from that, DSLs can of course be developed from scratch as well (like most external DSLs). In the book "Domain-Specific Languages" Martin Fowler describes a number of methods for developing language parsers (Fowler, 2010). Barrientos and Lopez present a new approach to DSL design by using functional programming combinators as well as the composite pattern from object-oriented programming (Barrientos & Lopez, 2009).

Since DSLs are used more and more often in modern software systems, there's a growing need for language maintainability. DSLs ought to help reducing maintenance costs in large software systems, but this can only work if maintaining the DSLs itself is not too costly. Pizka and Jürgens therefore introduce the term *language evolution* (Pizka & Jürgens, 2007). They present three special DSLs which can be used to switch between different *generations* of a language: the Grammar Evolution Language (GEL) describes the grammatical difference between two language versions, the Word Evolution Language (WEL) focuses on the syntax, and the Language Evolution Language (LEL) combines GEL and WEL to perform language transformations.

In order to reduce the cost of developing and maintaining a DSL, language modularization can be used. Hudak tries to reuse compiler components such as lexer and parser as well as semantic analysis (Hudak, 1998). Irazabal and Pons present a modularization technique based on Xtext (Irazabal & Pons, 2010). Therefore they import several partial language definitions and merge them into a new single one.

In recent times, domain-specific languages are also used to simplify the development of distributed programs and algorithms. For example, Lee et al. present the Delite Compiler Framework which allows users to describe parallel code execution using an abstract DSL (Lee, Brown, Sujeeth, & al., 2011). The framework is able to use the abstract description to generate optimized code for several, heterogeneous platforms such as different operating systems, mobile devices or GPUs via CUDA or OpenCL. Their work aims for hiding the specifics of each target platform from the user. In order to develop their languages, Lee et al. use the so-called *language virtualization* technique (Chafi & al., 2010). Components of a high level language (in this case Scala) such as lexer, parser and type checker will be reused. Based on that, components for optimization and code generation will be implemented. Sujeeth et al. use the Delite Compiler Framework and this technique to implement the OptiML language which is a DSL for distributed machine learning (Sujeeth & al., 2011).

Xtext[12] is a language workbench written in Java and based on the Eclipse Modeling Framework (EMF)[13] and ANTLR[14] as the underlying language recognition tool. Xtext allows domain-specific languages up to general purpose languages to be defined. In recent times, Xtext has gained prominence and is used for a wide range of applications in the Java community—e.g. the Xtend[15] language that adds useful features to Java, Spray[16] which is a DSL for the Graphiti framework[17], etc.

In Xtext you define a DSL by specifying a context-free grammar which is interpreted by ANTLR. This tool will automatically generate the lexer and the parser for the language. Language constructs will be mapped to an EMF model that Xtext automatically derives from the grammar. Additionally, the framework can generate an editor that can be embedded in Eclipse RCP applications. This editor offers features such as syntax highlighting, auto completion, etc.”

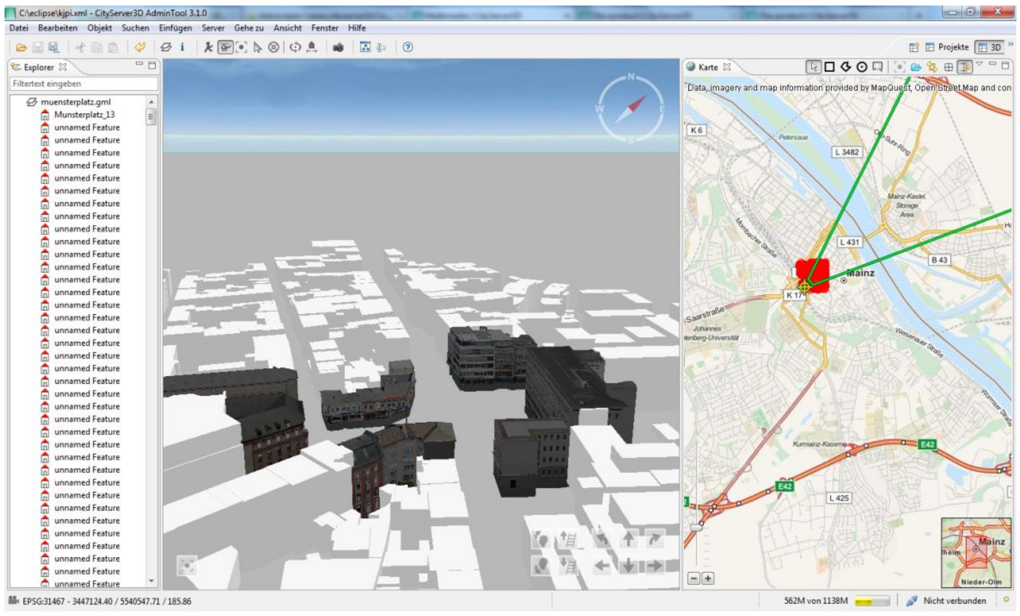
We found embedding the editor in an existing Eclipse RCP application to be quite hard to manage. First of all, Xtext creates OSGi bundles which have a large number of dependencies. It also depends on some parts of the Eclipse IDE. So, if you want to include an editor generated by Xtext

[12] http://www.eclipse.org/Xtext/index.html
[13] http://www.eclipse.org/modeling/emf/
[14] http://antlr.org/
[15] http://www.eclipse.org/xtend/
[16] http://code.google.com/a/eclipselabs.org/p/spray/
[17] http://www.eclipse.org/graphiti/

into an existing application you also will have to include a lot of additional bundles from the Eclipse IDE. These bundles also contain extensions to the application's main menu as well as additional views. It's rather hard to disable all that additional UI elements that you actually do not want to have in your application. You can use RCP activities for that, but this just seems to be a quick and dirty solution. In fact, it would be better to completely avoid these dependencies—which is, unfortunately, not yet possible with Xtext.

The framework makes heavy use of code generation. This raises the question whether generated code should be checked in into source control, and if not what additional steps have to be included in the build process and what additional plugins do the developers have to install in the IDE in order to locally generate the code.

For the various urbanAPI scenarios, we need different domain-specific languages, probably modular or hierarchical and configurable to a certain degree. With Xtext this task is rather hard as it requires the developer to generate new OSGi bundles for each single DSL which may eventually lead to a quite large number of bundles. We consider this situation hard to manage. A dynamic language interpreter without code generation would be much easier to handle.


C:\eclipse\kjpi.xml - CityServer3D AdminTool 3.1.0
Datei Bearbeiten Objekt Suchen Einfügen Server Gehe zu Ansicht Fenster Hilfe
Explorer
Filtertext eingeben
muensterplatz.gml
Munsterplatz_13
unnamed Feature
Karte
Projekte
3D
Data, imagery and map information provided by MapQuest, Open Street Map and con
Mainz
EPSG:31467 - 3447124.40 / 5540547.71 / 185.86
562M von 1138M
Nicht verbunden

5.3 Rule languages

This section contains an overview over state-of-the-art rule languages and how they compare to our approach.

5.3.1 Artifical intelligence

One of the most popular rule languages is *Prolog*, which is a declarative programming language for logical expressions. Prolog programs consist of a number of relations expressed in Horn clauses, a subset of first-order logic terms. There are two types of clauses: rules and facts. A rule consists of a head and a body. The body consists of a number of predicates that evaluate to true or false. The rule head :- body means that head is true if body is true. A rule without a body is called a fact.

After specifying rules and facts, the user defines queries which the system tries to answer with either “Yes/True” (the query was successful) or “No/False” (the query was not successful or it could not be answered).

Prolog mostly uses pure predicates that have no side effects, except for certain predicates that can have minor side effects, such as printing a value to the screen. Besides Prolog there are other rule languages for knowledge representation in the area of artificial intelligence such as F-Logic.

Other rule languages often used in the area of artificial intelligence allow for a wider range of predicates with side-effects, for example, the rule languages of *Jess and CLIPS*. Both engines use a *LISP* dialect for rule declaration. They both allow users to define production rules that have a right-hand side where some kind of output can be produced or the fact base can be altered. More precisely, in Jess and CLIPS rules consist of a left-hand side where a condition can be specified, and a right-hand side that contains an action. The action will only be executed if the condition evaluates to true. For example, this enables new facts to be inserted into the working memory if a certain condition is true. This type of fact inference is supported by the RETE algorithm which allows rules to be chained: if a fact is added to the working memory because a condition became true, another condition of another rule might become true which causes this rule to be executed (fired) as well. Since the RETE algorithm is event-based, it is also known to be very efficient when it comes to perform pattern matching on a fact base, in particular if the facts are changed frequently.

The rule languages presented in this section differ from the approach we propose in the following aspects:

- Prolog’s language is on the one side easy to learn, but one the other side requires the user to be able to express problems in first-order logic. Moreover, Prolog allows for defining problems from almost arbitrary application domains. The same applies to F-Logic. The domain-specific languages we propose are more targeted to a specific use case. They only contain vocabulary necessary in the specific application domain. They are not as flexible as Prolog's language, but easier to learn and easier to use for non-IT personnel.

