Garment

---A Mechanism for Abstraction and Encapsulation of Languages

Zhang Naixiao[1], Zheng Hongjun[2] and Qiu Zongyan[3]

(*Peking University, Beijing 100871, China*)

Abstract Domain-specific languages are closely related to interface languages of domain-oriented software. Thus, the specifications of such software can be abstracted to specifications of language systems, and implementation of such software can be abstracted to implementation of the language systems. As a unified model to support software development and research, a mechanism named *Garment* for abstracting and encapsulating languages is proposed. *Garment* provides a unified framework for defining languages (syntax and semantics) and describing relations between languages (which are classified as inheritance, shielding and extension). Finally, an experimental environment, which supports software development with Garment, is introduced briefly.

Keywords Garment Software development methodology
Software development language Software development environment

1 Introduction

Languages are tools for thinking and expressing. *Domain-specific languages*, i.e., the language of mathematics and the language of chemistry, are formed in the disciplines. They are powerful in describing domain-specific objects and processes, and reflect the research states of the discipline.

Due to the development of computer science and technology, *computation* has become another important means in scientific research, besides theoretical and experimental approaches. Many new branches of sciences have appeared and form a new trade, called *computational science*. Examples are computational physics, computational chemistry, computational mechanics, and computational linguistics. The flourishing of computational sciences comes from the fact that computational models of most research problems can be built on computers. Computational scientists deal with domain-specific problems with computer. It is desirable for these scientists to have powerful software for modeling objects

[1] Dept. of Information Sciences, School of Math. Sciences, Email : naixiao@pku.edu.cn

[2] Dept. of Computer Science and Technology,

[3] Dept. of Information Sciences, School of Math. Sciences, Email : zyqiu@pku.edu.cn

and processes in their fields. The systems should be efficient *programming environments* in building models for solving practical problems. It is more convenient if the models could be described directly by domain-specific languages. To the designers of domain-oriented software, the ideal goal is to provide an abstract machine on which the domain-specific language can be used directly (in some sense). In that case, the systems act as *virtual-reality* environment for the domain-scientists. Under this ideal situation, the relations between the discipline and the software, the domain-specific language and the interface language of the software, the domain-related research work and the programming are shown in Figure 1.

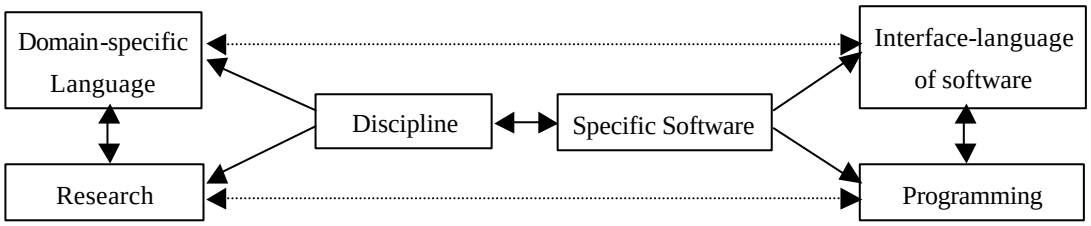


Figure 1. Discipline and Software

In this paper, we will focus on the domain-oriented software with programmable interface languages. To form a unified view and model for such software and its development, the nature of software is studied, and a new model for software development is formed.

2 MOSAT - A Systematic Development Method

For supporting development of domain-specific software, we propose a systematic development method called **MOSAT** (**M**odel-**O**riented **S**pecification **A**nd **T**ransformation [1]). In MOSAT, the development of program, software, and the environment is divided into three levels, as shown in Figure 2. The environment developers, taking views of software theory, build a unified software development environment. This environment includes a software development language (SDL) which is suitable for defining software and an interpreter of SDL. Software developers use SDL as tool to describe their software and its interface language, forming domain-specific abstract model. The compiler of the interface language is then 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 models are translated by the compiler into executable programs for solving of the problems.

In MOSAT, the concept "model" introduced in VDM [11] is extended, and divided into general models of software, abstract models of domains, and special models of practical problems. Theory and implementation of these models can be studied separately. In MOSAT, abstract data types (ADT) are the basic units in defining languages, and the programming language is fundamental objects in software research. The specification of software is

abstracted to the specification of language-system. The relations between languages are classified into three categories: *inheritance*, *shielding*[**] and *extension* [6]. Data abstraction is enhanced to *language abstraction*. As the type definition language of Polya [8,9], abstract and concrete grammar, syntax and semantics definitions are separated in our SDL, which will benefit *software reuse* [15] and *partial implementation* [14]. Referring to *transformational semantics* in CIP [13], both of the definition and implementation of languages can be described in SDL in unified form. This can not only improve the brevity of languages but also decrease the inconsistencies in development.

