Domain-specific language

A **domain-specific language** (**DSL**) is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language (GPL), which is broadly applicable across domains. There are a wide variety of DSLs, ranging from widely used languages for common domains, such as HTML for web pages, down to languages used by only one or a few pieces of software, such as MUSH soft code. DSLs can be further subdivided by the kind of language, and include domain-specific *markup* languages, domain-specific *modeling* languages (more generally, specification languages), and domain-specific *programming* languages. Special-purpose computer languages have always existed in the computer age, but the term "domain-specific language" has become more popular due to the rise of domain-specific modeling. Simpler DSLs, particularly ones used by a single application, are sometimes informally called **mini-languages**.

The line between general-purpose languages and domain-specific languages is not always sharp, as a language may have specialized features for a particular domain but be applicable more broadly, or conversely may in principle be capable of broad application but in practice used primarily for a specific domain. For example, Perl was originally developed as a text-processing and glue language, for the same domain as AWK and shell scripts, but was mostly used as a general-purpose programming language later on. By contrast, PostScript is a Turing complete language, and in principle can be used for any task, but in practice is narrowly used as a page description language.

Contents

Use

Overview
- In design and implementation
- Programming tools

Domain-specific language topics
- External and Embedded Domain Specific Languages
- Usage patterns
- Design goals
- Idioms

Examples
- GameMaker Language
- Unix shell scripts
- ColdFusion Markup Language
- Erlang OTP
- FilterMeister
- MediaWiki templates
- Software engineering uses
- Metacompilers
- Unreal Engine before version 4 and other games
- Rules Engines for Policy Automation
- Statistical modelling languages
- Generate model and services to multiple programming Languages

Gherkin
Other examples
Advantages and disadvantages
Tools for designing domain-specific languages
See also
References
Further reading
External links

Use

The design and use of appropriate DSLs is a key part of domain engineering, by using a language suitable to the domain at hand – this may consist of using an existing DSL or GPL, or developing a new DSL. Language-oriented programming considers the creation of special-purpose languages for expressing problems as standard part of the problem-solving process. Creating a domain-specific language (with software to support it), rather than reusing an existing language, can be worthwhile if the language allows a particular type of problem or solution to be expressed more clearly than an existing language would allow and the type of problem in question reappears sufficiently often. Pragmatically, a DSL may be specialized to a particular problem domain, a particular problem representation technique, a particular solution technique, or other aspects of a domain.

Overview

A domain-specific language is created specifically to solve problems in a particular domain and is not intended to be able to solve problems outside of it (although that may be technically possible). In contrast, general-purpose languages are created to solve problems in many domains. The domain can also be a business area. Some examples of business areas include:

- domain-specific language for life insurance policies developed internally in large insurance enterprise
- domain-specific language for combat simulation
- domain-specific language for salary calculation
- domain-specific language for billing

A domain-specific language is somewhere between a tiny programming language and a scripting language, and is often used in a way analogous to a programming library. The boundaries between these concepts are quite blurry, much like the boundary between scripting languages and general-purpose languages.

In design and implementation

Domain-specific languages are languages (or often, declared syntaxes or grammars) with very specific goals in design and implementation. A domain-specific language can be one of a visual diagramming language, such as those created by the Generic Eclipse Modeling System, programmatic abstractions, such as the Eclipse Modeling Framework, or textual languages. For instance, the command line utility grep has a regular expression syntax which matches patterns in lines of text. The sed utility defines a syntax for matching and replacing regular expressions. Often, these tiny languages can be used together inside a shell to perform more complex programming tasks.

The line between domain-specific languages and scripting languages is somewhat blurred, but domain-specific languages often lack low-level functions for filesystem access, interprocess control, and other functions that characterize full-featured programming languages, scripting or otherwise. Many domain-specific languages do not compile to byte-code or executable code, but to various kinds of media objects: GraphViz exports to PostScript, GIF, JPEG, etc., where Csound compiles to audio files, and a ray-tracing domain-specific language like POV compiles to graphics files. A computer language like SQL presents an interesting case: it can be deemed a domain-specific language because it is specific to a specific domain (in SQL's case, accessing and managing relational databases), and is often called from another application, but SQL has more keywords and functions than many scripting languages, and is often thought of as a language in its own right, perhaps because of the prevalence of database manipulation in programming and the amount of mastery required to be an expert in the language.

