Archives of Computational Methods in Engineering (2021) 28:3113–3134
https://doi.org/10.1007/s11831-020-09492-4

ORIGINAL PAPER

Domain-Specific Language Techniques for Visual Computing: A Comprehensive Study

Liming Shen[1] · Xueyi Chen[1] · Richen Liu[1] · Hailong Wang[1] · Genlin Ji[1]

Received: 10 February 2020 / Accepted: 27 August 2020 / Published online: 27 October 2020

Abstract
As a part of domain-specific development, Domain-Specific Language (DSL) is widely used in both the academia and industry to solve different aspects of the problems in engineering. A DSL is a customized language whose expressiveness is tailored to a well-defined application domain, so as to offer an effective interface for the domain experts. To mitigate the programming complexity of the General-Purpose Programming Languages, and meanwhile maintain the precise expression towards some exact engineering domains, DSLs present a higher level of abstraction than low-level interfaces, while providing much more flexibility than high-level interfaces. Nevertheless, it lacks a survey to have a systematic overview of the essential commonalities shared by those works. In this survey, we take a brand-new perspective, to categorize the state-of-the-art works into different categories, tailored to three fundamental implementation concerns of DSLs: abstract syntax, concrete syntax, and semantics. Specifically, they are characterized according to their parsing and mapping strategy (external/internal) between the abstract syntax and concrete syntax, the mapping results (textual/graphical symbols), and also the functions they emphasize (modeling, visualizing, etc.). Integrated with the literature, we finally summarized the research overview of DSLs.

1 Introduction

Domain-specific languages (DSLs) are languages that are tailored to specific engineering domains and offer significant improvements in expressiveness and ease of use compared with general programming languages. DSLs are highly specialized to a particular solution domain and provide the high-level abstractions which serve as building blocks helping analysts to quickly create programs. In addition, they rely on visual elements (such as layout, colors and lines) to represent the meaning of programs, which can help programmers with correctly understanding programs [1].

DSL development is difficult and requires domain knowledge and language development expertise. But as time proceeds, DSLs have been widely used in science and engineering projects. Scientists often struggle to manage data complexity and choose the version to control system and developers need to know their choice of tools and languages, and the obstacles these choices bring. DSLs can provide higher levels of abstraction in environments close to the domain of application expert [2], to quickly and correctly complete a program that meets the requirements, helping scientists and analysts deal with complex code and data and estimate future trends about the data use or context.

Although DSLs have been widely used in various scientific fields, the summary of development methods of DSLs is very limited. Thus, we have summarized the visualization work in the areas of physics, chemistry, biology, and meteorology that use DSLs, with some guidance as to whether or how scientists and domain experts use DSLs.

1.1 Related Surveys

In many scientific and engineering fields, such as genomics and astrophysics, scientists write programs to analyze data gathered from experiments. Jones et al. [4] reveal several opportunities to improve DSLs and related tools, such as helping scientists deal with data complexity and anticipate problems when choosing a language. They examined the number and purpose of DSLs used by scientists and the extent to which science projects switch between DSLs and/or other languages. They consider scientific modeling programs, each of which has the following three-step structure: (1) loading data about physical phenomena, (2) using this data to run one

✉ Richen Liu
richen@pku.edu.cn

[1] School of Computer and Electronic Information / School of Artificial Intelligence, Nanjing Normal University, Nanjing, Jiangsu, China

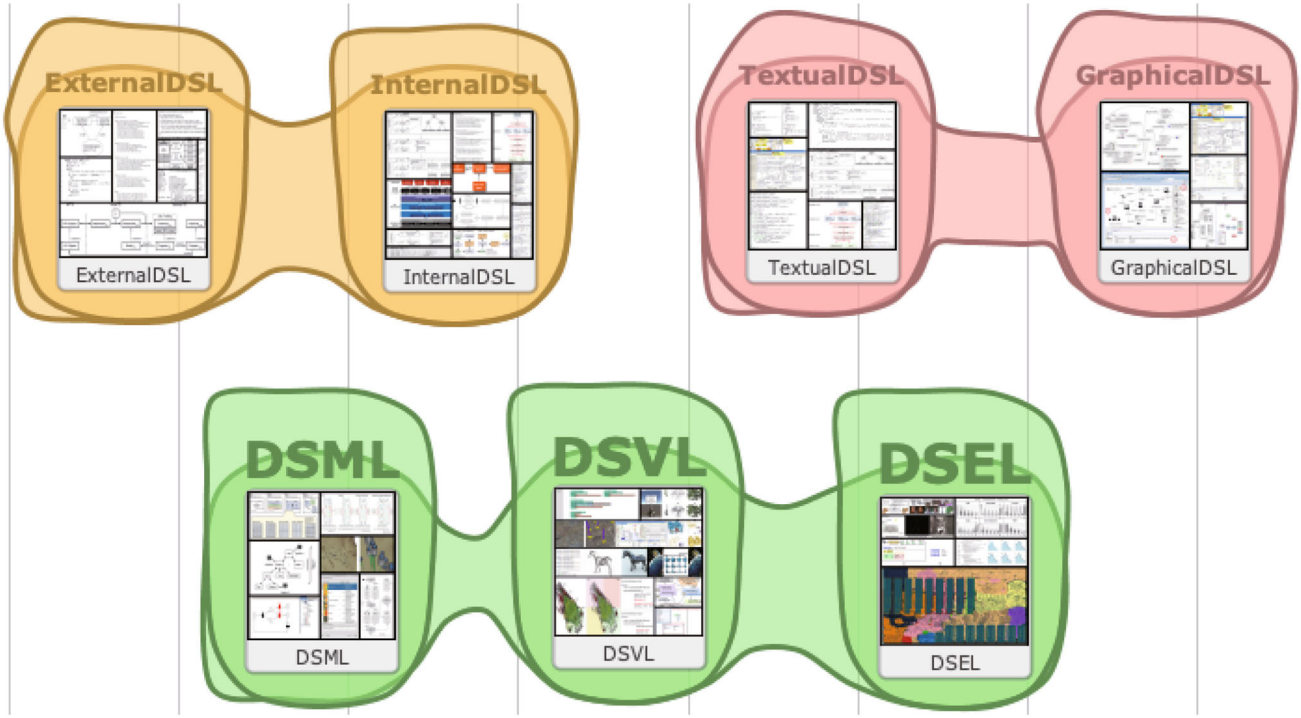


Fig. 1 An overview of three major categories of the survey visualized by bubbles [3]: External or Internal DSL, Textual or Graphical DSL and DS"X"L

or more abstract mathematical models to generate output, and (3) visualizing or otherwise post-processing the output (for example, using statistical tests).

Besides, high-performance FPGA programming is usually the exclusive domain of a few professional hardware developers. A review exploring the mapping of DSLs to high-performance FGPA [2] argues that DSLs allow programmers to describe computing in a way that is natural to the domain. This description hides the scope and variety of parallelism in the computation from the programmer, while still allowing the DSL compiler to leverage domain-specific concurrency for spatial acceleration. They divided the different DSLs for FPGA design into three categories: application-based knowledge, knowledge about a particular computing model, and FPGA implementation-specific knowledge. These three classes represent the gradual reduction of the level of abstraction from high-level application-oriented concepts to low-level bit operations.

In addition, in the field of machine learning, which is a technique for processing data and reasoning about it, Portugal et al. [5] summarize DSLs for machine learning, giving some examples, and introducing the classifications used in the survey.

1.2 Taxonomy of the Survey

Although DSLs have been widely used in a wide variety of scientific fields, the survey of DSLs development methods is very limited. We summarized the existing DSL work and found that many of them had common characteristics, so we categorized the survey based on the commonalities. For example, there are External DSLs (i.e., independent DSLs) and Internal DSLs (i.e., embedded DSLs) based on whether DSLs depend on other host languages. Besides, with respect to different symbols of the DSLs, there are Textual DSLs and Graphical DSLs. In addition, DSLs can be divided into DSVL (Domain-Specific Visual Language), DSML (Domain-Specific Model Language) and DSEL (Domain-Specific Embedded Language) according to the function and independence of DSL. These categories intersect and are not independent of each other. To keep readers interested in both related work and research trends in the field, we have improved the features of Bubble-Sets [3], an interactive tool that allows users to explore the literature and its relationships in Focus+Context techniques. As shown in Fig. 1, bubbles of different colors represent three categories, and small bubbles represent subcategories. Horizontal and vertical axes can be encoded by users into different information, such as year and number of references. In addition, Fig. 2a, b use bubbles to show individual work and subcategories respectively. The color of the bubbles represents different

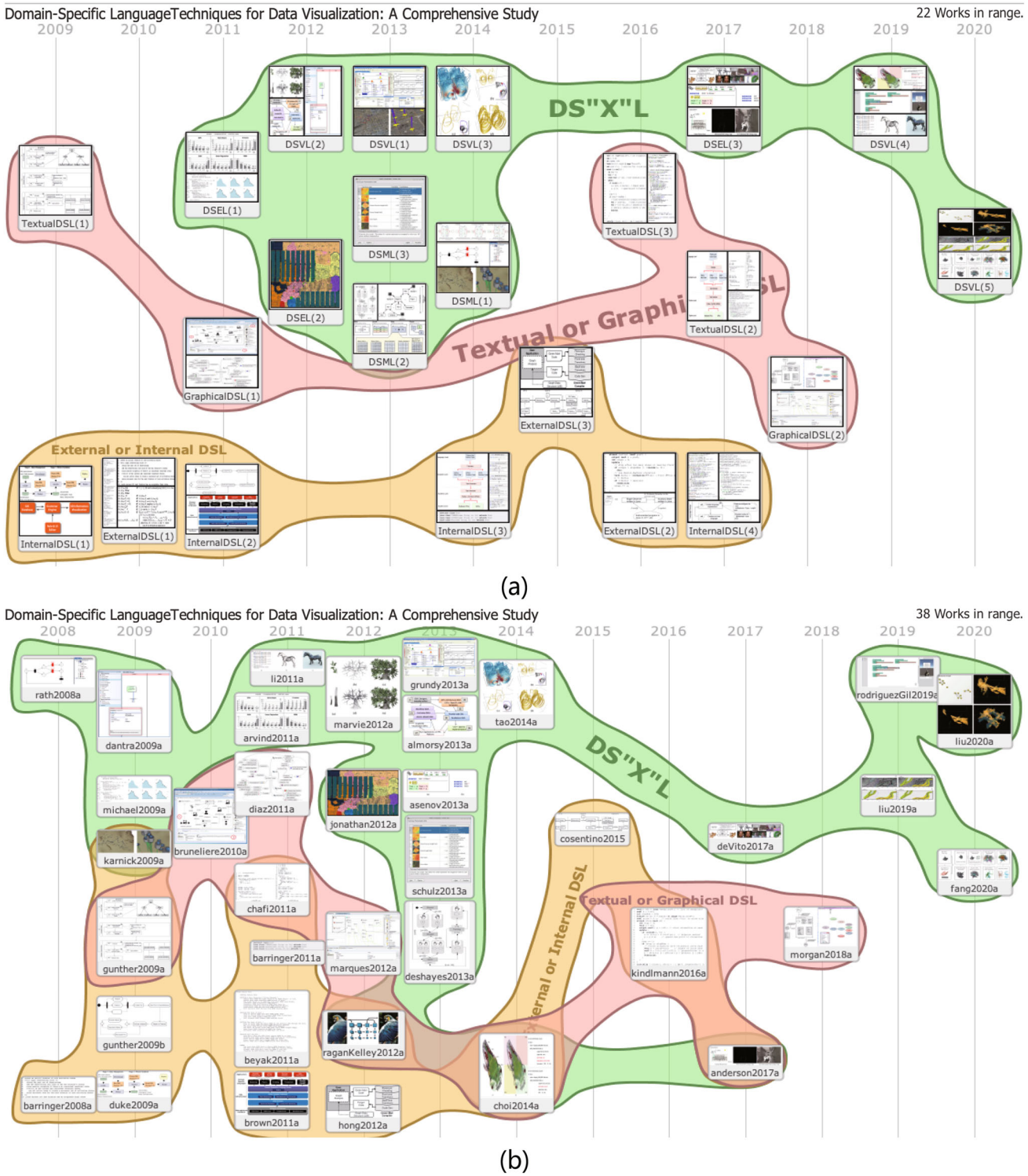


Fig. 2 An overview using BubbleSets [3] to show subcategories of each of the three main categories (**a**), and selected papers (**b**), respectively. Colors of the bubbles represent different categories or technologies they used, and each node represents a subcategory (**a**) or a paper (**b**). If papers share the same technology, each node will be surrounded by bubbles in the same color. The horizontal axis represents the publication time, and the vertical axis represents reference data from ScienceDirect (www.sciencedirect.com) and Google Scholar (scholar.google.com)

technologies, and each node represents a subcategory (a) or a paper (b). If multiple approaches share the same technology, the corresponding nodes will be surrounded by bubbles in the same color, representing the same technology they use. The horizontal axis represents the publication time, and the vertical axis represents reference data from the web site of ScienceDirect (www.sciencedirect.com) and Google Scholar (scholar.google.com).

Table 1 shows the papers from all categories. It is easy to conjecture that most papers herein share several analogous properties across various categories. All the selected papers are colored and stored in columns according to the main property they presented, which is related to the characteristic that they are External or Internal ones (Ext/Int), the Textual or Graphical ones (Tex/Gra), and their exact functions like modeling or visualization (DS"X"L).

2 External or Internal DSLs

A DSL is a domain-specific language whose expressiveness is tailored to a well-defined application domain, so as to offer an effective interface for the domain experts. To mitigate the programming difficulties of the GPLs (General-purpose Programming Languages) and meanwhile maintain the precise expression in some exact domains, DSLs present a higher level of abstraction than low-level interfaces. In doing this, it trades the generality for ease of use and conciseness, while providing much more flexibility than high-level interfaces [6].

