1

A Study on Meta-Programming Environments

by
Takao Tenma

January 1990

Dissertation submitted to the Faculty of the Information Engineering Course of the Graduate School of Engineering of Hiroshima University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

In order to increase the productivity of software development, a programming environment which supports programming activities should be provided. Recently, an integrated programming environment which provides language-oriented facilities through an interactive user interface enables the higher level of productivity. However, an integrated environment should be constructed for each programming language.

Since a development of a software strongly depends on the application domain, a language and an environment which is adapted to the domain are needed. During a software development, many kinds of software objects are created such as requirements, designs, programs, test cases, development plans, and management information. A software object requires facilities which are specialized to its language. Furthermore, a programmer or a project requires a method to customize the environment. In order to satisfy these requirements, a meta-programming environment which provides a domain- or language-oriented integrated environment from a description of a language and an application domain has been studied on.

A software is modified frequently by many reasons during its life, including development and maintenance, and its cost is very large. In particular, maintenance takes a major part of total software cost. Cost of modification is mainly caused by side-effects of a modification. When a programmer modifies a software he/she must correct a lot of side-effects. Most of these corrections may be routine works and also they waste creative ability of programmers.

Above discussion suggests that an automatic correction of side-effects will improve the productivity of a software development rapidly. However, no intensive studies on the modification support which aims to automate corrections of side-effects can be seen.

This dissertation discusses meta-programming environments, and presents an meta-programming environment MUSE which supports multiple-languages and provides higher describable meta-description language based on an object-oriented meta-model. Next, an approach to correct side-effects and a modification support system developed on MUSE for a type modification are proposed.

First, in Chapter 1, the background and motivation of the research are summerized.

In Chapter 2, studies on meta-programming environments are discussed and an approach taken in this dissertation is briefly explained.

In Chapter 3, a meta-programming environment MUSE, is presented. In a software development, many kinds of software objects are created. However, most of meta-programming environments can support only one language at one time. MUSE has been designed (1) to support multiple kinds of software objects including design languages and management information, (2) to provide facilities which are specialized to each kind of software objects, and (3) to provide highly describable meta-description language, in addition to provide benefits of an integrated programming environment.

In order to realize these goals, MUSE introduces an active meta-model based on an object-oriented concepts. A software object is presented as a collection of objects, and an object is an instance of a class to represent one of software components such as syntactic elements, declarations, design scheme, files, modules, and test cases. MUSE has multiple language knowledge and switches them for each software object. In addition to provide ordinary supported facilities, MUSE provides attribute propagation and static semantic checking based on attribute relations, design editing, management, and modification support.

In Chapter 4, a modification support system on MUSE is presented. First, the author discusses problems, treatment, and supporting facilities of a modification and proposes an approach taken in the system. The target of the system is to automate corrections of side-effects in a program caused by a type modification. Side-effects are detected by propagating type attributes on an internal representation of a program. When a side-effect is detected on an object, the modification support system corrects it by replacing the object which causes an inconsistency with another object as follows: (1) Find a class which resolves the inconsistency in a class hierarchy. (2) Instantiates the founded class. (3) Customize the instantiated object. (4) Replace the original objects with the customized object. This approach and the system are realized on meta-level.

Finally, Chapter 5 summarizes the results of the dissertation and some future directions are given.

Acknowledgements

The author would like to take this opportunity to express his deep appreciation to those who supported him throughout the course of his study.

The author is very grateful to Professor T. Ichikawa, Associate Professor M. Tanaka, and Dr. M. Hirakawa for their invaluable guidance, and to Professor Ichikawa in particular for his supervision.

The author is deeply indebted to Professor T. Ae, Professor K. Onaga, and Professor M. Yamashita of the dissertation examining committee for their valuable comments and suggestions for the improvement of this dissertation.

The author would like to thank Professor Y. Isomichi, Professor K. Kinoshita, Professor I. Kubo, Professor W. Maeda, Professor K. Mizukami, Professor A. Nakamura, and Professor T. Shohoji of the Information Engineering Course in the Graduate School of Engineering, Hiroshima University, for their general advice for preparing this dissertation.

Thanks due also to the past and present members of Information Systems Research Laboratory for their discussion and cooperation. In particular, the author would like to acknowledge Dr. H. Tsubotani, M. Yoshino, and Y. Nagara graduated members of the architecture and programming environment research group and Y. Sato, Y. Morimoto, K. Okamoto, and Y. Miyanaga of the current members of the programming environment research group for their efforts and cooperation in designing and implementing a system prototype.

The author would like to thank Mr. Shaun Doig and Mr. Murray McDonald for his advice and support in preparing presentations and refining the manuscript in English.