Further blurring this line, many domain-specific languages have exposed APIs, and can be accessed from other programming languages without breaking the flow of execution or calling a separate process, and can thus operate as programming libraries.

Programming tools

Some domain-specific languages expand over time to include full-featured programming tools, which further complicates the question of whether a language is domain-specific or not. A good example is the functional language XSLT, specifically designed for transforming one XML graph into another, which has been extended since its inception to allow (particularly in its 2.0 version) for various forms of filesystem interaction, string and date manipulation, and data typing.

In model-driven engineering, many examples of domain-specific languages may be found like OCL, a language for decorating models with assertions or QVT, a domain-specific transformation language. However, languages like UML are typically general-purpose modeling languages.

To summarize, an analogy might be useful: a Very Little Language is like a knife, which can be used in thousands of different ways, from cutting food to cutting down trees. A domain-specific language is like an electric drill: it is a powerful tool with a wide variety of uses, but a specific context, namely, putting holes in things. A General Purpose Language is a complete workbench, with a variety of tools intended for performing a variety of tasks. Domain-specific languages should be used by programmers who, looking at their current workbench, realize they need a better drill and find that a particular domain-specific language provides exactly that.

Domain-specific language topics

External and Embedded Domain Specific Languages

DSLs implemented via an independent interpreter or compiler are known as *External Domain Specific Languages*. Well known examples include LaTeX or AWK. A separate category known as *Embedded (or Internal) Domain Specific Languages* are typically implemented within a host language as a library and tend to be limited to the syntax of the host language, though this depends on host language capabilities.[1]

Usage patterns

There are several usage patterns for domain-specific languages:[2][3]

- Processing with standalone tools, invoked via direct user operation, often on the command line or from a Makefile (e.g., grep for regular expression matching, sed, lex, yacc, the GraphViz toolset, etc.)
- Domain-specific languages which are implemented using programming language macro systems, and which are converted or expanded into a host general purpose language at compile-time or realtime
- **embedded domain-specific language** (**eDSL**),[4] implemented as libraries which exploit the syntax of their host general purpose language or a subset thereof while adding domain-specific language elements (data types, routines, methods, macros etc.). (e.g. jQuery, React, Embedded SQL, LINQ)
- Domain-specific languages which are called (at runtime) from programs written in general purpose languages like C or Perl, to perform a specific function, often returning the results of operation to the "host" programming language for further processing; generally, an interpreter or virtual machine for the domain-specific language is embedded into the host application (e.g. format strings, a regular expression engine)
- Domain-specific languages which are embedded into user applications (e.g., macro languages within spreadsheets) and which are (1) used to execute code that is written by users of the application, (2) dynamically generated by the application, or (3) both.

Many domain-specific languages can be used in more than one way. DSL code embedded in a host language may have special syntax support, such as regexes in sed, AWK, Perl or JavaScript, or may be passed as strings.

Design goals

Adopting a domain-specific language approach to software engineering involves both risks and opportunities. The well-designed domain-specific language manages to find the proper balance between these.

Domain-specific languages have important design goals that contrast with those of general-purpose languages:

- Domain-specific languages are less comprehensive.
- Domain-specific languages are much more expressive in their domain.
- Domain-specific languages should exhibit minimal redundancy.

Idioms

In programming, idioms are methods imposed by programmers to handle common development tasks, e.g.:

- Ensure data is saved before the window is closed.
- Edit code whenever command-line parameters change because they affect program behavior.

General purpose programming languages rarely support such idioms, but domain-specific languages can describe them, e.g.:

- A script can automatically save data.
- A domain-specific language can parameterize command line input.

Examples

Examples of domain-specific languages include HTML, Logo for pencil-like drawing, Verilog and VHDL hardware description languages, MATLAB and GNU Octave for matrix programming, Mathematica, Maple and Maxima for symbolic mathematics, Specification and Description Language for reactive and distributed systems, spreadsheet formulas and macros, SQL for relational database queries, YACC grammars for creating parsers, regular expressions for specifying lexers, the Generic Eclipse Modeling System for creating diagramming languages, Csound for sound and music synthesis, and the input languages of GraphViz and GrGen, software packages used for graph layout and graph rewriting.