- Jess and CLIPS are production rule systems. This kind of rule-based system definitely fits better to our approach of using rules for data integration, visualisation, etc. For example, in Jess it's possible to change an object's color[18] if some condition is true (if its type is a building, for example). However, Jess and CLIPS use a LISP-dialect as their rule language. Although many computer scientists don't have problems with writing LISP scripts, end-users with no background in functional programming will surely have. At least it will be a lot harder for them, in particular since the excessive use of parentheses in LISP often scares people off, even experienced programmers. The rule languages we propose are a lot easier to read since they almost look like natural language. At the same time they have a well-defined grammar and syntax and are readable by machines and humans.

5.3.2 Rule markup languages

Another area where rule languages are used is the Semantic Web. In section 5.1 we already introduced rule-based systems falling into this category, but here we focus more on the languages used.

A popular markup language for rules in the semantic web is *RuleML*. RuleML is actually a family of rule languages serialised in XML. Its main focus is interoperability. RuleML is modular and supports subfamilies such as SWRL, RQL, or OWL. Existing rule-based systems such as CLIPS, Jess, or ILOG JRules support RuleML as rule exchange format.

RuleML differs from our approach in terms of readability and usability. RuleML is an exchange format for rules meant to be read by machines and not humans. Its standardised format supports this very well. Our domain-specific languages, on the other hand, are designed to be easy understandable by humans, in particular domain experts. Of course, since our DSLs have a well-defined syntax and grammar they're also parseable by machines, but the main focus is to provide a good usability.

Another format worth mentioning is *RIF*. Just like RuleML it's XML-based. Its main focus is also rule exchange, hence its name *Rule Interchange Format*. It compares to our approach just the same as RuleML.

[18] Jess is not an object-oriented rule engine. So, in fact, it's not possible to change an object's color, but to introduce a fact that relates a color to some other fact.

Compared to RuleML and RIF there are rule languages in the semantic web area that are more readable. One of these is *SWRL* (Semantic Web Rule Language). SWRL has a XML-based syntax, but also a so-called Human-Readable Syntax. SWRL is mainly used to define logical expressions through Horn clauses. It can be compared to Prolog's approach although it is more targeted to the semantic web. Compared to our domain-specific languages it differs in the following aspects:

- It is definitely more readable than the XML-based approaches presented above, but it still requires the user to have experience in expressing problems in first-order logic. Our DSLs are targeted to a specific application domain and are therefore a lot easier to read and use for non-IT personnel.
- Although SWRL is targeted to the Semantic Web, it is still very generic and can be used to express a wide range of problems. Our DSLs are targeted to specific problems only.

5.3.3 Object-oriented rule languages

There are some rule languages that focus on object-oriented problems more than the other rules described above. One of these languages is the *Object Constraint Language (OCL)* which is part of UML. OCL can be used to describe constraints on objects through invariants, preconditions, postconditions, etc. For example, it can be used to express that some kind of condition must hold before an object's method can be called, or that a field's value lies between a specified range.

OCL can actually be used well to define policy constraints like it has been done by Sohr et al. to define constraints for user authorisation policies (Sohr, Ahn, & Migge, 2005). However, OCL differs from our approach in the following aspects:

- It is rather generic since it focuses on expressing constraints on arbitrary objects, so OCL requires the user to have an understanding of object-oriented design. Our languages are tailored to the urban management domain and are therefore a lot easier to use for experts from this domain.
- OCL's syntax is very readable for software developers. However, we aim for languages that look like natural prose, although our languages also have a well-defined syntax and grammar for them to be readable by machines.
- In some use cases we need production rules as we want the user to be able to change the visualisation of geodata, for example. With OCL this would not be possible.

Another well-known object-oriented rule engine is *Drools*. We already introduced this framework in section 5.1. Drools has its own rule language that is also very generic. It does not target a specific application domain, but is on the other hand very flexible. Drools comes with its own support for domain-specific languages. It uses regular expressions which the developer has to map to expressions in Drools' rule language. Regular expressions are not as flexible as an external DSL with a complete grammar and syntax. Instead they can only be used for a limited variety of use cases.

However, as described in section 5.1 we use Drools as the underlying rule engine whenever possible.

The third object-oriented rule engine worth to mention is the commercial product *ILOG JRules*. JRules was developed by a company called ILOG, which was bought by IBM in 2009. The framework mostly focuses on business rules. It contains two languages: the ILOG Rule Language (IRL) and the Business Action Language (BAL). Both languages are rather easy to read and use, although IRL is meant to be used by developers instead of domain experts. BAL on the other hand consists of a number of predefined constructs that make rules define in this language almost look like natural prose. For example, the following rule does exactly what it says:

```
If the name of customer contains "Peter"
then print "Hello " + the name of customer
```

This rule is perfectly readable for domain experts, even if they have no background in programming, semantic web, or first-order logic—as they would have for the languages described above. This is exactly the same kind of language that we are aiming for. ILOG JRules even contains an editor that supports syntax highlighting and auto-completion. This tremendously speeds up the rule definition process. Besides, it helps inexperienced users define rules on their own. In urbanAPI we also aim for developing a similar editor for our rule languages.

Unfortunately, ILOG JRules is a commercial, closed-source product. So, it's not possible to integrate it into urbanAPI. Apart from that, the Business Action Language BAL only contains a number of predefined constructs that are very generic. It is not meant to be used in the urban management domain. Some of these predefine constructs are:

```
all of the following conditions are true
any of the following conditions is true
the name of this rule
it is not true that <condition>
the number of <objects>
```

Of course, the underlying object model can be arbitrary, so it would indeed be possible to use an urban management domain model as well. However, this only applies to the objects and properties that can be used in the language. The general constructs that make up a natural-sounding sentence in a rule cannot be changed.

5.4 Graphical rule editors

There are a number of graphical user interfaces the support users in the process of rule definition. One of these editors comes with the rule engine Drools (already described above). Figure 1 shows a screenshot of this editor.