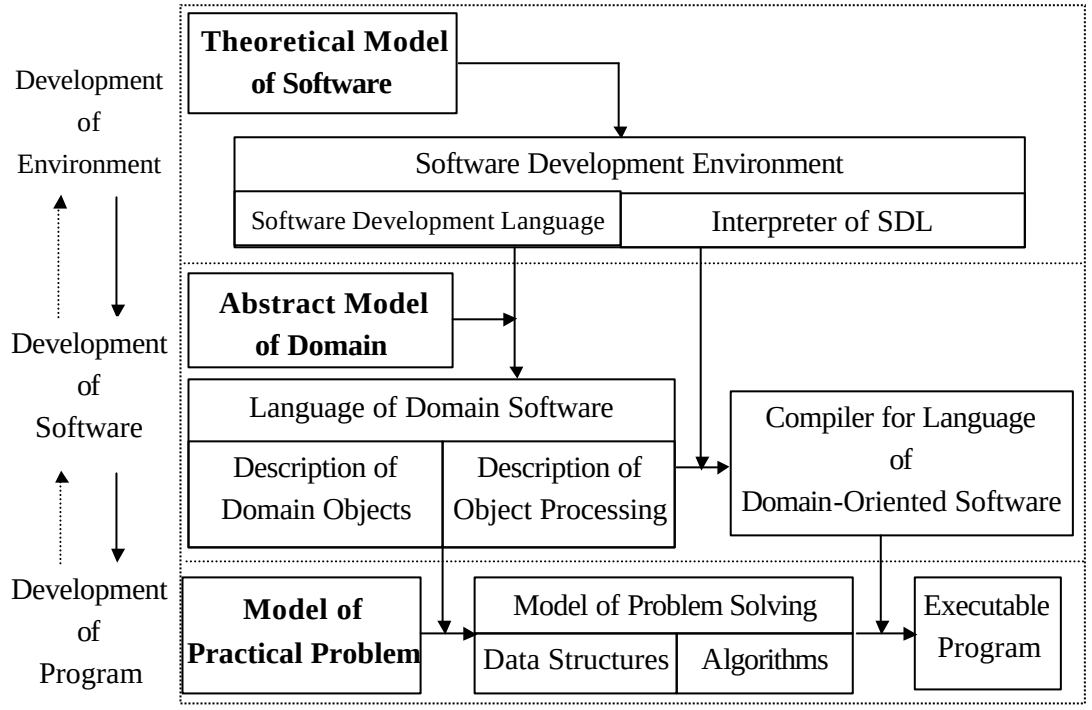


Figure 2. Three different levels in MOSAT

MOSAT supports software reuse in many facets. For productivity, all languages developed in the environment may be organized, according to inheritance relations, into a language family, and stored in a *Language Knowledge Base* (LKB) [4]. In developing a new software system, a suitable language in LKB can be chosen as the *parent language* to define the new language of the software, by ways of inheriting, shielding and extending. The semantics of extension parts can be defined by transformation rules. The inherited parts can be redefined, when necessary. All of the description can be thought as a specification of the new software. Afterwards, the software, including the complete definition of the interface language and the internal implementation of facilities, can be generated by the environment

[**] Shielding is a special case of inheritance, namely, partial inheritance. Language L1 shields some constructs of L2 means that these constructs are not usable for users of L1, but are usable for implementers of L1.

automatically. In this approach, the development of software could be think as "*specification + transformation*".

The example in figure 3 illustrates something under MOSAT. In the figure, the base software of the environment provides language L0. L0 is usually an implemented language (such as C or Pascal). It is called *kernel language* in the language family. The base software provides a programming environment for L0. A software for theoretical mechanics (for example) can be built; its language L1 is defined upon L0. Then, the software for elastic mechanics and fluid mechanics can be developed, which provide language L2 and L3. As an alternate, the software for fluid mechanics could also be developed on the elastic mechanics software. In this case, the parent language of L3 is L2. In choice of the parent language, two factors are important to consider. From which language the newer can inheritance more, i.e., reuse more existing components and implementations. From which language the new extensions can be implemented more easily and efficiently.

