A Component-based Framework and Reusability in Garment*

Naixiao Zhang
naixiao@pku.edu.cn
Department of Informatics, School of Mathematical Sciences, Peking University
Ying Liu
liuying@water.pku.edu.cn
Department of Informatics, School of Mathematical Sciences, Peking University

Abstract

Garment is a mechanism for abstraction and encapsulation of languages. It aims to make the best support on the definition and implementation of new languages, especially DSLs (Domain Specification Languages). Garment originally provided a unified framework for defining languages and describing relations between languages. The framework is very convenient for defining and describing new languages. However it is not flexible enough to design some similar DSLs.

We propose, in recent work, a component-based framework for the design of DSL where software reuse is a very important feature. In this paper, the component-based framework will be briefly introduced. A conceptual analysis of reusability in Garment is also made from the different points of view and different levels here.

1. Introduction

Domain Specification Languages (DSLs) [2][6] are also called task-specific, application-oriented, or problem-oriented, such as HTML for web pages, Excel macros for spreadsheet applications, VHDL for hardware design and so on. In addition, so-called fourth-generation languages (4GLs) are usually DSLs for database applications. They have been used widely in many kinds of fields, and more and more DSLs need to be designed. How to define and implement new DSLs becomes very important. Though DSLs are similar to general-purpose programming languages, such as Pascal, Fortran, and C, they are different. DSLs are often simpler than general-purpose languages in their purposes. Without thinking of the domain knowledge, designing and implementing DSLs is easier than that of general-purpose languages because they are succinct. Then, it is possible to provide a uniform mechanism for defining and implementing DSLs. In [7], the authors had pointed out that domain-specific languages are closely related to interface language of domain-oriented software. Thus, the specification of such software can be abstracted to specifications of language systems. As a unified model to support software development and research, a mechanism named Garment for abstracting and encapsulating languages is proposed.

In [7], a systematic development method called MOSAT(Model-Oriented Specific And Transformation) is proposed. In MOSAT, the development of program, software, and the environment is regarded as three different levels. Where, the environment developers design a unified software development environment on the basis of software theory model. This environment includes a software development language (SDL) and an interpreter of this SDL. Software developers use SDL as a tool to describe their software and its interface language, then domain-specific abstract model can be built. The compiler of the interface language is generated by SDL interpreter. On the third level, the program developers (computational scientists) focus on domain problems, build their problem-solving models (programs) using the developed language. These programs are translated into the executable programs for solving of the problems by the

* This paper was supported by the National Natural Science Foundation of China under Grant. No. 69983001.

compiler. To implement MOSAT method, it is crucial to design an SDL for describing DSLs.

A framework of Garment had been discussed in [7]. Many experiments have showed that software developers can easily describe a language with it. But, if a software developer wants to develop two similar languages, he has to repeatedly do some similar works, such as describing similar tokens, statements, and expressions. The framework seems not flexible. Therefore, a new component- based framework in Garment, which makes full use of features of software reuse, is proposed in this article.

The component-based framework in Garment is used for the specifications of DSLs. Then a SDL is defined for developing software or language system. In SDL, the new component-based framework of Garment is used to define new DSL with the beginning notation — "garment". If a DSL is described in SDL, the definition of this DSL is called a garment. DSLs developers can describe a DSL with many components, such as token_component, declaration_component, expression_ component, statement_component, type_component, and program_component. All components and garments can be stored in a repository, which is called knowledge repository. While defining a new DSL, the developer can choose some components[3] even a garment from this knowledge repository and reuse them in the new DSL. In addition, language transformation is the means of implementing DSLs in SDL. A DSL developer must choose a target language for the new DSL. Transformation rules are used to describe the transformation from the description of a new DSL to the target language. The transformation rules can also be reused while defining a new DSL using SDL if the transformation is the same as before. Therefore, there are three reuse levels in SDL: To reuse a whole garment is the highest reuse level in SDL; To reuse some kinds of syntax components is the second level; To reuse the concrete syntax, for example, a special statement or type, is the lowest reuse level in SDL.

A general source-to-source program transformation system is used to implement a DSL in Garment. Software reuse is one of the most important features of transformation system. The reusability in this transformation system will be described in this paper.

