Survey on Domain Specific Languages Implementation Aspects

Eman Negm[1], Soha Makady[2], Akram Salah[3]
Faculty of Computers and Artificial Intelligence
Cairo University, Giza, Egypt

***Abstract*—Domain Specific Languages (DSLs) bridge the gap between the business model and the technical model. DSLs allow the technical developer to write programs with the business domain notations. This leads to higher productivity and better quality than General Purpose Languages (GPLs). One of the main challenges of utilizing DSLs in the current software process is how to reduce the implementation cost and the knowledge required for building and maintaining DSLs. Language workbenches are environments that provide high level tools for implementing different language aspects. The purpose of this paper is to provide a survey on the different aspects of implementing DSLs. The survey includes structure, editor, semantics, and composability language aspects. Furthermore, it overviews the approaches used for each aspect and classify the current workbenches according to these approaches.**

***Keywords*—Domain Specific Language (DSL); language workbench; language implementation aspects; software language engineering**

I. Introduction

Domain Specific Languages (DSLs) [1] are languages that are designed and implemented to express and solve a specific class of problems. This class represents the domain of the language. SQL (Structured Query Language) is a DSL specialized in database domain, HTML (Hyper Text Markup Language) is another DSL that concerns with the web domain. Throughout the paper, *MiniIoT* DSL is used to describe the different aspects of DSL implementation, it is a fake DSL for Internet of Things (IoT) domain. Listing 1 shows a sample of *MiniIoT* code that defines sensors and actuators to send alarm and open the extinguisher in case of the temperature exceeds some threshold in a specific building.

DSLs have many advantages over General Purpose Languages (GPLs) for representing a specific domain. On one hand, it provides higher abstractions for the given domain which raises the productivity and the quality of the development process [2]–[4]. The user, who utilizes *MiniIoT*, will use concepts like devices, sensors, and actuators to write IoT programs. In case of utilizing one of the GPL like Java and C, the user is forced to use concepts that are not related to his domain like classes, fields, and arrays to represent his problem. The latter representation consumes more time and effort and may generate more errors than the direct *MiniIoT* representation. DSL also provides better validation and verification for the output programs since it utilizes domain specific constraints. The DSL developer could define constraints to verify that the generated program is meaningful. For example, *MiniIoT* will not allow to configure the same sensor in two different locations in the same time. In addition, the error messages are also more meaningful since it utilizes the domain concepts. *MiniIoT* will display error messages like "*This sensor is already located in building No. 11 in this time*". On the other hand, the use of domain notations in the DSLs allows more involvement for the domain expert in the development process. This leads to bridge the gap between the business model which is owned by the domain expert and the technical model which is owned by the programmer. As a result, The quality of the final product is enhanced [5], [6].

Listing 1: MiniIoT Code Sample

```
Define Sensor temperatureSensor measures
    Temperature located in Building 11
Define Actuator alarmActuator that send Alarm
    to number 911 located in Building 11
Define Actuator extinguisherActuator that open
     fire extinguisher located in Building 11
if ( temperatureSensor.temperature >
    TEMPERATURE_THURSHHOLD ) {
 alarmActuator.execute
 extinguisherActuator.execute
}
```

Despite the advantages mentioned above, DSL doesn't have the expected role in the current software development life cycle. There are many reasons for this, one of the main reasons is the cost of designing, implementing, and composition of a new DSL. Designing a DSL, that covers all domain aspects, is not an easy task. Implementing a new language from scratch, in case of no suitable one is available, is a very difficult task if the normal GPL techniques are used. Building a compiler or an interpreter from scratch consumes a lot of time and effort, plus special technical skills are required. Most of the current applications involve more than one domain, consequently the modularity and composability of the DSL are very important aspects when creating a new DSL or even choose an existing one.

Language workbenches, the term is proposed by Martin Fowler [7], are comprehensive environments that try to provide a solution for the above problems by providing high level tools. These tools facilitate the development, maintaining, and composition of DSLs. Most of language workbenches apply the idea of modular language and language extension. They enable the language developer to create languages' modules that are fully integrated with each other.

Although the DSL concept is not a new one, there are no standard approaches for DSLs development. There are many aspects that should be handled by the language workbenches to

TABLE I. DSL Workbenches

Workbench Name	URL
Ensō	http://www.enso-lang.org/
Intentional Software	http://www.intentsoft.com/
LISA	—–
Mas	http://www.Mas-wb.com/
MetaEdit+	http://www.metacase.com/
MontiCore	http://www.monticore.de/
MPS	http://www.jetbrains.com/mps/
Neverlang	http://neverlang.di.unimi.it/
Rascal	http://www.rascal-mpl.org/
Silver	http://melt.cs.umn.edu/silver/
Spoofax	http://www.spoofax.org/
SugarJ	http://www.sugarj.org/
Whole Platform	http://whole.sourceforge.net/
Xtext	http://www.eclipse.org/Xtext/

achieve their goals. Each workbench follows its own approach to implement these aspects. Some of these approaches are published in scientific papers and others are in the workbench documentation. Up to our knowledge, no recent study done to discuss and classify these approaches.

This paper discusses the different approaches for achieving the various DSL implementation aspects. It includes structure, editor, semantics, and composability language aspects. There are other aspects that are not included in this survey like language validation and testing aspects. The language workbenches are classified based on the approaches used in each aspect. The survey includes 14 language workbenches (Table I). Unfortunately, there are some workbenches that the authors couldn't fetch their approaches in some aspects due to poor documentation or due to their being commercial workbenches.

