Java-centered Translator-based Multi-paradigm Software Development Environment

Xiaohong (Sophie) Wang *xswang@salisbury.edu*
Department of Mathematics and Computer Science
Salisbury University
Salisbury, MD 21801, USA

Abstract

This research explores the use of a translator-based multi-paradigm programming method to develop high quality software. With Java as the target language, an integrated software development environment is built to allow different parts of software implemented in Lisp, Prolog, and Java respectively. Two open source translators named *PrologCafe* and *Linj* are used to translate Prolog and Lisp program into Java classes. In the end, the generated Java classes are compiled and linked into one executable program. To demonstrate the functionalities of this integrated multi-paradigm environment, a *calculator* application is developed. Our study has demonstrated that a centralized translator-based multi-paradigm software development environment has great potential for improving software quality and the productivity of software developers. The key to the successful adoption of this approach in large software development depends on the compatibility among the translators and seamless integration of generated codes.

Keywords: Software Development Environment, Translator, Multi-paradigm.

1. INTRODUCTION

Improving the quality of software products and the productivity of software developers has been an enormous challenge for the software industry. To respond to the challenge, many new design and development methodologies and programming paradigms have been introduced. The availability of modeling tools and rich sets of libraries and the adoption of design patterns and application frameworks all contribute to produce better software systems today. Another rapid evolving frontier in this campaign is the development of programming languages based on different paradigms. In the context of computer science, a programming paradigm is defined as a computational model [1] that a programming language is based on, i.e., the style or approach a programming language uses to express problem solving plans. In the past forty years, several generations of programming languages have been introduced based on the following four dominant programming paradigms: imperative, functional, logic and object-oriented. Since real world problems are much diversified, it is not surprising that some styles are better suitable to solve some problems than others. Another observation is that for large sophisticated software, it is likely that a single paradigm may not be enough to develop all parts of the system. This naturally led to the pursuit of software development using programming languages with different paradigms, i.e., multi-paradigm programming. The overall objective of multi-paradigm programming is to allow developers to choose a paradigm best suited for the part of the problem to be solved. As for how multiple paradigms can be deployed to build a single application, many different routes have been taken to try to answer this question.

The translator-based multi-paradigm programming was first proposed in [6]. This approach allows multi-paradigm programming by translating the source code written in different paradigms into a target language code before they are integrated. [7] has demonstrated the feasibility of this approach by developing a compiler for a functional programming language. However, there are still some questions remain to be answered. How feasible and realistic is it to use this approach in large scale real world application development? What are the main obstacles of deploying this approach in real world?

The rest of this paper is organized as following. In the second section, the background of the translator-based multi-paradigm is discussed. In the third section, we describe our experiment with the translator-based multi-paradigm programming by implementing a centralized software development platform *SourceMerge*, which allows for program development with logic, functional and object-oriented programming languages. Using this platform, a *calulator* application is developed with expression validation, evaluation and GUI components written in Prolog, Lisp and Java respectively. Issues encountered during this experiment are also discussed in this section. The final section summarizes our current work.

2. TRANSLATOR-BASED MULTI-PARADIGM PROGRAMMING

A programming paradigm is often defined as a computational model ([1]) that a programming language is based on. In general, a programming language implements only a single paradigm. For example, the imperative paradigm, with C as an example language, is identified by the use of variables, assignment statements and explicit flow of control. The functional paradigm, with Lisp as a representative, distinguishes itself in function definitions, recursion and the ability to create high-order functions. Logic paradigm depends on rules and logic, and a main language supporting this paradigm is Prolog. The object-oriented paradigm, featured by Java, uses class inheritance hierarchy and polymorphism to create applications with dynamically reusable code. Some modern programming languages can support more than one paradigm. For example, C++ supports both imperative and object-oriented paradigm and SICStus Prolog supports both logic and object-oriented paradigm. Each paradigm has its strength and weakness in representing the concepts and carrying out the actions of a specific application. Multi-paradigm programming is to explore different ways to integrate the best features of each paradigm in software development.

Generally speaking, multi-paradigm programming can be accomplished either inside the same programming language (i.e., language extension using multi-paradigm languages) or in a system that assures a certain way of integration and interaction among separate processes or modules. [2] and [3] proposes to use a multi-paradigm language. Rather than deciding what the correct paradigm is to use, using a language that implements every paradigm can easily solve the issue theoretically ([2]). The problem with this approach is that a language with many paradigms intertwined would be too difficult for most programmers to learn and would be rather hard to understand and debug applications written in such language. An emerging theme is the ability to access one programming language from another ([4]). A good practice would suggest keeping paradigms separated to allow for understandable code. [6] and [7] approach the problem by translating a single language program into a target language such as C. While the approach is credible, it restricts the abilities of multi-paradigm programming to only allowing source and target paradigms to be used together. As one can see, all those approaches are limited in either the number of paradigms can be combined and the degree of integration can be achieved.