DSLs developers need to develop DSLs with SDL. Then an environment which implements a SDL interpreter must be provided for DSLs developers. Of course, this environment should include some other auxiliary functions for developing new DSLs conveniently. This environment is called Garden. Because any new DSL can be developed with Garden, Garden can be regarded as a DSLs generator. Once a DSL is developed with Garden, computation scientists can describe application systems with it. This kind of DSLs can also be regarded as application generators. Software reuse is also one of the most important features of application generators. In this paper, the reusability in the application generators is also discussed succinctly.

This article is organized as follows: The new component-base framework in Garment is discussed in Section 2. In Section 3 we make a conceptual analysis of reusability in Garment. Conclusion and final remarks are given in Section 4.

2.Component-based framework of Garment

While developing a new DSL, analyzing this application domain at the beginning is necessary. After finishing analyzing, the next step is to define the DSL. The definition of a DSL includes several components, such as token_component, decl_component, expr_component, stmt_component, type_component, and prog_ component. Afterward, how to describe the DSL using a high-level specification language is one of the most important tasks. The following step is to implement the specification, i.e. to generate the DSL's processor. We can adopt the method to generate target codes directly. But in order to improve the productivity and reuse existing software, DSLs can be implemented as a source-to-source translator composed with a processor for another language [6].

According to above steps of developing DSLs, SDL provides a component-based framework for the specification of a DSL. The SDL's processor is used to produce a DSL processor from its specification. Source-to-source transformation system is used to support the implementation of DSLs. We use a set of transformation rules to translate a DSL program into the program in target language.

First, we discuss how to define a new DSL in SDL. Here, a garment encapsulates a whole definition of a DSL including its syntax and semantic. Syntatically, a garment

begin with keyword — garment, with its components indicated by token_component, decl_component, expr_component, stmt_component, type_component, and prog_component respectively. Every component includes the syntax of DSL, which includes abstract syntax, concrete syntax, and some transformation rules. Finally, a garment ends with keywords — **end garment**;.

The syntax of garment is given bellow. Boldface words are keywords of the SDL.

garment ::= **garment** id1 [**extend** id2]
[token_component]
[decl_component]
[expr_component]
[stmt_component]
[type_component]
[prog_component]
end garment;

id1 is the name of the new DSL, *id2* is regarded as *id1*'s parent language. The definition of a new DSL is correlative with its parent language.

The definition of a DSL is composed of several components. *Token_component* is used to define all lexical elements of a DSL, such as identifiers, digitals, and strings. *Decl_component* is used to define all kinds of declarations of a DSL, such as constant, variables, functions, and procedures. *Expr_component* is used to define the format of expressions. *Stmt_component* is used to define the format of statements, such as assign, procedure calling, and condition statements. These components' grammar structures are different, but they are similiar. In this paper, we will mainly introduce the syntax of *token_component*. The differences among these components will be pointed out when they appear. In addition, *Type_component* is a kind of compound component, it define all the types of systems of a DSL including the format of all types and operations with respect to them. Because of its particularity, it also will be introduces in detail.

- ***token_component***

The syntax of token_component is given bellow. For convenience, BNF is extended: The notation {}-s is used to describe a non-empty list of element separated by s.

token_component:: =
token_component [**include** token_component_name]**:**
{token_rule}-@
end [**token_component] ;**

token_component_name denotes an existing token component. It may be an independent component. It may also be the name of an existing DSL. If *token_component_name* is the name of an existing DSL, the token component of this DSL can be reused here. *Token_rule* is the most important part of *token_component*. The specification of *token_rule* takes the form:

token_rule ::=
[**aux**][rule_name][token_local] **in** syntax [==> trans]

aux is the modifier of this rule, and this modifier can be removed. When there is not any modifiers in a rule, this rule is called interface rule which can be used anywhere of any component in the garment. On the contrary, a rule is called auxiliary rule which only can be used inner the component when the rule have a modifier — **aux**. *Rule_name* is a rule's name, of course, a rule may have no name, then, *rule_name* is removed. *Token_local* makes comments of this rule's local lexical, they describe all syntax compositions and their properties which are used in the concrete syntax. Then *syntax* is the left part of a rule, it defines the concrete syntax of the DSL. The right part of this rule, *trans*, defines the semantics of the structure by means of its target language. Each rule in the definition plays a role of a transformation rule, in which, *syntax* describes the match pattern and match conditions, and *trans* describes the substitution form of target language.