The rest of the paper is structured as follows, Section II describes some concepts of the language implementation and workbenches. Section III looks at related surveys on DSL implementation. Sections IV, V, VI, and VII introduce the different aspects of the DSL implementation. They illustrate each aspect and the main approaches, that are used in the current workbenches, to achieve these aspect. Section IX concludes the paper and presents prospective ongoing research directions.

II. Background

Programing languages are a set of languages used to give instructions to the computers to preform a specific task. There are two types of programming languages: General Purpose Languages (GPLs) and Domain Specific Languages (DSLs). GPL is a language that is used to write programs for any domain. It does not contain any constructs that are related to a specific domain. Java, C++, and Python are examples of GPLs. DSL is a language that concerns with a specific class of problems that represents a specific domain. It contains abstractions and optimizations for the given domain. A specialized editor is often provided with the DSL to provide a specialized editor services and error messages. SQL, HTML, and CSS are examples of DSLs.

DSL could be implemented as internal or external DSL. Internal DSL is the DSL that is embedded into a general purpose language. This type is limited with the grammar of the host language. Additionally, there is a lack in the editor support since the editor is not aware of the grammar and constraints of the embedded DSL. In contrast, external DSL are implemented independently from any host language. It has its own grammar and editors which makes the language more flexible and the editor could assist the programmer during the development process with the domain knowledge. Throughout the paper, The term DSL is used to refer to the external DSL.

DSL development process involves five stages: 1) Domain Analysis: this stage includes understanding and analyzing the concerned domain, getting the concepts and relations of the domain, and determining the boundaries of the domain, 2) Design: this stage defines the design of the DSL that could encode the problems of the domain determined in the previous phase, 3) Implementation: the concrete language is implemented in this stage using one of the existing tools, 4) Evaluation: where the new DSL is evaluated to determine if it satisfies the business's need or not, and 5) Maintenance: it is a continues stage to update the new developed DSL to satisfy the continues changes in the business requirements. This survey concerns with the approaches used in the implementation stage only.

There are two general approaches for programing languages implementation: compilation and interpretation. In the compilation approach, the compiler translates the programs, written in a high level language, into a low level language that will be executed by the machine. In the interpretation approach, the interpreter executes the actions written in the given program directly on the machine without any translation. Building a compiler or an interpreter from scratch is a complex task that needs a lot of effort and time. It also needs special technical skills. DSLs are lightweight languages than GPLs since they cover limited domain. It should be implemented faster with less effort and knowledge. Therefore, using the classical GPL implementation approaches for DSL implementation is not realistic.

Language Workbench is an environment for DSLs development. It provides high level tools for implementing, evaluating, and maintaining DSLs. Language Workbench reduces the effort and knowledge needed for building DSLs and hides the complexity of the GPL implementation approaches. Fowler [7] lists three components that should be supported by any language workbench to create a new DSL: 1) The abstract representation that includes defining the language structure, 2) The editor that allows the user to manipulate the abstract representation, and 3) The generator that transforms the abstract representation into an executable coded. The current approaches used to support the above three components are described in Sections IV, V, and VI. In addition, the language workbench should support the integration among DSLs which is described in Sections VII.

Throughout the paper, the language developer is the person who is responsible for developing the DSL, and the user is the person who utilizes the DSL to write programs. Fig. 1 shows the relation among language developer, language user, and language workbench.

III. Related Work

Although, there are many research directions in the domain specific languages field, there is a limited number of survey papers that cover the aspects of implementing DSLs. Van

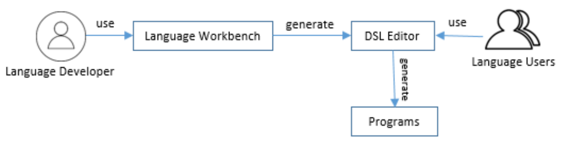


Fig. 1. DSL Implementation Stockholders

Deursen et al. [8] lists the implementation approaches for the semantic aspect only. Kosar et al. [9] provides a preliminary study on ten DSL implementation approaches. A unified DSL is designed to be implemented by the ten approaches to compare the DSL implementation effort and the end-user effort to build programs with the given DSL. Similar work is done by [10] to evaluate four DSL implementation approaches based on unified state machine DSL. It concludes that no single approach is valid for all scenarios. Oliveira et al. [11] provide a theoretical survey on DSLs that does not describe any implementation techniques. Erdweg et al. [12] focus on the language composability aspect and how it is covered by different language workbenches.

A systematic mapping study is provided by [13] that includes research questions related to the techniques and methods of DSLs, the existing DSLs with their related domains, and the tools for DSL creation and usage. The study lists some techniques used in DSLs and it provides statistics about the DSL research types and their respective domains. The results show that defining external DSLs for the different domains gains a lot of attention. This indicates that utilizing of DSLs is very useful and applicable in many domains.

Language Workbench Challenge 2013 (LWC'13) published a work where the authors propose a feature model for languages workbenches and classify the workbenches according this model [14], [15]. It provides a unified challenge (i.e. a DSL for questionnaires) to be implemented by ten workbenches that are included in LWC'13. The paper compares the features provided by the different workbenches in the different aspects. It shows that no one language workbench supports all the required features. The authors is concerned with the supported features of each workbench more than the approaches used to implement these features.

Kosar et al. [16] and Mernik [17] performed another systematic mapping study on the research papers published from 2005 to 2013. The authors include the papers published after their survey paper in 2005 [18].The systematic mapping study includes research questions that try to catch the research space and trends of the DSL field within the given period. One of the main conclusions of this study is that the DSLs will be the main programming languages for the next period. Additionally, it reports that one of the open problems in DSL field is how to facilitate the DSL development for domain expert. This survey discusses the current approaches that contribute in solving the above problem. Thanhofer-Pilisch et al. [19] preformed another systematic mapping study that focuses on DSL evolution only.