GameMaker Language

The GML scripting language used by GameMaker Studio is a domain-specific language targeted at novice programmers to easily be able to learn programming. While the language serves as a blend of multiple languages including Delphi, C++, and BASIC, there is a lack of structures, data types, and other features of a full-fledged programming language. Many of the built-in functions are sandboxed for the purpose of easy portability. The language primarily serves to make it easy for anyone to pick up the language and develop a game.

Unix shell scripts

Unix shell scripts give a good example of a domain-specific language for data[5] organization. They can manipulate data in files or user input in many different ways. Domain abstractions and notations include streams (such as stdin and stdout) and operations on streams (such as redirection and pipe). These abstractions combine to make a robust language to describe the flow and organization of data.

The language consists of a simple interface (a script) for running and controlling processes that perform small tasks. These tasks represent the idioms of organizing data into a desired format such as tables, graphs, charts, etc.

These tasks consist of simple control-flow and string manipulation mechanisms that cover a lot of common usages like searching and replacing string in files, or counting occurrences of strings (frequency counting).

Even though Unix scripting languages are Turing complete, they differ from general purpose languages.

In practice, scripting languages are used to weave together small Unix tools such as grep, ls, sort or wc.

ColdFusion Markup Language

ColdFusion's associated scripting language is another example of a domain-specific language for data-driven websites. This scripting language is used to weave together languages and services such as Java, .NET, C++, SMS, email, email servers, http, ftp, exchange, directory services, and file systems for use in websites.

The ColdFusion Markup Language (CFML) includes a set of tags that can be used in ColdFusion pages to interact with data sources, manipulate data, and display output. CFML tag syntax is similar to HTML element syntax.

Erlang OTP

The Erlang Open Telecom Platform was originally designed for use inside Ericsson as a domain-specific language. The language itself offers a platform of libraries to create finite state machines, generic servers and event managers that quickly allow an engineer to deploy applications, or support libraries, that have been

shown in industry benchmarks to outperform other languages intended for a mixed set of domains, such as C and C++. The language is now officially open source and can be downloaded from their website.

FilterMeister

FilterMeister is a programming environment, with a programming language that is based on C, for the specific purpose of creating Photoshop-compatible image processing filter plug-ins; FilterMeister runs as a Photoshop plug-in itself and it can load and execute scripts or compile and export them as independent plug-ins. Although the FilterMeister language reproduces a significant portion of the C language and function library, it contains only those features which can be used within the context of Photoshop plug-ins and adds a number of specific features only useful in this specific domain.

MediaWiki templates

The *Template* feature of MediaWiki is an embedded domain-specific language whose fundamental purpose is to support the creation of page templates and the transclusion (inclusion by reference) of MediaWiki pages into other MediaWiki pages.

Software engineering uses

There has been much interest in domain-specific languages to improve the productivity and quality of software engineering. Domain-specific language could possibly provide a robust set of tools for efficient software engineering. Such tools are beginning to make their way into the development of critical software systems.

The Software Cost Reduction Toolkit[6] is an example of this. The toolkit is a suite of utilities including a specification editor to create a requirements specification, a dependency graph browser to display variable dependencies, a consistency checker to catch missing cases in well-formed formulas in the specification, a model checker and a theorem prover to check program properties against the specification, and an invariant generator that automatically constructs invariants based on the requirements.

A newer development is language-oriented programming, an integrated software engineering methodology based mainly on creating, optimizing, and using domain-specific languages.

Metacompilers

Complementing language-oriented programming, as well as all other forms of domain-specific languages, are the class of compiler writing tools called metacompilers. A metacompiler is not only useful for generating parsers and code generators for domain-specific languages, but a metacompiler itself compiles a domain-specific metalanguage specifically designed for the domain of metaprogramming.

Besides parsing domain-specific languages, metacompilers are useful for generating a wide range of software engineering and analysis tools. The meta-compiler methodology is often found in program transformation systems.

Metacompilers that played a significant role in both computer science and the computer industry include Meta-II,[7] and its descendant TreeMeta.[8]

Unreal Engine before version 4 and other games

Unreal and Unreal Tournament unveiled a language called UnrealScript. This allowed for rapid development of modifications compared to the competitor Quake (using the Id Tech 2 engine). The Id Tech engine used standard C code meaning C had to be learned and properly applied, while UnrealScript was optimized for ease of use and efficiency. Similarly, the development of more recent games introduced their own specific languages, one more common example is Lua for scripting.