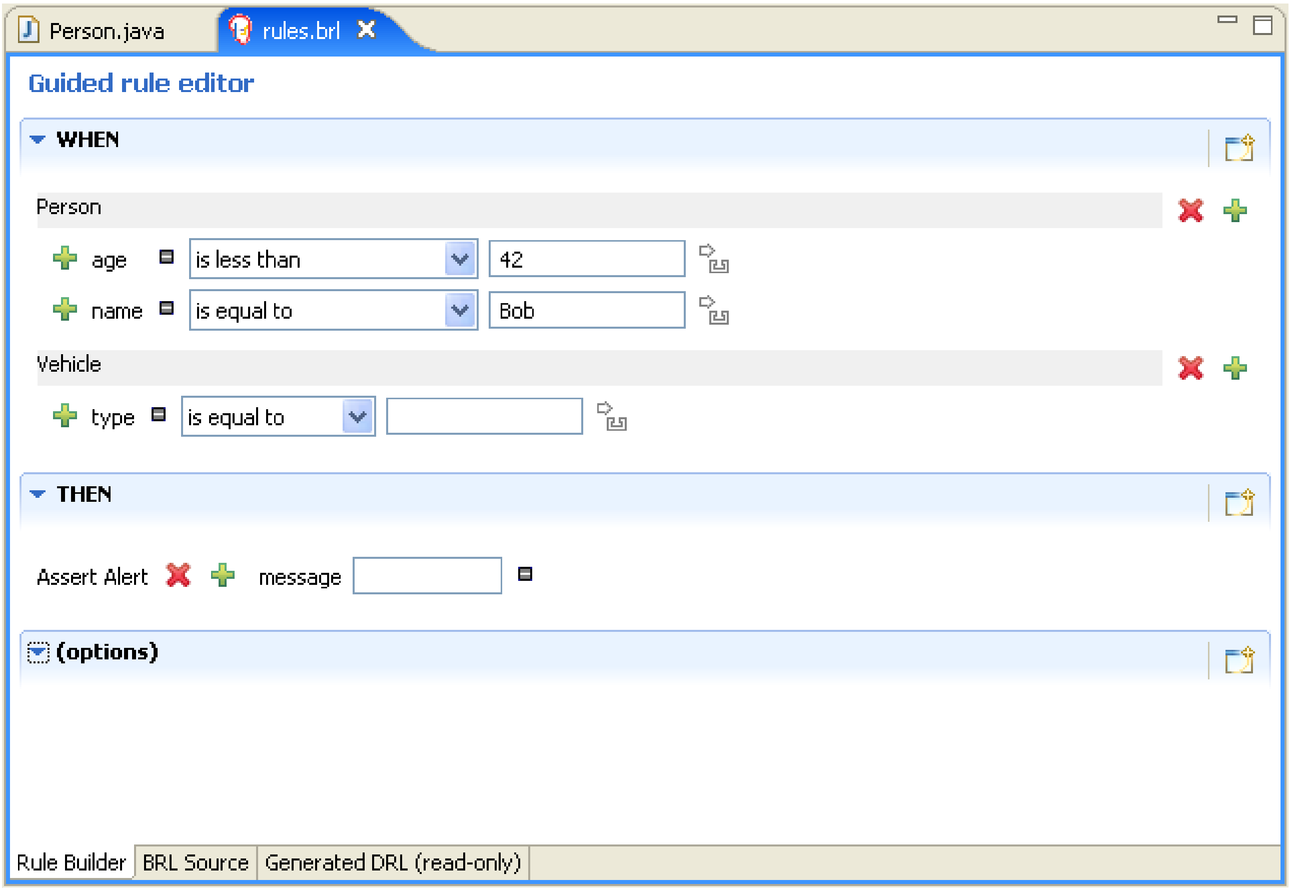


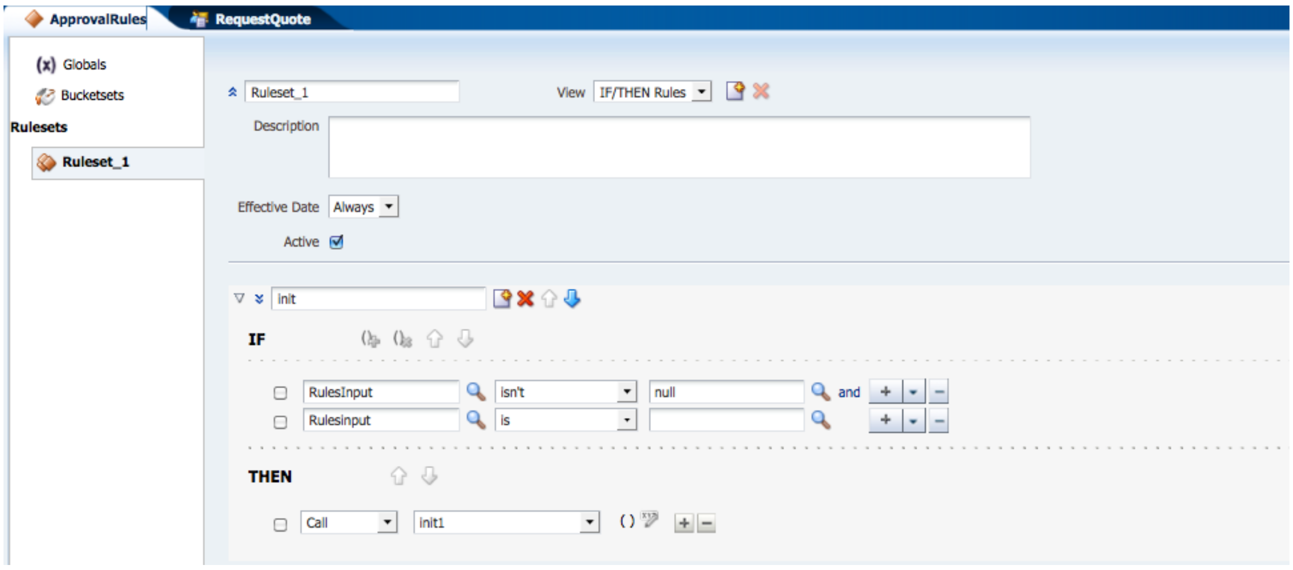
Figure 1 The graphical rule editor of Drools (source: http://www.jboss.org/drools/drools-expert.html)

The editor is divided into two parts, a *when* section and a *then* section. In the when section, the user can add several conditions. The then section contains actions that should be performed when the conditions evaluate to true.

The editor builds up on Drools' own rule language. The selections made by the user are translated directly to a textual rule script. Compared to the rule editor we present in section 6 it is very generic. Just like Drools' rule language it can be used for almost arbitrary problems that can be expressed with production rules. Our rule editor on the other hand is tailored to be used in the area of geospatial applications, in particular urban management and is therefore easier to use for experts of this domain.

Another editor similar to the one of Drools is the Oracle Business Rules editor that is part of Business Process Composer. The rule language used in Oracle Business Rules is quite similar to the one used in Drools. Rules consist of a WHEN and THEN clause, defining the condition and the action respectively. The grammar is easy to read if you're used to declarative rules, but rather hard for domain experts.

The editor has a similar interface to the one of Drools. It guides the user through the rule definition process by providing forms whose contents directly translate to the textual rule language. (see Figure 2).



Ruleset_1 | View: IF/THEN Rules
Effective Date: Always
Active: ✓
init
IF
RulesInput isn't null and
RulesInput is
THEN
Call init1

Figure 2 The guided rule editor of Oracle Business Process Composer

Another graphical editor for rules is Visual Rules Modeler by Bosch (see Figure 3). The tool supports modelling rules through flow diagrams and decision tables. The diagrams are easy to read and clearly show what decisions lead to which result.

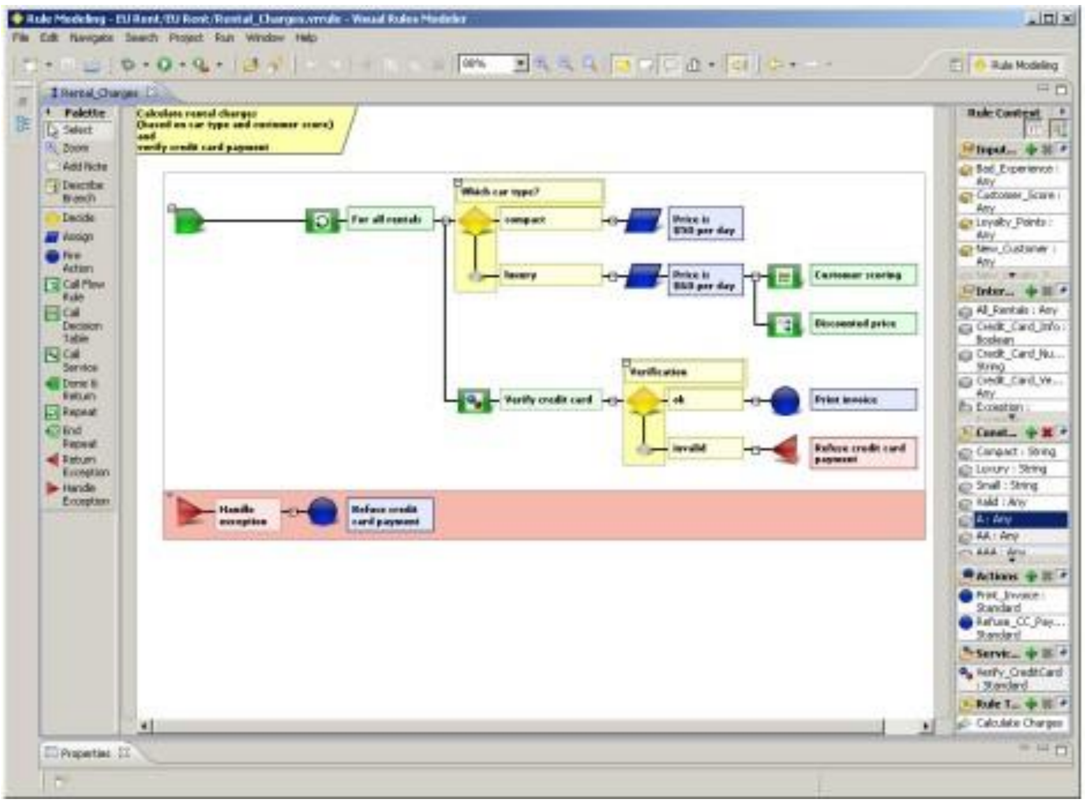


Figure 3 The Visual Rule Editor by Bosch (source: http://www.bosch-si.com/technology/business-rules-management-brm/brm-komponents/visual-rules-modeler.html)

The visual rule language consists of components you also find in UML flow charts. There are start and end symbols, generic processing steps (actions), subroutines, input/output symbols, and conditionals. These components are connected by lines to indicate the flow from the start symbol to the end symbol. The diagrams are read from the left to the right.

Visual Rules Modeler is a commercial product that cannot be integrated into urbanAPI. Additionally, it is not meant to be used in the geospatial domain, hence it lacks features such as spatial indexes, etc.

The fourth graphical rule editor presented here is the one that comes with *OntoStudio* from Semafora Systems GmbH. The rule editor has been completely reimplemented within the ONTORULE project. ONTORULE was co-funded by the European Commision from January 2009 to December 2011 under the FP7 programme. The OntoStudio developers figured that RIF rules (see section 5.3.2) are very hard to read for domain expert, so they implemented a graphical editor that supports the user in rule editing:

“The standard RIF is a format for exchanging rules between different tools. It’s intention was not to be used for modelling purpose and the representation syntaxes are not easily understandable by non-experts. Looking at the internal representation format ObjectLogic of the ontology and rule modelling environment OntoStudio also reveals that the syntax is not easy to use by non-experts. To provide an editor that can be used by non-experts ontoprise developed a graphical rule editor which hides the complexity of the syntax and provides easy drag&drop functionality as well as auto-completion.”

— Eva Maria Kiss, Ontoprise GmbH, August 2011

http://ontorule-project.eu/showcase/OntoStudio_Graphical_Rule_Editor

(last visited: 2013-09-07)

The OntoStudio rule editor is based in concepts from UML. The developers claim that UML is well-known to many users and so the rule editor should be a lot easier to use than writing rules directly in RIF. Figure 4 shows a screenshot.