The author would like to thank researchers and programmers in UNIX networking society for their public domain and free software in developing a system prototype.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To meet the demands to increase the productivity of software development, it becomes more important to provide software tools and environments which support programming activities.

1.1 Integrated Programming Environments

Since needs for software increase, the productivity of software development becomes more important. Research on software engineering proposes many philosophies, techniques, methodologies, and languages. In order to apply these technologies in actual development efficiently, useful software tools based on the software technologies such as editors, compilers, interpreters, and debuggers have been developed. These tools improve the productivity of software development and the reliability of software.

From 70's, many works aim to develop a programming environment which integrates software tools. A programming environment is not only a set of tools but also provides a framework for combining and cooperating tools in the environment. UNIX[1] [Dolotta 78] is a one of most famous operating system and a programming environment.

In some cases, the term *programming environment* means a software system which support only one programmer, and the term *software development environment* is used for an environment to support a project team [Dart 87]. And also, the term *programmer* has a restricted meaning, i.e., programmers are those write programs in only coding phase. This dissertation, however, use wider definitions. A programming environment supports both of a project team and a programmer, and a programmer means a man who works in any phases of software development. The term *software object* is used to represent one of documents or information created in a software development such as specifications, designs, programs, test cases, and management information.

[1]UNIX is a trademark of AT&T Bell Laboratories.

Some commercial programming environments developed on an advanced personal workstation have been released from '80s. These are called *integrated programming environments*. Examples of them are Interlisp-D [Teitelman 81], Smalltalk-80[2] [Goldberg 83b], Mesa [Sweet 85], Cedar [Swinehart 86], Symbolics/Genera [Walker 87], and ELIS/TAO. These are developed for a single programming language and provides rich and sophisticated functions for the target language. These are characterized by the following features.

(1) Language-Oriented

An integrated environment provides rich facilities and tools specialized to a particular programming language. Examples of such tools are *structured editor* [Barstow 84a], *inspector*, and *file package* [Sandewall 78] in Lisp environment and *class browser* in Smalltalk-80. Since an integrated environment is written in the target language itself, a programmer can customize the environment and accesses system defined tools and functions in the target language level. Furthermore, the command language in the environment is the same as the target language. The programmer does not need to know other languages. On the contrary, UNIX, which is not an integrated environment, requires a programmer to learn many programming and command languages such as C, MAKE, C-shell, dbx, awmrc[3], Xdefaults[4], and Emacs-Lisp.

(2) Integration

Integration enables programmers to combine software tools in an uniformed way. This feature is realized to unify an *internal representation of programs* which are accessed and shared with software tools in the environment [Delisle 84]. The internal representation should be a higher level structure and a data structure of the target language. Examples of such internal representation are *list structure* in Lisp environment and *object hierarchy* in Smalltalk-80. The UNIX uniforms structure of files, programs, and communication data between processes as a text and byte stream [Kernighan 81], but this is a low level structure.

(3) Interactive

Recent hardware technology in high performance workstation with a bitmap display and a mouse enables a high-band width interactive communication between an environment and a programmer. Some small techniques on a bitmap display such as multiple windows, popup and pull-down menus, icons, push-buttons, and scrollbars support a more friendly communication. An interactive means that an environment

[2] Smalltalk-80 is a trademark of Xerox Corporation.

[3] awmrc is a definition format for awm window manager.

[4] Xdefaults is a resource definition format for X11

accepts a user operation at any time and responses to the operation quickly. The first interactive programming environment is Interlisp [Dandwall 78].

(4) Incremental

In an incremental environment, a size of programming activities, to which an environment responses, kept as small as possible. An incremental environment checks correctness for each of programming activities and updates related data such as internal representation of a program and an execution code, immediately. Since there is no inessential works such as compilation and linking, he/she can concentrates his/her creative works. Usually, a programmer creates a program through an editing, compilation, linking, and debugging cycle. In an incremental environment, he/she can create a program without quitting an editor and without his/her mental context switching. This is called *modeless programming*. Modeless programming strongly depends on integration and interactive.

An *incremental syntactic and static semantic checking* detects an erroneous status of a program for each editing operations. An *incremental compilation* [Schwartz 84] and an interpreter/debugger enable an immediate execution after a break of execution and editing operation. The *incremental re-execution* technique removes unnecessary parts of execution for debugging by reusing previous execution data [Karinthi 87].

An integrated programming environment has been mainly developed for an interpreter-based language, like Lisp. Recently, many works have been tried to develop an integrated programming environment for other compiler-based programming languages, such as Ada[5], Pascal, Modula-2, C, and C++. Table 1.1 summarizes history of integrated programming environments.