Token_local takes the following form:

token_local::= {(rule_name | char_set) item_name }-,
char_set ::= **{**char_list**}** |
char_set+char_set |
char_set-char_set |
char_set ***** char_set
char_list ::= { obs_char }- | obs_char **..** obs_char
item_name ::= id

We have mentioned that there are some differences among these components. Now let us study the following syntaxes of decl_rule, expr_rule and stmt_rule:

decl_rule::=[**aux**][rule_name] local **in** syntax [==> trans]
expr_rule::=[**aux**][rule_name] local **in** syntax **return** type
[==> trans]
stmt_rule::=[**aux**] [rule_name] local **in** syntax [==> trans]

Obviously, they are different to token_rule. Firstly, the defining of local variables is not same. The above three rules have the same syntax in defining their rules' local variables:

local ::= {item_decl}-,
item_decl ::= kind item_name [**:** type]
kind ::= **decl** | **expr** | **stmt** | **type** | rule_name

In addition, the concrete syntax of expr_rule makes it clear that the concrete syntax of any expression should have return type.

- ***type_component***

Type_component is more complex than other components. It is used to define all types of a DSL. The syntax of it is given as follows:

type_component::=

type_component [**include** type_component_name] **:**

{type_def}-%

end [**type_component**] **;**

Every type_def define a type, include its syntax and some operations relative to it. A type_def includes the definitions of literals, operators, expressions, functions, and procedures. The definition of type_rule is given as following:

type_def::=

type type_name **repr** rep_type [**with** impl_mod_name]

[comment]

[**literals:** token_rules]

[**operatiors:** op_rules]

[**expressions:** expr_rules]

[**functions:** func_rules]

[**procedures:** proc_rules]

end [abs_type_name]

In the above definition, *abs_type_name* is the name of a type. It is defined by *type_def*, and *rep_type* is the representation of this type in the target language. In addition, there must be some relationship between *abs_type_name* and *rep_type*, and it is illustrated by the invariant, which is included in the comment. The other parts of type_def, such as *toke_rules*, *op_rules*, *expr_rules*, *func_rules*, and *proc_rules* are similar to that of the other components. There is another composition after with, *impl_mod_name*, in type_def. *Impl_mod_name* is the name of an existing module, which has been implemented in the target language. This module can be a package that is implemented by Ada, a class implemented by C++, or a file implemented by C. Any function that was defined in this module can be used in the compositions of *type_def* directly. A type that is defined in a *type_component* is similar to a class. So that it is very convenient to be inherited and extend an existing type.

While developing a domain application system, the development process is divided into two separate stages. At the first stage, the application domain is analyzed. And the corresponding DSL is designed and defined using the SDL. The DSL processor can be produced automatically. At the second stage, the application system is developed using the DSL. This approach is efficient for the application domains if there are many application systems to be developed with low cost. Obviously, once a suitable DSL is implemented, the second stage will become considerable easy.

3. Reusability in Garment

Software reuse [1] is regarded as a potential powerful means to improve the practicability of software engineer. Garment as a new mechanism of abstraction and encapsulation for languages is provided mainly because of software reuse. Just because of the reusability in Garment, it is very convenient and flexible to design domain application system.

On the one hand, SDL provides a new component-based framework. This component-based framework provides language or software developers with several kinds of software reuse levels. On the other hand, garments that were defined in SDL are implemented on the basis of transformational system. However, transformational system is one of the most important application cases of software technique reuse. Thereby we will discuss the reuse ability in Garment mainly because of the above two aspects. We have implemented a Garden that can be regarded as a kind of application generator. Application generator is also one of common application cases of applying the technique of software reuse. A conceptual analysis of reusability in Garment is the main topic of this section.

3.1 Reusability in Defining DSLs

Software reuse is an important target in software engineering. Rickard[6] had pointed out that abstraction was regarded as an important part of software reuse. Therefore, how to improve the abstraction level while defining a garment is one of the most important targets. SDL provides a component-based framework for defining DSLs. This component-based framework allows users to define DSLs in a high abstract. While defining new DSLs, some features of existing garments can be inherited. Then heavy work to define a new DSL from scratch can be avoided.

There are three reuse levels while defining a new DSL. The first level is to reuse a whole garment. It means that DSLs developers can reuse all features of a garment. Then the reused language is the new DSLs' parent-language. This level can be regarded as the highest reuse level. The second level is to reuse some components of a garment or some independent components. Because some DSLs may have

same parts, but these DSLs usually are not completely same. Then it is not necessary to rewrite all the components of new DSL. The developers can only inherited one or more components of a garment. DSLs developers also can reuse some independent components, which do not belong to any garment. The third level is to reuse some concrete composition of a component, for example, some rules in a component can be reused while defining the same component or other similiar components. This level is regarded as the lowest reuse level.