Nevertheless, DSLs are also complimented through three fundamental implementation concerns: abstract syntax, concrete syntax, and semantics, same as GPLs [7]. The abstract syntax, which could be expressed by grammars, specifies the language constructs and their relationships according to the exact domain. While the concrete syntax, realized as a mapping of the abstract syntax, maps those language constructs to a set of symbols. The domain experts then leverage those symbols to generate programs conforming to the original abstract syntax with a supportive editor serving as GUI. Eventually, the semantics provides a concrete meaning to the language constructs of the DSLs [8].

Generally-normally, DSLs could be categorized into 2 categories, named External DSLs, i.e. Standalone DSLs, and Internal DSLs, i.e. Embedded DSLs, separately. The architectural decision about using which one of them, is essentially about which of the parsing and mapping strategy between the abstract syntax and concrete syntax should be resorted to, and would eventually impact the shape of the resulting language [9].

Broadly speaking, External DSLs are completely standalone languages. Such a DSL allows total design freedom [10] while eventually requiring the realization of its own dedicated infrastructure (i.e. editors, compilers, interpreters, etc. [8]) to some degree, due to its independence. Hence its concrete syntax and semantics would be built from scratch [11], which might lead the development process to be extremely time-consuming and challenging. Whereas Internal DSLs are usually embedded in a host language (i.e. an existing GPL), generally are implemented as libraries in order to leverage the existing functionality from the GPLs [12] to some extent. For concreteness, the syntactic features from the original GPLs and corresponding language infrastructure could be reused, which might cause some kind of platform-dependence. Considering these, an appropriate host language is significant [13] to the Internal DSLs.

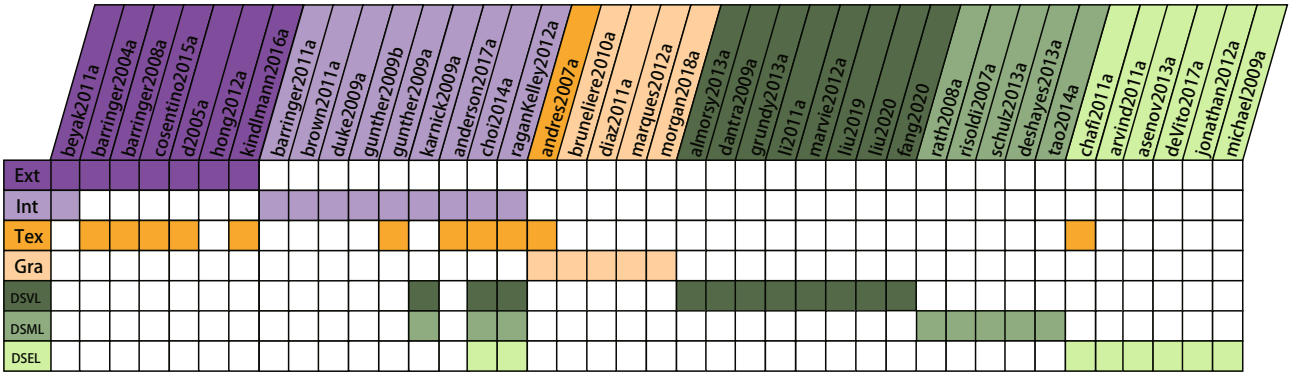
2.1 External DSLs

The development of an External DSL could be almost considered as the creation of a brand-new language [12], since it is completely independent of any existing GPLs. Mernik et al. [12] summarized patterns for the decision, analysis, design, and implementation phases of DSL development which improve and extend the early work on DSL design patterns. They also discussed domain analysis tools and language development systems that can help speed up DSL development. The developers are allowed to design the syntax and features tailored to the requirements of the exact application domain without the limitation of original GPLs, thus hiding the fine-grained programming details for ease of use [14]. The costumed facilities and environments help to ease coding, by assisting static validations and special-designed optimizations [15].

To illustrate, in the field of scientific visualization and image analysis, Kindlmann et al. present Diderot [16] to bridge the semantic gap between the mathematical concepts of the algorithms and how they are described in source code. Since many of the elementary abstractions in this domain (e.g. fields) are not directly supported by GPLs, they present a portable parallel DSL to mitigate the obscurity caused by the low-level implementation, by providing high-level mathematical programming notation to express such concepts directly in code. The code of a dense sampling example of an isocontour via Newton–Raphson iteration written by Diderot is shown in Fig. 3a. To avoid syntax and platform constraints, Diderot is not embedded into a host language. In doing this, they could better depict the field properties and express mathematical operators precisely in Unicode. It is worthy to mention that it could be compiled to a C/C++ library with an API for settings.

Also in the domain of graph analysis, to tackle the performance and implementation challenges, Hong et al. propose a DSL named Green-Marl [17], as shown in Fig. 3b. With the high-level constructs provided by it, developers are allowed to intuitively delineate their graph analysis algorithms, whose

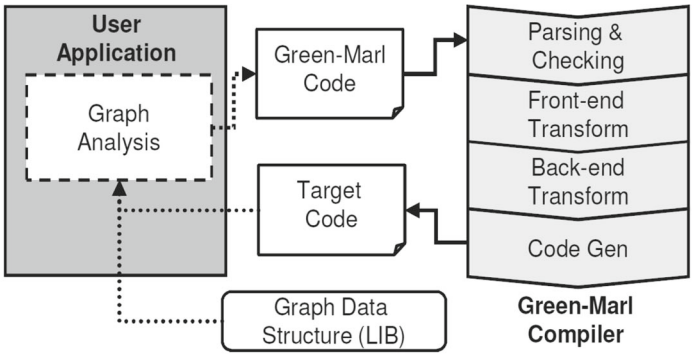
Table 1 Some representative papers selected from each category



The horizontal axis represents different papers while the vertical axis means different properties selected in the process of the design of DSLs, including that the targeted DSL is External (Ext) or Internal (Int), and the DSL is Textual (Tex) or Graphical (Gra), and the DSL is which kind of DS"X"L according to their exact function, like DSML for modeling, DSVL for visualization, and DSEL for embedded DSLs. Many papers herein share several analogous essential properties across various categories. All papers are classified (with different colors) and sorted in columns according to the major property they presented

```
1  strand isofind (vec2 pos0) {
2    output vec2 x = pos0;
3    int steps = 0;
4    update {
5      // Stop after too many steps or leaving field
6      if (steps > stepsMax || !inside(x, F))
7        die;
8      // one Newton-Raphson iteration
9      vec2 delta = -normalize(∇F(x)) * F(x)/|∇F(x)|;
10     x += delta;
11     if (|delta| < epsilon)
12       stabilize;
13     steps += 1;
14   }
15 }
```

(a)



(b)

Fig. 3 Several typical External DSLs. **a** A dense sampling example of an isocontour via Newton-Raphson iteration in Diderot [16]. The heart of it is the update function that expresses one iteration of the algorithm. **b** Overview of Green-Marl DSL-compiler Usage [17]

high-level semantics and inherent parallelism are captured by this DSL. By making use of those high-level information supported by such DSL, the corresponding compiler then translates the algorithm description written in Green-Marl into an optimized parallel implementation. Though Green-Marl is an External DSL so far, Hong et al. are considering the viability of implementing it as an Internal DSL in the future. They demonstrate that the final output of their compiler, which is written in a GPL rather than a machine language, performs as well as or even better than traditional implementations.

External DSLs are also used in the field of verification, with their own parsers. Barringer et al. develop a series of External DSLs for trace analysis, named EAGLE [20], HAWK [21], RULER [18], and LOGSCOPE [19], separately. To illustrate, the basic monitoring algorithm of [18] is shown in Fig. 4a, and the simplified log file of [19] is shown in Fig. 4c. Eventually, they share their development experiences in [22] and conclude that such products designed as External DSLs are hard to extend or change once completed. Meanwhile, they claim that the additional features proposed by users are hard to handle.

As talked above, it is widely acknowledged that the development of an External DSL is considerably tedious due to its independence on existing GPLs. This asks the developers to define its very own syntaxes and semantics [23] while designing corresponding environments. To alleviate this problem, Cosentino et al. proposed a prototype-tool named DSLit [15]. With such a tool, they could analyze the API and automatically generate a semantically equivalent External DSL with its relevant development environment. The model-driven engineering (MDE) techniques are adopted to bridge between the API and DSL technique spaces, as shown in Fig. 4b. Three DSL templates contained in this prototype are designed to address three categories of APIs, by selecting one of them the tool is able to custom the External DSL, generat-

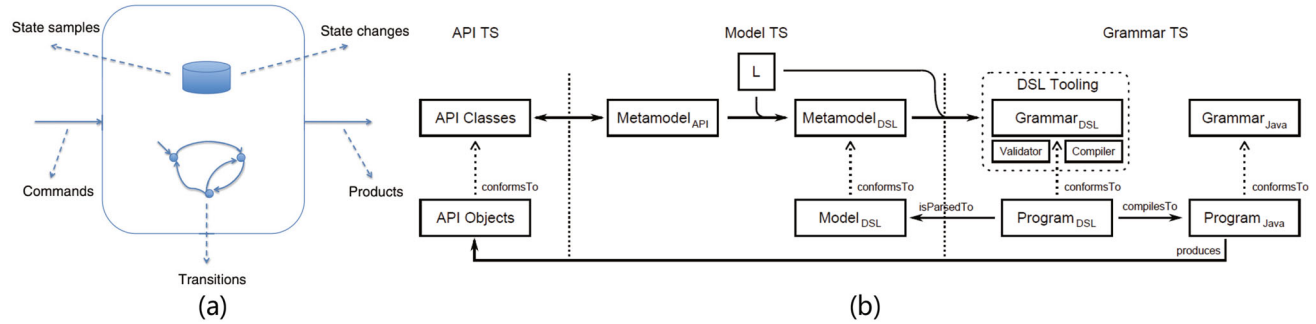


Fig. 4 Another group of typical external DSLs. **a** A basic monitoring algorithm for a trace of observation states against a set of named rules defined by a rule system [18]. **b** An overview of the linguistic architecture of DSLit [15] that spanning over three technical spaces (TS): the API TS, the Model TS, and the Grammar TS. **c** The core elements of [19], i.e. the events, are referred to in this log as COMMAND, EVR, CHANNEL, CHANGE and PRODUCT

ing the domain-specific development environment based on Xtext [24].

Furthermore, in the industrial world, to address the productivity of the teams in the video game development studio, SAGA [25] is developed by Beyak et al. as an External DSL for story management. They noticed that the whole development process requires the programmers to engage in frequently, while the communication among them and other departments is usually inefficient and failure-prone. SAGA is designed especially for domain experts in story design, so that they could "code" their ideas directly, and the programmers are allowed to focus on the places that need software design more significantly. It is worthy to mention that even though SAGA that focuses on story is an External DSL, its internal model of the generated code, which can be rendered into chosen GPL should be viewed as an Internal DSL.

Though an External DSL is completely standalone, it could be mapped to an arbitrary targeted host language or platform via generative techniques [9]. Nonetheless, we should notice that though developers are allowed to design the External DSL's features tailored to the domain-specific requirements in any possible way [9], there is a trade-off between such freedom and development cost. The design process of specifying those detailed language features with dedicated formalisms (which could express each definition precisely and correctly) could be more than cumbersome and tedious. There are also a lot of substantial efforts for the realization of related environments. All of these bring about a potential difficulty for the integration of multiple DSLs since they might hold different syntaxes and semantics. Traditionally, their extensibility might be weaker than Internal ones due to the adaption of the costumed tools [26], etc.

2.2 Internal DSLs

To avoid potential side effects discussed above, numerous works choose to embed their DSL in a GPL, i.e. a host language, so as to lower the development cost to a large degree. This is called Internal DSL, aka DSEL [27] for sometimes. The high-level abstractions of domain-specific concepts are designed, by reusing related language features as well as corresponding environments, i.e. editors, parsers, or compilers [8], etc. They seem to be more extensible since the additional features could be better handled with the help of GPLs [22].

Considering their dependency on host languages, the selection of an appropriate GPL could be significant according to the requirements of the exact domains. Traditionally, GPLs that offer more syntactical flexibility (and hence more convenient for the construction [26] of Internal DSLs) seem to be more popular as host languages. As discussed by many related works [9,22,26,27], scripting languages and dynamic languages are seemingly ideal for Internal DSLs, on account of their semantic modifiability, syntactic flexibility, and straightforward adaption (to a certain extent).

The abundance utilization of languages like Scala, Python, Ruby, etc. could illustrate the phenomenon. TraceContract presented by Barringer et al. [28] is a well-designed Internal DSL for trace analysis. To concisely and appropriately delineate the formalized properties of the trace, i.e. a sequence of events, while better trade off its extensibility with expressiveness, they choose to embed their DSL into Scala, with constructs for temporal reasoning, as shown in Fig. 5a. By reusing partial built-in notion of functions and pattern matching, their data parameterization is well handled. As a hybrid of both state machines and temporal logic, TraceContract performs well on the analysis of trace and program execution log files, thanks to the combination of object-oriented and functional programming features supported by Scala.

