2IMP20 Domain Specific Language Design

Individual report

Quartile 4 - 2021-2022

Full Name	Student ID	Study
Mingzhe Shi	1665073	Computer Science and Engineering

Eindhoven, July 11, 2022

Contents

1 Reflection

In this quartile's Domain Specific Language (DSL) Design course, we started with an initial review of regular and context-free grammar to understand and learn the concept of metamodeling. By learning the metamodeling language: Rascal, we successfully used Rascal to define the corresponding abstract syntax of a domain-specific language: HCL and established its concrete syntax through the Rascal language workbench(LWB). Following this, we learned the Eclipse Modeling Framework (EMF) techniques and understood the form that defines the static and dynamic semantics of the language. After learning model transformation to develop model-to-model transformation, we completed the modelling of HCL under the Eclipse EMF workbench. We converted it into the corresponding Graph Model with the help of $QVTo$.
In the second half of the course, we learned the relevant knowledge of code generators and got to know $lex\&yacc$, $Xtext$, $Xtend$, MPS and other DSL development workbenches. Finally, we combined the knowledge learned above, from syntax definition and semantics implementation to code generation, to design and develop our own DSL. In the process, We learned to be a language developer and gained valuable experience. Below I will discuss my gains and reflections on three aspects.

1.1 The use of flex&bison

Lex and Yacc were the first popular and efficient lexer and parser generators, and flex and bison were the first widely open source versions compatible with the original software [4]. When we choose a language workbench to develop our DSL, the first thing that comes to our mind is $flex\&bison$. We learned about the grammar of $lex\&yacc$ in class and how to use them for syntactical analysis. We were impressed by the precise handling of precedence, association, ambiguity and contradiction in this LWB. Considering their extremely convenient environment configuration (in fact, they are already installed by default in our mac OS) and relying on GCC compilation, which allows us to easily implement syntax implementation and code generation.

1.1.1 How we use flex&bison

Flex&bison makes it easy for us to use VSCode to develop our lexical analyzer, parser and code generator. We define the lexical patterns of our language in the lexer. Specifically, we create records for all special tokens and use regular expressions to match statements and constants. In this way, when the complete flex lexical analyzer is compiled through C to generate an executable, we can scan the DSL file we input and execute the corresponding C code according to the matched token. We import the executable from flex into bison and improve and write our concrete syntax tree according to the previous abstract syntax definition. In addition, we can define operator precedence and association in this parser and resolve shift-reduce conflicts through left recursion. The code generator part is also in this bison code file. Our DSL is designed with python grammar as a reference, so we also choose python as the target generated language. We declare the IO stream at the beginning of the code by adding matching python codes for the corresponding syntax between the concrete syntax tree. In order to check the well-formedness of the language, we define multiple symbol tables. Adding the identifiers between the concrete syntax trees to the symbol table, comparing and checking whether the declaration is valid. We can get the final code generator executable by compiling the complete bison code file. The code generator executable inputs and compiles our DSL file, and the matching python file can be generated to complete the code generation.

1.1.2 Weaknesses of flex&bison

Although the convenience and popularity of $flex\&bison$ can help us quickly start DSL development, it also brings many limitations. Strictly speaking, $flex\&bison$ is not a complete language workbench, they act more as a language parsing tool and do not provide corresponding help in the process of language development and code generation. Lack of mature IDE help to a certain extent, increased the difficulty of our development. At the same time, the general-purpose language they provide to support defining DSLs is C, making it very uncomfortable for us who are unfamiliar with C during the development process.

1.2 Our Implemented DSL: AIL

When designing this DSL, we thought about areas in which the DSL can help improve productivity and efficiency. After reading some papers and materials, we decided to design a DSL in the field of agricultural irrigation. Its primary function is to provide the staff with the agricultural irrigation information language

to help farmers develop customized real-time monitoring programs for irrigation information according to the actual needs of the farmland and orchards on the farm and the hydrological conditions around the farm. We adopt a modular design, so that developers can build the entire program through plants, actions, assets, observers and controllers. For specific design, please refer to the technical documents we provide.