• ***To reuse a whole garment***

To reuse a whole garment is the highest reuse level in three levels. After a garment is defined, it is stored in knowledge repository. When a user wants to define a new DSL, he can search for an existing garment that the new DSL could be defined on the basis of it from the knowledge repository. If there is an existing garment that can be reused, the user can reuse the whole garment. So, the language that the garment has defined will be regarded as the new DSL's parent-language. So the new DSL is regarded as the parent-language's child-language. In the definition of garment in the section 2, id2, which locates behind "extend", is the name of the language, which will be reused by the id1. So id2 is id1's parent-language. Reusing a garment means reusing all the components of this garment. A small example about it is given as follows.

```
garment CALCULATOR_1 extend CALCULATOR
      type_component:
          type cal_1 repr Float
            leterals:
cal_1_literal num wh, num frac, sign s in [s] wh "." frac
            operators:
                  "/" i:cal, j:cal       return cal_1
                  ==> "Float(" i ")" "/" "Float(" j ")"
                @ ……
            …..
           end
      end type_component;
      prog_component:
            program_1 expr r:cal_1 in "go" r ==> ……
      end prog_component
end garment;
```

A new language CALCULATOR_1 is defined, and it inherits all features from its parent-language — CALCULATOR, which had been defined and existed in the knowledge repository. Then CALCULATOR_1 includes all components of its parent-language besides some new defined type, cal_1, in its type_component and new prog_rule, program_1, in its prog_component.

• ***To reuse some components***

If a garment can be reused by a new garment, the task of defining a new DSL becomes easy. However, some DSLs are not same completely. It is not necessary to reuse all the components of a garment. Then the component-based framework of SDL permits DSLs developers to reuse some components of a garment or some independent components. To reuse some components is the second reuse level, and component is another reusable unit.

There are some garments and independent components in the knowledge repository. While defining a new DSL, the developer can search for the new DSL's parent-language from the knowledge repository firstly. If the parent-language does not exist, he can search for a garment which includes components that can be reused by the new DSL. If the new garment reuses some of the components of a garment, the developer only needs to add the garment's name behind of a keyword — include, at the beginning of the definition of the new defined component.

DSLs developers also can search for some independent components from the knowledge repository, which can be reused by the new garment. If an independent component is reused, the independent component's name is added behind of the keyword — include, at the beginning of the definition of the new defined component.

The following example is about how to reuse a component of a garment.

```
garment CALCULATOR_1
      type_component include CALCULATOR:
          type cal_1 repr Float
            leterals:
      cal_1_literal num wh, num frac, sign s in [s] wh "." frac
            operators:
                  "/" i:cal, j:cal       return cal_1
                  ==> "Float(" i ")" "/" "Float(" j ")"
                @ ……
            …..
           end
      end type_component;
      ……
end garment;
```

CALCULATOR_1 only reuse CALCULATOR's type_ component not the whole garment. This reuse level of components makes it more flexible to define a new DSL because we usually only need to reuse some of the components of a garment not the whole garment.

• To reuse some concrete compositions of a component

To reuse some concrete compositions of a component is the lowest reuse level. Sometimes it is not necessary to reuse the whole component. Then the component-based framework supports to reuse some concrete compositions of a component mainly referring to some rules. The following example is about type_componet.

```
type_component:
    type rational repr "type ration_number is"
                       "record"
                       "num, denom: integer;"
                       "end record;"
     literals:
          ……
     operatiors:
       "+"      x: rational, y: rational return rational ==>
  "(" x ".num" "*" y ".denom" "+" y ".num" "*"
 x ".denom" ")" "/" "(" x ".denom" "*" y ".denom" ")"
    @ ……
     expressions:
       id_expr id r: rational in r return rational ==> r
    @ neg_expr   expr r: rational in "-" r return rational
        ==> "-" r
    @ ……
     end
end;
```

expr and *id* both have been defined in token_component and expr_component. In this definition of type, they are reused directly not defined repeatedly.