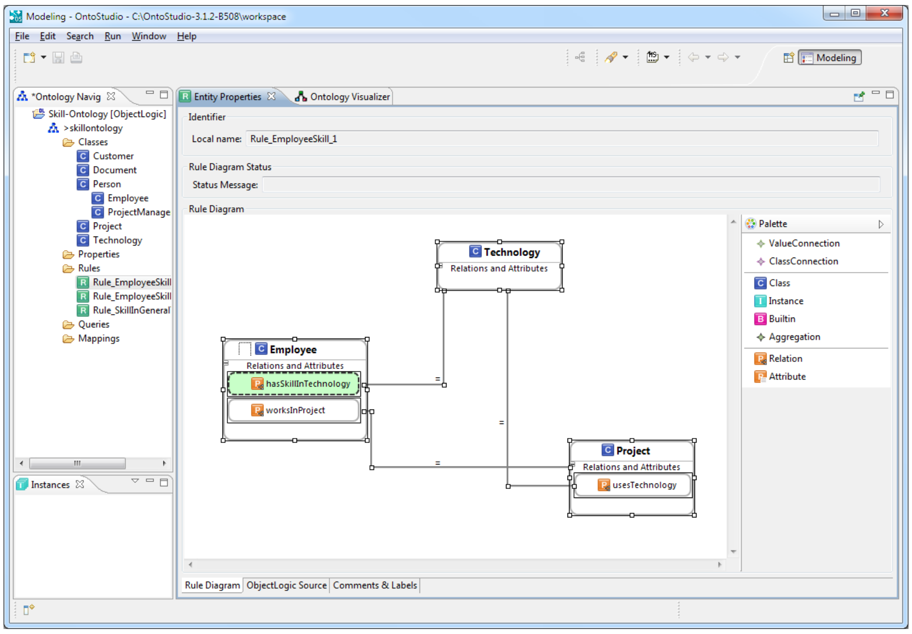


Figure 4 The graphical rule editor of OntoStudio (source: http://www.semafora-systems.com/en/products/ontostudio/, last visited: 2013-09-07)

Again, the graphical rule editor is very flexible and can be used for a wide range of applications, but it is not meant to be used in the geospatial area and in particular not for urban management applications. Of course, it might be possible to use it in these areas, but the editor would be harder to use than one that is directly tailored to these application domains.

5.4.1 Summary

One of the major differences between the 3rd party graphical rule editors described above and our editor presented in the next section lies in the vocabulary used. Our editor consists of a very limited set of conditions and actions (called recipes) that are meant to be used in the geospatial domain. Recipes are a lot more powerful than most of the expressions in the 3rd party tools. For example, with the 3rd party tools users operate on objects and their attributes on a very low level using conditionals, variable assignments, etc., whereas our editor contains powerful actions such as the 3D “extrusion” operation which implements a whole geometrical algorithm in only one expression.

Of course, the limited vocabulary also limits the use of the editor to the use cases it has been designed for. It is not overly flexible and cannot be used in arbitrary scenarios. One the one hand the 3rd party tools are superior in respect to their feature-richness and flexibility, but on the other hand the limited vocabulary of our editor consisting of terms from the geospatial domain makes the editor easier to learn for domain experts. In order to improve the reusability of our editor we implement the following.

- Recipes can have parameters which allows them to be adapted to various situations.
- Our editor is very modular, so adding new recipes or changing the behaviour of an existing one is very easy and can be done in a short amount of time.

Another difference between our editor and the Bosch Visual Rules Modeler, for example, is that we rely on the capabilities of the underlying rule-based system Drools to forward chain rules to create a complex workflow. In Bosch Visual Rules Modeler you mostly work on very large flow diagrams that can become confusing very quickly, especially for users who are not experienced in rule modelling. In our editor you work with small rules, where each of them models exactly one aspect. Together, through forward chaining, these rules can be combined to create a complex workflow. For the domain user it’s easier to work on single aspects at a time without having to deal with the whole workflow all the time.

5.5 Graphical editors in the geospatial area

Please note that the editors presented above are very flexible and can be used for a wide range of applications, but they are not directly tailored to the geospatial domain. The use of rules and techniques from the semantic web in geospatial applications is generally considered useful (van Oosterom, 2009) and so there are some extensions to existing rule systems such as stSPARQL or GeoSPARQL. However there are no graphical editors for geospatial rules yet. You mostly find graphical rule editors in the area of business modelling or semantic web but not geospatial information management or even urban management.

Nevertheless, there are tools that can be used to perform geospatial operations, in particular there's ESRI's *ArcGIS ModelBuilder* which is part of the ESRI's ArcGIS Spatial Analyst. The ModelBuilder is controlled through a graphical editor that allows users to define complex geospatial workflows. With this editor users can select functions attached with data that matches given criteria perform operations such as classification or colourisation. Figure 5 show a screenshot of this tool.

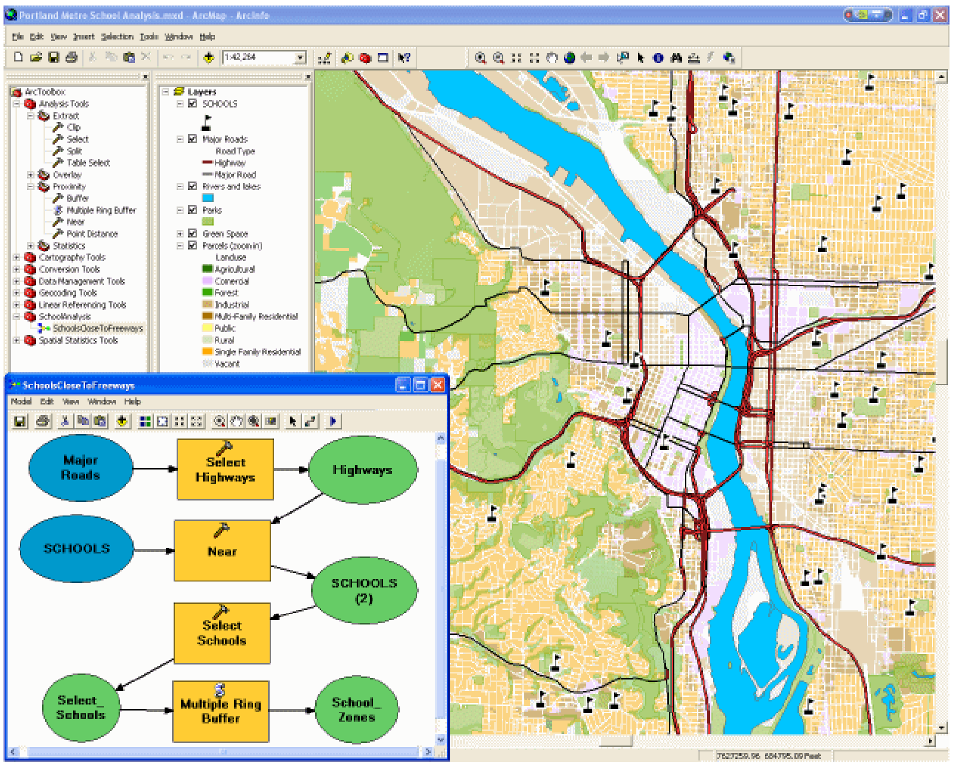


Figure 5 The graphical editor of ESRI ArcGIS ModelBuilder (source : http://www.esri.com/software/arcgis/extensions/spatialanalyst/key-features/graphical, last visited: 2013-09-07)

The ArcGIS ModelBuilder is targeted to geospatial applications and therefore only contains operations needed in this domain. This includes spatial indexes (and operations such as "near" or "inside") as well as geometrical operations (such as building buffer polygons). The tool supports 2D data and 2,5D raster data. However, one of the scenarios of the urbanAPI project is about 3D VR applications, in particular 3D city models. Such kind of data cannot be processed with ArcGIS. Apart from that, ArcGIS is a commercial product and cannot be easily integrated into the urbanAPI toolset.

6 The CityServer3D's Graphical Rule Editor

The CityServer3D is a client-server system for the storage, visualization, and processing of spatial data. Geo information from different sources is integrated into an object-relational database and placed in the web at the disposal of different clients. In addition to 2D and 3D geometries in different levels of detail (LOD) the CityServer3D can also store and process technical and metadata.

It offers access to geodata via standardized interfaces. When connected to already existing data infrastructures, the CityServer3D helps you to efficiently manage your 3D city models. The integrated continuation processes grant a sustainable use of the information. Integrating multimedia content can positively affect the presentation of geodata. The CityServer3D can be used for urban planning, simulation and data analysis.

6.1 CityServer3D – AdminTool

The AdminTool is a desktop application with a large feature set. Import and export of geodata in a local working area or into the CityServer3D Server, quality assurance, rule definition, data editing and of course the visualization are only a few of its features.

The user interface can be customized with different views, depending on the customers' requirements. The most used and most important views (arranged from left to right in the picture above) are the explorer view, the 3D view and the 2D view. The explorer displays a hierarchic overview of scene currently loaded. The 3D view visualizes the geodata and the 2D view is mainly used for navigation and orientation. In addition there are pre-built perspectives representing different views in different alignments.