As per the previous paragraphs, no recent survey papers cover the latest approaches of implementing DSLs. Unlike the above work, this survey describes the recent approaches used in four different aspects. It focuses on the internal approaches used in the current workbenches rather than the provided features of each workbench. The provided survey could be a starting point for the DSL researchers to catch the current status of DSL implementation aspects.

IV. Language Structure Aspect

The language structure aspect includes defining the structure of the new DSL. This structure represents the concepts and the relations that reflect the target domain. In this aspect, the language developer defines the concepts that are included in the *MiniIoT* language. Fig. 2 shows a subset of the concepts of *MiniIoT* DSL and their relations. *MiniIoT* contains two types of devices: sensors and actuators. The device is located in some place. The Sensor generates a specific observation and the actuator does some action.

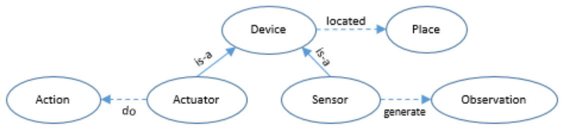


Fig. 2. *MiniIoT* Concepts

There are two main approaches, used in the current language workbenches, for defining the language structure: Grammar-driven approach and Model-driven approach. The subsequent sections describe each approach in details. Table II classifies the included workbenches according to the above mentioned approaches.

TABLE II. DSL Workbenches Classification according to Language Structure Aspect

Approach	Workbench	Used Techniques
Grammar-driven Approach	*Spoofax*	SDF
	SugarJ	SDF
	Silver	Attribute Grammar
	LISA	Attribute Grammar
	Neverlang	Grammar DSL
	MontiCore	Grammar DSL
	Rascal	Grammar DSL
	Xtext	Grammar DSL
	Ensō	Object Grammar
Model-driven Approach	*Whole Platform*	Meta-model DSL
	MPS	Meta-model DSL
	Mas	Meta-model DSL
	MetaEdit+	GOPPRR Model
	Intentional Software	Meta-model DSL

A. Grammar-Driven Approach

In the grammar-driven approach, the definition of the language depends on defining the grammar of this language. Context Free Grammar(CFG) is used as the formal definition for the concrete syntax of the language. The meta languages, that are used to define CFG, are reused to define the DSLs with some modifications to be more appropriate for DSL definition. Backus–Naur Form (BNF) is a one of the main formal meta-languages that are used to define CFG. Extended Backus–Naur Form (EBNF) is a meta-language that extends BNF by adding more notations and symbols like multiplicity symbol.

Attribute grammar is a context free grammar with attributes and evaluation rules. It can contain syntax and semantics information while (E)BNF contains syntax only. The values of the attributes are evaluated by a compiler or a parser according to the given evaluation rules. Attribute grammar defines two types of attributes: synthesized attributes and inherited attributes. The values of the synthesized attributes are computed from the values of the attributes of the children nodes. They pass semantic information down in the abstract syntax tree. However, the values of the inherited attributes are computed using the values of the attributes of the parent nodes. They pass semantic information up in the abstract syntax tree. *Silver* [20] and *LISA* [21] are examples of systems that utilize the attribute grammar to define the structure of the DSL.

Syntax Definition Formalism [22] (SDF) is another meta-language for defining the grammar of a language. SDF is richer than (E)BNF and more modular. It allows to divide the grammar into modules which facilitates the language's embedding and reusing. In addition, SDF provides more declarative grammar definition and declarative disambiguation. Moreover, It enables to define both lexical and context-free syntax. For the above reasons, SDF is more suitable to be utilized for DSL definition. SDF implementation provides a scannerless parser for the SDF definition. *Spoofax* [23] and *SugarJ* [24] are examples of workbenches that use SDF for defining language structure.

Another technique for defining the grammar of the language is to develop a new DSL for the grammar definition. The main advantage of this technique is that the new developed DSL is designed to be easier and more user friendly than the above standards. In addition, most of the proposed languages support modularity and reusability. *Xtext* [25] workbench proposed a grammar DSL which is an EBNF-like DSL developed by *Xtext* itself. This grammar DSL defines the concrete syntax and how it is mapped to the semantic model of the new DSL. *Rascal* [26] workbench has a meta-programming DSL that includes notations for defining the syntax of the new DSLs.

Neverlang [27], [28] also proposes a DSL that represents the syntax by a set of productions coded in BNF. *MontiCore* [29], [30] proposes Grammar Definition Language for defining both concrete and abstract syntax. The benefit of using the same language for both elements is to reduce the inconsistency and redundancy between the two elements. One of the drawbacks of this technique is that it does not support languages that have multiple concrete syntax and single abstract syntax.

Object grammar [31], [32] is another technique proposed in *Ensō* workbench. It provides a declarative mapping between text and object graph. Object grammar extends EBNF by adding constructs to build an object graph while parsing the text. It also defines how to transform the generated graph back into text. Object grammar produces a graph as an output for the parsing process instead of the parse tree in the above techniques. The graph is described by a schema. One of the main advantages of Object Grammar that it supports language reusabilty easier than traditional grammar-based techniques.

B. Model-Driven Approach

The Model-driven approach does not use grammar rules to define the language structure. In contrast, it utilizes the meta-modeling for defining the structure of the language. The program, written by the user, is the model that conforms to the meta-model defined by the language developer. A Language structure meta-model is used to describe the abstract syntax not the concrete syntax. It defines the concepts, relations, and constraints of the language. This meta-model does not describe how the user will edit these concepts and relations. The concrete syntax is determined by a projection process that describes how to edit the concepts and relations defined in the previous meta-model. The projection process is described in more details in the Language Editor Aspect (Section V-B). One of the main advantages of this technique is that it provides higher abstraction level than the grammar-driven techniques. The language developer doesn't have to know the grammar details or the parser techniques. He defines a semantic model rather that a concrete syntax which makes the definition process more easier.