While developing new DSLs using Garment, developers only need to concentrate on how to describe the new DSL using SDL not the detailed implementation, which will definitely improve the productivity of new DSL. [1] have pointed out that software is developed in two phases with transformational system: Software developers describe the semantic behavior of a software system using a high-level specification language; Software developers then apply transformations to the high-level specifications. At the first phase, software developers create an executable system in a language that has relatively small cognitive distance from the developer's informal requirements for the system. SDL is a specification language for the first phase of the transformational system. The reusability of SDL will definitly improve the productivity of DSLs.

3.2 Reusability in Implementing DSL

After defining a DSL, how to implement is another important topic. In the component_based framework, a general source-to-source program-transformation system is used to support the implementations of DSLs[5]. User can choose an existing target language for new DSL. The chosen language is called the implementation language, which serves as the target of the transformation.

As a transformational system, software reuse is the most important feature. We will analyze the reusability in the transformational system from the points of view of reusable artifacts. The following three sections describe three different kinds of reusable artifacts in the transformation system.

• A whole target language

While language developers implementing a language from scratch, they must experience a process of lexical analyzing, parsing, syntactic and semantic error analyzing, and code generating. Obviously, it is very complex and difficult for developers. Then, a method of providing a general interpretor for implementing new DSLs is adopted with Garment. Programs written in new DSLs are transformed into programs in target language. Just because of adopting transformation method, the complex process is avoided. While developing a new DSL, we can choose an appropriate existing language as the target language firstly. Secondly, developers make an abstract description for the new DSL and translate it into the target language. In fact, to reuse an existing language is equal to reusing this language's compiler.

• Primitive structures of a target language

Once a DSL is implemented, the primitive structures of its target language can be reused while application system developers developing new application systems in the DSL. In addition, these structures also can be reused to design higher level structures. Then, there are two possibilities to reuse some primitive structures of the target language.

First, a DSL can embed some of these structures without modifying. Thus, users can use the structures in their DSL programs directly. While describing new DSLs, none of any transformation actions need to be implemented. Obviously, this reuse level is provided for the DSL programmers. An example is given below.

stmt_component:

Proc_call ident name **in** "call" name

@ ……

end stmt_component

The structure of calling statement to procedure remains unchanged. DSL developers need not redesign the structure.

Secondly, some structures can be reused to implement higher level structures of a new DSL. This kind of usage is not obvious, but very important. This reuse level is also provided for DSL developers. An example is also given as follows:

stmt_component:

enum_ass variable a, **expr** b **in** $1{a}-"," ":=" $1{b}-","
==>#1:"declare"
$1{"task" **temp0** ";"
"task body" **temp0** "is"
"begin"
a.#2 ":=" b.#1";"
"end" **temp0** ";"}
"begin null; end;"
"declare"
$1 {"task" **temp1** ";"
"task body" **temp1** "is"
"begin"
a.#1 ":=" a.#2";"
"end" **temp1** ";"}
"begin null; end;"
#2: $1{b.#2}
#3: $1{a.#3 ":=" a.#2 ";"}
$1{a.#2 ":=" b.#2 ";"}
$1{"if" a.#3 "/=" a.#2 "then return False; end if;"}
@ ……
end;

This example describes how to implement multiple assignment statement in a parallel language. When a statement is an enumerated assignment, every sub-variable is assigned in parallel. Then the new DSL needs to implement the parallelization. In the above example, it is clear that the new DSL don't provide any parallel structures. Because Ada95 is chosen as the new DSL's target language, and there is a fixed grammar structure, task, which can describe parallel semantics. Then this existing structure is used to implement parallel processing for the new DSL programs. Although this kind of reusability for the primitive structure of the target language is not obvious, it is still very important.

- **Existing sub-program:**

This reuse level can be introduced from two different points of view. First, existing sub-program in target languages, such as user-defined procedures, functions, data-types, etc., can be reused as usual. In the specification of a DSL, one can ask for the use of those existing target-language level entities, as building blocks for the implementation of the DSL. This reuse level is provided for DSLs developers. An example about it is given as follows:

type Comx **with** Complex
literals: …….
operators:
"+" x:Comx, y:Comx **return** Comx ==> x "+" y
@ ……
end

Complex is a defined type in Ada95, which is the new DSL's target language. Many declarations and operations, such as "+", "-", and "*", about this data type have been defined in Ada95, and all those definitions have been put into a program package whose name is Complex. Then, the op_rule can use any of Complex's operation, "+".