Translator-based programming has been discussed in [1], [6], and [9]. The main idea is that different parts of an application can be written in different programming languages; later those different parts are translated into one target language; finally the translated source code in the target language are compiled into a final executable by the target language compiler. A similar approach surfaced after [6] allows for two languages to be written together in the same source file(s) and to be translated/interpreted during compile time ([4]). This approach is much like translator-based multi-paradigm programming except the code are written as if part of the same language instead of being written in separate files as separate programs. [1] argues that after comparing with all others, the translator-based multi-paradigm approach appears to be the most viable and expandable solution since theoretically it allow any number of paradigms to be integrated in a natural way and it is a much better compromise between ease of use and degree of integration.

3. JAVA-CENTERED TRANSLATOR-BASED MULTI-PARADIGM PLATFORM

To answer the questions such as how feasible and realistic is it to use translator-based multi-paradigm approach in large scale real world application development and discover the main obstacles of deploying this approach, we developed a Java-centered translator-based multi-paradigm platform *SourceMerge*. *SourceMerge* provides a simple interface for selecting source files written in different paradigms and translated them into the target paradigm. In this case, Java is selected as the target language and programs written in Lisp and Prolog representing logic and functional paradigm are prime candidates to be translated. Once the selected source files written in Prolog and/or Lisp are translated by their corresponding compilers respectively, *SourceMerge* can collect them into a single location and generate all required libraries and packages and use Java compiler to generate a single, standalone Java application.

3.1 Design Considerations

Java is selected as the target language in *SourceMerge*. Java's object-oriented paradigm is much more suitable for large-scale applications due to its capability for abstraction, inheritance and polymorphism, easy-to-use language interface and its portability to any system that runs a Java Virtual Machine. Java's strength in describing real world objects and their behaviors with class structure makes it natural to represent the concepts and actions to be carried in other paradigms (such as functions in functional paradigm and predicate logic in logic paradigm). Java's sandboxing provides a security blanket that can protect a user from system crashes caused by errors in translated code.

We choose logic and functional paradigms as the two candidate paradigms in *SourceMerge*. However, as can be seen from Figure 1, the design of *SourceMerge* allows the inclusion of a new paradigm to the system can be done easily (as demonstrated by the dashed line portion in Figure 1) since the translation process for each paradigm is independent from the rest of the system. *SourceMerge* acts as an adapter to translate multiple paradigms into a single target paradigm. As long as a language translator follows the rules for generating output defined by *SourceMerge*, it can be easily integrated into the system.

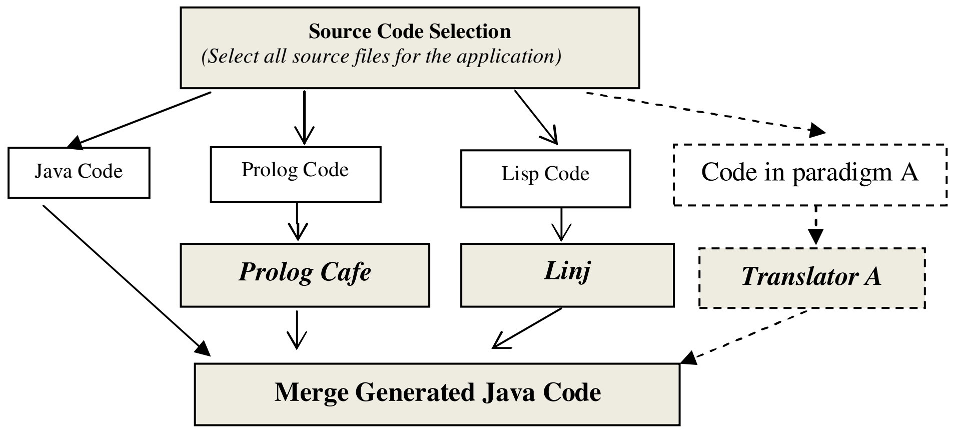
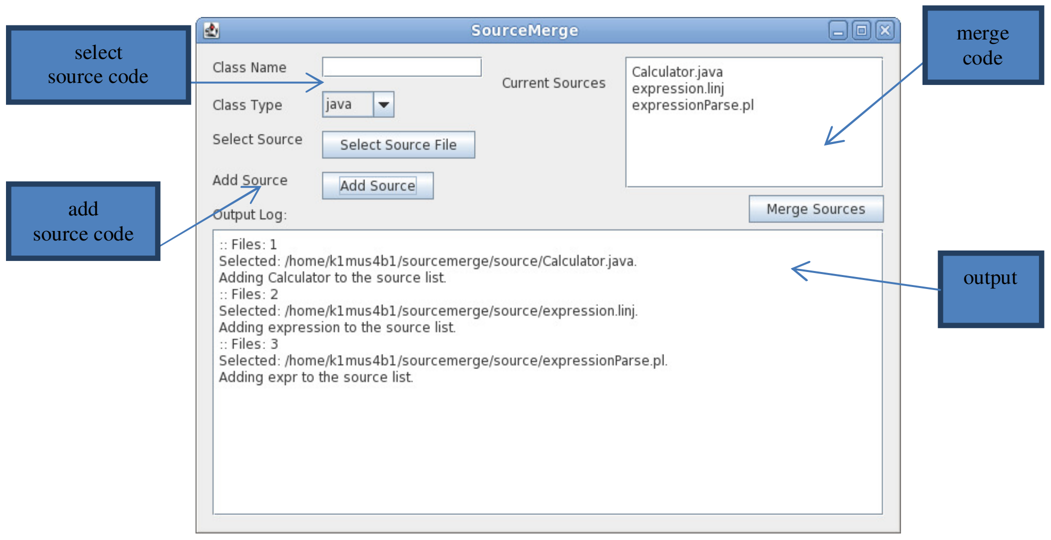