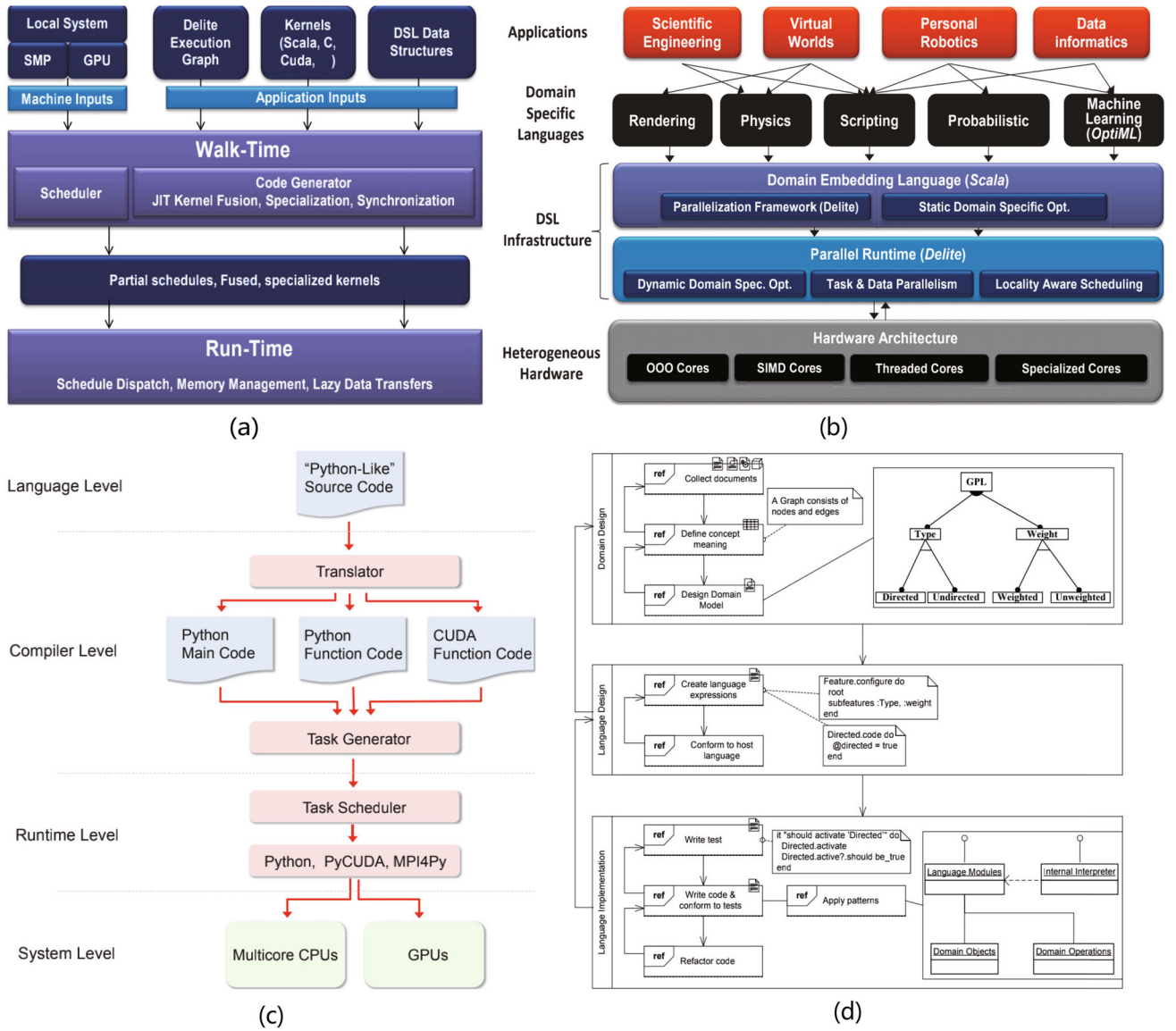


Fig. 5 Several typical internal DSL approaches. **a** The SCALA classes define the type Event of events, and three specific kinds of events in the work [28]. **b** The Language Design part of the DSL engineering process [29]. Ruby is chosen as their host language. **c** Constructing new implicitly parallel DSLs automatically target heterogeneous hardware presented by the work [10]. The first layer is a way of embedding a DSL within Scala. **d** The Delite class diagram for the Vector object in the OptiML DSL [30]. **e** The overview of Vivaldi [31]. The input source code is translated into different native functions that stored in Python main, Python functions and CUDA functions, while low-level data communication and function execution are handled by Python, CUDA and MPI runtime functions

Also based on Scala, in order to alleviate the challenge of the construction of parallelism DSLs, Delite Compiler Framework and Runtime [10] is provided by Brown et al., to offer the expertise in parallelism and hardware by helping to lift embedded DSL programs to an intermediate representation (IR). Their DSL infrastructure consists of many layers, as shown in Fig. 5c, and the approach of embedding a DSL within Scala called Lightweight Modular Staging [32] is on the first layer. As an example of DSL written in Delite, OptiML [30] is provided for the domain of machine learning. This tool inherits static type safety and type inference from Scala, obtaining support from related tooling and infrastructures (Fig. 5d).

Additionally, in the field of volume processing and visualization on distributed heterogeneous systems, Vivaldi [31] is designed to provide high-throughput performance (Fig. 5e). By taking advantage of Python, Vivaldi hides the programming details while offering intuitive abstractions. The translation from Python-style code to CUDA code is also implemented, so that the GPU acceleration can be used.

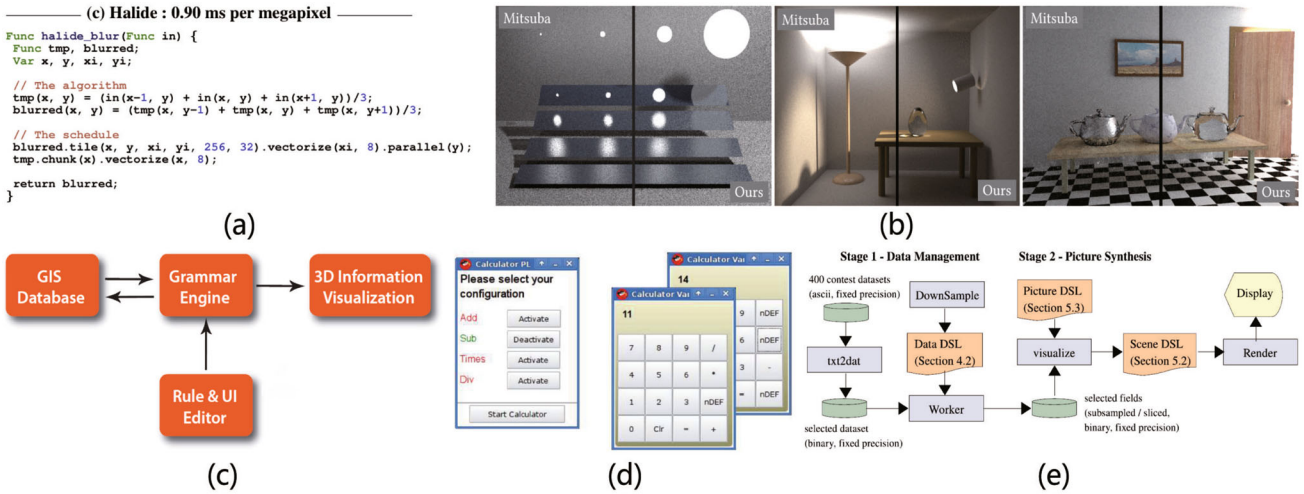


Fig. 6 Another group of typical internal DSLs. **a** A comparison of an algorithm in clean C++, optimized fast C++ without readability and portability, and Halide [33] embedded in C++. By separating the algorithm description from its schedule, the same performance could be achieved without making the same sacrifices. **b** The path tracer code illustrated in [34], which is embedded in C++. **c** The GIS Procedural Modeling System [35] whose grammar engine is implemented in C++. **d** The basic workflow of rbFeature [36] based on Ruby. **e** The system architecture shows how the DSLs that embedded in Haskell help mediating the transformation from data to rendered image [37]

In the case of Ruby, an object-oriented dynamic programming language [29,38] introduces a brand-new engineering process for Internal DSL, as shown in Fig. 5). They stress the combination of the lightweight agile development process with implementation patterns. It makes the whole process works in small iterations, supporting Language Modeling, Language Integration, and Language Purification. By making use of Ruby and related infrastructure, the generated DSL could keep fresh and dynamic, which means opening to modification and quick adaption to new environments. Such a development process has been used in the design of rbFeatures [27,36], which is implemented as an extension of Ruby (Fig. 6d).

Besides, functional programming languages, such as Haskell, seems well-suited to be a host language [22]. For example, the visualization system provided by Duke et al. [37] for a challenging astrophysics visualization problem embeds three DSLs in Haskell. The 3 DSLs are designed for large dataset management, for low-level rendering and interaction, and for high-level description of the desired picture, separately. By reusing the features of Haskell, animations with the elegant specifications can be generated, while realizing the visualization both correctly and concisely, as shown in Fig. 6e.

In addition, abundant works choose to embed their DSLs into C or C++. To illustrate, in the field of image processing, a prototype DSL named Halide [33] is presented for functional algorithm and schedule specification. In their context, schedule refers to the decisions about storage and the order of computation. The conflating of them might cause the sacrificing of readability, portability, and modularity of the code that express the computational photography algorithms. The imaging process pipeline designed with Halide embedded in C++ could separate the intrinsic algorithms from the schedule, enabling to compile straightforward expressions of the pipeline to implementations with outstanding performance across a diversity of devices, as shown in Fig. 6a. As the host language, C++ acts as the meta-programming layer offering reusable structures. Also based on C++, Aether [34] is implemented as an Internal DSL for concise and correct realization of Monte Carlo rendering. By offering primitives for sampling and probability code, the time needed for the implementation of unbiased light transport algorithms are reduced to a large degree, as shown in Fig. 6b. Considering that most renders are constructed in C++, they choose it as the host language for easy composition. Other than these, the grammar engine of [35] is also implemented in C++. A method of generating interactive visualization for GIS pipeline is proposed, by using the shape grammars specified by script-like language (Fig. 6c).

Undeniably, the adoption of Internal DSLs could also be a two-edged sword, which means that the abuse or misapplications of them could also bring out potential unfavorable effects, just like the usages of the external ones. The development of such embedded DSLs seems to be considerably easier, thanks to the reusable functionalities provided by GPLs. Their dependence on host languages seems to entail some limitations, nevertheless.

To illustrate, not only their syntax features are closely tied to the GPLs, the corresponding infrastructure like the parser,

interpreter and related tooling also relies significantly on what the host languages supported. This phenomenon might further cause platform-dependence [9] to some extent. The programming paradigm [8] and the environmental requirements of the original GPL might restrict the capabilities of an Internal DSL, and obstruct the well-performance of some optimizations and static analyses [10]. The readability might be affected, and the whole implementation might hard to be comprehended, which is obscured by the advanced features of the GPL, further making the extensions less straightforward [26]. Besides, users still might need to get an insight into a certain host language to better leverage the Internal DSL based on it.

3 Textual or Graphical DSLs

As mentioned above, the concrete syntax, one of the three fundamental implementation concerns of DSLs, maps those language constructs specified by abstract syntax to a set of symbols. These symbols would be leveraged to generate programs by domain experts [8]. Generally and normally, the symbols herein are either textual or graphical, which brings about the 2 categories.

The Textual DSLs own textual concrete syntaxes that underline the mapping between the meta-model concepts and their textual representation, that is, textual characters and symbols [29]. Those concepts are familiar to the DSL designers and domain experts with certain programming experiences, since there exist a lot of texture GPLs like Java and C. As the major choice of most DSL designers [12,39], they perform series of advantages, like high readability, convenient integration and version control, fast editing style [40,41], etc. They also promote the communication among the experts to some extent, due to the textual forms that are easy to be presented in many cases.

The Graphical DSLs, aka visual DSLs, on the contrary, provide graphical symbols, which means that they can be built up from a series of diagrammatic conventions [42]. Though they are scarce compare with Textual DSLs [12,39], there are some scenarios that Graphical DSLs would fit in with. They allow the solution to be visualized as diagrams, providing a straightforward overview, as well as facilitating communication and collaboration in a direct way, compared with Textual ones. The elements of the languages are depicted in many diagrammatic forms, like rectangles, arrows, dashed lines, etc., arranged on the visual surface for the browsing of workflows and/or dataflows.

To match particular domain problems, the two kinds of DSLs are utilized in different scenes, maximizing their own strengths. But sometimes there exist hybrid schemes which perform both textual and/or graphical viewpoints in a single system, such as systems designed with Textual DSLs for input and Graphical DSLs for output [42], or Multi-View DSL [43, 44] (Fig. 7a), etc.

3.1 Textual DSLs

Textual DSLs that emphasize textual characters and symbols are widely used for a long time considering their benefits. They are highly-readable for code construction and review, fast to edit (including search and replace operations), friendly to integration, version control [40] and even refactor, etc., which ensure their scalability and convenience. Furthermore, mature GPL and DSL editors set lots of conventions about editing styles and textual generation, like error makers, quick fixers and so on [40], which have been leveraged by people for a long time. They also facilitate efficient communication among the experts since the textual forms are easy to be spread. Besides, the edge of Textual DSLs is shown when the solution performs large-scale and sophisticated, since they could naturally handle such a situation in a concise way.

To construct a textual language, it is necessary to establish mapping relations between metamodel elements and syntactic structures of it [45]. Cook et al. [42] conclude several ways of defining a Textual DSL, including using a parser-generator (i.e. External Textual DSL), embedding the configuration code in a host language (i.e. Internal Textual DSL), taking advantage of XML, etc. The development of them, whereas, can be a tough and time-consuming task, especially for the External ones, as we mentioned in Sect. 2. That is why for the design of such DSLs, and also for the Graphical DSLs, a trade-off between the easiness of development and the low-cost of learning and using must be considered.

For example, Cosentino et al. proposed a model-driven method, named DSLit [15], to generate External Textual DSLs based on object-oriented APIs, aiming to mitigate the development cost of DSLs. As a proof-of-concept implementation, DSLit is realized as a prototype DSL generator on Eclipse platform, which is able to analyze Java APIs and generate External Textual DSLs based on Xtext framework. The generation of the DSL and the related structure could be controlled by developers by editing the API representation and design choices. Three different DSL templates are contained in distinct APIs, which allows the customization. Additionally, in the field of image processing and visualization, Diderot [16] proposed by Kindlmann et al. should be counted as one of the External Textual DSLs. It is designed to provide the domain-specific elementary abstractions in code, which could not be represented by GPLs directly (Fig. 7d). Textual symbols could help them to better propose mathematical programming notations, so as to further express related abstract concepts (e.g., fields) and their properties. In doing so users could custom algorithms on parallel hardware conveniently.