Rules Engines for Policy Automation

Various Business Rules Engines have been developed for automating policy and business rules used in both government and private industry. ILOG, Oracle Policy Automation, DTRules, Drools and others provide support for DSLs aimed to support various problem domains. DTRules goes so far as to define an interface for the use of multiple DSLs within a Rule Set.

The purpose of Business Rules Engines is to define a representation of business logic in as human-readable fashion as possible. This allows both subject matter experts and developers to work with and understand the same representation of the business logic. Most Rules Engines provide both an approach to simplifying the control structures for business logic (for example, using Declarative Rules or Decision Tables) coupled with alternatives to programming syntax in favor of DSLs.

Statistical modelling languages

Statistical modelers have developed domain-specific languages such as R (an implementation of the S language), Bugs, Jags, and Stan. These languages provide a syntax for describing a Bayesian model and generate a method for solving it using simulation.

Generate model and services to multiple programming Languages

Generate object handling and services based on an Interface Description Language for a domain-specific language such as JavaScript for web applications, HTML for documentation, C++ for high-performance code, etc. This is done by cross-language frameworks such as Apache Thrift or Google Protocol Buffers.

Gherkin

Gherkin is a language designed to define test cases to check the behavior of software, without specifying how that behavior is implemented. It is meant to be read and used by non-technical users using a natural language syntax and a line-oriented design. The tests defined with Gherkin must then be implemented in a general programming language. Then, the steps in a Gherkin program acts as a syntax for method invocation accessible to non-developers.

Other examples

Other prominent examples of domain-specific languages include:

- Emacs Lisp
- Game Description Language
- OpenGL Shading Language
- Gradle

Advantages and disadvantages

Some of the advantages:[2][3]

- Domain-specific languages allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. The idea is that domain experts themselves may understand, validate, modify, and often even develop domain-specific language programs. However, this is seldom the case.[9]
- Domain-specific languages allow validation at the domain level. As long as the language constructs are safe any sentence written with them can be considered safe.
- Domain-specific languages can help to shift the development of business information systems from traditional software developers to the typically larger group of domain-experts who (despite having less technical expertise) have a deeper knowledge of the domain.[10]
- Domain-specific languages are easier to learn, given their limited scope.

Some of the disadvantages:

- Cost of learning a new language vs. its limited applicability
- Cost of designing, implementing, and maintaining a domain-specific language as well as the tools required to develop with it (IDE)
- Finding, setting, and maintaining proper scope.
- Difficulty of balancing trade-offs between domain-specificity and general-purpose programming language constructs.
- Potential loss of processor efficiency compared with hand-coded software.
- Proliferation of similar non-standard domain-specific languages, for example, a DSL used within one insurance company versus a DSL used within another insurance company.[11]
- Non-technical domain experts can find it hard to write or modify DSL programs by themselves.[9]
- Increased difficulty of integrating the DSL with other components of the IT system (as compared to integrating with a general-purpose language).
- Low supply of experts in a particular DSL tends to raise labor costs.
- Harder to find code examples.

Tools for designing domain-specific languages

- JetBrains MPS is a tool for designing domain-specific languages. It uses projectional editing which allows overcoming the limits of language parsers and building DSL editors, such as ones with tables and diagrams. It implements language-oriented programming. MPS combines an environment for language definition, a language workbench, and an Integrated Development Environment (IDE) for such languages.[12]
- Xtext is an open-source software framework for developing programming languages and domain-specific languages (DSLs). Unlike standard parser generators, Xtext generates not only a parser but also a class model for the abstract syntax tree. In addition, it provides a fully featured, customizable Eclipse-based IDE.[13]
- Racket is a cross-platform language toolchain including compiler, JIT compiler, IDE and command line tools designed to accommodate creating both domain-specific languages and completely new languages.[14][15]

See also

- Language workbench
- Architecture description language
- Domain-specific entertainment language
- Language for specific purposes
- Metalinguistic abstraction
- Programming domain