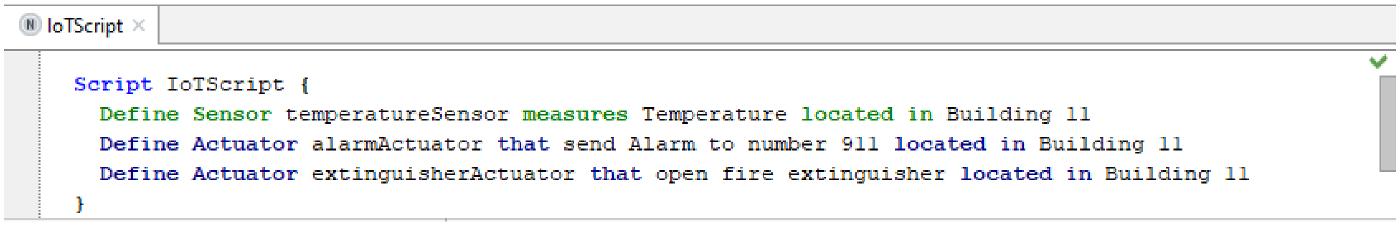
Most of the language workbenches, that follow this approach, propose a DSL to define of the language structure meta-model. *MPS* [33], *Whole Platform* [34], *Mas*, and *Intentional Software* [35] workbenches follow this approach. *MPS*'s DSL is called *Structure Language* which is a DSL for representing and editing the Abstract Syntax Tree (AST) of the program. It defines the concepts, attributes and relations of the nodes of the AST. *Whole Platform* has a modeling framework that utilizes a DSL called *Models Language*. *Models Language* defines the language as a set of entities, features, and types. *Mas* also has its own DSL to define the structure of the language. It a web-based language that allows the users to define the abstract syntax of the DSL.

Intentional Software Workbench [36] proposed a tree representation for the software programs that merges the abstract syntax and the concrete syntax. It represents any program as a so-called intentional program tree which is a kind of syntax tree. This tree consists of a set of nodes and references; the nodes represent the program elements and the references represent relations among elements. The intentional program tree includes different types of nodes that hold semantic and syntactic information. This mix between abstract and concrete syntax facilitates the textual representation of the semantic model as described in Section V-B.

MetaEdit+ [37] is a language workbench that supports only graphical DSLs. It proposes a model called GOPPRR Model to define the abstract syntax of the graphical DSL. GOPPRR stands for Graph, Object, Property, Port, Role, and Relation which are the elements of the language structure in *MetaEdit+*. The user creates the GOPPRR model by filling in a set of forms or by specifying the model graphically.

V. Language Editor Aspect

In language editor aspect, the language developer defines how the user views and edits the language structure defined in the above aspect. In this aspect, the developer of the *MiniIoT* language should define the editor of the sensors and the actuators concepts. The editor should allow the user to define new sensors and actuators and to define the relations among them. In addition, It should support all the language editor services like highlighting, error checking, and code completion. Fig. 3 shows a sample textual editor for *MiniIoT* script that defines sensors and actuators. This editor is created by *MPS* workbench.



```
Script IoTScript {
  Define Sensor temperatureSensor measures Temperature located in Building 11
  Define Actuator alarmActuator that send Alarm to number 911 located in Building 11
  Define Actuator extinguisherActuator that open fire extinguisher located in Building 11
}
```

Fig. 3. *MiniIoT* Editor

TABLE III. DSL WORKBENCHES CLASSIFICATION ACCORDING TO LANGUAGE EDITOR ASPECT

Editing Mode	Workbench	Supported Language Notations			
		Textual	Tabular	Symbols	Graphical
Parser-based	*Spoofax*	✓			
	Neverlang	✓			
	MontiCore	✓			
	Silver	✓			
	Rascal	✓			
	Ensō	✓			✓
	SugarJ	✓			
	Xtext	✓			
	LISA	✓			
Projectional	*Whole Platform*	✓	✓	✓	✓
	MPS	✓	✓	✓	
	Mas	✓	✓		
	MetaEdit+		✓	✓	✓

The user should be able to edit either the grammar-driven or the model-driven definition. The usability of the language editor is one of the key factors for the success of any DSL. There are two editing approaches implemented in the language workbenches: the parser-based editing and the projectional editing (i.e. parser-less editing). Table III classifies the workbenches according to the above editing approaches.

A. *Parser-based Editing*

Parser-based editing depends on a parser that translates a textual code into an AST. Users interact with the concrete syntax by entering a sequence of characters into a text buffer. The parser then matches this sequence with the language grammar to construct the abstract syntax tree (AST) of the program. This editing approach is the main approach that are applied in the grammar-driven workbenches. The parser is automatically generated after the grammar of the DSL is defined by the language developer. The generated parser is utilized to build the final editor of the DSL. Most of the parser-based workbenches generate the editor as an Eclipse plugin based on the generated parser. Consequently, The parser-based editors support textual DSLs only. *Ensō* is an exceptional case, it is a grammar-driven workbench that supports graphical notations since it depends on the object grammar that defines the bidirectional mapping between the text and the object graph (Section IV-A).