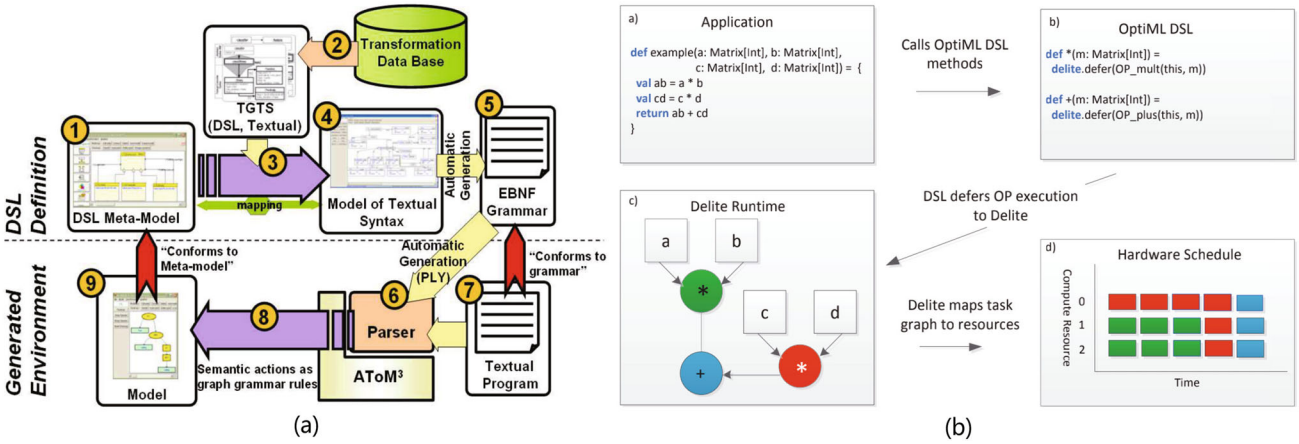


Fig. 7 Some typical approaches on Textual DSLs. **a** The definition of a multi-view DSVL [44] in the object-oriented continuous simulation domain with both textual and graphical views. **b** Side-by-side comparison of OptiML [30] (left-side) and MATLAB (right-side) code for Gaussian Discriminant Analysis (GDA). **c** A Textual DSL embedded in Ruby [29]. **d** A dense sampling code written in Diderot [16] that based on C++

On the opposite, as an illustration of Internal Textual DSL, Gunther et al. [29] introduced a novel DSL engineering approach (Fig. 7b) emphasizing both light-weighted agile development process and implementation patterns, aiming at Ruby-based Internal Textual DSL. The agile process working in small iterations, including domain statements collection and domain model definition, syntax design and iterate with test specifications (language expressions). Patterns help with productivity by applying proved solutions to similar problems, helping with Language Modeling, Language Integration and Language Purification. By reusing the existing language infrastructure, the construction and custom of DSL could be efficient and dynamic.

In addition, Halide [33] could be considered as an Internal Textual DSL based on C++. The form of textual code could help to organize the execution and storage globally, optimize the inner loops and further transform the program comprehensively. The imaging pipeline they proposed separates the algorithm from its schedule, achieving high performance without the scarification of readability and portability. Their basic expressions include constants, domain variables and calls to Halide functions, from which C++ operator overloading is used to build logical operators, etc.

Furthermore, in the field of volume processing and visualization, Vivaldi [31] is designed for heterogeneous systems, providing a highly abstracted programming environment. Their DSL has Python-like syntax, presenting functions and numerical operators for customized programming of volume rendering and processing. The basic program of it encompasses the main function that served as a driver to organize customized functions, each function call generates tasks handled parallelly. Also aiming at heterogeneous hardware for DSL, Brown et al. [10] present a heterogeneous parallel framework named Delite for DSLs. They claim that the DSL should be designed to attain both productivity and performance. This means to provide high abstraction, enable mappings among parallel implementations and abstractions, and also support static and dynamic optimizations, as they showed in the design of the Textual DSL realized by Delite: OptiML [30], which is illustrated in Fig. 7b.

The corresponding language workbench (i.e. the provided tool) which supports DSL end-programmers to create, edit and maintain programs [46] is also vital for the design of DSL. As for Textual DSL workbenches, they usually provide functions like syntax highlighting, code completion, and components like scanner, parser, etc. In particular, Pfeiffer et al. [47] make a comparison of 4 different tools support for Textual DSLs, i.e. IMP [48], MontiCore [49,50], oAW [51] and MPS [52], describe their pros and cons separately.

3.2 Graphical DSLs

Though it seems that less such works have been developed compared with Textual ones, still, Graphical DSLs have several advantages over Textual ones in some circumstances. Graphical DSLs are built up from a series of diagrammatic conventions [42]. They facilitate further engagement of the non-technical and non-IT experts [40,53], maximizing the cognitive benefits [54] of the graphical languages (rather than textual ones). The learning cost of such users is mitigated since they do not need to comprehend the logic or the declarations and semantic rules [55]. Graphical DSLs are highly abstracted, which means that they could express intuitively, offering an overview while hiding redundant low-level details, giving insights about how workflow or dataflow maps on the visual surface.

Graphical DSLs are more than diagrams, as underlined by Cook et al. [42], the designers need to further create actual models that conceptually represent the target system, together with the several different diagrams to represent their contents, since each of the diagrams delineates a single aspect of the model. Also, as summarized by [39], a successful DSL design not only depends on adopting the appropriate abstractions, but also producing as less as disturbance to existing practices. In fact, there are several aspects that need to be defined in the design of Graphical DSLs: notation, model, generation and final integration [42,56], etc.

- **Model**, i.e. domain model, is a model of the target concepts that delineated by the language graphically. It mainly consists of domain classes, representing the concepts of the problem domain which are usually mapped as shapes, and domain relations, which can be mapped as either connectors or layout among the shapes. A set of peculiarities are contained in both the domain classes and the domain relations. The definition of constraints is used to check the validity of the constructed diagrams.
- **Notation** means a set of diagram symbols used to edit and/or represent the target model, i.e. the graphical representation of domain classes and domain relations. Their basic elements are shapes and connectors laid out on a 2-D screen that are used to build up the Graphical DSLs by reusing. Decorators contained by them could offer the textual information and icons if needed.
- **Generation** means we need to generate other artifacts like data, configuration files, code together with the model built up with designed DSL, whenever the diagram is changed.

By making use of the DSL design tools could ease the whole process to some extent. But also like the Textual ones, as summarized by Cook et al. [42], building up a Graphical DSL based on existing notations and conventions could be categorized as creating an Internal DSL, which means to map the notational elements to the target domain. Similarly, creating the Graphical DSL from scratch requires the brand-new design of basic elements and more experts, which is akin to the creation of an External DSL.

For example, in the domain of digital games development, Barroca et al. [41] propose a Model-Driven Development (MDD) based Graphical DSL for Role-Playing Game (RPG) production (Fig. 8d). Though there usually exist some troubles in the development of games like the scarce of code reusing considering cross-platforms, the basic common game concepts and logic shared by a lot of RPGs could be summarized so as to realize the code reuse in a high level. According to this, they design a Graphic DSL, which could enable the game developers to model a game with different properties like characters, rules, etc., and generate the code for the target platform. Their visual DSL allows the game designers to see the big picture, grasp how the whole game process will be mapped and supports them to detect errors. This is quite different from SAGA [25] mentioned above, which is also designed for game developments in the industrial world. SAGA mainly focuses on story management as a Textual DSL.

Similarly, it meets the needs of the industrial world, in the domain of cloud calculation, Silva et al. [57] introduce a Cloud DSL. It describes could platform entities in the form of graphics presented by diagrams, so as to improve cloud portability. Cloud portability means the ability to mitigate an asset deployed in one cloud to another [59]. Considering that different cloud platforms that offer similar services could own various names, characteristics and functions, it is significant to describe the semantics of cloud entities in a mapping way so as to facilitate smooth migration of applications across distinct platforms. Their DSL makes use of a common cloud vocabulary covering a wide variety of cloud IaaS services, supporting the description of different types of clouds. More importantly, as shown in Fig. 8b, the graphical editor provided by their Graphical DSL not only boost the visualization of cloud entities, but also benefit efficient communication of the cloud strategy among different stakeholders. The property of diagramed presentation is vitally different from Cloud# [60], a Textual DSL, focusing on cloud computing services.

In wiki construction, Diaz et al. [39] proposed a "wiki scaffolding", which is specified by DSL, for the initialization of corporate wikis, as illustrated in Fig. 8a. Wikis are defined as "the simplest online database that could possibly work" [61] as a popular knowledge collaboration method [62], and are more and more adopted among companies [63]. Wiki deployment is different when targeting at various organizations due to their peculiarities. For corporate wikis, such as their users, roles, documents, permissions and project milestones are already exist before the wiki creation, which would frame both wiki users and editing. Considering these, they introduce "wiki scaffolding" for company wikis installation, where some categories and permissions are initialized to mimic the corporate background. In doing this, they could delineate the contextual setting for wikis deployed in an existing organization. The Graphical DSL they introduced for "wiki scaffolding" specification, i.e., Wiki Scaffolding Language (WSL), is built upon FreeMind [64], an open-source tool for mindmaps creation. Such a DSL could abstract from the technicalities that go in setting those parameters down to wiki code [39], and further promote wiki deployment and boost user participation. More importantly, like the Cloud DSL [57], the form of Graphical DSL could benefit collaboration and sharing since its expressions are denoted as mindmaps, which are commonly used in recording and

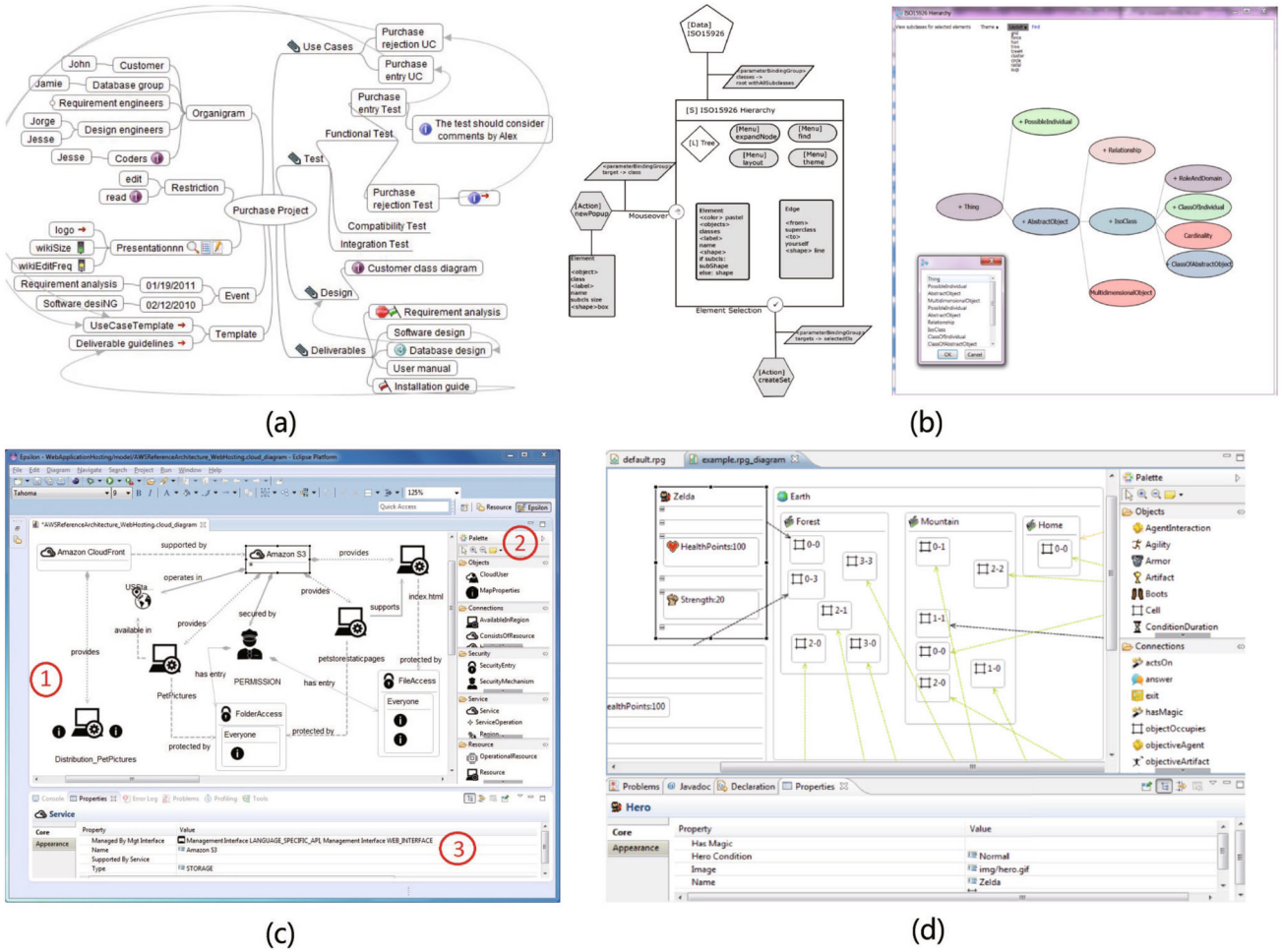


Fig. 8 Some typical methods on Graphical DSLs. **a** An example for the Purchase Project as a WSL mindmap [39]. **b** The Cloud DSL [57] that consists of 3 parts: a canvas that presents cloud entities and their relationships by graphical components, a palette that presents cloud meta-model entities, and the properties tab that shows the properties of each selected entity. **c** VizDSL [58] that models and implements an interactive visualization of the ISO 15926 hierarchy. The View Container of this visualization contains Elements on the classes collection and Edges from each Element to its superclass. **d** An example of the graphical editor in [41]. The entities of the RPG model and the relations amongst them are presented in the main window, and the attributes of those RPG entities are assigned in the properties window below

expressing, encouraging the participation and communication among domain experts.