FIGURE 1: *SourceMerge*'s Select, Translate, Merge and Compile Process.

Figure 2 show the GUI of *SourceMerge* application. Three major tasks can be completed on this interface: selection of source files, translation of the source code and merging of the final executable. The execution status of translation and compilation is also displayed on the interface.



Class Type: java

Current Sources:

```
Calculator.java
expression.linj
expressionParse.pl
```

Output Log:

```
:: Files: 1
Selected: /home/k1mus4b1/sourcemerge/source/Calculator.java.
Adding Calculator to the source list.
:: Files: 2
Selected: /home/k1mus4b1/sourcemerge/source/expression.linj.
Adding expression to the source list.
:: Files: 3
Selected: /home/k1mus4b1/sourcemerge/source/expressionParse.pl.
Adding expr to the source list.
```

FIGURE 2: *SourceMerge* Main Interface.

3.2 Translation Tools

Due to the time constraint for this study, we decided to use existing open source Lisp and Prolog translators. After a thorough research over the Internet, we find that *Prolog Cafe* [9] and *Linj* [10] are the only two freely available tools exist today that specifically meet the needs of translating the Prolog and Lisp languages to Java.

The Prolog to Java translator *Prolog Cafe* is built on the *de facto* standard for Prolog compilers, the Warren Abstract Machine (WAM) ([5], [11]). After translating Prolog sources to Java, the execution model of the generated Java classes are also based on the WAM. Though this produces difficult-to-read code and layers of terms to sort through, the translated output executes cleanly and, in the Prolog aspect, quickly. *Prolog Cafe* is written in Java and therefore portable to any platform with Java compiler and runtime environment. The generated source code requires the inclusion of the *Prolog Cafe* Java libraries that implement the WAM algorithms. Additionally, the compiled program still depends on a standalone interpreter within *Prolog Cafe*. This makes it unsuitable to be embedded into other Java programs and we will address this development issue further later.

Linj, the Lisp to Java translator, is open source and it translates from one source paradigm to Java. To make the translation algorithm efficient and allow the programmer to follow the Lisp programming conventions and still have the translated source follow all of Java's rules, *Linj* comes with its own language, respectively named *Linj* ([10]). The *Linj* language is syntactically the same as Common Lisp except for packages, something that is ignored for this study, and the existence of a null-term as opposed to an empty list. The *Linj* translator is written in Common Lisp and the translated source code is purposed to be human-readable and efficient. There are no specially required Java libraries to include so the individual classes generated can be compiled as standalone programs, or embedded into other Java applications.

3.3 Encountered Issues

A major hindrance towards the research for this project is that: both of the translators, *Prolog Cafe* and *Linj*, are no longer supported by their creator. Also, due to the limited user base of the tools, community support and resources are scarce, if not nonexistent.