6.2 Graphical rule editor

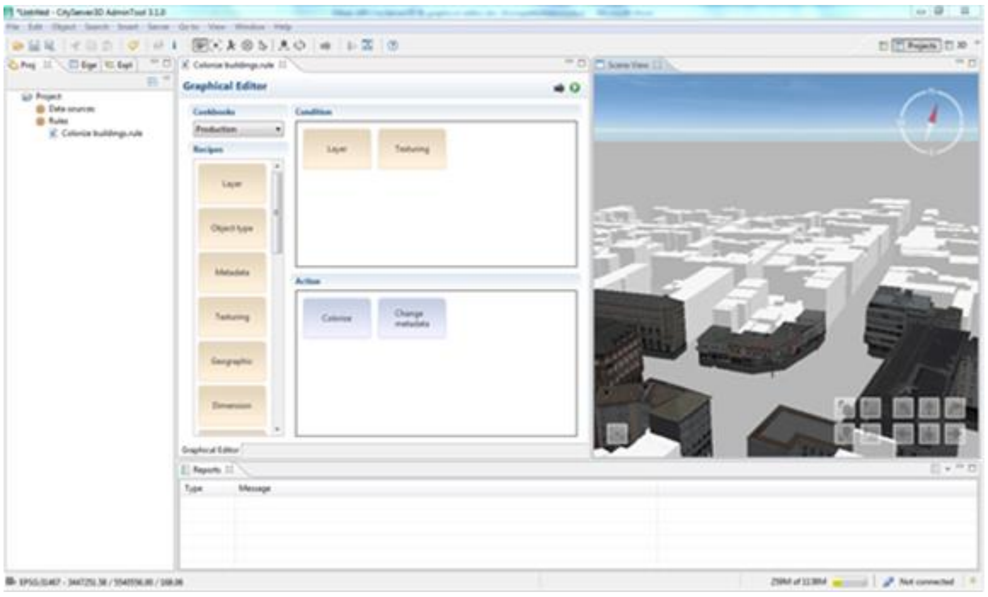


Figure 6 The CityServer3D contains a graphical rule editor

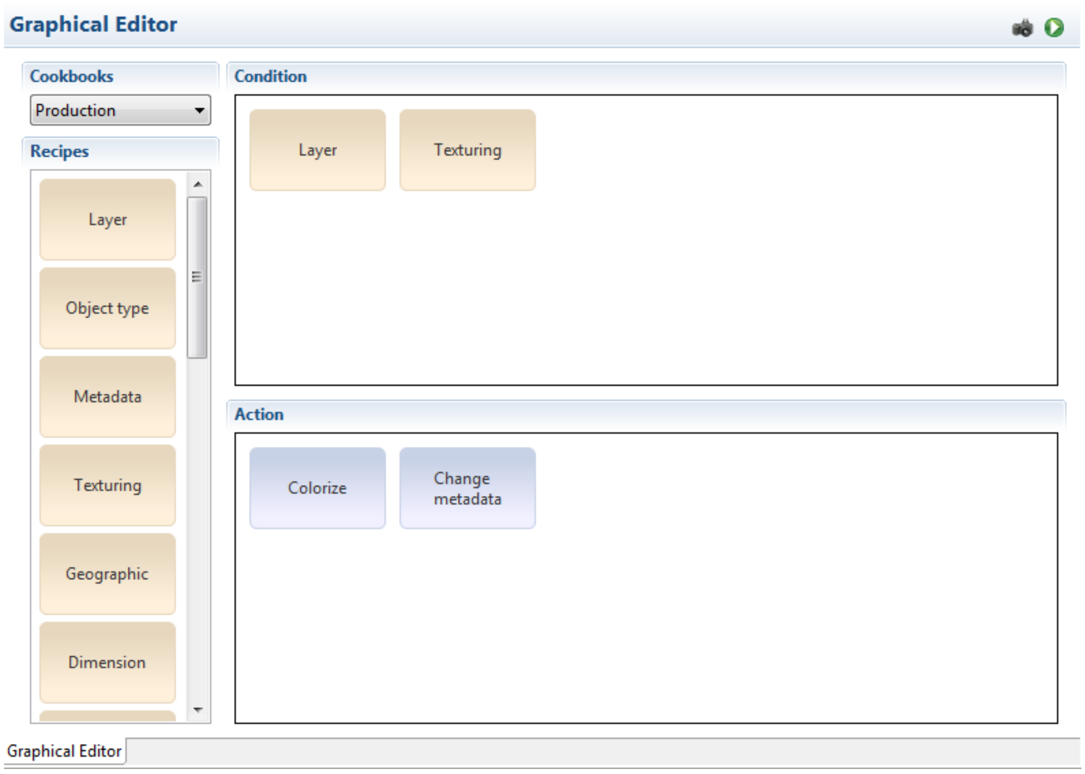
During the UrbanAPI project Fraunhofer IGD has extended the CityServer3D AdminTool by a graphical rule editor. This editor allows automated processes to be created. In the product view the user can define not only data sources to be processed but also rules containing several workflow steps.

Currently there are various workflow steps, which you can combine in the rule editor with just one click. For example, it is possible to colorize all objects of a certain type in the city model such as buildings or trees, or objects with a certain level of detail. This process can be saved and executed whenever it is required. It allows the integration of different data sources into a common data basis.

The projects view (on the left side of Figure 6) allows the user to create several projects with its own rules and data sources. Thereby fast import and rule execution is granted.

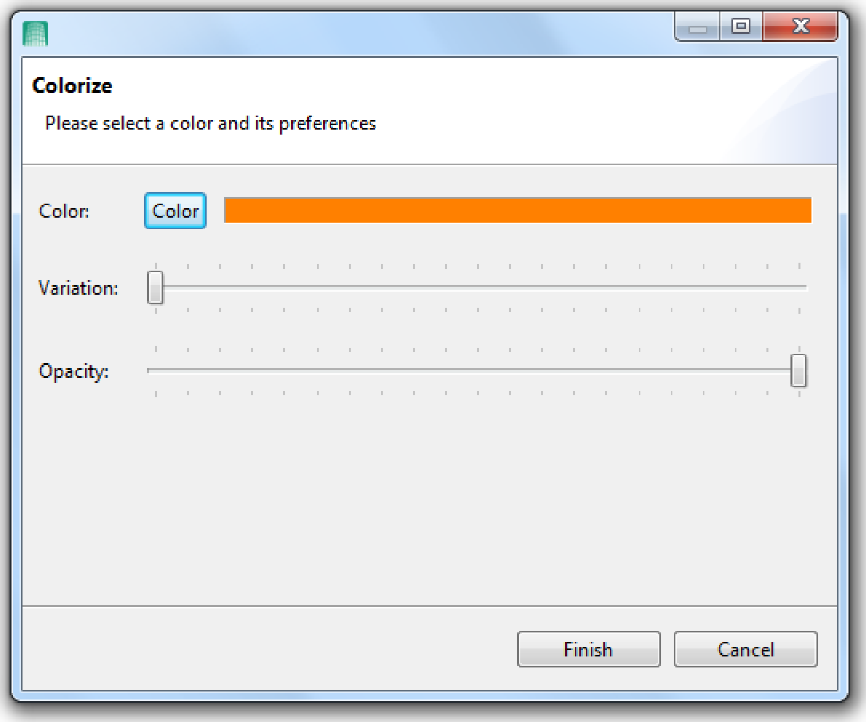
The report view (on the bottom of Figure 6) is used for feedback while executing a rule. If an object could not be changed or filtered by the rule for any reason, a report will appear there containing a feedback message and a link to the said object. For example if a rule filters objects by a value of a metadata, the report view will provide a list of all objects without the requested metadata.

The graphical editor contains two major components: The recipes and the cookbooks. Each recipe is represented with one block and the cookbooks separate the large number of recipes into groups for a better overview. The available cookbooks are on the top of the tool bar.



The graphical editor itself is subdivided in three parts. The tool bar on the left side contains all recipes available in the selected cookbook. The brown recipes are conditions. They filter objects by different criteria—e.g. appearance, location, size or metadata. The other recipes are actions and differ by their function. Blue recipes are for editing, green ones are for creating and red ones are for deleting data. Those action recipes are placed on the lower side of the editor.

Each recipe contains several parameters such as the colour recipe which contains the colour and values for opacity and variation. When adding or editing a recipe, a wizard will be displayed where the user can enter this information.



The granularity of the recipes is chosen based on the criteria described in section 7.