Secondly, a DSL developer and programmers can define many sub-programs, and put them into the DSL's functions library. Then other programmers can reuse those existing sub-programs while programming. This reuse level is the same as usual reusability in general-purpose languages, and it is for programmers.

In fact, the reuse level is more than that we have mentioned before, for example, a newly developed DSL can be chosen as the target language. Once a new DSL is chosen as a target language, it means that another reuse level appears.

In order to provide DSLs developers with an environment for developing new DSLs, we have set up a Garden. The software development process in Garment approach can be divided into three classes:

- The design of the SDL and development of a Garden;
- Analyzing application domains, then defining and implementing a new DSL;
- Developing domain application systems in a DSL.

Garden can be regarded as a DSLs generator. For DSLs developers, it is clear that DSLs' specification is separated from its implementation. While developing a new DSL, making a specification of the new DSL is the only thing. On this level of abstraction, it is possible for even

non-programmers to be familiar with concept of an application domain to create DSLs. All DSLs developers can reuse the global system architecture, major subsystems within this global architecture, and very specific data structures and algorithms [6]. This reuse level is provided for DSLs developers.

DSLs developed in the second step can also be regarded as an application system generator. Once a DSL is generated, programmers can develop domain application systems with it. This application system generator is similar to the traditional programming language compiler. However, it differs from traditional compilers in that the input specifications are typically very high-level, special-purpose abstractions from an application domain. This reuse level is special for domain application system developers.

4. Conclusion

Garment is a mechanism for abstraction and encapsulation of languages. It aims to support the definition and implementation of new languages, especially DSLs. However, we think that Garment is mainly used for DSLs because DSLs are usually simple and succinct. It is well known that DSLs are used more and more widely. Moreover, some people think that design and effective implementation of DSL will become an important field in the near future. It is necessary to provide an effective mechanism to design DSLs. Garment plays an important role in defining DSLs. In addition, the correctness and effectiveness are both important for a DSL. Therefore, the two properties may be guaranteed through the type checking and optimization in Garment.

The component-based framework of Garment is a progress because of reusability. For defining DSL, this framework supports several reuse levels. DSLs developers can conveniently and flexibly describe new DSLs by defining some components and inheriting some features from its parent-language, some independent components, and some concrete compositions of components. Transformation system is used to implement DSLs. Reusability is an important feature in transformation system. In addition, Garden can be regarded as an application generator. It is well known that reusability is also an important feature of application generator. In a word, this component-base framework is a product of reusability. Although we don't give formal definition of the reusability in Garment like [4], a conceptual analysis has been made in this paper.

In this paper, the component-based framework and reusability were discussed mainly from the point of view of being used in Garment. In fact, this idea is common to sofeware engineering. We have pointed out before that domain-specific languages are closely related to interface language of domain-oriented software. Thus the specification of such software can be abstracted to specifications of language systems. While developing a new domain software system or language system, it is very important to reuse some existed component. It is also necessary to provide a framework for the users. The component-based framework introduced in this paper is an special case for impertive language development. However, it can be extended to object-oriented language and others specification language easily. We are doing something about it.

5.References

[1] Charles W. Krueger, Software Reuse, ACM Computing Surveys, Vol. 24, No. 2, June 1992, P131-183.

[2] David Atkins, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, Kenneth Rehor, Experience with a Domain Specific Language for Form-based Services. In USENIX Association, Conference on Domain-Specific Languages, P37-49, October, 15-17, 1997.

[3] P.A.V. Hall, Architecture-driven Component Reuse. Informational and Software Technology 41(1999) P963-968.

[4] Rym Mili, Jules Desharnais, Marc Frappier, Ali Mili, Semantic Distance Between Specifications, Theoretical Computer Science 247(2000), P257-276.

[5] Zhang Naixiao, A Notation of Program Transformation in Programming Languages — on Transformation Programming Languages. Journal of Software Transaction, Vol. 4, No. 5, P17-23, October, 1993.

[6] Rickard E. Faith, Lars S. Nyland, and Jan F. Rrins, KHEPERA: a system for rapid implementation of domain specific languages. In UNENIX Association, Proceeding of Conference on Domain-Specific Languages, Santa Barbara, California, P 243-255, October 15-17, 1997.

[7] Zhang Naixiao, Zheng Hongjun and Qiu Zongyan, Garment — A Mechanism for Abstraction and Encapsulation of Languages, ACM SIGPLAN Notices, Vol. 32, No. 6, p53-60, 1997.