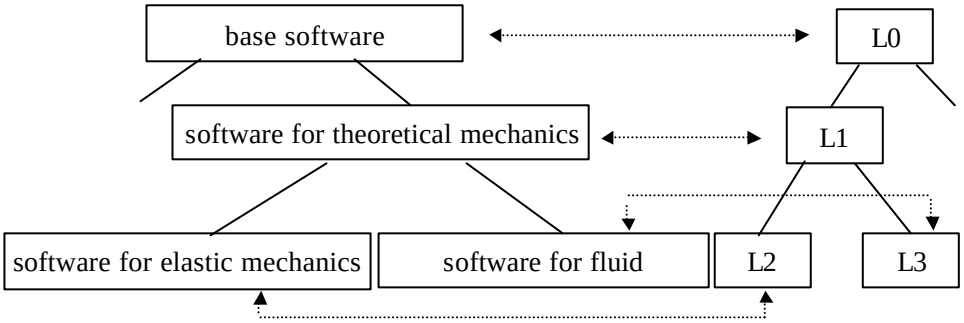


Figure 3. An Example

3 Garment — A Mechanism for Abstraction and Encapsulation of Programming Languages

To implement MOSAT method, it is crucial to design an SDL for describing software of various domains. The SDL should have mechanism to indicate parent language selection, describe inheritance, shielding and extension regarding the parent language, and to express transformational semantics for new extended constructs and modified inheritance constructs. A mechanism called **Garment** is defined in our SDL. A Garment could be think as an abstraction and encapsulation of a language. We discuss the conceptual framework of Garment here.

Syntactically, a Garment begins with keyword **garment**, with its specification and implementation parts indicated by **spec** and **impl** respectively. In the **spec** part, language constructs are defined with abstract grammar and concrete grammar. The part of abstract grammar specifies grammatical properties—such as kind, type, and scope —possessed by the basic constructs of the language. The concrete grammar part specifies concrete

representations of these constructs as used in real programs. Language constructs are grouped into two catalogs: control structures (represented by structure statements) and data structures (represented by abstract data types). The transformational semantics of language constructs are defined in the **impl** part. Transformation rules are used to define the semantics of structure statements and abstract data types of the new language with respect to its parent. A key point is that the parent language is not only part of the specification language for the semantics of the new language, but also its implementation language.

The syntax of Garment is given bellow. For convenience, BNF is extended: the notation [] is used to describe non-empty list, []-*s* a non-empty list of element separated by *s*, {} an optional entity. Boldface words are keywords of the SDL.

```
garment ::= garment  id1  from  id2          /*  head  */
            spec                             /* specification */
                 import  inher_part ;
                 [type_def  |  struc_def ]-;
                 {with  op_list}
            impl                             /* implementation */
                 [trans_def  ]-;
            end  id1
```

Where *id*1 is the name of the language to be defined, *id*2 is name of the parent language. The specification of inheritance relation takes the form

```
inher _part ::= [id3]-; | all { except  [id3]-, }
```

where *id*3 are the names of constructs, that is, names of structure statements and abstract data types in the parent language. Listing constructs are inherited (this means that other constructs are shielded to users of this language). All of the parent language can be inherited (**all** is for this), also can only list names to be shielded behind keyword **except**.

The syntax for description of new abstract data type and structure statement is as:

```
type_def  ::= type  id4          { '(' parameter_part ')' }
              { literal            [  literal_part  ]-;  }
              operation            [ operation_part ]-;
struc_def ::= struc  id5         { '(' parameter_part ')'}
              [  struc_part  ]-;
```

where *id*4 and *id*5 are names of constructs. *Parameter_part* describes type parameters or length parameters of constructs. For example, qualified type "stack" can be represented by *stack*(**t*), where **t* is a type parameter. *Literal_part* and *operation_part* describe grammar of constants and operations of type *id*4. The operations can be defined by procedures, expressions, operators, or statements. The *Struc_part* is used to define structure statements, including their constructs. The details of these definitions take a way

similar to type definitions in Polya language[9].