Also emphasizing easy-to-use and easy-to-comprehend, in the domain of modern operational systems, Morgan et al. [58] propose VizDSL (Fig. 8c) for interactive information visualization. Nowadays, operational systems are large-scale and heterogeneous, distributed, which leads to the challenge that the development process might cause unknows, risks and a series of underlying unforeseen issues. These could eventually result in overspend, etc. Domain experts like engineers in the energy industry that handle those problems need to grasp such complex information contained in these specifications in right semantic, which could be cumbersome and time-consuming. Though visualization could somehow alleviate this problem and help them to comprehend relations amongst those systems, customizing visualizations could still be hard since it needs IT expertise. In this situation, VizDSL is proposed based on Interaction Flow Modeling Language (IFML) as a Graphical DSL, so as to design and create highly interactive visualizations based on model-driven engineering principles. As a Graphical DSL, their tool aims to offer easy-to-use describe, model, interactive visualize and share functions for non-IT experts, so that they could conveniently explore comprehensive semantic structures. They make use of existing standards to ensure standards-based interoperability, i.e., IFML that follows MDE principles.

We need to notice that some of the Graphical DSL designs share common defects due to their intrinsic properties. To illustrate, considering the limitation of the 2-D screen, it would be very challenging to perform large-scale problems

with Graphical DSLs. The inappropriate arrangement of the layout might lead to visual clutter and further hiding the detailed information. Besides, for the developers, it is necessary to consider the different and merge of the versions, and also the interface for the operations like search, review and refactors. Anyhow, the DSL designed, no matter Textual ones or Graphical ones, should be easy to develop for the designers and easy to use for the domain experts [40].

4 DS"X"L

Most DSLs are based on the characteristics of the domain, providing glyphs customized for the application domain, which can be used in model-driven development or programming, and relying on visual elements (such as layout, colors and lines) to represent the meaning of programs. Compared with the DSLs developed for visualization, some others are designed to simplify domain-specific models. In addition, there are DSLs that can be embedded in other languages (C++, C, Java, JavaScript, etc.) to improve program performance and flexibility.

When we review the DSL related work, we found there are several techniques can be named as DS"X"L, such as DSVL (Domain-Specific Visualization Language, developed for domain-specific visualization), DSML (Domain-Specific Modeling Language, simplifying simulations of domain-specific models), DSEL (Domain-Specific Embedded Language, based on a framework or model to embed the DSL in other languages). Therefore, in this section, we divide DSL related literatures into DSVLs, DSMLs, and DSELs in the form of DS "X" L.

4.1 DSVL

DSVL refers to the DSLs that performed as concise and useful tools that allow rapid development of applications and/or structures within a specific domain, with the advantages of simplicity and high-level abstractions. By taking advantage of DSVL meta-tools, developers can create new DSVLs based on the provided templates and graphical representations. Thus, DSVL combines the flexibility of programming languages with the simplicity of graphical interfaces to help users without much programming experience use high-level representations.

The similarity measurement of the integral curve is one of the important tasks of visualization. FlowString [66] is proposed as a new method of extracting shape invariant features from streamlines based on the string. In this method, the streamlines are resampled according to the local characteristic scale of the streamlines, and then the resampling points are classified according to the shape similarity of the local adjacent areas of the streamlines. It also encodes streamlines into a selected string of shape characters to construct meaningful words to query and retrieve. In addition, the method captures the intrinsic streamline similarity that would not change under translation, rotation, and scaling, and uses the suffix tree to efficiently search for streamline patterns of arbitrary length. The intuitive interface and user interaction design support flexible queries that can be executed at the character or word level as shown in Fig. 9b. Also, customizing character or word searches are supported and precise and approximate searches are allowed.

In order to handle the challenges of global geometric design of surfaces, Li et al. [67] also introduced the concept of shape grammar, similar to the work of FlowString [66]. They improved the field-guided shape syntax for plane and three-dimensional object model geometry or organic patterns, the core of which is the use of fields in all aspects of the shape as shown in Fig. 9c. Their systems can also use scalar, vector, or higher-order tensor fields with orders greater than two. They incorporate vector fields and tensor fields into the grammar and use collision detection and shape merging to modify the behavior of rules.

GPU Shape Grammars support interactive program generation, adjustment, and large-scale environment element visualization. Marvie et al. [68] proposed a method based on rule interpreter of expression instantiation and the parallel grammar extension based on segment for real-time process modeling and rendering of complex environment elements to dynamically generate geometric shapes in graphics hardware. In their work, the generated model is streamed through a graphical pipeline to avoid storing overly detailed models. Figure 9d shows the tree model rendered by this work [68]. A new approach [65] to creating web-based intelligent tutors and conversational agents (CAs) relies on a simple DSL-based on the Google Blockly library, to create visual block programming languages, and create domain-independent agents that can be integrated into external platforms. The main view of the authoring tool is shown in Fig. 9a and agent authors can control exactly what the bot says and when. What is new in this approach is the use of a non-programmer oriented visual DSL to define proxy behavior.

As the need for high-throughput processing and visualization of high-volume data becomes more urgent, Vivaldi [31], a DSL that supports execution on distributed GPU architectures, can be used for volume processing and visualization on distributed heterogeneous computing systems. The Python-like grammar and parallel processing abstraction of Vivaldi provide common functions and numeric operators for customized visual and high-throughput image processing applications, making it easy for users to write customized and high-quality volume rendering and processing applications. A downsampled zebrafish dataset is rendered by Vivaldi as shown in Fig. 10a. In engineering fields, there are also some works proposed. For example, Liu et al. [72] propose a series

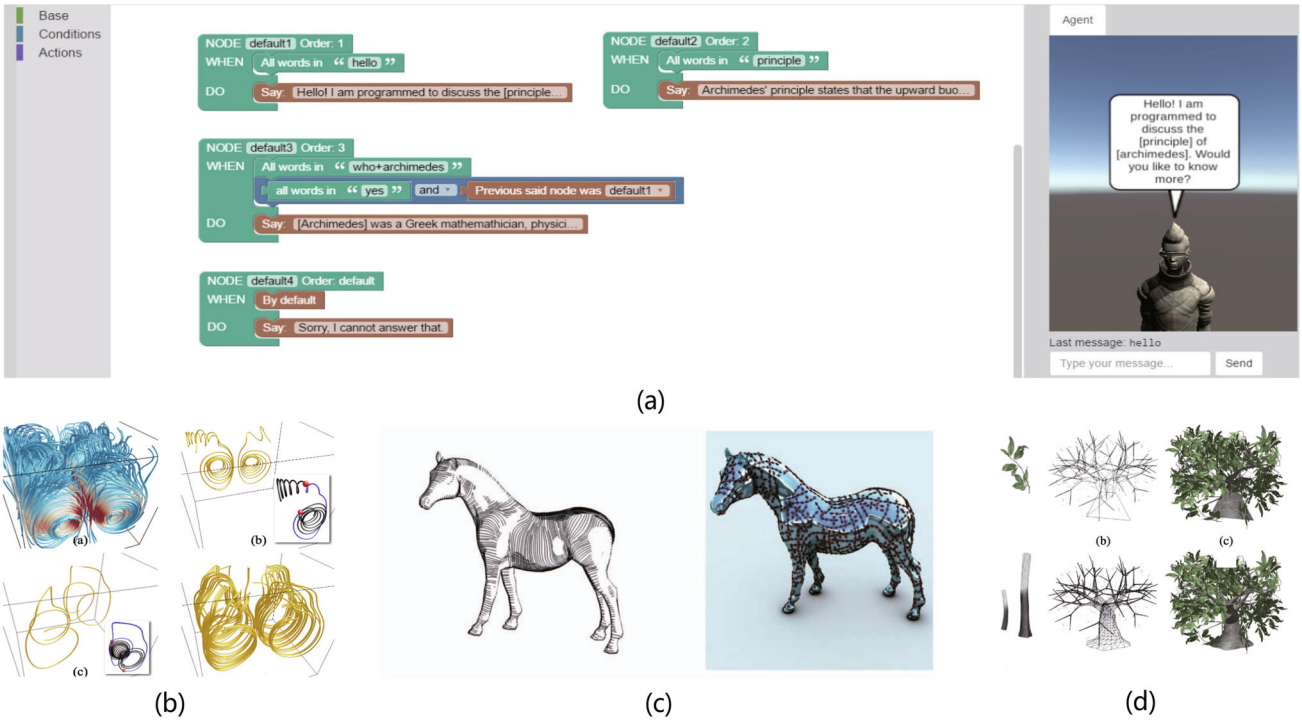


Fig. 9 Several typical DSVLs. **a** The main view of the authoring tool, the agent authors can control exactly what the bot says and when [65]. **b** A case study of FlowString [66] for the two swirls data set which shows all streamlines and the query result. **c** A shape grammar that is guided with a user-designed tensor field and the final result is shown [67]. **d** The tree model rendered by GPU shape grammars [68]. **e** The end-user interface of the report writer [69]

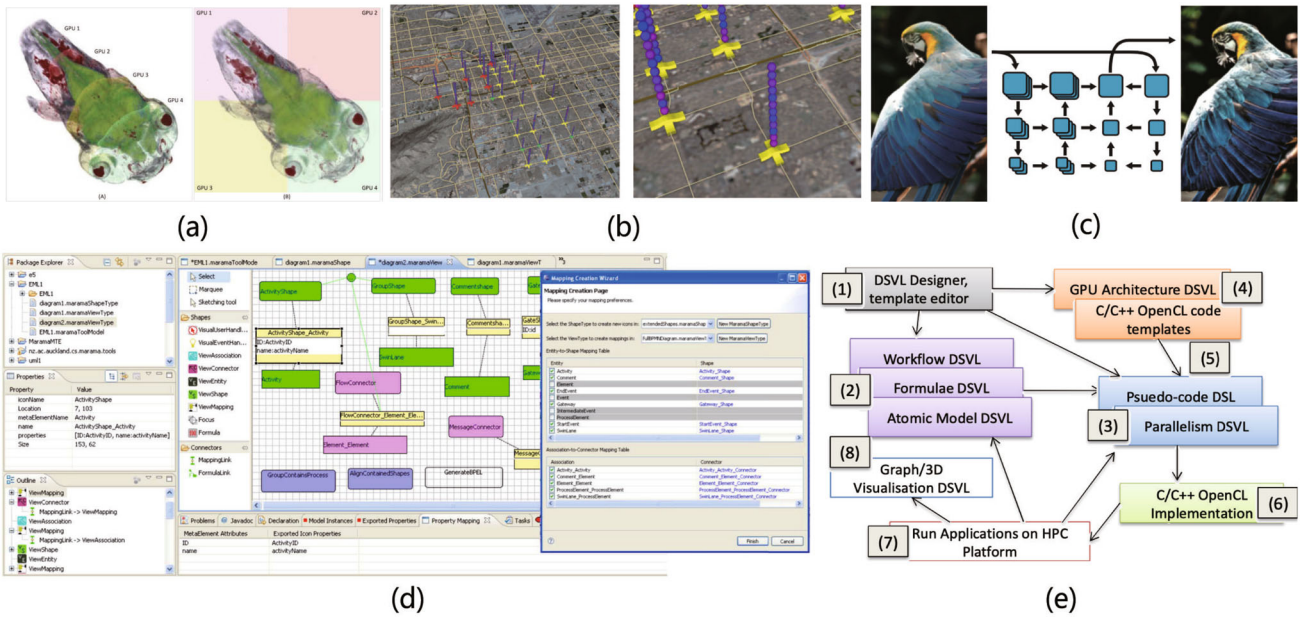


Fig. 10 Some typical DSVLs. **a** A downsampled zebrafish dataset is rendered using Vivaldi [31]. **b** A model-driven scientific and applied engineering method based on DSVLs [70]. **c** The effect of enhancing local contrast [33]. **d** Two examples of DSVL from MaramaEML [71]. **e** An example of integrating static and dynamic data for traffic analysis [35]

of domain-specific visualization systems [73–75] to explore both seismic slice and seismic volume data. Almorsy et al. [70] developed a scientific application designer designed to enable scientists to be deeply involved in writing low-level source code. This work is based on DSVLs, which allow users to define a variety of DSVLs for domain-specific modeling at different levels of abstraction, and then use these DSVLs to develop applications that allow scientists to describe the workflows, processes, entities, formulas, calculations, and final implementation code of different computing platforms, as shown in Fig. 10b. DSVL can not only be used in the scientific field, but also has a demand in the industry. Enterprise tasks and processes are often complex and require extensive end-user training and support. A report writer based on DSVL [69] can add the text report writing language for companies to help quickly design and implement reports for enterprise systems and databases in the print and graphic arts industries. This is done by creating a meta-modeling framework that helps developers modify and add new elements to the report language. The meta-model is presented to the reporting staff in a visual language. Figure 9e shows the end-user interface. To avoid conflating algorithms defined by computation with decisions about the order of storage and computation, Ragan-Kelly et al. [33] proposed a feedforward imaging pipeline representation method that separates algorithms from scheduling, including tiling, fusion, recalculation and storage, vectorization and parallel selection The effect of enhancing local contrast is shown in Fig. 10c.

Karnick et al. [35] proposed GIS shape syntax based on GIS glyph visualization, which can model multiple aspects of 2D or 3D visualization, including initialization, shape attributes, glyph spatial location, glyph separation, representation of instances, and interaction. They begin by defining advanced syntax to generate a visualization of a scenario consisting of one or more independent data sets called layers. The user then specifies the import of the visual data, and sets the grammar rules and then creates and locates the data symbols based on the geographical location, and finally the grammar engine processes the rules and generates the visualization. In addition, the actual geometry and appearance definitions are implemented by rules attached to the layers as child elements. Fig. 10e shows an example of integrating static and dynamic data for traffic analysis and the spheres at each intersection represent the number of traffic accidents per month. DSVL has meta-models and visual notations that allow domain users to represent complex models in one or more visual forms. A visual language approach that supports DSVL definitions can be an active approach to the design and construction of domain-specific modeling environments [71]. The approach to generating DSVL tools from various high-level visualization specifications in a meta-library is called Marama, with the core being a set of visual, declarative specifications for domain metamodels, visual representations and views. Two examples of DSVL from MaramaEML are shown in Fig. 10d.