Spoofax and *SugarJ* use SDF for the grammar definition, SDF generates scannerless generalized LR parser [38]. *Spoofax* also has a configuration DSL called *Editor SerVice (ESV)* to configure the editor services. *Silver* has a parser and context-aware scanner generator called *Copper*. *LISA* tool generates a source code of the scanner, parser, interpreter, and compiler in Java. *Silver* and *LISA* do not support editor generation. *Xtext* and *MontiCore* workbenches also generate a parser based on the given grammar. In addition, they generate an Eclipse plugin as a textual editor for the language. *Neverlang* has a modular LR parser generator called *DEXTER* but it does not support editor generation.

B. *Projectional Editing*

Projectional editing is an editing approach that depends on the abstract representation as the main source of information. It is also called parser-less editing since no parsing is needed to build the AST. Unlike parser-based editing, projectional editing allows the user to directly build and edit the AST. No need for transformation from the concrete syntax to the abstract syntax. Instead, the projection process generates a visual representation from the AST. This representation could be textual, graphical, or tabular. The workbenches that apply this approach are called projectional workbenches, they follow the model-driven approach for the language structure definition. One of the main advantages of the projectional editing is the ability to represent the same abstract representation with different visual representations according to the user's need.

As in the language structure aspect, new meta-languages are defined for projecting the abstract representation to the end user and defining the editor services like coloring and error marking. *MPS*, *Whole Platform*, *Intentional Software*, and *Mas* are examples for projectional language workbenches. *MPS* has a DSL for creating the projectional editors called *Editor Language*. It defines the editor as a set of cells [39], [40], each cell can contain a static symbol or a user defined symbol. These cells allow the language developer to define different types of notations for the DSL (i.e. textual, tabular, symbols, or graphical) since they reflect the AST directly without parsing.

Whole Platform has a Model-Based Editing (Mbed) framework that provides the *Editors DSL*. It is a DSL that allows the user to create editors based on the language structure defined by *Models DSL*. Mbed is implemented based on the Eclipse Graphical Editing Framework[1] (GEF). Whole Plateform editors are generated as an Eclipse plugin.

Intentional Software uses the intentional program tree described in (Section IV-B) to facilitate the textual editing. This is done by continuous loop of unparsing, editing, and parsing steps. The unparsing step transforms the tree to a sequence of tokens, each token is a sequence of characters or

[1]http://www.eclipse.org/gef

an extended formate (e.x. image). The token also holds other information like formating information and author information. The generated token sequence is the textual representation of the given tree (i.e. program). The user can edit the token sequence by adding new tokens or deleting existing ones. Then, the updated sequence is parsed to generate an updated tree. The final step is to unparse the updated part of the tree to derive tokens that will be displayed through the editor. The main goal of this method is to keep the advantage of the model-driven approach while simulating the normal textual editors behavior.

C. Language Notations

This section discusses the types of notations that could be supported by the DSL editors. A DSL could include textual, tabular, symbols, and graphical notations. It could be restricted to only one type or it can mix different types of notations. The types of notations supported by the editor is totally dependent on the used editing approach. Table III lists the notations types supported by different workbenches.

Parser-based workbenches (*Spoofax*, *Neverlang*, *Monti-Core*, *LISA*, *Silver*, *SugarJ*, and *Xtext*) support textual notations only since they are based on parser-based editing. However, *Ensō* supports graphical notations since it depends on the object grammar that include bidirectional mapping between the text and the object graph. *MetaEdit+* is the only workbench that does not support textual notations, it supports graphical, symbol, and tabular notations only. Projectional workbenches simplify the integration among different types of notations since they are parser-less. Consequently, *MPS*, *Mas*, and *Whole Platform* support mixing among different notations in the same DSL.

D. Projectional Editing vs. Parser-based Editing

Table IV lists a comparison between the projectional editing and the normal parser-based editing. Language modularity, notational freedom, and program representation flexibility could be achieved easily by projectional editing while it more difficult to be implemented using parser-based editing. Projectional editing is still not compatible with many systems in the current infrastructure like source control systems. One of main advantages of the normal parser-based editing over the projectional one is avoiding the tool lock-in, the user is not limited to specific editors with specific versions. The user can open the code and edit it with any textual editor. Additionally, the code is always open even it contains incomplete or wrong syntax. Finally, the usability of the textual parser-based editors overcomes the usability of the projectional ones. The main reason for this situation is that the user works directly with the program tree in the projectional editors rather than the character sequence in the parser-based editors. Consequently, a special handling should be done for deletion, insertion, and copy/past actions done by the user which is not the case for the parser-based editors.

TABLE IV. PARSER-BASED EDITING VS PROJECTIONAL EDITING

Criteria	Parser-based Editing	Projectional Editing
Editor Usability	Normal text editors	Specialized editors
Language Modularity	Done by merging Grammars (Ambiguity)	Done by merging ASTs
Notational Freedom	Support textual notations only	Support non-textual notations
Language Evolution	Code is always opened	Editors should take special care for incompatible models
Infrastructure Integration	Can be easily integrated with current tools	Not compatible with many tools and copy/past from projectional editor to/from text editor is still an issue.
Tool Lock-in	Any text editor can work	Users are limited to specific editors with specific versions.
Program Representation	Only textual representation	Providing several projections

VI. LANGUAGE SEMANTICS ASPECT

With the structure and the editor aspects, the language developer defines the structure of the DSL and how to edit this structure. In the language semantics aspect, the language developer defines how the program, written in the given DSL, will be executed. Language semantics describes the meaning of the language notations. It describes the steps, the computer should follow, to execute the given program. There are two ways for defining the semantics of a DSL: translational semantics and interpretive semantics.