References

1. Fowler, Martin; Parsons, Rebecca. "Domain Specific Languages" (https://martinfowler.com/books/dsl.html). Retrieved 6 July 2019.
2. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.doi:10.1145/1118890.1118892 (https://doi.org/10.1145%2F1118890.1118892)
3. Diomidis Spinellis. Notable design patterns for domain specific languages (http://www.spinellis.gr/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html). *Journal of Systems and Software*, 56(1):91–99, February 2001. doi:10.1016/S0164-1212(00)00089-3 (https://doi.org/10.1016%2FS0164-1212%2800%2900089-3)
4. Felleisen, Matthias; Findler, Robert Bruce; Flatt, Matthew; Krishnamurthi, Shriram; Barzilay, Eli; McCarthy, Jay; Tobin-Hochstadt, Sam (March 2018). "A Programmable Programming Language" (https://cacm.acm.org/magazines/2018/3/225475-a-programmable-programming-language/fulltext). *Communications of the ACM*. **61** (3): 62–71. doi:10.1145/3127323 (https://doi.org/10.1145%2F3127323). S2CID 3887010 (https://api.semanticscholar.org/CorpusID:3887010). Retrieved 15 May 2019.
5. "Data definition by The Linux Information Project (LINFO)" (http://www.linfo.org/data.html). *www.linfo.org*. Retrieved 2016-01-14.
6. Heitmeyer, C. (1998). "Using the SCR* toolset to specify software requirements" (https://web.archive.org/web/20040719135712/http://chacs.nrl.navy.mil/publications/CHACS/1998/1998heitmeyer-WIFT.pdf) (PDF). *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*. IEEE. pp. 12–13. doi:10.1109/WIFT.1998.766290 (https://doi.org/10.1109%2FWIFT.1998.766290). ISBN 0-7695-0081-1. S2CID 16079058 (https://api.semanticscholar.org/CorpusID:16079058). Archived from the original (http://chacs.nrl.navy.mil/publications/CHACS/1998/1998heitmeyer-WIFT.pdf) (PDF) on 2004-07-19.
7. Shorre, D. V. (1964). "META II a syntax-oriented compiler writing language". *Proceedings of the 1964 19th ACM National Conference*: 41.301–41.3011. doi:10.1145/800257.808896 (https://doi.org/10.1145%2F800257.808896). S2CID 43144779 (https://api.semanticscholar.org/CorpusID:43144779).
8. Carr, C. Stephen; Luther, David A.; Erdmann, Sherian (1969). "The TREE-META Compiler-Compiler System: A Meta Compiler System for the Univac 1108 and General Electric 645" (https://apps.dtic.mil/docs/citations/AD0855122). *University of Utah Technical Report RADC-TR-69-83*.
9. Freudenthal, Margus (1 January 2009). "Domain Specific Languages in a Customs Information System". *IEEE Software*: 1. doi:10.1109/MS.2009.152 (https://doi.org/10.1109%2FMS.2009.152).
10. Aram, Michael; Neumann, Gustaf (2015-07-01). "Multilayered analysis of co-development of business information systems" (http://www.jisajournal.com/content/pdf/s13174-015-0030-8.pdf) (PDF). *Journal of Internet Services and Applications*. **6** (1). doi:10.1186/s13174-015-0030-8 (https://doi.org/10.1186%2Fs13174-015-0030-8). S2CID 16502371 (https://api.semanticscholar.org/CorpusID:16502371).

11. Miotto, Eric. "On the integration of domain-specific and scientific bodies of knowledge in Model Driven Engineering" (https://web.archive.org/web/20110724223732/http://adams-project.org/standrts09/proceedings/miotto_vardanega_standrts09_final.pdf) (PDF). Archived from the original (http://adams-project.org/standrts09/proceedings/miotto_vardanega_standrts09_final.pdf) (PDF) on 2011-07-24. Retrieved 2010-11-22.
12. "JetBrains MPS: Domain-Specific Language Creator" (https://www.jetbrains.com/mps/).
13. "Xtext" (https://eclipse.org/Xtext/).
14. Tobin-Hochstadt, S.; St-Amour, V.; Culpepper, R.; Flatt, M.; Felleisen, M. (2011). "Languages as Libraries" (http://www.ccs.neu.edu/scheme/pubs/pldi11-thacff.pdf) (PDF). *Programming Language Design and Implementation*.
15. Flatt, Matthew (2012). "Creating Languages in Racket" (http://cacm.acm.org/magazines/2012/1/144809-creating-languages-in-racket). *Communications of the ACM*. Retrieved 2012-04-08.