At the end of specification part, keyword **with** introduces an *op_list* to specify precedence and associatively of operators of language *id*1. The semantics of language constructs is specified by a set of transformation rules named *transform* [8]:

```
trans_def ::= transform    id3    { '(' parameter_part ')' }
              {    repr    type
                                /*  represent  invariant  */  }
                   rule    [rule]-®
```

A transform describes the semantics of an abstract data type or a structure statement of the new language (child language) with regard to its parent. Semantics of all extended constructs must be defined by transforms. For inherited constructs without a transform, the original semantics is inherited. For new extended abstract data types, it is necessary to introduce a representation type, following **repr** in the transform. Representing invariants are recommended as annotation to describe relations between new types and their representing types.

A rule has two parts which are separated by a key word **repl**. The left part is a pattern (for the child language) that must be satisfied when the rule is applied to make a transformation, right part is its replacement (in the parent language). For details of definitions and effects of transform and transformation correctness, please refer to [2][3].

Here is a simple example, where L0 is the parent language and child language L1 inherits all constructs of L0 and extends it to L1 with stack type *stack* and a loop structure *loop*:

```
garment  Li  from  Lj        /*  head  */
spec                          /*  specification  */
import   all ;
type       stack(*t)
  operation
    makeempty ( s:stack (*t) ) ;
    push ( e:*t, s:stack (*t) ) ;
    pop ( s:stack (*t) ) ;
    top ( s:stack (*t) ) *t ;
    isempty ( s:stack (*t) ) BOOL
struc      loop
   loop   {[stmt-s]-;} exp-e {[stmt-t]-;}   (e:BOOL)      as
          "LOOP"{ [s]-;} "WHEN" e  " EXIT " { " ; " [t]-; } " ENDLOOP "
impl                          /*  implementation  */
  transform    stack(*t)
    repr     RECORD
               a: ARRAY (*t, 100) ;
               i: INT
             END
```

```
        rule
          makeempty(s)   repl     s.i:=0
        ® push(e,s)                    repl     s.a[s.i], s.i:=e, s.i+1
/*precondition s,i<100*/
        ® pop(s)          repl     s.i:=s.i-1              /*precondition  s,i>0   */
        ®  top(s)         repl       s.a[s.i]
        ® isempty(s)                 repl                    s.i=0
      transform     loop
        rule
         LOOP [s]-; WHEN e EXIT; [t]-; ENDLOOP
                              repl VAR temp:BOOL;
                                 temp:=true;
                                  WHILE  temp   DO   [s]-;
                                                IF  e  →   temp := false
                                                       not(e) → [t]-;
                                                FI
                                                OD
        ® LOOP   WHEN e EXIT; [t]-; ENDLOOP
                 repl WHILE not(e) DO   [t]-; OD
        ® LOOP   [s]-;   WHEN e EXIT    ENDLOOP
                 repl   REPEAT   [s]-; UNTIL   e
        ® LOOP    WHEN e EXIT    ENDLOOP
                 repl   REPEAT   SKIP   UNTIL   e
        end        Li
```

Type *stack* is represented by a *record* as usual. The operations of stack are defined as procedures. Designers can use other representation types and other ways, such as operators, expressions, or even statements, to define the operations according to their needs and taste. The definition of *loop* statement follows D. Knuth's paper on GOTO [16]. Before keyword **as** is the abstract grammar of *loop*. Where the two parts *s* and *t* are optional statement sequences and *e* is a logic expression. Following the **as** is the concrete grammar. A transform named *loop* describes the implementation of *loop*, in which four rules make possibly for the transformation system to select the best one according to the presence of statements. The assumption in the example is that types BOOL, RECORD, and statements IF, WHILE and REPEAT are all constructs of language L0.