1.3 Lessons learned in designing and implenenting a DSL

We have considered a lot in the design. We hope to implement a DSL to monitor and send corresponding information in real-time and integrate functions such as function definitions, structures, expressions, judgments, IO and cron jobs. However, late in the development process, we found that the functionality we added was too complex and redundant. A large part of the reason for this design and implementation gap is that we think in terms of python when we design features. However, when implementing the scheduling work, we found that this cannot be achieved out of thin air. We need to build a whole foundation to achieve the function we want on the roof. For example, we took expression and loop into our DSL as a matter of course at design time, which is logically simple. However, even at the time of implementation, and express abstract syntax tree that covers various cases has already cost us. two days. This miscalculation of the amount of work resulted in a DSL that we implemented only a naive version of the DSL we designed.
At the same time, as I see, we also underestimate the role of LWB. In the middle of the course, we learned Model-based DSL Definition and learned about the related workbench, but we did not understand the help of this development thinking in DSL design. We chose lex&yacc as the workbench, which allowed us to It was easy to get started at first. Nevertheless, when we started working on symbol table, well-formedness checking, and python code generation, we had many headaches with C's low compatibility and limitations. Although we found some LISP-type embedded dependencies that could be called to help with development, this also Increased our learning cost. This assignment made me realize the critical role of large-scale LWB in developing complex DSL, and also understand that the process of DSL design and development should be circular. The original intention of DSL design is to solve the inconvenience of working in specific fields caused by the complexity of GPL, so a concise and refined DSL should be what we pursue.
Finally, through studying this course and the exercise of related assignments, I have truly realized the charm of domain-specific language development. We will continue to improve this DSL after submitting the DSL's initial version. At the same time, I also hope to participate in related industrial-grade DSL design and development in the future.

2 Paper Discussion

Many papers and materials helped us greatly during the course study and DSL design development. Based on the course slides, we have deepened our understanding of DSL concepts and implementation ideas by reading some papers in the field of DSL. The following are three representative papers I selected. The first paper is about the timing and design methods of DSL, and the second paper is about agricultural domain-specific language design similar to the DSL we designed. The third paper is about DSL code generation research. I will discuss my insights and reflections on them in conjunction with our DSL design and development.

2.1 Paper1: When and How to Develop Domain-Specific Languages [1]

This article, published in 2005, focuses more on educating developers about the needs for domain-specific language design. Provide general ideas to help DSL developers by researching DSL development patterns and other common problems.

2.1.1 Findings and interpretations

In the article's introduction, the author put forward a very interesting point: *"Any General Programming Language (GPL) can act as a DSL"*. For example, the well-known Application Programmer Interface (API) can be regarded as a domain-specific library consisting of methods, functions and classes. When we need to call them in GPL, our purpose is to use them to realize the capabilities of some specific fields.
The advantage of DSLs is that they provide domain-specificity better than the object creation and method calls that are often necessary when using APIs. Specifically, the advantages of DSL can be summarized in the following three parts:

1) DSL can eliminate the limitation of operator symbols provided by GPL and can use appropriate domain-specific operators according to users' needs in specific domains.

2) The domain-specific construction and abstraction of DSL are more straightforward than the library.

3) The code pattern of DSL is simpler than GPL, so it is easier to use.

These advantages enable DSLs to be more efficient, more closely related to domain-specificity, and easier to analyze, debug, optimize, and transform.
Then, the author introduces the executability and reusability of DSLs. The author does not strictly limit whether a DSL must be executable but gives several levels. Combined with the DSLs we have learned, this point of view knows that it is not outdated now. The HCL we implement in our homework is not executable, but this language is not designed for execution. This DSL not meant to be executable can also use it in visualization tools, consistency analyzers, and checkers. DSLs can also be designed to reuse well-established software specifications and patterns, such as XML, and they kennel the exploitation of the syntax of an existing language. This reuse is also an important reason for the popularity of DSLs.
The advantages and characteristics of DSL also bring difficulties to its development:

1) It needs to have both the knowledge of the target domain and the ability of language development.

2) Its development process is different from GPL, which requires more careful consideration of involved factors.

3) Depending on the size of the DSL's target group, support and maintenance are also areas of consideration later in the development life cycle.

Before we designed our DSL, we did not have a deep understanding of the relevant knowledge of the agricultural irrigation field we were targeting, which led us to spend more time thinking about whether certain functions were applicable during the development process.
The author gives a DSL development pattern of phases in the article: ***decision, analysis, design, implementation, and deployment***, and explains this model in detail. I focused on his elaboration on the design part. The author divides the design methods of DSL into two types, one is the relationship between DSL and existing language, and the other is the formal nature of design description. When designing the DSL, we use the first way, which is also the easiest way. Another approach brings me new insights, extending the existing language to cater to the concept of domain. This way of development is doubly difficult because it not only conforms to the GPL's existing standards but also needs to be designed to make it accessible to non-programmers. The author does not explain where this method mainly applies, but this approach may be used for specific domains with extremely complex functional requirements.