4.2 DSML

DSML was proposed by Deshayes et al. [77] to reduce the accidental complexity of existing tools and languages to an acceptable level. This tool facilitates the development of interactive applications by providing DSML-based executable modeling tools. It allows the creation of behavioral models of 3D objects in a specific modeling environment, as well as the dynamic execution of models to model gesture interactions. In addition, the DSML they proposed provides concrete visual syntax that includes domain-specific elements such as the body parts involved in the interaction, the type of gesture (orientation, duration, shape), and the state of the modeled object. The model of the interaction with a virtual book using the DSMLs is shown in Fig. 11d. When a Domain-Specific Model (DSM) is projected into an existing simulation tool, the simulation results are provided at the mathematical model level rather than directly at the DSM level, which is a limitation for domain experts. Rath et al. [76] focused on DSL-based dynamic semantics for efficient simulation directly in a specific modeling environment and proposed a generic discrete event simulation framework for describing the system behavior of DSVL. The language allows complex model changes to be made at each simulation step, and the dynamic semantics are captured by the model transformation language of VIATRA2 [80], which combines abstract state machines and graph transformations into a rules-based specification formalism to express arbitrarily complex model changes. As shown in Fig. 11a, the Petri net(formally, Place/Transition nets with inhibitor arcs) simulation is displayed in the concrete syntax model (diagram). As in so many other specific domains, user interface modeling in the control systems domain has its own set of requirements and challenges that are sometimes difficult to solve through standard general-purpose modeling languages. Risoldi et al. [78] proposed a DSL for user interface modeling of complex control systems, called Cospel (Control system specification language), which focuses more on formalizing the structure and behavior of the system. The language is designed by the model-driven architecture approach [81], defined by the metamodel, and the semantics are given by converting the metamodel into a CO-OPN form. In addition, by defining metamodel based transformations, users can convert DSLs represented by the metamodel into other languages. This language allows domain experts and software engineers to more easily specify, validate and simulate activities. An example(drink vending machine) of hierarchical control system is shown in Fig. 12b.

A representation of a feedforward imaging pipeline [33] can separate the algorithm from its schedule and simplify

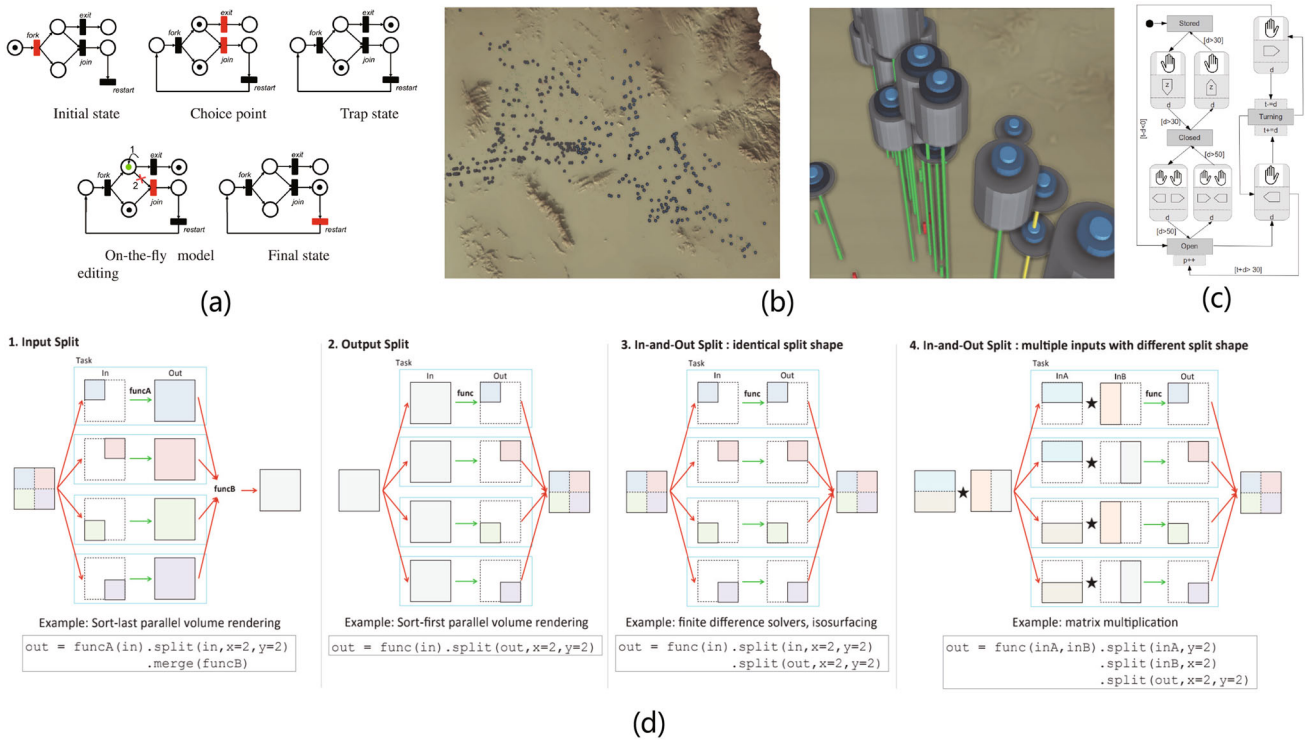


Fig. 11 Some typical methods on DSMLs. **a** The Petri net(formally, Place/Transition nets with inhibitor arcs) simulation is displayed in the concrete syntax model (diagram) [76]. **b** The overview of the well and a close-up 3D view of the well glyphs [35]. **c** The various parallel processing models can be implemented in a single line of Vivaldi code [31]. **d** A model of the interaction with a virtual book using DSMLs [77]

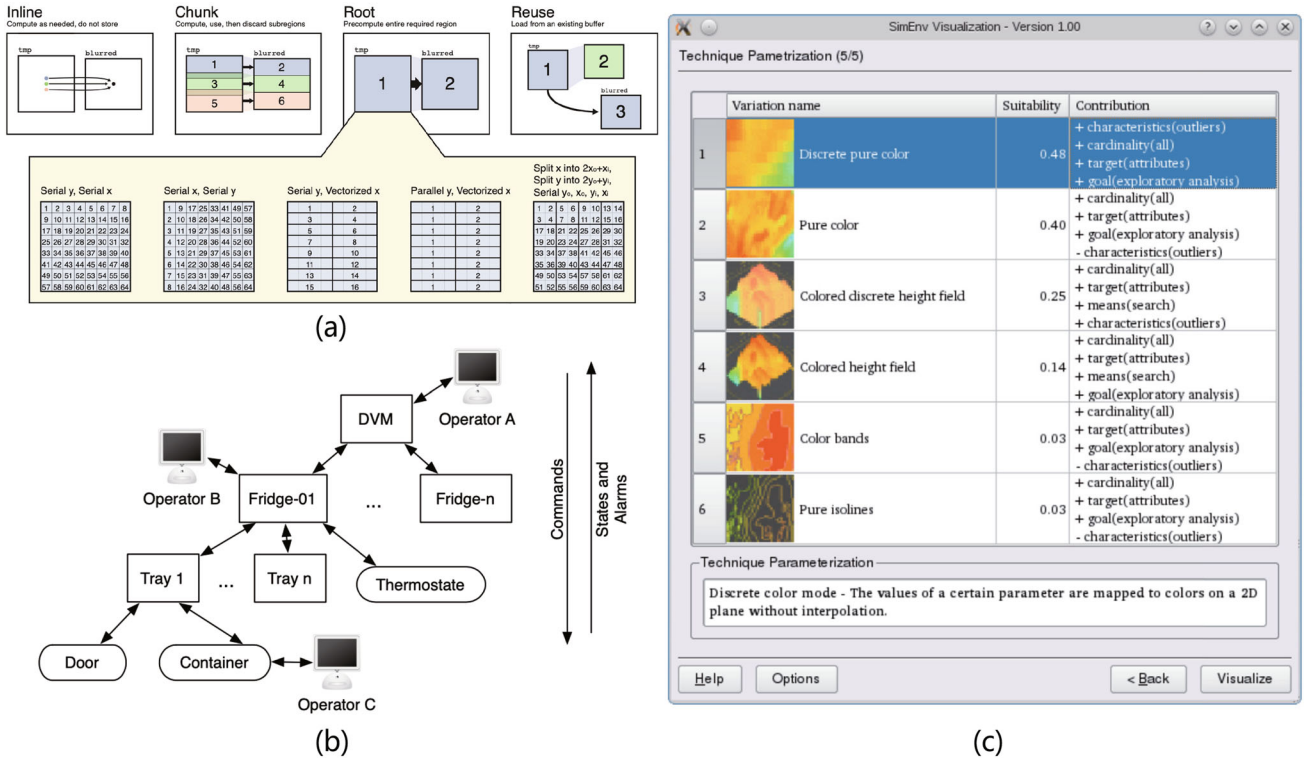


Fig. 12 Other typical methods on DSMLs. **a** The scheduling of the imaging pipeline is modeled as a set of choices that each phase must make about how to evaluate each input [33]. **b** An example(drink vending machine) of hierarchical control system [78]. **c** An example of selecting appropriate visualization techniques for climate model output data [79]

the algorithm specification. The image and the intermediate buffer become functions over an infinite integer domain without explicit storage or boundary conditions. In addition, the imaging pipeline is a combination of functions. This functional language provides a natural model for separating "what", "when", and "where", allowing for the omission of boundary conditions so that the image works in the domain of infinite integers. The scheduling of the imaging pipeline is modeled as a set of choices that each phase must make about how to evaluate each input, as shown in Fig. 12a. Compared with the method which consists of three dimensions ("what", "when", and "where") proposed by Ragan-Kelley et al., the design space of the visual task provided by Schulz et al. [79], is made up of five design dimensions related to "5w" such as "why", "what", "where", "who", "when" and "how". This approach derives an abstract design space that brings together different aspects of existing task taxonomies and models to make visual tasks clear and assess the applicability and compatibility of individual task taxonomies in a given situation. As shown in Fig. 12c, users can rank visual results based on calculated suitability scores and display the most influential design requirements, ranked by their contribution to the final score. In addition, this abstract design space can be instantiated on its own to be used in concrete use cases.

Shape grammar has been used in the construction and analysis of architectural structure design, plant modeling and other aspects. To quickly model the GIS visualization based on glyph, Karnick et al. [35] proposed a solution based on procedural modeling, for 2D and 3D visualization modeling. This method uses the shape grammar to place the glyph in appropriate location of the geographic space, and adds additional functionality for syntax in the context of geospatial visualization, including visual geometry content, the method of resolving conflict in the layout and interaction methods. The overview of the well and a close-up 3D view of the well glyphs are shown in Fig. 11b. Similarly, Choi et al. [31] developed Vivaldi, a DSL for volume processing and visualization of distributed heterogeneous computing systems, with the help of shape grammar. Vivaldi provides high-level programming language abstractions for parallel processing models commonly used in visualization and scientific computing, and supports a variety of parallel processing strategies that map well to common visualization and scientific computing pipelines. The various parallel processing models can be implemented in a single line of Vivaldi code as shown in Fig. 11c. Users can easily leverage the computing power of the most advanced distributed systems to perform visualization and computing tasks without requiring much knowledge of parallel programming.

A proper modeling language is one of the keys to a Model-Driven Development (MDD) approach. Since UML is considered as the standard language for software modeling, many MDD methods integrate the modeling requirements into UML to use UML as a DSML. Giachetti et al. [82] integrate the DSML into UML by automatically generating UML profile to promote the use of UML in the context of MDD, and provides a solution that DSML meta model is the input to generate the UML profile automatically. DSMLs incorporate concepts that represent domain-level knowledge. As a result, the systems analyst does not have to reconstruct these concepts from scratch [83]. At the same time, DSMLs help improve model quality, such as model integrity, because the DSML contains constraints that would otherwise have to be added manually.

4.3 DSEL

In 1994, an excellent technique for specifying and supporting DSL came to be called "Domain-Specific Embedded Language", or DSEL [89]. The technique which tended to produce inefficient interpreted implementations, is based on the idea of extending the "host" programming language with domain-specific libraries. DSELs leverage the host language's tooling infrastructure, but also limit the host language's syntax and IDE support. This restriction negatively affects the productivity of programmers by preventing them from using convenient specialized symbols. Asenov et al. [85] came up with the concept of a structured code editor that allows DSELs developers to customize how DSEL code is rendered and the interactions are available. They proposed Envision which was the prototype of a structured code editor for the .NET Code Contracts library. Because notations are sometimes inconvenient as a result that they need to conform to the rules of the host language, a listener for expression modifications is added to solve the problem as shown in Fig. 13b. This editor allows API designers to easily customize the look and feel of the DSELs they provide, separating the storage format from the visualization and interaction, and selecting the visualization based on the context. In addition, it allows for custom interactions and makes each visualization interactive, making it easy to create simple customizations, facilitate composition, and support advanced customizations.