6.2.1 Recipes

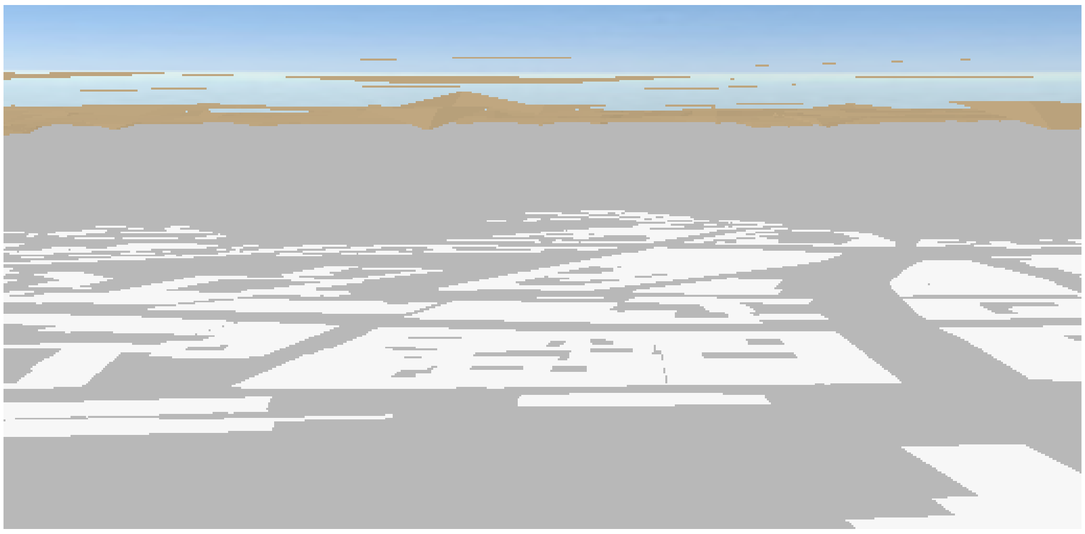
In this chapter we explain the recipes that have been developed especially for the UrbanAPI project. The description will include recipe parameters and examples for practical use. In addition there will be a complete workflow based on example data from Cologne.

Tile Terrain (Action)

This recipe divides the digital terrain model into several parts. The user can define the amount of parts with a slider in the wizard page of this recipe.

Map to Terrain (Action)

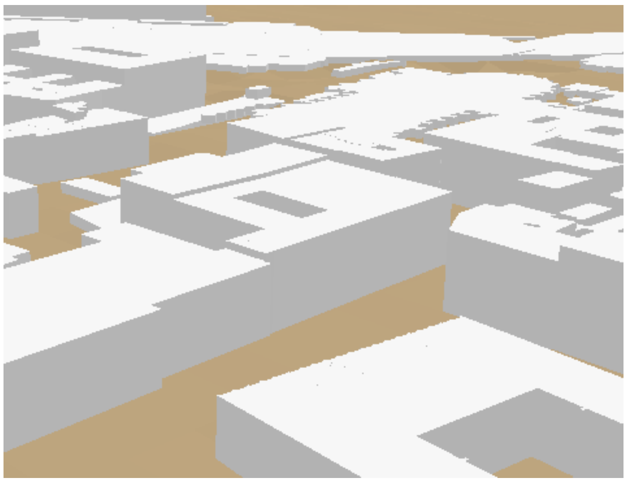
This recipe can map 2D items from shape files as textures to digital terrain models in the scene. The parameters are the resolution of the texture and the colour. The colour can be a default colour or it can depend on a metadata of the object.



6.2.1.1 Example: Cologne

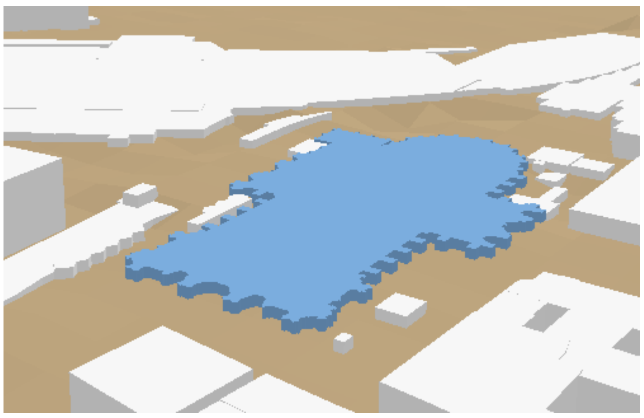
This demo is an example of data continuation. We got two different data sets: The digital terrain model and the buildings footprints.

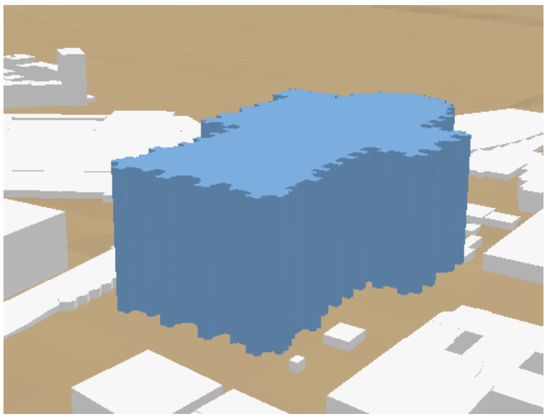
The footprints contain different metadata which can be used to change the data. First of all we will explain the single steps, followed by the required recipes and finally the three rules needed for this scenario.



First of all we copy the buildings footprints, so the original won't be changed.

After that we take the footprints and lift them to the height of the digital terrain model and extrude them. We use the Metadata NumberOfFloorsAboveGround for extrusion.





Some buildings don't have that metadata, e.g. the Cologne Cathedral. Those buildings are extruded per default by one floor. We will get a report of those buildings and can delete them from the scene. In this case we delete the Cologne Cathedral. Now we can finish with the third rule.

We select the buildings original footprints and add a duplicate filter, so that already changed buildings won't be extruded again. Then we can extrude them by 45 meters, no matter what metadata they got. As a result the Cologne Cathedral will appear within the city model with a height of 45 meters.

The different recipes used for this scenario:

Conditions:

Layer

This recipe is for layer selection. You can specify multiple layers here and only objects within those layers will be changed.

Duplicate Filter

You can specify all layers which should be controlled and this filter ignores all objects which are already part of these layers.

Actions:

Copy

This recipe makes a copy of all objects and the copies won't be changed by any following action recipes.

DGM Match

With this recipe you can lift or lower buildings to the height of a digital terrain model. It requires no parameters, because it takes the terrain model below or above the building.

Extrude

This recipe extrudes a 2D footprint with two parameters: The amount of floors via metadata and the floor height. In this scenario we use three meter floor height and the metadata NumberOfFloorsAboveGround.

Move into layer

This recipe moves all objects to a specified layer.

Rules:

The different rules in the order of execution:

Rule #1:

Layer, Copy, Move into Layer

Rule #2:

Layer, DGM Match, Extrude

Now we delete the Cologne Cathedral manually.

Rule #3:

Layer, Duplicate Filter, DGM Match, Extrude

7 Some words on granularity

The granularity of the language constructs must be chosen carefully. They shouldn’t be too trivial or too powerful, since the usability would not be viable. The language constructs, using the principle separation of concerns, will always capture particular attributes from one of the following categories:

- Source: What is the source of this data? (E.g. Database, local file, scene)
- Properties: Which properties does the object have? (E.g. Colour, texture, metadata, level of Detail)
- Geographical: What is the size or the location of the object?

In addition to the categories the actions are divided into three types; *add*, *update* and *delete*. In this way actions are very specific and the three types are an increased security method to avoid too powerful language constructs.

Due to this granularity, the individual language constructs in their function and their information are distinct, however, must not be used in combination, so each component has its own use. In this manner, complex rules are produced without using an excessive amount of language constructs. This will lead to user-friendly, self-explanatory rules.

8 Policy modelling rules

In section 1 we saw that production rules can be of use for interactive policy modelling. For example, they can be used to automatically update the visualization or the user interface when a certain condition becomes true.

The urban planning scenario in Bologna requires a user interface that allows users to insert, move or delete waste bins in the city. During interactive policy modelling rules can be very useful. The policy that is modelled here is how many waste bins should be available in a certain area. The user wants to find the right balance between too few bins—which would eventually lead to dirty streets, because people would throw waste on the ground—and too much bins—which would probably have a negative influence on the cityscape. A rule could be used here to control a visual indicator that goes from green (good) to red (bad). The farer the number of waste bins within an area is away from the ideal number, the more the indicator will lean to red. Event-based rule execution can be used here to update the indicator whenever a waste bin is added or removed.

In the following we will describe how domain-specific languages for policy modelling applications can be designed. We will start a general description of an ontology modelling process for domain-specific languages. Then we will present sample scenarios and sample languages. Finally, we derive entities and operations used in the domain-specific languages and show examples.

8.1 Ontology modelling

In the urbanAPI project we aim for creating domain-specific languages that use terms from the user's domain. In the area of semantic web ontology building is used to identify concepts and relations from a given application domain. Ontologies can be useful for the definition of domain-specific languages where they act as the basis from which the taxonomy, vocabulary, and parts of the grammar are derived.

Ontology building is typically divided into several parts and iterations (Nicola, Missikoff, & Navigli, 2009). In order to create a domain-specific language, the following workflow can be used:

1. Analyse the application domain
2. Create scenarios/storyboards
3. Look for subjects and objects. Create a concept in the ontology for each subject/object.
4. Look for predicates. Use them in the ontology as relations to connect the subjects and objects.
5. Build sample scripts that use the created ontology
6. Review and reiterate if needed

In the urbanAPI user workshop that was held from 2nd to 4th of July 2013 in Darmstadt we used this workflow to generate ontologies for the 3D VR applications. The first two steps, the domain analysis as well as the creation of scenarios or storyboards, had already been done before. The results of this can be found in deliverable D2.1.