2.1.2 Reflections

Although this paper was published in 2005, the author's many viewpoints in the paper have brought thinking and reflection to our DSL development. The first is that the author's definition of executable attributes for DSLs helped me solve some of the confusion about DSL design. We spent a lot of time thinking about what kind of feedback it should give when we ran the file after it was converted into a python executable when we discussed our DSL's functionality. Still, we didn't need to force anything out. As a DSL, it can provide more of a model for optimization, analysis and inspection of the domain. So we only kept a $MONITOR$ directive at the end of the designed DSL to enable our automated detection of irrigation assets.
Furthermore, the focus on domain-specific factors in DSL development mentioned in the article is similar to the difficulties we encountered. The lack of research in a domain can lead to a lack of domain knowledge in the development process to design a set of control flows clearly and accurately, which gives us empirical reflection. The previous domain analysis and research need to be more concerned when developing other DSLs in the future. Of course, much of the article's discussion of DSL development support tools have been integrated into complete IDEs today. At the same time, the development difficulty brought about by different dimensions of design methods may no longer be a crucial problem for more convenient IDEs. These require new insights from new research.

2.2 Paper2: A domain-specific language framework for farm management information systems in precision agriculture [2]

This article is similar to the domain concerned by the DSL we designed. It introduces the ideas and models of farm management DSL frameworks design in detail and has a certain reference for our language design. At

the same time, the author's research on the design decisions and evaluation brought about by the difference between GPL and DSL and their research on precision agricultural requirements in the article is also helpful.

2.2.1 Findings and interpretations

At the beginning of the article, the author also compares GPLs and DSLs, but he focuses on analyzing the advantages of DSLs in precision agriculture. His point can be summed up as follows: DSLs support co-development approaches because they are generally declarative and easier to understand than GPLs. At the same time, DSL can take advantage of the development of precision agriculture systems, thereby closing the conceptual gap and improving the productivity and quality of developing software systems.
As we all know, when we implement a DSL, we often need to put it into production practice. Rather than focusing on a specific area or problem, workers in certain fields, such as farmers in this paper, need a more complete and straightforward solution. DSLs seek a balance between flexibility and structure and are tailored to a specific domain that can reduce the learning curve. This ensures the feasibility and necessity of DSL development focused on precision agriculture.
Next, the author begins to introduce the design of their DSL from the perspective of requirements and operational needs. From their conceptual diagram, it can be observed that there are three types of actors, of which DSL Developer and Software Engineer are two types of actors. In fact, in the author's conception of the framework, software engineers and farmers are DSL users. The difference is that software engineers use DSL to develop DSL software or tools, while farmers use this set of DSL software, and the author starts DSL design on this basis.
The author's model-based systems engineering approach is instructive in the DSL design process. First, by reading the relevant materials, a formal domain model with main concepts is established for the DSL. Then the elements in the formal model are combined to design the meta-model, feature model and the corresponding context diagram of the precision agriculture system of the DSL. Subsequently, the DSL framework's structure, syntax and features will be explained.
The author also proposes 6 kinds of ***DSL framework alternatives*** for precision agriculture in the field, which is very helpful, in my opinion. The DSL we designed is the fourth design pattern of reference. We also use the definition of assets, actions, and the declaration and invocation of the above definitions for our DSL. This setup ensures that our DSL technical details and functionality are separated from the farm configuration while being less complex and easier to develop.
To evaluate the applicability and practicality of the developed DSL framework, the authors conduct controlled experiments in the second half of the article. Experiments show that The DSL framework is effective and practical in using three DSL languages. The results elaborate that the DSL is easy to use and learn with a small investment of time. In addition, the DSL framework can be extended for other concerns or aligned with developments in the corresponding field. Other projects may even choose other DSL framework alternatives, which depend on the defined evaluation criteria and the subjective assessment of those criteria. From the study of this article, I can see the multiple benefits of the DSL framework for precision agriculture in practice.

2.2.2 Reflections

This paper is beneficial both for our DSL design and development and for understanding DSL design theory. In my opinion, the author provides a very mature and industrialized DSL system framework analysis, development, and evaluation pipeline. This brought me a lot of new insights and reflections.
First of all, in the first half of the analysis of the advantages of the DSL framework, the author's definition of the core requirements of DSL product users impressed me deeply. Combined with his subsequent insights that DSL users should be divided into software developers and domain-specific users, I realized that in designing DSL, the purpose is only to provide domain-specific users. However, often, in the end, The result is a language more suitable for DSL software developers. The fundamental reason is that we did not clarify this user distinction in the initial requirements analysis process. Domain-specific users such as farmers want a concise and complete method, which is not provided by DSL developers but developed by DSL software engineers.
In addition, the author's idea that the modular DSLs can call each other and integrate libraries to construct a DSL framework, the six framework alternatives, and the corresponding convincing evaluation methods are all novel concepts DSL that I have not come across before. Therefore, I highly recommend this paper.