Unlike structured code editors that allow developers to customize the work of DSELs, many other work embed DSLs directly into other languages. Aether [34], an embedded domain-specific language for Monte Carlo integration. Since most renderers are written in C++ or C, Aether is embedded in C++ to facilitate composition. It provides some basic methods for writing concise, structure-correct sampling and probability code. It helps ease the task of programmers, such as deriving and implementing probability density functions, performing explicit metric transformations, and handling bookkeeping and combinations of different sampling strategies. Aether requires the user to write sampling codes when deriving the corresponding PDF, and the compiler automat-

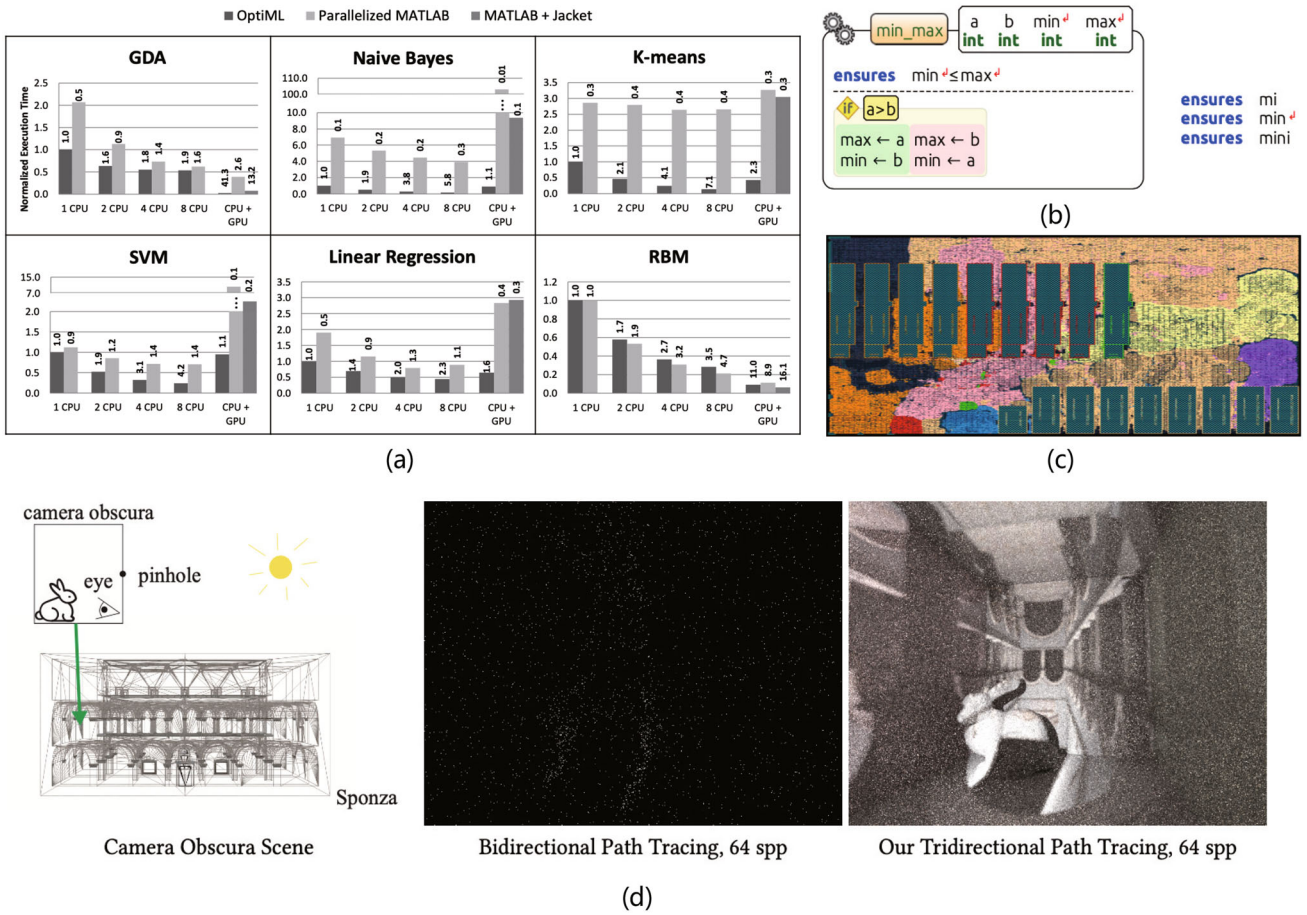


Fig. 13 Some typical approaches on DSELs. **a** The execution time of OptiML [84] applications compared to MATLAB. **b** A listener for expression modifications is added to solve the problem that notations are sometimes inconvenient. [85]. **c** The data-parallel processor layout results. [86]. **b** A scene based on Aether rendered using a challenging traditional Monte Carlo method [34]

ically generates the code needed to evaluate the PDF. The programmer needs to specify an integrand (for example, surface area) in the parameterization of a single selection and the compiler will automatically consider the metric change. Finally, it can calculate conditional probabilities based on the requirements of the Metropolis-Hastings algorithm. A scene based on Aether rendered using a challenging traditional Monte Carlo method is shown in Fig. 13d. Ragan-Kelley et al. [33] demonstrated the power of this representation by expressing a series of recent image-processing applications in a DSEL which is called Halide and is embedded in C++, and compiling them into ARM, x86, and GPU. This algorithm can effectively process the original data pipeline, bilateral grid, fast local Laplace filter and image segmentation.

High-level charts and low-level vector diagrams are at opposite ends, and their interfaces are quite different from the underlying model. There is still a gap between low-level graphical systems and high-level visual systems, and the abstractions used by visual systems may be unfamiliar to the designer. Michael et al. [87] proposed Protovis, a JavaScript-based embedded domain-specific language that allows for lower levels of control over design by combining simple graphical tags such as bars, lines, and labels to build visualizations. Protovis makes interactive visualization more accessible to web and interaction designers, who can define the visualization as a hierarchy of tags, and the visualization properties as functions of the data to focus on the composition of the data representing graphic elements using concise and easy-to-learn notations. The various example applications are shown in Fig. 14a. Cache consistency protocols are key components in many parallel and distributed systems. Teapot proposed by Chandra et al. [90] is used to write cache consistency protocols, and because Teapot is written specifically for consistency protocols, the protocol programmer needs to consider the logical structure of the protocol. In addition, it facilitates automatic protocol validation, making it difficult to find protocol errors (such as deadlocks) that can

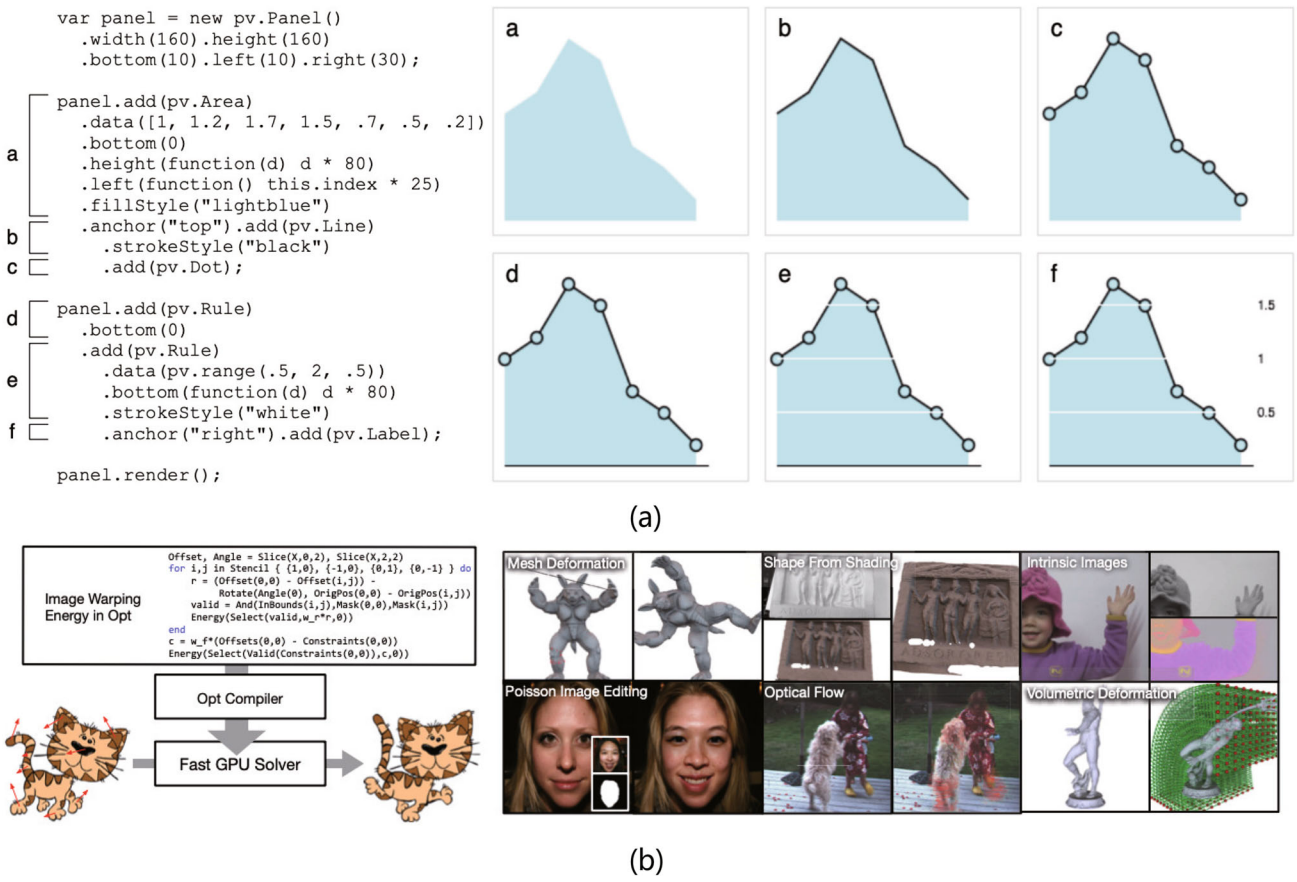


Fig. 14 Some typical approaches on DSELs. **a** The various example applications in the work [87]. **b** A high-performance GPU-based optimizer for many graphical problems [88]

be detected and fixed during actual execution. Opt, proposed by DeVito et al. [88], is used to write objective functions represented by many graphical and visual problems in visual data such as images and grids. The language can generate different solver variants, so users can easily trade off between numerical precision, matrix-free methods, and solver methods. Opt is embedded in the Lua programming language and uses operator overloading to create symbolic representations of energy, thus providing a high-performance GPU-based optimizer for many graphical problems as shown in Fig. 14b.

High-performance FPGA programming is usually the exclusive domain of a few professional hardware developers. DSLs can limit the set of programming constructs to minimize programmer errors, while also supporting a rich set of domain-specific optimizations and program transformations. Chisel [86], an embedded DSL supporting hardware construction, has powerful abstractions that make it easy to reuse components and take advantage of the features of the Scala language to provide more layers of DSLs. It supports the use of highly parameterized generators and the hierarchical domain-specific hardware to advanced hardware design language. In addition, Chisel can be precisely specified low-level hardware pieces and also be easily extended to capture many useful advanced hardware design patterns. The data-parallel processor layout results are shown in Fig. 13c. The hardware design abstraction is enhanced by providing concepts such as object orientation, functional programming, parameterized types, and type inference. When applying machine learning algorithms, it is often necessary to use multiple parallel programming models for different devices, such as CPU and GPU, to take advantage of modern hardware. OptiML [84], a DSL method for machine learning, embeds DSLs into Scala using the Delite [30] framework to perform domain-specific analysis and optimization, and automatically generates CUDA code for the GPU. In the heterogeneous system composed of multi-core CPU and GPU, the performance is better than the MATLAB code directly parallelized as shown in Fig. 13a.

A DSEL is an ultimate abstraction, allowing users to reason about the program within the domain semantics, rather than within the semantics of the programming language [27]. The advantage of DSEL is that its syntax is extensible. Users can add new constructs to the language by simply defining new functions. However, because of the rules of the host

language, extensibility is limited by the underlying representation of semantics and abstract syntax. C++ is often used as the general-purpose language for embedding DSEL. If several DSELs based on the same host language are designed for different domains, users in different domains can share a common core language and all its associated tools. Therefore, DSELs can contribute to the programming language ecosystem [91].

5 Discussion and Conclusions

DSLs are designed to alleviate the programming complexity of the GPLs for the domain experts. By making use of the simplified interfaces specially customized for an exact domain, they could construct their projects both flexibly and conveniently, even for the non-IT scientists. Techniques related to DSLs are summarized and categorized in this survey since they are either used individually or synthetically in both the academic world and the industrial world, according to their parsing and mapping strategy between the abstract syntax and concrete syntax, the mapping results, and also the exact functions they emphasize.

As far as we noticed, parsing and mapping strategy between the abstract syntax and concrete syntax of the DSLs could be depicted as External or Internal, focusing on whether the DSL is based on a host language (i.e., a GPL) or not. While the mapping results could be categorized as Textual or Graphical symbols provided to users for further utilization. Furthermore, the exact functions that the DSLs emphasize are mainly related to the targeted domains that they designed for, like the DSVL for visualization and the DSML for modeling, etc.

To our best knowledge, no survey papers summarizing the DSL-related literatures from the viewpoint of such categories. Overall, we hope that this survey inspires novel ideas on the design of DSLs in multiple domains.

Acknowledgements This work was supported by the National Nature Science Foundation of China (NSFC) Grant Nos. 61702271 and 41971343, and Postgraduate Research & Practice Innovation Program of Jiangsu Province (No. SJCX20_0445). The authors would like to thank Ziqi Sha, Sitong Fang, Shunlong Ye, Guang Yang, and Ziyu Yao for their participating in paper collections.