A. Translational Semantics

Translational semantics defines the meaning of the language by translating this language into another target language. The compiler is an example for translational semantics that describes the semantics of a high level language by translating it into a low level language. For DSLs, the target language is often one of the general purpose languages, and the AST represents the model of the language. Listings 2, 3, 4, and 5 show a sample code that describes the meaning of the *MiniIoT* code shown in Fig. 3 using Java language.

Listing 2: *MiniIoT* Java code generated - Device.java

```
public class Device {
 protected String place;
 public String getPlace() {
  return place;
 }
 public void setPlace(String place) {
  this.place = place;
 }
}
```

Listing 3: *MiniIoT* Java code generated - Actuator.java

```
public class Actuator extends Device {
 private String action;
 public Actuator(String place, String action)
     {
```

```
  super();
  this.place = place;
  this.action = action;
 }
 public String getAction() {
  return action;
 }
 public void setAction(String action) {
  this.action = action;
 }
}
```

Listing 4: *MiniIoT* Java code generated - Sensor.java

```
public class Sensor extends Device {
 private String observation;
 public Sensor( String place, String
     observation) {
  super();
  this.place = place;
  this.observation = observation;
 }
 public String getObservation() {
  return observation;
 }
 public void setObservation(String observation
     ) {
  this.observation = observation;
 }
}
```

Listing 5: *MiniIoT* Java code generated - IoTScript.java

```
public class IoTScript {
 public static void main(String[] args) {
  Sensor temperatureSensor = new Sensor("
      Temperature", "Building 11");
  Actuator alarmActuator = new Actuator("send
      Alarm to number 911", "Building 11");
  Actuator extinguisherActuator = new Actuator
      ("open fire extinguisher", "Building 11")
      ;
 }
}
```

The translation could be implemented by two ways: model transformation and code generation [41]. The model transformation (i.e. model-to-model) translates the model of the source language into the model of target language independently of the concrete syntax of both languages. The classical approach for implementing model transformation is to construct the target AST while traversing the source AST. *Xtext* and *MPS* support this approach. Another approach is to build a relation between the source AST and the target AST. Then, utilizing this relation to implement the translation between them [42]. The advantage of this approach is to support bidirectional mapping between the source and the target ASTs.

The code generation (i.e. Model-to-Text) translates the model of the source language into the source code of target language directly. Template languages are used for implementing code generation where the source code of the target language is embedded as a text within the template language. Consequently, the tool used for editing template language is not aware of the target language. *Xtext* supports code generation by template languages. By composing the template language and the target language, the tool becomes aware of both languages which facilitates the translation definition. *MPS* and *Spoofax* utilize language composition and template languages to support code generation.

Xtext uses *Xtend*[2] language for code generation and model transformation. *Xtend* is a general purpose language very similar to Java syntax but with less linguistics redundancy. *Xtext* supports code generation and model transformation by generating the target text or the target AST while traversing the source AST.

MPS also supports code generation and model transformation. However, the code generation is only used at the end of the chain where the AST is translated into a GPL code to be passed into the compiler. This translation is done by *textgen* language. *MPS* supports model transformation by templates and macros. The template code is an instance of the target model that defines the actual transformation. These templates contain a set of macros that are used to define the dependences between the target model and the source model. Macros also define queries over the source model that are utilized to build the target one.

Whole Platform supports code generation by providing a Java model generation framework that translates the model to Java compilation units. The framework provides a set of APIs, that is based on Eclipse platform Java tools[3], to allow the language developer to define the translation from the model to the Java compilation units. *Whole Platform* supports model transformation by a traversal framework that facilitates traversing the source model to implement the translation process to the target model.

Spoofax has a DSL for program transformation called *Stratego* [43]. It utilizes rewrite rules and rewriting strategies to support code generation and model transformation. The rewriting strategies contain a set of rewrite rules in a specific order and conditions. If the right-hand sides of the rewrite rules are the final text of the target language, then it will preform a code generation. However, If the right-hand sides are parts of the model of the target language, then it will preform a model transformation. *SugarJ* also uses Stratego for defining the language's translational semantics.

B. Interpretive Semantics

Unlike translative semantics, interpretive semantics defines the meaning of the programs by executing them directly without translation to another language. The language developer defines how the different language constructs will be evaluated (i.e. executed). The semantic actions, defined by the language developer, are executed while traversing the AST. The developer of *MiniIoT* in this case will build an interpreter that generates an executable code directly, no intermediate Java code will be generated.

[2]https://www.eclipse.org/xtend/
[3]http://www.eclipse.org/jdt/ index.html

Xtend is utilized by *Xtext* to allow the language developer to build interpreters for DSLs. *MPS* utilizes the BaseLanguage, a Java similar language, to define interpreters. Baselanguage is defined by *MPS* itself, so it could be extended to support more features for building interpreters. The developer uses Xtend in *Xtext* or BaseLanguage in *MPS* to define how the DSL statements will be executed and how the expressions will be evaluated. In addition, he defines the changes in the program's variables and states.

Whole Platform supports interpretive semantic through its traversal framework, the language developer utilizes the APIs provided by this framework to traverse the model and define the corresponding execution actions. *Spoofax* and *SugarJ* follow a different way to build interpreters. It utilizes the rewrite rules to define the current state of the program and to define the transformation among different states to reach the final execution state.

VII. Language Composability Aspect

DSL is a language that is specialized in a specific domain, while the real life programs may contain more than one domain. Merging more than one DSL in the same program is called language composition. Language composability aspect defines how the new DSL will be composed with other DSLs. Assuming that the developer of *MiniIoT* has a new requirement for extending *MiniIoT* to support robots functionalities. *MiniRobot* is a ready DSL for robotic application development. Accordingly, the developer will decide to compose *MiniRobot* DSL with *MiniIoT* DSL to satisfy the new requirements.