In the workshop we therefore started with step 3. Together with users from the cities we analysed the scenario descriptions of Vitoria-Gasteiz and Bologna. We were looking for subjects and objects that we needed to add to the ontologies as concepts. We then set the concepts into relation by connecting them with predicates. The domain analyses can be found in sections 8.1.1 and 8.1.2. The resulting ontologies are presented in sections 8.1.3 and 8.1.4.

8.1.1 Ontology analysis for the 3D VR scenario in Vitoria-Gasteiz

The following text has been taken out of deliverable D2.1. It is the scenario description for the 3D VR application for Vitoria-Gasteiz. Terms important for the domain and hence for the ontology are marked in bold. Note that the terms have been selected during the user workshop. The text certainly contains more terms, but these are the ones that the users consider most important and that should go into the domain-specific languages.

*"[…] the Department in charge of Water Management studies the **permeable** pavement in the new section, the application helps them to calculate the water volume of run-off in different raining scenarios and this way they decide what capacity new sustainable **drainage** systems must have. They are also able to evaluate the amount of stormwater that will be released from going to the traditional **sewage** system to the natural depuration system.*

*The Parks and Gardens Department visualizes the state of the project to short, medium and long term in different scenarios corresponding to different epochs of the year and with different vegetable species. This way, they can decide which are the most appropriate species for each of the **streets**, according, for example, to shelter/**shadow**, biocapacity, biodiversity and management criteria and indicators.*

*The Architecture Department identifies the main **buildings** to be rehabilitated. They set a list of green factors that can be introduced in order to improve **energy consumption** and production, as well as increase biodiversity and biocapacity rates. The application helps its visualization in the urban scenario and shows the quantification of the new rates.*

*The Public Space Department plans a new distribution for the different modes of mobility, including the reduction of the on street parking in order to increase public space for **pedestrians**. The application helps to calculate the changes of uses in the new section. Following to this, they also realize a new type urban **lighting** could be more efficient, and the application is also capable of visualizing the **energy** savings that this change will mean in the set of the streets."*

8.1.2 Ontology analysis for the 3D VR scenario in Bologna

The following text has been taken out of deliverable D2.1. It is the scenario description for the 3D VR application for Bologna. Terms important for the domain and hence for the ontology are marked in bold. Note that the terms have been selected during the user workshop. The text certainly contains more terms, but these are the ones that the users consider most important and that should go into the domain-specific languages.

"Mr. Rossi loads 3D VR application […]. On the first screen he gets various options and text explaining what he can do. Options are related to the main objects which have been identified as the most important ones for the rehabilitation of the ***district****. Mainly these options are:*

- *Waste collection,*
- ***Green*** *areas such as park,* ***plant, tree*** *and garden, etc*
- *Car* ***parking spaces****,*
- *Street/Public space benches*
- ***Bike*** *sharing and racks*
- *Junction boxes,*
- *Traffic markings and road sings,*
- *Pedestrian* ***ways****,*
- *Open spaces for bars/restaurants, and*
- *Public street lighting.*

[…]

Following these steps, he clicks on the city maps of the district. He zooms to a street. He selects a ***garbage bin*** *located on the 3D street maps. The left side window/panel contains all the editing options he can apply to this object. He decides to eliminate it from the* ***street****.*

*Mr. Rossi decides to intervene on other options which are listed in the main left side window panel. For example, he selects "***Green***" option. The subset of this option has two other options, such as "***Trees***" and "***Plants***". […] Mr. Rossi drags one plant into the 3D screen and places on the location from where he earlier removed the* ***garbage bin****. Mr. Rossi also wants to insert a tree into the* ***street****. He again goes to the left panel and selects "***Trees***" option and as a result on the right side of the screen a new palette window opens showing different tree objects. He tries to drag one of the tree objects on the* ***roadway*** *but it doesn't work. A help box indicates that trees cannot be planted on roadways and suggests other nearby places to plant the tree. He again drags the tree on a little garden beside the street and everything works fine. […]"*

8.1.3 Ontology model for the 3D VR scenario in Vitoria-Gasteiz

After identifying the most important terms we put them as concepts into an ontology model and created relations between them. Some terms were merged to more general ones, some terms had to be added since they did not appear in the scenario descriptions yet. Note that the ontologies have been worked out together with the users during the user workshop. They contain concepts and relations the users consider most important for the domain-specific languages.

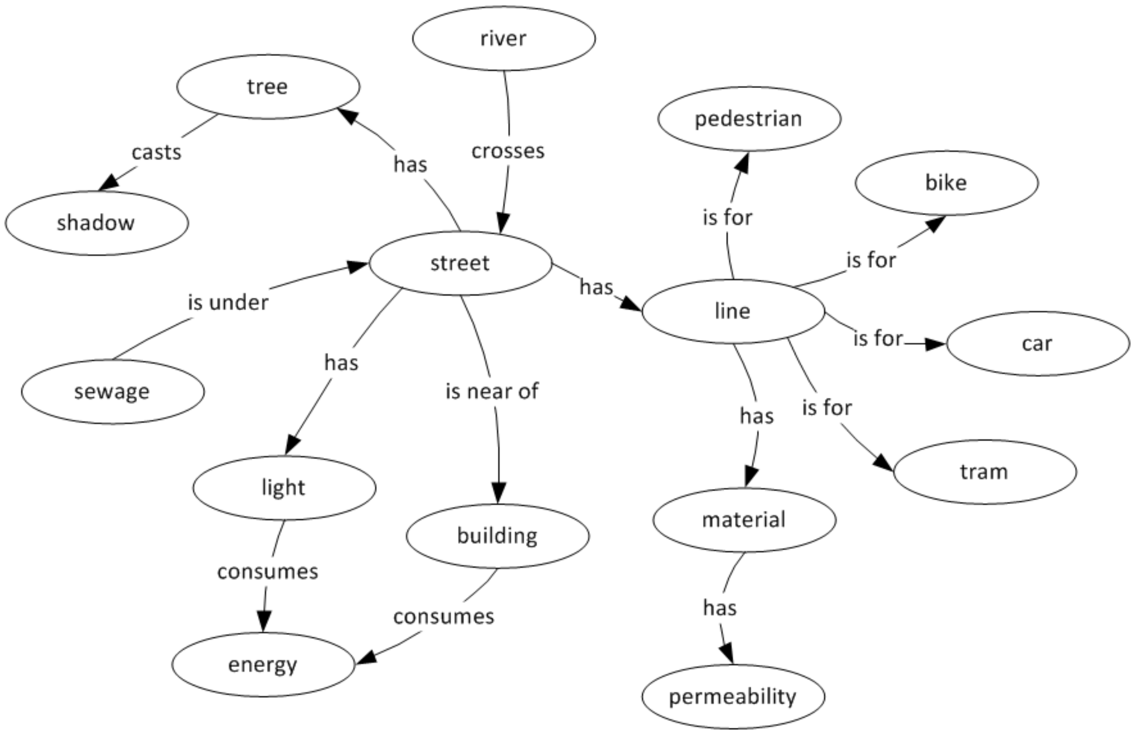


Figure 7 Ontology for the 3D VR scenario in Vitoria-Gasteiz

8.1.4 Ontology model for the 3D VR scenario in Bologna

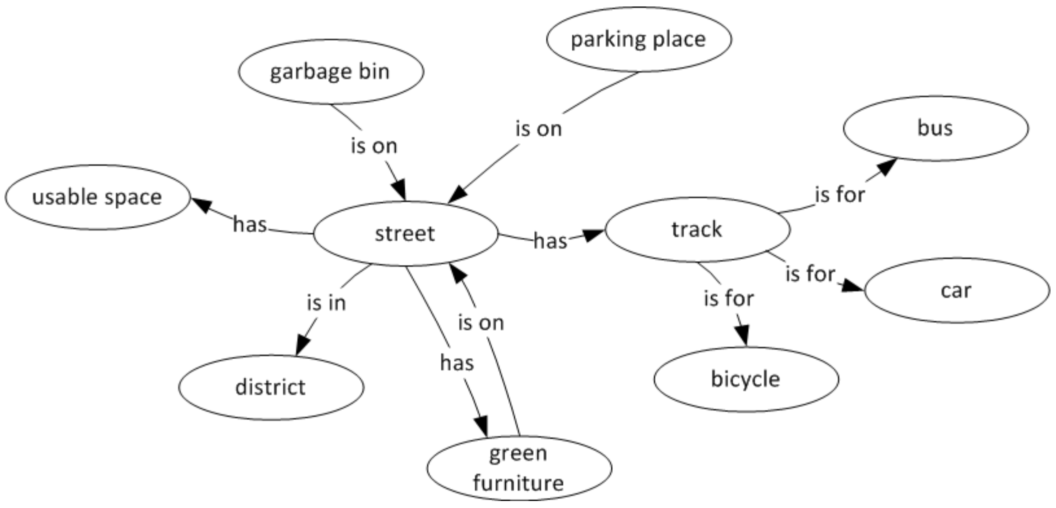


Figure 8 Ontology for the 3D VR scenario in Bologna

8.2 Sample scripts and rules for the 3D VR scenario in Vitoria-Gasteiz

After we created the ontologies, together with the users from the cities, we created example scripts or rules written in a domain-specific language—or precisely, multiple languages (see comment in section 8.5 below). We tried to incorporate the ontologies as much as possible into the syntax and grammar of these languages.