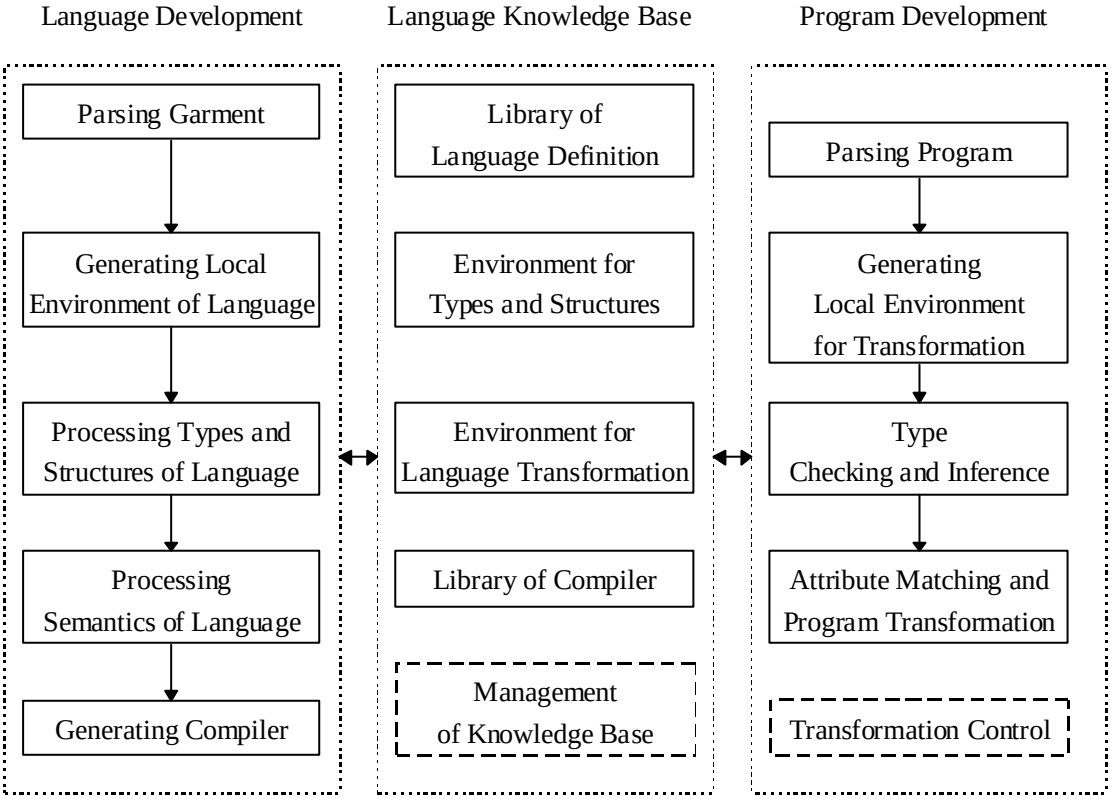
4 Garden - An Implementation of Garment

We implemented a **Gar**ment **d**evelopment **en**vironment named **Garden**. The language knowledge base (LKB) in Garden supports building of the language family organized as a tree. The language family focus to application domains and could have various abstraction levels [4]. People can select an existing language from LKB if it is good enough for solving

their domain problems. Otherwise, they can develop their suitable language with Garment, based on some existed language in LKB.

Garden implements two major functions. One is to implement languages specified by SDL. The other is to transform abstract models of problems described in a language in the LKB, into the kernel language. The latter amounts to program implementation.

Figure 4 illustrates the architecture of Garden. All the syntax and semantic information of existing languages are stored in the LKB. Language definitions are stored as text for reference. Constructs of languages, underlying relations (inheritance, shielding and extension), type and structure environment of languages, the language transformation environments are all stored in inner representation for language development and program development. The compilers generated for languages are stored in a separated compiler library.



Firgure 4. Architecture of Garden

In language development, the definition of a new language is first parsed. With the parent language in the LKB, the system generates an environment for developing the new language. In addition, **Garden** deals with new types and structures in the language, updates the environment for the types and structures in the LKB. It performs transforms, updates the environment for language transformation, and generates a compiler of the language.

Supported by LKB, the program development system parses the user programs, forms local environments for the model transformation. Furthermore, the system makes type

checking by attribute matching and program transformation, and generates the final program.

Figure 5 shows the work flow in **Garden.** The Garment interpreter is the heart of the system. The specification of a language L_{ij} for some domain is represented by Garment G_{ij}. It is interpreted by the Garment interpreter. Information of the parent language in LKB is added to results of the interpretation. The block of language knowledge is formed and stored into LKB. The lexical analyzer and parser of language L_{ij} are formed in the interpretation. Joining with some general program modules, through compiling and linking, a compiler of language L_{ij} is generated. In program development, if solution of a practical problem is described in L_{ij}, forms a program P_{ij}, the compiler of L_{ij} is called to compile P_{ij} into an equivalent program P_i in parent language L_i. This process is repeated until a kernel language program P_0 is formed.