Erdweg et al. [12] proposed a theoretical classification for the different types of language composition. The authors specify four types of language composition: Language Extension, Language Unification, Self-Extension, and Extension composition. These types describe the relations among the languages that will be composed. The composition process includes combining the different language aspects: Language Structure, Language Editor, Language Semantics, and Language Validation.

Parser-based workbenches support composability by merging the grammars of the combined languages. *Xtext* supports only language extension. It defines the grammar in a modular way by the grammar DSL, where the module could extend another module. The grammar in *Xtext* only inherits from one base grammar, consequently it could not support language embedding, extension composition, and language unification. The limitation of *Xtext* composability is due to utilizing ANTLR's LL(*) algorithm [44]. Alternatively, workbenches that depend on generalized parsing techniques support different types of grammar composition.

Spoofax and *SugarJ* uses SDF and Stratego for composing different language aspects. SDF applies scannerless generalized LR parsing, which enables language unification. *Rascal* is another example for workbenches that utilizes generalized parser. *Spoofax*, *SugarJ*, and *Rascal* support language semantics and validation composition by combining the rewrite rules.

MontiCore supports composability by grammar inheritance and language embedding. The grammar inheritance enables language extension without changing the base language. The extension adds new productions or overrides existing ones. The language embedding between two grammars is done by defining external nonterminals that should be filled by the embedded language.

Neverlang depends on feature orientation and modular language development to implement language composability. It proposes a feature oriented composition model that defines the language's implementation as a set of components. The model defines the relations among the components as dependencies. The dependency is a property that is required by the component but it is not defined in the same component. The dependency is represented by a placeholder in the syntax level or by a semantic property in the semantic level.

Ensō achieves composability by implementing a merge operator between the two object grammars that define the combined languages. The merge operator is a union operator that merges the object graphs of the two languages and overrides the duplicate objects and attributes by the values of the second language. This allows the second language to extend and modify the first one.

Projectional workbenches support composability aspect easier than parser-based ones, since the composition is based on the integration between the abstract representations rather than the grammar rules. *MPS* supports the idea of modular language [45]. Modular language is a bridge between large languages and small languages. It depends on a small core language and a set of language modules (extensions) come with its own syntax, editor, and IDE tooling. The composition in *MPS* is very similar to object oriented programming.

VIII. Discussion

The survey indicates that the approaches of developing DSL could be classified into two main classes: Model-based approaches class and Text-based approaches class. The model-based approaches represent the language as a meta model and the program as an instance model of the given meta model. The meta model is the abstract representation of the language. The definition of the meta model represents the language structure aspect. The definition for the process of editing the meta model represents the language editor aspect. The composition aspect is achieved by merging the meta models of the given languages.

On the other hand, the text-based approaches represent the language as a set of grammar rules and the program as a text that follows the given rules. The definition of the grammar rules represents the language structure aspect. The definition of the parser that translates the text into an abstract syntax tree represents the language editor aspect. The composition aspect is achieved by merging the given grammar rules.

The language semantic aspect defines the translation or the interpretation of the abstract representation to other representation or executable code. Since the language semantic aspect depends on the abstract representation, the same techniques are used in model-based approaches and text-based approaches to implement this aspect.

The text-based approaches are similar to the approaches used for developing general purpose languages, accordingly they are more mature than the model-based approach. In addition, they are applied by more language workbenches than model-based approaches. Alternatively, the model-based approaches are considered as a new direction for implementing programming languages. They facilitate the implementation of many language aspects and they provide editing capabilities more than text-based approach.

IX. Conclusion

This paper introduces a survey of the different aspects of implementing a new DSL. The survey covers structure, editor, semantics, and composability language aspects. It lists the approaches used for achieving each aspect and describes how different workbenches apply this aspect.

The survey concludes that there are no standards for applying the different DSL implementation aspects. More research is needed to set standards for DSL implementation. Additionally, the key aspect of the DSL implementation is the structure aspect. The technique used in the structure aspect determines the techniques used in the other aspects. Finally, no one existing approach facilitates all DSL implementation aspects.

Future work should consider other aspects like validation and testing aspects. Further research is needed to cover the missing workbenches in each aspect. The survey shows that the projectional editing approach is a very promising approach that could facilitate the implementation of many aspects. Accordingly, further studies will be done to address the current challenges of applying the projectional editing approach.

References

[1] R. Lämmel, “A story of a domain-specific language,” in *Software Languages*. Springer, 2018, pp. 51–86.

[2] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton, “A software engineering experiment in software component generation,” in *Proceedings of the 18th international conference on Software engineering*. IEEE Computer Society, 1996, pp. 542–552.

[3] S. Kelly and J.-P. Tolvanen, “Visual domain-specific modelling: Benefits and experiences of using metacase tools,” in *International Workshop on Model Engineering, at ECOOP*, vol. 2000. Citeseer, 2000, pp. 1–9.

[4] A. N. Johanson and W. Hasselbring, “Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 2206–2236, 2017.

[5] T. Kosar, M. Mernik, and J. C. Carver, “Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments,” *Empirical software engineering*, vol. 17, no. 3, pp. 276–304, 2012.

[6] T. Kosar, S. Gaberc, J. C. Carver, and M. Mernik, “Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments,” *Empirical Software Engineering*, vol. 23, no. 5, pp. 2734–2763, 2018.

[7] M. Fowler, “Language workbenches: The killer-app for domain specific languages,” 2005.