Further reading

- Dunlavey (1994). *Building Better Applications: a Theory of Efficient Software Development*. International Thomson Publishing. ISBN 0-442-01740-5.
- Heitmeyer, Constance (1998). "Using the SCR Tool-set to Specify Software Requirements" (https://apps.dtic.mil/dtic/tr/fulltext/u2/a465026.pdf) (PDF). *Proceedings, Second IEEE Workshop on Industrial Strength Formal Specification Techniques, Boca Raton, FL, Oct. 19, 1998*: 12–13. doi:10.1109/WIFT.1998.766290 (https://doi.org/10.1109%2FWIFT.1998.766290). ISBN 0-7695-0081-1. S2CID 16079058 (https://api.semanticscholar.org/CorpusID:16079058).
- Mernik, Marjan; Heering, Jan & Sloane, Anthony M. (2005). "When and how to develop domain-specific languages" (https://ir.cwi.nl/pub/10893). *ACM Computing Surveys*. **37** (4): 316–344. doi:10.1145/1118890.1118892 (https://doi.org/10.1145%2F1118890.1118892). S2CID 207158373 (https://api.semanticscholar.org/CorpusID:207158373).
- Spinellis, Diomidis (2001). "Notable design patterns for domain specific languages". *Journal of Systems and Software*. **56** (1): 91–99. doi:10.1016/S0164-1212(00)00089-3 (https://doi.org/10.1016%2FS0164-1212%2800%2900089-3).
- Parr, Terence (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages*. ISBN 978-0-9787392-5-6.
- Larus, James (2009). "Spending Moore's Dividend". *Communications of the ACM*. **52** (5): 62–69. doi:10.1145/1506409.1506425 (https://doi.org/10.1145%2F1506409.1506425). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782). S2CID 2803479 (https://api.semanticscholar.org/CorpusID:2803479).
- Werner Schuster (June 15, 2007). "What's a Ruby DSL and what isn't?" (http://www.infoq.com/news/2007/06/dsl-or-not). C4Media. Retrieved 2009-09-08.
- Fowler, Martin (2011). *Domain Specific Languages*. ISBN 978-0-321-71294-3.
- Brambilla, Marco; Cabot, Jordi; Wimmer, Manuel (2012). *Model Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. **#1**. Morgan & Claypool. ISBN 978-1-62705-708-0.

External links

- "Minilanguages (http://www.faqs.org/docs/artu/minilanguageschapter.html)", *The Art of Unix Programming,* by Eric S. Raymond
- Martin Fowler on domain-specific languages (http://martinfowler.com/dsl.html) and Language Workbenches (http://www.martinfowler.com/articles/languageWorkbench.html). Also in a video presentation (http://www.infoq.com/presentations/domain-specific-languages)

- Domain-Specific Languages: An Annotated Bibliography (http://www.st.ewi.tudelft.nl/~arie/papers/dslbib.pdf)
- One Day Compilers: Building a small domain-specific language using OCaml (http://www.venge.net/graydon/talks/mkc/html/index.html)
- Usenix Association: Conference on Domain-Specific Languages (DSL '97) (http://www.usenix.org/publications/library/proceedings/dsl97) and 2nd Conference on Domain-Specific Languages (DSL '99) (http://www.usenix.org/publications/library/proceedings/dsl99)
- Internal Domain-Specific Languages (http://philcalcado.com/content/research_on_domain_specific_languages.html#toc-internal)
- The complete guide to (external) Domain Specific Languages (https://tomassetti.me/domain-specific-languages/)
- jEQN (https://sites.google.com/site/simulationarchitecture/jeqn) example of internal Domain-Specific Language for the Modeling and Simulation of Extended Queueing Networks.

Articles

- External DSLs with Eclipse technology (http://www.theserverside.com/tt/articles/article.tss?l=PragmaticGen)
- "Building Domain-Specific Languages over a Language Framework". CiteSeerX 10.1.1.50.4685 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.4685).
- Using Acceleo with GMF : Generating presentations from a MindMap DSL modeler (http://www.acceleo.org/pages/using-acceleo-with-gmf/)
- UML vs. Domain-Specific Languages (http://www.methodsandtools.com/archive/archive.php?id=71)
- Sagar Sen; et al. "Meta-model Pruning". CiteSeerX 10.1.1.156.6008 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.6008).