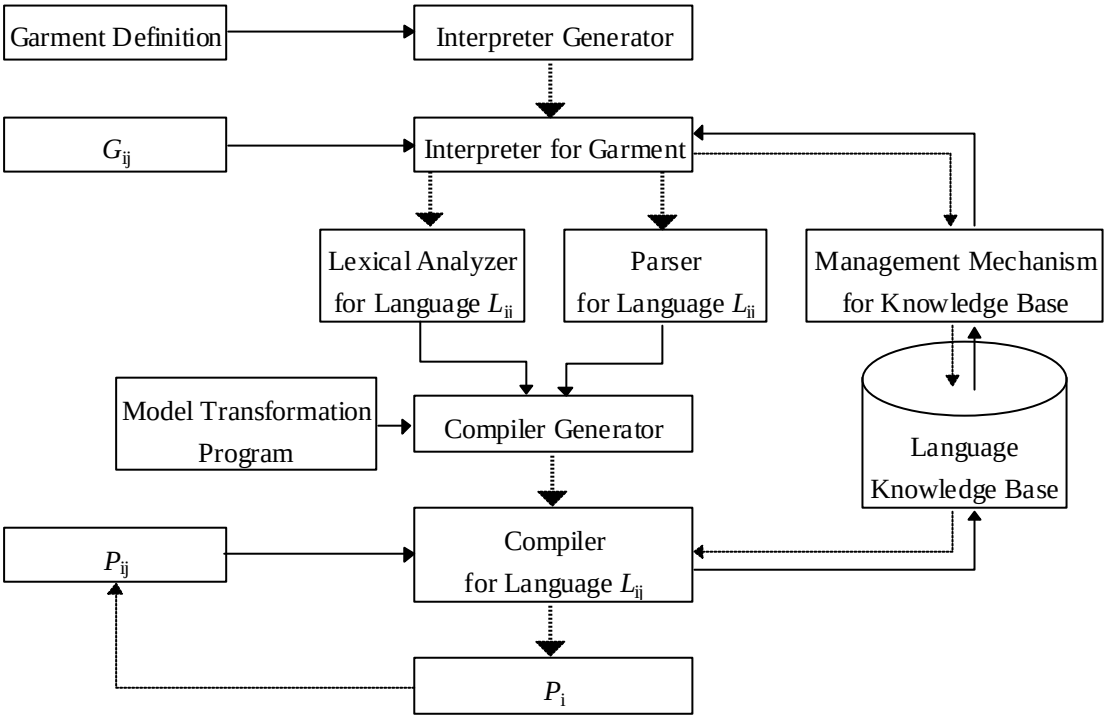


Figure 5. Work Flow of Garden

General speaking, there are two major ways to improve software quality in Garden. First, partial implementation [4] can improve representations of important types and implementations of common operations in these types. Second, using of existed definitions and implementations in LKB can reduce costs of software development.

Garden is only an attempt for testing Garment mechanism and supporting the MOSAT method. With it, we have developed by self-application a program transformation system and modeled a telephone exchange system. The feasibility and effectiveness of this method are verified [5]. Afterwards, a Pascal-like language named Poly0 is defined as the kernel

language. Using Garment, Poly0 is augmented to language Poly1 with types linked list, tree, stack, and pair. Binary tree traversing and heap sorting are implemented in Poly1. Furthermore, language Poly2 is defined on Poly1, with set, sequence and bag etc. The Huffman algorithm and a prototype of library management system are implemented in Poly2. Poly0, Poly1 and Poly2 and some other languages construct a small language family. All of these are beneficial explorations in developing software with MOSAT method and accumulating experiences for using and improving Garment.

5 Software and Langauge

In views of its architecture, *Garden* may appear to be simple. It reflects our ideas on software. The statements and results in this section should be taken with a grain of salt. They reflect our concern for explaining and reconciling the ideas. It is hoped that they provide some insights leading to more substantial work on the questions.

The software crisis motivated computer scientists to dig into the field. In the last twenty years, many inspiring results have been developed. Nevertheless, the software crisis has not disappeared yet. In our opinion, one of the important reasons is that the concept, software, has not been distinguished clearly from "program". As the result, occur often chaos in dealing with *software development* and programming, *software development language* and programming language, *software development environment* and programming environment etc. In the language research area, this problem appears as mixing *extending definitions* and *declarations* in languages. Meta-features have tended to be included into programming languages that increase the complexity of programming and affect the efficiency of the implementation.

The word *software* refers to a larger entity than a program. Software, such as a control system, application system, or utility tool, refers to systems developed for users. In software development, people pay attention to building abstract computation systems for specific domains and abstract models. For program, they concern on suitable algorithms, data structures, and their implementations. Research on software concentrates on usefulness, flexibility, reliability and robustness of the models (and final systems), while research on programs focuses on correctness and efficiency.