In Vitoria-Gasteiz the following rules could be used to calculate the energy consumption of all lights on the Avenue de Gasteiz:

```
The light of type A consumes 3000 kW/h

Street “Avenue de Gasteiz” has 50 lights of type A
```

Other, similar rules could be used to calculate how much water would drain away depending on the material used for the various lines on the Avenue de Gasteiz:

Material A has a permeability of 10^{-8} m/s

Material B has a permeability of 10^{-3} m/s

```
All pedestrian lines have material A

All tram lines have material B
```

8.3 Sample scripts and rules for the 3D VR scenario in Bologna

In Bologna the following rules could be used to alter the visualization depending on given parameters. For example, the following rule can be used to mark bicycle tracks that are long enough and not interrupted by other tracks. Such tracks should be displayed in green to indicate they are considered ‘good’ or ‘preferable’ tracks.

```
If a bicycle track is longer than 1 km
then display it in green
```

A similar rule could be used to display streets in green that have enough usable space.

```
A garbage bin takes 3 units of usable space

If the usable space of street A is more than 200 units
then display it in green
```

```
If the usable space of street A is less than 200 units
then display it in red
```

It is also possible to define interactive rules, i.e. rules that define how the system behaves depending on the user’s interaction:

```
When a garbage bin is removed on street B
then replace it by green furniture
```

```
When a street is free of garbage bins and parking places
then replace it with a bicycle track
```

8.4 Identified entities and operations

The domain-specific languages presented in the previous sections consist of the concepts identified in the domain analysis. In particular, the following **entities** are used:

Vitoria-Gasteiz:

- Light
- Street
- Material
- Permeability
- Pedestrian line
- Tram line

Bologna:

- Bicycle track
- Garbage bin
- Usable space
- Street
- Green furniture
- Parking place

Note that *usable space* can be used as a property of *street*. The ontology defines a *has* relation from *street* to *usable space*.

The following list contains **parameters** for the entities. Some of them are derived from concepts that do not directly appear as entities in the DSLs.

- Energy (expressed in kW/h)
- Permeability (expressed in m/s)
- Usable space (expressed in units)
- Lights
  - Number of lights
  - Type
- Material
  - Type
- Tracks and lines
  - Length

The following list contains **operators** and **operations** that can be applied to entities. Parameters are put in angle brackets.

- Light **consumes** <energy>
- Street <name> **has** <n> lights **of type** <t>
- A <entity> **takes** <n> units of <entity>
- **Display** <entity> in <color>
- **Replace** <entity> by/with <entity>

The DSLs allows users to define rules using the following **conditional expressions**:

- **If** <entity> **is longer/shorter than** <n> **then** …
- **If** <entity> **is more than/is less than** <n> **then** …
- **If** <entity> of <entity> **is more/less than** <n> **then** …
  (in the case of "usable space of street")

The DSLs allow users to define actions that should be performed as a response to **events**:

- **When** <entity> is removed **then** …
- **When** <entity> is added **then** …

In order to make the DSLs more readable and usable a number of filler words have been added. These words exist merely for readability. They are skipped during parsing and do not change the semantics of the rules:

- a
- the

In conditional expressions the pseudo entity **it** can be used as a placeholder for the expression’s subject as follows:

- If **<entity>** is longer/shorter than <n> then display **it** in <color>
- If <entity> of **<entity>** is more/less than <n> then display **it** in <color>
- etc.

8.5 Next steps and conclusion

In the previous sections domain-specific languages for two scenarios have been defined (including their ontology, syntax and grammar). The languages will now be implemented so the scripts can be tested with real data in the UrbanAPI web portal.

As stated above, domain-specific languages should be very easy to understand for the domain expert. Therefore DSLs consist of a very limited vocabulary where single expressions may be quite powerful. Unlike general purpose languages, DSLs are not very flexible. This means that such a language may only be used in a very specific application domain. This is of course the main goal of DSLs, but it also shows one drawback: the easier a language is to learn or understand (i.e. the fewer tokens it contains in its vocabulary) the less flexible it is. Vice versa: the more general or flexible a language is the more complicated or harder it is to learn. What is actually needed is a balance between these two goals: flexibility and usability. This can be achieved by reusing language constructs.

The ontologies presented above show that there are common concepts shared by both scenarios. By identifying even more common concepts (also by analysing the other project scenarios from

Vienna and Ruse) we think that it is possible to create a basic domain-specific language that consists of common terms. It should be able to derive languages specific to each use case from the basic language. This will make the language definition process a lot easier and faster. Identifying this basic language is subject to research in the upcoming project months.

9 Conclusion

In this report we presented the motivation for using rule-based systems and domain-specific languages in the urbanAPI project. Rule-based systems can be used quite well to automate complex workflows in the geospatial domain, but they are rather complex and hard to understand for non-IT personnel. Domain-specific languages are tailored to a very specific application domain and help users express workflows in their own language.

In this report we also presented an overview over the state of the art and made a comprehensive comparison of different rule-based systems. We finally decided for Drools since if fulfils all our requirements and can be integrated very well into the existing infrastructure.

We presented the current status of development in the CityServer3D, where we already implemented a graphical rule editor.

Finally, we presented a concept for textual domain-specific languages on which we will continue to work in the upcoming project months.

10 References

Barrientos, P., & Lopez, P. (2009). Developing DSLs using combinators. A design pattern. *International Multiconference on Computer Science and Information Technology, IEEE*, (pp. 635-642).

Chafi, H., & al. (2010). Language Virtualization for Heterogeneous Parallel Computing. *Proc of ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications* (pp. 835-847). ACM Press.

Coors, V., & Krämer, M. (2011). Integrating quality management into a 3D geospatial server. *28th Urban Data Management Symposium UDMS at 40 Years: Making Contributions to the Future* (pp. 7-12). International Society for Photogrammetry and Remote Sensing (ISPRS).

Doorenbos, R. B. (1995). *Production Matching for Large Learning Systems.*

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 17-37.

Fowler, M. (2010). *Domain-Specific Languages.* Amsterdam: Addison-Wesley Longman.

Hudak, P. (1998). Modular domain specific languages and tools. *Proceedings of the Fifth International Conference on Software Reuse* (pp. 134-142). IEEE Comput. Soc.

Irazabal, J., & Pons, C. (2010). Supporting Modularization in Textual DSL Development. *XXIX International Conference of the Chilean Computer Science Society* (pp. 124-130). IEEE.

Krämer, M. (2013). *Internal report: Dissertation Proposal.* Technical University of Darmstadt, Fraunhofer IGD.

Krämer, M., Ludlow, D., & Khan, Z. (2013). Domain-specific rule languages for urban policy modelling (To be submmitted). *European conference on modelling and simulation*.

Lee, H., Brown, K., Sujeeth, A., & al. (2011). Domain-specific Languages for Heterogeneous Parallel Computing. *IEEE Micro 31, vol. 5*, pp. 42-53.

Mernik, M., Heering, J., & Sloane, A. (2003). *When and how to develop domain-specific languages.* Unversity of Maribor, CWI Amsterdam, and Macquarie University.

Miranker, D. P. (1987). TREAT: A better match algorithm for AI production systems. *Elements*, 42-47.

Nicola, A. D., Missikoff, M., & Navigli, R. (2009). A software engineering approach to ontology building. *Information Systems 34*, pp. 258–275.

Perlis, A. (1982). Epigrams on Programming. *Sigplan Notices 17, 9*, 7-13.

Pizka, M., & Jürgens, E. (2007). Automating Language Evolution. *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07)* (pp. 305-315). IEEE.

Reitz, T., Krämer, M., & Thum, S. (2009). A processing pipeline for X3D Earth-based spatial data view services. *Proceedings Web3D 2009: 14th International Conference on 3D Web Technology* (pp. 137-147). ACM Press.

Sohr, K., Ahn, G.-J., & Migge, L. (2005). Articulating and enforcing authorisation policies with UML and OCL. *In Proceedings of the 2005 workshop on Software engineering for secure systems—building trustworthy applications (SESS '05), ACM, New York, NY, USA*, pp. 1-7.

Sujeeth, A. K., & al. (2011). OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. *Proc of the 28th Int'l Conf. Machine Learning (ICML 11)* (pp. 609-616). ACM Press.

Thum, S., & Krämer, M. (2011). Reducing Maintenance Complexity of User-centric Web Portrayal Services. *Proceedings Web3D 2011: 16th International Conference on 3D Web Technology* (pp. 165-172). ACM Press.

van Oosterom, P. (2009). Research and development in geo-information generalisation and multiple representation. *Computers, Environment and Urban Systems 33, 5*, (pp. 303-310).