The first obstacle is that *Prolog Cafe* generates Java source code as a standalone application running through a command-line interpreter that comes with *Prolog Cafe*. This execution model does not fit into our design of the centralized environment that merges Java classes generated from different paradigms and compiles them into a single Java application. Since the command-line interpreter for *Prolog Cafe* is also open source, and it also includes the same Java libraries that are required to be included in the generated source code, this became the starting point of tackling this issue. After stripping away the bells and whistles of the command-line interface of the interpreter, it became visible that the same procedures are called to execute any translated code each time. Therefore, for each Prolog source code, a specially defined template class, which gets dynamically modified by *SourceMerge*, is used to produce the source suitable for integration with other Java classes.

Along with no longer being supported, the documentation for *Linj* was never completed. There are also no instructions as to how to install *Linj* nor the system requirements for installation. The *Linj* translator uses direct Linux commands and the translation process must be performed under a Linux system. This information is missing from the unfinished documentation. Another challenge is that all required Lisp packages are not specified in the document either. So significant amount of effort were made to determine the system requirements, write a parser to search through the *Linj* translator to identify all packages used before the final *Linj* to Java translator running Linux with the Steel Bank Common Lisp compiler was installed successfully.

Two similar issues affected both the *Prolog Cafe* and *Linj* translators from producing usable code and neither were ever hinted at in documentation or other sources. The required Java libraries by the source generated from *Prolog Cafe* don't exist, until one translates them from Prolog to Java. Since the interpreter provided with *Prolog Cafe* has a set of pre-compiled libraries required, initially no errors were encountered when it was used alone. However, to include them into other sources, they needed to be in their raw source and the errors started to show up. With *Linj*, the translator successfully translates all basic Lisp programs to Java without an issue. However, whenever non-basic expressions were used, such as a built-in function, compilation would silently fail without any error messages. After significant time and effort spent on debugging, we found that *Linj* requires a Lisp-package for each of the types being translated (such as mathematical expressions, or vectors). Both of the issues were resolved eventually by letting *SourceMerge* automatically supply the required Java libraries during translation.

3.4 Application Development Demonstration

To demonstrate the functionalities of *SourceMerge*, an arithmetic calculator program was created under the *SourceMerge* environment. The *Calculator* application was written using all three distinct paradigms allowed by *SourceMerge*: functional, logic, and object-oriented. Each paradigm was used to write the part of the application that highlights the best features of this paradigm.

The *Calculator* can perform input validation and evaluate arithmetic expression. Java, the object-oriented paradigm with rich GUI libraries, was used to create the *Calculator*'s GUI (see Figure 3-4).

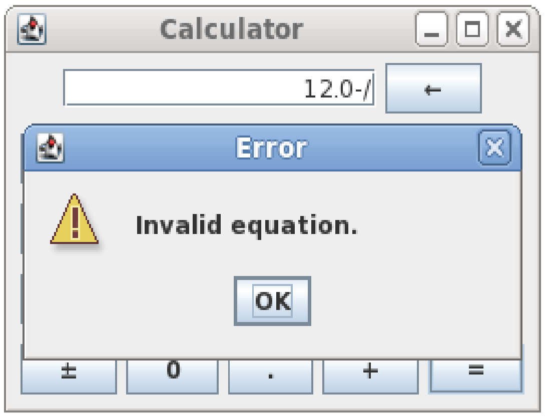

Invalid equation.

FIGURE 3: Expression Validation.

During the development of the *Calculator*, Prolog, the logic paradigm language, was used to implement the validation function for the *Calculator* (Figure 3). Using predicate logic, the validation of a proper arithmetic expression can be achieved by the following block of Prolog code:

```
expr(L) :- num(L).
expr(L) :- append(L1, [+|L2], L), num(L1), expr(L2).
expr(L) :- append(L1, [-|L2], L), num(L1), expr(L2).
expr(L) :- append(L1, [*|L2], L), num(L1), expr(L2).
expr(L) :- append(L1, [/|L2], L), num(L1), expr(L2).
num([D]) :- number(D).
```

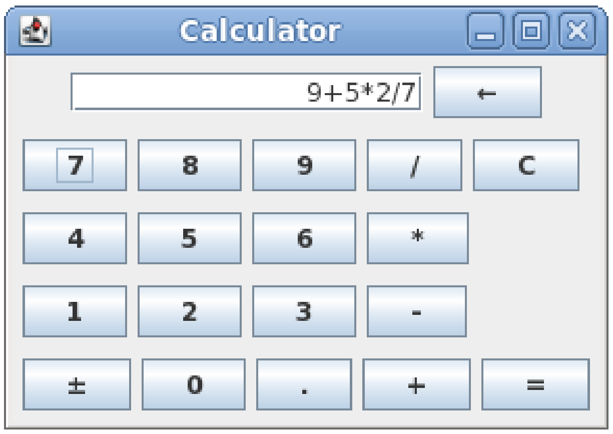


FIGURE 4: Expression Evaluation.