Software systems are in fact abstract machines for dealing with problems of certain kinds. A software system implements a new level of computation facilities. Consequently, it forms a new face between computer and users as a "*garment*" draped over the original computer system. Problem-solving models for specific problems can be constructed on it.

Programming language systems (compiling systems) are the earliest successful software. High-level languages are abstract models of programming. The development of language systems makes computer "understand" high-level languages and provides better programming environments. They are very important in the progress of programming science. In the case of computational sciences and compiling systems, the differences between

software development and programming are obvious. They are similar to the differences between building a lab and doing experiments in the lab (in *experimental sciences*), or creating an axiom system and proving theorems in it (in *theoretical sciences*).

It is the time to develop separated *software science* apart from programming to some extent. In this way, we can concentrate on the nature of software in itself and the laws of software development. The general approach of computational sciences is also good for software science, which may lead to some approaches in which the software development could be treat as what was done in programming [7].

The framework introduced in this paper is only a first step toward the goal. In this work, we analyzed the theories and techniques about typing, formal semantics, program transformation, data abstraction, software reuse, partial implementation and so forth. Many of our ideas are affected by Polya, VDM, CIP and Ada. We also have some discussion of a mathematical model of Garment [6] and built an abstract model for programming languages. The model is used to describe the semantics of language inheritance, shielding, and extension, and to show the inherent connections between languages in a language family.

Our further researches will concentrate on properties of the kernel language and the semantics of language transformation. Another interesting direction would be uniformity of language extension.

Acknowledgments

The authors wish to thank Prof. Wu Yunzeng, Prof. Xu Zhuoqun and Prof. Qu Wanling for their support and their valuable interaction about the research work of *MOSAT*. We are especially thankful to Prof. David Gries and Prof. Dines Bjørner for taking a great interest in this paper and suggesting many improvements in its exposition. Finally, we have benefited greatly from Chen Guang, Song Jin, Cong Xinri and Chen Zhongmin's partly implementation of *Garden* and the development cases.

References

[1] Zhang Naixiao, Xu Zhuoqun and Qu Wanling. On Model-Oriented Transformational Software Development Methodology. (in Chinese) *Theoretical Computer Science*, edited by Wu Wenjun, *vol*.2 (1994), 54-64.

[2] Zhang Naixiao. Notation of Program Transformation in Programming Languages - on Transformational Language. (in Chinese) *Chinese Journal of Software* 4, 5 (1993), 17-23.

[3] Zhang Naixiao. Analysis and Design of Program Transformation Process. (in Chinese) *Chinese Journal of Computers* 17, 6 (1994), 473-476.

[4] Zhang Naixiao. The Trinary-tree Representation of Knowledge and the Implementation of Inference Procedure. (in Chinese) *Chinese Journal of Computers* 13, 1 (1993) 32-41.

[5] Qu Wanling and Zhang Naixiao. Simulation of Developing a Telephone Exchange System by Transformational Method. (in Chinese) *Computer Research and Development* 32, 7(1995), 11-16.

[6] Zheng Hongjun and Zhang Naixiao. An Abstract Model for Programming Languages. *Proc. of CICS*'95, 69-75.

[7] John E. Hopcroft *et al*. Computer Science - Achievements and Opportunities. *Society for Industrial and Applied Math*, 1989.

[8] David Gries and DennisVolpano. The Transform - A new language construct. *Tech. Rpt. CS Dept. Cornell Univ*. 1989.

[9] Dennis Volpano and David Gries. Type Definition in Polya. *Tech. Rpt. CS Dept. Cornell Univ*, TR 90-1085, 1990.

[10] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys* 17, 4 (1985), 471-522.

[11] Cliff B. Jones. Systematic Software Development Using VDM. *PHI LTD*, 1990.

[12] Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* 20, 1 (1987), 29-70.

[13] Helmut.A.Partsch. Specification and Transformation of Programming. *Springer-Verlag*, 1990.

[14] Jan F.Prins. Partial Implementations in Program Derivation. *Ph.D. Dissertation, CS Dept. Cornell Univ*. TR 87-854, 1987.

[15] Charles W.Krueger. Software Reuse. *ACM Computing Surveys* 24, 2 (1992), 131-183.

[16] Donald Knuth. Structured Programming with Goto Statement. *ACM Computing Surveys* 6, 4 (1974), 261-301.