2.3 Paper3: An Evaluation of Domain-Specific Language Technologies for Code Generation [3]

When developing the code generation part of the DSL, we were interested in the efficiency of different production languages and language workbenches, the differences in performance, and how to evaluate them. This article compares and assesses the performance of several LWBs in two target generation languages: C++ and Scala, through scientific methods.

2.3.1 Findings and interpretations

The purpose of the author of this paper using the DSL is to alleviate the errors and time consumption when implementing the optimization algorithm mapping task and improve the work efficiency of the programmer. DSL programs are more portable and extensible because the compiler derives all details. Converting a DSL program into an executable file requires three efforts:

1) Experts in the domain concerned by the DSL study which algorithms are most helpful for the realization of the task so that the compiler can convert this expertise through the corresponding application put. A concrete algorithm replaces the abstract syntax in the DSL.

2) Experts with a deep understanding of the hardware architecture evaluate the target system running the executable program, thereby modifying the implementation of the previously defined algorithm.

3) The actual developer confirms that the appropriate technology is used to obtain the best performance. These optimization techniques need to be applied to the previously chosen implementation method.

The authors chose Spoofax/IMP for academic-derived LWBs and Rascal for metaprogramming-based LWBs when choosing the language workbench and C++ and Scala as GPL-based workbenches. The transformation process begins by reading a DSL file containing definitions of computational domains and variables, boundary conditions, mathematical operators, and basic statements such as loops, conditions, input/output, and mathematical operations. It is then stored by converting variable and computed field definitions to array definitions and operators to functions. Finally, print the stored structure as C++ code. The authors then evaluate the above-selected workbenches in the paper. Using Spoofax/IMP as the basis for a DSL is possible and offers several benefits, especially in generating the end-user interface. Features like syntax highlighting and code completion, error checking, flagging, and recovery are automatically generated and distributed as standalone Eclipse plugins. But Spoofax/IMP is not very stable, and authors often encounter error restarts while developing and testing.
The authors consider Rascal a well-thought-out tool for creating DSLs and their corresponding compilation framework. It provides several nice and useful features, such as automatically annotating the location of parser completions or generating a visual representation of an abstract syntax tree (AST). Rascal is stable but still needs to mature in terms of usability. Error messages are usually not very helpful and make it difficult to understand the workflow when debugging.
On a C++-based workbench, flex&bison is the core. The basic logic is that if the specified grammar rule matches the token stream provided by the scanner, the relevant AST construction code or other code will be executed. This is also a viable way to create a custom DSL build framework with excellent compile-time and runtime performance, as is the developer's support of various IDEs and their debugging capabilities. Since C++ is the accepted standard, extensive documentation and many third-party components are available. The fundamental problem is that developers must put great effort into simulating the required but missing functional aspects.
Scala, on the other hand, is a popular tool for designing embedded DSLs. Scala provides powerful parser combinators that allow the use of context-sensitive grammar. Transformations can be described shortly and expressively, and all the information collected before can be easily accessed. However, since DSL IDE generation is not supported like C++, these must be done manually.
The author scored these four code generation schemes through the designed evaluation criteria in the second half of the paper. Finally, C++ and Scala, the two GPL development methods, received higher evaluations. The authors speculate that such results stem from the lack of complete dependencies between Spoofax and Rascal, which prefer to focus on specific target communities, making it difficult for development speed, intensity, and diversity to match small teams with specialized methods.

2.3.2 Reflections

The reflections that this article brought me to focus on interpreting the normative process for code generation and evaluating GPL-based workbenches and specific workbenches. The theoretical explanation of code generation aligns with our ideas when designing the DSL. The method explanation of parsing in the C++ workbench section also provides a clear direction for our work.

Nonetheless, this article's author's evaluation criteria are not rigorous enough. In my opinion, although the possible reasons for the low scores of Spoofax and Rascal are explained at the end, the paper lacks elaboration about the benefits of the IDEs during the evaluation and the functions of syntax highlighting, code completion, error checking, marking, and recovery. In fact, according to our experience with flex&bison, the author's evaluation of the various LWBs in the paper is reasonable. However, the conclusions are less convincing in the absence of actual development examples, especially for tools such as Spoofax, Rascal and even Jetbrains MPS.

References

[1] Marjan Mernik, Jan Heering, and Anthony Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–, 12 2005.

[2] Desirée Groeneveld, Bedir Tekinerdogan, Vahid Garousi, and Cagatay Catal. A domain-specific language framework for farm management information systems in precision agriculture. *Precision Agriculture*, 22:1–40, 08 2021.

[3] Christian Schmitt, Sebastian Kuckuk, Harald Köstler, Frank Hannig, and Jürgen Teich. An evaluation of domain-specific language technologies for code generation. 06 2014.

[4] Tony Mason John Levine, Doug Brown. *lex yacc, 2nd Edition*. O'Reilly Media, Inc., 1992.