The expression evaluation of the *Calculator* was implemented using Lisp, the functional paradigm language (Figure 4). Avoiding the easy-way of using Lisp's eval, the source was designed to use the prefix notation to operate on two numbers. The used method is as follows:

```
(defun expression(x/float operator/string y/float)
        (cond ((string-equal operator "+") (+ x y))
                ((string-equal operator "-") (- x y))
                ((string-equal operator "*") (* x y))
                ((string-equal operator "/") (/ x y))
                (t 0)))
```

As discussed earlier, during the implementation of the *Calculator*, each of the three involved paradigms was used to implement a component that showcases the paradigm's features most suitable for the functionality of the component. The three kinds of source files were sent into *SourceMerge* (see Figure 2) and the Prolog and Lisp files were successfully translated into Java classes; these Java class files were all merged together; and a compiled *Calculator* application was presented.

4. SUMMARY AND DISCUSSIONS

The Java-centered multi-paradigm software development environment *SourceMerge* built in this study confirms that translator-based multi-paradigm software development approach is, theoretically feasible for producing good quality software efficiently. *SourceMerge* demonstrates a way to take source code from Prolog and/or Lisp, translates them into Java classes and has them merge together to form a single application. This is an important and exciting step. What is even important is to know what factors make this approach deployable in real world software development. The experience in this study has shed some lights in answering these questions.

First of all, our experience shows that a centralized development similar to *SourceMerge* is crucial to the success of translator-based multi-paradigm programming. It would be very difficult and frustrating if a user has to go through many complex processes to accomplish the translation and deal with the inconsistent behaviors and interfaces of the generated code. All the productivity increase and quality improvement promised by the translator-based multi-paradigm programming will be diminished by this difficulty.

Secondly, when building a centralized environment for multi-paradigm software development, many important factors should be taken into considered. Multi-paradigm means not just one, two or three paradigms to be used. It means that a centralized environment should be scalable enough to accommodate as many paradigms as possible. To achieve this goal, the design of the system should follow the Open-Closed design pattern, i.e., a new paradigm should be added to the system easily and the update on the current paradigm translation process should not affect other translation processes at all.

Thirdly, the selection of translators is the key to building a centralized multi-paradigm system. In this study, two open source translators were selected. This decision was made due to our time constraint. During our development process, we have encountered many unforeseen obstacles, such as limited documentation, unknown system requirements and missing features. Another obvious drawback with using existing translators is that the output code generated by the translators of different paradigm may not be consistent. This will definitely make the integration of the final executable very different if not impossible. We recommend that standards for the translated code should be established and the design for each translator should base on the pre-defined standards so that they can have consistent behaviors and interfaces. Although this approach requires an upfront investment to build the translators, this will make the integration of translated sources and the future extension of the system easier.

5. REFERENCES

1. R. Horspool and M. Levy. “Translator-Based Multiparadigm Programming”. Journal of Systems and Software, 25, 39-49, 1993.

2. T. Budd and R. Pandey. “Never Mind the Paradigm, What About Multiparadigm Languages?” SIGCSE Bulletin, 27, (2), 25-30, 1995.

3. D. Spinellis. “Programming Paradigms as Objective Classes: A Structuring Mechanism for Multiparadigm Programming,” Ph.D. Thesis, University of London, 1994.

4. M. Carlsson et al. “SICStus Prolog User’s Manual”. Swedish Institute of Computer Science, 2011.

5. H. Ait-Kaci. “Warren’s Abstract Machine: A Tutorial Reconstruction”. MIT Press Cambridge, 1991.

6. P. Codognet and D. Diaz. “wamcc: Compiling Prolog to C”. In 12th International Conference on Logic Programming, MIT Press, 317 – 331, 1995.

7. M. Levy and R. Horspool. “Translating Prolog to C: a WAM-based approach”. In Proceedings of the Second Computing Network Area Meeting on Programming Languages, and the Workshop on Logic Languages, 1993.

8. X. Wang, “Compiling Functional Programming Languages Using Class Hierarchies”. M.Sc. Thesis, Department. of Computer Science, University of Victoria, 1992.

9. M Banbara, N. Tamura, and K. Inoue. “Prolog Cafe: A Prolog to Java Translator System”. INAP, 45-54, 2005.

10. A. Leitão. “Migration of Common Lisp Programs to the Java Platform -The Linj Approach Linj”. 11th European Conference on Software Maintenance and Reengineering, 243 – 251, 2007.

11. D. Warren. “An abstract Prolog Instruction Set”. Technical Note 309, SRI International, Menlo Park, CA, 1983.