References

1. Nardi Bonnie A (1993) A small matter of programming: perspectives on end user computing. MIT Press, Cambridge
2. Kapre N, Bayliss S (2016) Survey of domain-specific languages for FPGA computing. In: International Conference on Field Programmable Logic and Applications (FPL), pages 1–12. IEEE
3. Collins C, Penn G, Carpendale S (2009) BubbleSets: Revealing set relations with isocontours over existing visualizations. IEEE Trans Visual Comput Graph 15(6):1009–1016
4. Jones M, Scaffidi C (2011) Obstacles and opportunities with using visual and domain-specific languages in scientific programming. In: 2011 IEEE symposium on visual languages and human-centric computing (VL/HCC). IEEE, pp 9–16
5. Portugal I, Alencar P, Cowan D (2016) A Preliminary Survey on Domain-Specific Languages for Machine Learning in Big Data. In: IEEE International Conference on Software Science, Technology and Engineering. IEEE, pp 108–110
6. Rautek Peter, Bruckner Stefan, Grller Eduard, Hadwiger Markus (2014) Vislang: a system for interpreted domain-specific languages for scientific visualization. IEEE Trans Visual Comput Graph 20(12):2388–2396
7. Harel D, Rumpe B (2004) Meaningful modeling: what's the semantics of "semantics"? Computer 37(10):64–72
8. Méndez-Acuña D, Galindo José A, Degueule T, Combemale B, Baudry B (2016) Leveraging software product lines engineering in the development of external DSLs: a systematic literature review. Comput Lang Syst Struct 46:206–235
9. Zdun U, Strembeck M (2009) Reusable architectural decisions for DSL design: foundational decisions in DSLs development. In: European Conference on Pattern Languages of Programs. CEUR-WS, pp B6(1–37)
10. Brown Kevin J , Sujeeth Arvind K , Lee Hyouk J, Rompf T, Chafi H, Odersky M, Olukotun K (2011) A heterogeneous parallel framework for domain-specific languages. In: 2011 International conference on parallel architectures and compilation techniques. IEEE, pp 89–100
11. Johanson AN, Hasselbring W (2014) Hierarchical combination of internal and external domain-specific languages for scientific computing. In: Proceedings of the 2014 European conference on software architecture workshops, pp 1–8
12. Mernik M, Heering J, Sloane AM (2005) When and how to develop domain-specific languages. ACM Comput Surv (CSUR) 37(4):316–344
13. Renggli L, Gîrba T (2009) Why smalltalk wins the host languages shootout. In: Proceedings of the international workshop on smalltalk technologies, pp 107–113
14. Jézéquel J-M, Méndez-Acuna D, Degueule T, Combemale B, Barais O (2015) When systems engineering meets software language engineering. Complex systems design & management. Springer, New York, pp 1–13
15. Cosentino V, Tisi M, Izquierdo Javier LC (2015) A model-driven approach to generate external DSLs from object-oriented apis. In: International conference on current trends in theory and practice of informatics. Springer, New York, pp 423–435
16. Kindlmann Gordon L, Chiw Charisee, Seltzer Nicholas, Samuels Lamont, Reppy John H (2016) Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. IEEE Trans Visual Comput Graph 22(1):867–876
17. Hong S, Chafi H, Sedlar E, Olukotun K (2012) Green-Marl: a DSL for easy and efficient graph analysis. In: Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, pp 349–362
18. Barringer H, Rydeheard D, Havelund K (2008) Rule systems for run-time monitoring: from eagle to ruler. J Log Comput 20(3):675–706
19. Barringer H, Groce A, Havelund K, Smith M (2010) Formal analysis of log files. J Aerospace Comput Inform Commun 7(11):365–390
20. Barringer H, Goldberg A , Havelund K, Sen K (2004) Rule-based runtime verification. In: International workshop on verification, model checking, and abstract interpretation. Springer, New York, pp 44–57

21. d'Amorim M, Havelund K (2005) Event-based runtime verification of java programs. ACM SIGSOFT Software Eng Notes 30(4):1–7
22. Barringer H, Havelund K (2011) Internal versus external DSLs for trace analysis. In: International conference on runtime verification. Springer, New York, pp 1–3
23. Stahl T, Voelter M, Czarnecki K (2006) Model-driven software development: technology, engineering, management. Wiley, Hoboken
24. Eysholdt M, Behrens H (2010) Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pp 307–309
25. Beyak L, Carette J (2011) SAGA: a DSL for story management. arXiv preprint arXiv:1109.0776
26. George L, Wider A, Scheidgen M (2012) Type-safe model transformation languages as internal DSLs in scala. In: International conference on theory and practice of model transformations. Springer, New York, pp 160–175
27. Paul H (1996) Building domain-specific embedded languages. ACM Comput Surv 28(4es):196es
28. Barringer H, Havelund K (2011) TraceContract: a scala DSL for trace analysis. In: International symposium on formal methods. Springer, New York, pp 57–72
29. Günther S (2009) Agile DSL-engineering and patterns in ruby. Otto-von-Guericke-Universität Magdeburg, Technical report (Internet) FIN-018-2009
30. Chafi H, Sujeeth AK, Brown KJ, Lee HJ, Atreya AR, Olukotun K (2011) A domain-specific approach to heterogeneous parallelism. ACM SIGPLAN Notices 46(8):35–46
31. Choi H, Choi W, Quan TM, Hildebrand D, Pfister H, Jeong WK (2014) Vivaldi: a domain-specific language for volume processing and visualization on distributed heterogeneous systems. IEEE Trans Visual Comput Graph 20(12):2407–2416
32. Rompf T, Odersky M (2010) Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In: Proceedings of the ninth international conference on Generative programming and component engineering, pp 127–136
33. Ragan-Kelley J, Adams A, Paris S, Levoy M, Amarasinghe SP, Durand F (2012) Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans Graph 31(4):32:1–32:12
34. Anderson L, Li T-M, Lehtinen J, Durand F (2017) Aether: an embedded domain specific sampling language for monte carlo rendering. ACM Trans Graph 36(4):99:1–99:16
35. Karnick P, Jeschke S, Cline D, Razdan A, Wentz E, Wonka P (2009) A shape grammar for developing glyph-based visualizations. Comput Graph Forum 28(8):2176–2188
36. Günther S, Sunkle S (2009) Feature-oriented programming with ruby. In: Proceedings of the first international workshop on feature-oriented software development, pp 11–18
37. Duke DJ, Borgo R, Wallace M, Runciman C (2009) Huge data but small programs: visualization design via multiple embedded DSLs. In: Practical aspects of declarative languages, 11th international symposium, PADL 2009, pp 31–45
38. Thomas D, Hunt A, Fowler C (2005) Programming Ruby: the pragmatic programmers' guide. Pragmatic Bookshelf, Raleigh
39. Díaz O, Puente G (2011) a DSL for corporate wiki initialization. In: International conference on advanced information systems engineering. Springer, New York, pp 237–251
40. Merkle B (2010) Textual modeling tools: overview and comparison of language workbenches. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pp 139–148
41. Marques E, Balegas V, Barroca BF, Barisic A, Amaral V (2012) The RPG DSL: a case study of language engineering using MDD for generating RPG games for mobile phones. In: Proceedings of the 2012 workshop on domain-specific modeling, pp 13–18
42. Cook S, Jones G, Kent S, Wills AC (2007) Domain-specific development with visual studio DSL tools. Pearson Education, London
43. Guerra E, De Lara J (2006) Model view management with triple graph transformation systems. In: International conference on graph transformation. Springer, New York, pp 351–366
44. Andrés FP, De Lara J, Guerra E (2007) Domain specific languages with graphical and textual views. In: International symposium on applications of graph transformations with industrial relevance. Springer, New York, pp 82–97
45. Jouault F, Bézivin J, Kurtev I (2006) TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th international conference on Generative programming and component engineering, pp 249–254
46. Fowler M (2005) Language workbenches: The killer-app for domain specific languages. In: Lambda the Ultimate
47. Pfeiffer M, Pichler J (2008) A comparison of tool support for textual domain-specific languages. In: Proceedings of the 8th OOPSLA workshop on domain-specific modeling, pp 1–7
48. Charles P, Fuhrer RM, Sutton Jr SM (2007) Imp: a meta-tooling platform for creating language-specific ides in eclipse. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering, pp 485–488
49. Grönniger H, Krahn H, Rumpe B, Schindler M, Völkel S (2008) Monticore: a framework for the development of textual domain specific languages. In: Companion of the 30th international conference on Software engineering, pp 925–926
50. Krahn H, Rumpe B, Völkel S (2014) Efficient editor generation for compositional DSLs in eclipse. arXiv preprint arXiv:1409.6625
51. Friese P, Efftinge S, Köhnlein J (2008) Build your own textual DSL with tools from the eclipse modeling project. In: Eclipse Corner Article
52. Fowler M (2005) A language workbench in action-mps. Online http://martinfowler.com/articles/mpsAgree.html
53. Moody D (2009) The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans Software Eng 35(6):756–779
54. Sendall S, Kozaczynski W (2003) Model transformation: the heart and soul of model-driven software development. IEEE Softw 20(5):42–45
55. Schlee M, Vanderdonckt J (2004) Generative programming of graphical user interfaces. In: Proceedings of the working conference on advanced visual interfaces, pp 403–406
56. Sleiman HA, Sultán AW, Frantz RZ , Corchuelo R et al (2009) Towards automatic code generation for EAI solutions using DSL tools. In: JISBD, pp 134–145
57. Silva GC, Rose LM, Calinescu R (2014) Cloud DSL: a language for supporting cloud portability by describing cloud entities. In: CloudMDE@ MoDELS, pp 36–45
58. Morgan R, Grossmann G, Schrefl M, Stumptner M, Payne T (2018) VizDSL: a visual DSL for interactive information visualization. In: International conference on advanced information systems engineering. Springer, New York, pp 440–455
59. Petcu D, Macariu G, Panica S, Craciun C (2013) Portable cloud applications from theory to practice. Future Generat Comput Syst 29(6):1417–1430
60. Liu D, Zic J (2011) Cloud#: a specification language for modeling cloud. In: 2011 IEEE 4th international conference on cloud computing. IEEE, pp 533–540
61. Cunningham W et al (2002) What is wiki. WikiWikiWeb. http://www.wiki.org/wiki.cgi
62. Raman M (2006) Wiki technology as a "free" collaborative tool within an organizational setting. Inf Syst Manag 23(4):59–66
63. Carlin D (2007) Corporate wikis go viral. Business Week Online, 12

64. Mueller J et al (2005) Freemind. http://freemind.sourceforge.net. Retrieved the 31st of December
65. Rodriguez-Gil L, Zubía JG, Orduña P, Villar-Martinez A, López-de-Ipiña D (2019) New approach for conversational agent definition by non-programmers: a visual domain-specific language. IEEE Access 7:5262–5276
66. Tao J, Wang C, Shene C-K (2014) Flowstring: partial streamline matching using shape invariant similarity measure for exploratory flow visualization. In: IEEE Pacific visualization symposium, pp 9–16
67. Li Y, Bao F, Zhang E, Kobayashi Y, Wonka P (2011) Geometry synthesis on surfaces using field-guided shape grammars. IEEE Trans Visual Comput Graph 17(2):231–243
68. Marvie JE, Buron C, Gautron P, Hirtzlin P, Sourimant G (2012) GPU shape grammars. Comput Graph. Forum 31(7–1):2087–2095
69. Dantra R, Grundy J, Hosking J (2009) A domain-specific visual language for report writing using microsoft dsl tools. In: IEEE symposium on visual languages and human-centric computing
70. Almorsy M, Grundy J, Sadus R, van Straten W, Barnes DG, Kaluza O (2013) A suite of domain-specific visual languages for scientific software application modelling. In: IEEE symposium on visual languages and human-centric computing
71. Grundy JC, Hosking J, Li KN, Ali NM, Huh J, Li RL (2013) Generating domain-specific visual language tools from abstract visual specifications. IEEE Trans Software Eng 39(4):487–515
72. Liu R, Ji G, Su M (2020) Domain-specific visualization system based on automatic multi-seed recommendations: extracting stratigraphic structures. Software: Pract Exp 50(2):98–115
73. Liu R, Guo H, Yuan X (2014) Seismic structure extraction based on multi-scale sensitivity analysis. J Visual 17(3):157–166
74. Liu R, Chen S, Ji G, Zhao B, Li Q, Su M (2018) Interactive stratigraphic structure visualization for seismic data. J Visual Lang Comput 48(2018):81–90
75. Liu R, Shen L, Chen X, Ji G, Zhao B, Tan C, Su M (2019) Sketch-based slice interpretative visualization for stratigraphic data. J Imaging Sci Technol 63(6):60505–1
76. Rath I, Vago D, Varro D (2008) Design-time simulation of domain-specific models by incremental pattern matching. In: IEEE symposium on visual languages and human-centric computing
77. Deshayes R (2013) A domain-specific modeling approach for gestural interaction. In: IEEE symposium on visual languages and human-centric computing
78. Risoldi M, Buchs D (2007) A domain specific language and methodology for control systems gui specification, verification and prototyping. In: IEEE symposium on visual languages and human-centric computing
79. Schulz HJ, Nocke T, Heitzler M, Schumann H (2013) A design space of visualization tasks. IEEE Trans Visual Comput Graph 19(12):2366–2375
80. Varró D, Balogh A (2007) The model transformation language of the VIATRA2 framework. Sci Comput Program 68(3):214–234
81. Lewis GA, Meyers BC, Wallnau K (2006) Workshop on program generation and model-driven architecture. Technical report, Carnegie Mellon University
82. Giachetti G, Marín B, Pastor O (2009) Using UML as a domain-specific modeling language: A proposal for automatic generation of uml profiles. In: International conference on advanced information systems engineering. Springer, New York, pp 110–124
83. Frank U (2010) Outline of a method for designing domain-specific modelling languages. Technical report, ICB-research report
84. Sujeeth AK , Lee HJ, Brown KJ , Rompf T, Chafi H, Wu M, Atreya AR, Odersky M, Olukotun K (2011) OptiML: an implicitly parallel domain-specific language for machine learning. In: International Conference on Machine Learning
85. Asenov D, Muller P (2013) Customizing the visualization and interaction for embedded domain-specific languages in a structured editor. In: IEEE symposium on visual languages and human-centric computing
86. Bachrach J, Vo H, Richards B, Lee Y, Waterman A, Avižienis R, Wawrzynek J, Asanović K (2012) Chisel: constructing hardware in a scala embedded language. In: DAC design automation conference 2012. IEEE, pp 1212–1221
87. Bostock M, Heer J (2009) Protovis: A graphical toolkit for visualization. IEEE Trans Visual Comput Graph 15(6):1121–1128
88. DeVito Z, Mara M, Zollhöfer M, Bernstein G, Ragan-Kelley J, Theobalt C (2017) Pat Hanrahan, Matthew Fisher, and Matthias Nießner. Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. ACM Trans Graph 36(5):171:1–171:27
89. Hudak P, Jones MP (1994) Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity. Contract 14(92-C):0153
90. Chandra S, Richards B, Larus JR (1999) Teapot: A domain-specific language for writing cache coherence protocols. IEEE Trans Software Eng 25(3):317–333
91. Hudak P (1998) Modular domain specific languages and tools. In: Proceedings of fifth international conference on software reuse (Cat. No. 98TB100203). IEEE, pp 134–142