[8] A. Van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography,” *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[9] T. Kosar, P. E. Martı, P. A. Barrientos, M. Mernik *et al.*, “A preliminary study on various implementation approaches of domain-specific language,” *Information and software technology*, vol. 50, no. 5, pp. 390–405, 2008.

[10] N. Vasudevan and L. Tratt, “Comparative study of dsl tools,” *Electronic Notes in Theoretical Computer Science*, vol. 264, no. 5, pp. 103–121, 2011.

[11] N. Oliveira, M. J. Pereira, P. Henriques, and D. Cruz, “Domain specific languages: A theoretical survey,” *INForum’09-Simpósio de Informática*, 2009.

[12] S. Erdweg, P. G. Giarrusso, and T. Rendel, “Language composition untangled,” in *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. ACM, 2012, p. 7.

[13] L. M. do Nascimento, D. L. Viana, P. A. S. Neto, D. A. Martins, V. C. Garcia, and S. R. Meira, “A systematic mapping study on domain-specific languages,” in *Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA’12)*, 2012, pp. 179–187.

[14] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, “The state of the art in language workbenches,” in *International Conference on Software Language Engineering*. Springer, 2013, pp. 197–217.

[15] S. Erdweg, T. Van Der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, “Evaluating and comparing language workbenches: Existing results and benchmarks for the future,” *Computer Languages, Systems & Structures*, vol. 44, pp. 24–47, 2015.

[16] T. Kosar, S. Bohra, and M. Mernik, “Domain-specific languages: A systematic mapping study,” *Information and Software Technology*, vol. 71, pp. 77–91, 2016.

[17] M. Mernik, “Domain-specific languages: A systematic mapping study,” in *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 2017, pp. 464–472.

[18] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[19] J. Thanhofer-Pilisch, A. Lang, M. Vierhauser, and R. Rabiser, “A systematic mapping study on dsl evolution,” in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2017, pp. 149–156.

[20] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, “Silver: An extensible attribute grammar system,” *Science of Computer Programming*, vol. 75, no. 1, pp. 39–54, 2010.

[21] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer, “Lisa: An interactive environment for programming language development,” in *International Conference on Compiler Construction*. Springer, 2002, pp. 1–4.

[22] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, “The syntax definition formalism sdf—reference manual—,” *ACM Sigplan Notices*, vol. 24, no. 11, pp. 43–75, 1989.

[23] L. C. Kats and E. Visser, *The spoofax language workbench: rules for declarative specification of languages and IDEs*. ACM, 2010, vol. 45, no. 10.

[24] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, “Sugarj: library-based syntactic language extensibility,” in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 391–406.

[25] M. Eysholdt and H. Behrens, “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*. ACM, 2010, pp. 307–309.

[26] P. Klint, T. Van Der Storm, and J. Vinju, “Easy meta-programming with rascal,” in *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 2009, pp. 222–289.

[27] W. Cazzola and E. Vacchi, “Neverlang 2–componentised language development for the jvm,” in *International Conference on Software Composition*. Springer, 2013, pp. 17–32.

[28] E. Vacchi and W. Cazzola, “Neverlang: A framework for feature-oriented language development,” *Computer Languages, Systems & Structures*, vol. 43, pp. 1–40, 2015.

[29] H. Krahn, B. Rumpe, and S. Völkel, “Monticore: a framework for compositional development of domain specific languages,” *International journal on software tools for technology transfer*, vol. 12, no. 5, pp. 353–372, 2010.

[30] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, “Systematic composition of independent language features,” *Journal of Systems and Software*, vol. 152, pp. 50–69, 2019.

[31] T. van Der Storm, W. R. Cook, and A. Loh, "Object grammars," in *International Conference on Software Language Engineering*. Springer, 2012, pp. 4–23.

[32] T. Van Der Storm, W. R. Cook, and A. Loh, "The design and implementation of object grammars," *Science of Computer Programming*, vol. 96, pp. 460–487, 2014.

[33] M. Voelter and V. Pech, "Language modularity with the mps language workbench," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1449–1450.

[34] R. Solmi, "Whole platform," Ph.D. dissertation, PhD thesis, University of Bologna, 2005.

[35] C. Simonyi, M. Christerson, and S. Clifford, "Intentional software," in *ACM SIGPLAN Notices*, vol. 41, no. 10. ACM, 2006, pp. 451–464.

[36] D. Waggoner, M. A. Jensenworth, P. Kwiatkowski, and C. Simonyi, "System and method for combining text editing and tree encoding for computer programs," Jun. 13 2017, uS Patent 9,678,724.

[37] S. Kelly, K. Lyytinen, and M. Rossi, "Metaedit+ a fully configurable multi-user and multi-tool case and came environment," in *International Conference on Advanced Information Systems Engineering*. Springer, 1996, pp. 1–21.

[38] E. Visser *et al.*, *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.

[39] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *International Conference on Software Language Engineering*. Springer, 2014, pp. 41–61.

[40] F. Steimann, M. Frenkel, and M. Völter, "Robust projectional editing," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2017, pp. 79–90.

[41] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, "Dsl engineering-designing, implementing and using domain-specific languages (2013)," *URL: http://voelter.de/dslbook/markusvoelter-dslengineering-1.0. pdf, http://dslbook. org*, 2013.

[42] OMG, "Mof 2.0 query/view/transformation (qvt) adopted specification." 2005, oMG document ptc/05-11-01.

[43] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. a language and toolset for program transformation," *Science of computer programming*, vol. 72, no. 1-2, pp. 52–70, 2008.

[44] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[45] M. Voelter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with mps," *Software Language Engineering, SLE*, vol. 16, no. 3, 2010.