An integrated programming environment has contributed to increase the productivity of program development. However, it supports only coding phase. To increase the total software development productivity, an environment which provides features of integrated environment and supports various programming activities in wider range of software development phases should be developed.

1.2 Meta Approaches in Software Engineering

The term *meta* has two meaning in software engineering. One is a programming language which can modify an interpreter of the language. To modify an interpreter is called *meta-programming*. Another is a *meta-tool* and a *meta-programming environment* which can be programmed to meet a particular language and an application

[5]Ada is a trademark of Ada Joint Program Office.

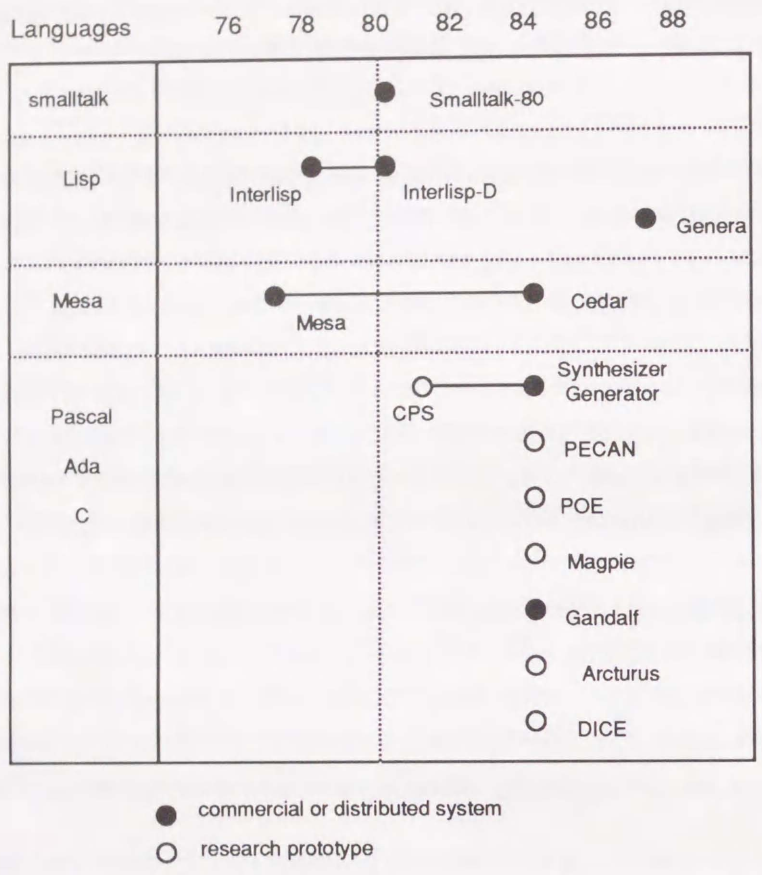


Table 1.1: History of Integrated Programming Environment

domain. The target of this dissertation is the later. Each software development phase requires software objects written in languages specialized to the phase. So, through a software development, many kinds of software objects with a formal syntax and a semantics are created. Furthermore, an application domain of software needs a particular language specialized to the domain and a language-oriented environment for the language.

In order to create language-oriented environments easily, meta-environments and meta-tools have been researched on. They generate a language-oriented tool or an environment from a description of the target language and a domain knowledge.

A typical meta-tool is a *compiler-compiler* which generates a compiler from a language description [Aho 77]. Yacc [Johnson 75] and Bison[6] [Donnelly 88] are parser generators and Lex [Lesk 75] is a lexical analyzer generator.

These are used widely on UNIX. Some compiler-compilers for other parts, for ex-

[6] Bison is distributed from Free Software Foundation.

ample, static semantic checking, optimizing, and code generating, are researched on [Aho 86]. Many generators for syntax-directed or language-oriented editors have been developed by, e.g., [Donzeau-Gouge 80], [Teitelman 81], [Fischer 84], [Horgan 84], and [Tenma 86]. Generated editors check a syntactic error incrementally. An *attribute grammar* and an incremental attribute propagation technique realizes to incorporate an incremental static semantic checking into a generated editor. Most of these editors provide only template-based editing operations. Recent research on incremental parsing techniques makes possible to edit a program in textually.

Emacs[7] [Stallman 81] is a most famous and widely used display-oriented screen editor, and it is one of meta-tools. Emacs provides *Emacs-Lisp* for customizing and extending functions in Emacs. Lots of packages have been written n Emacs-Lisp and distributed. Packages which are specialized to languages such as Lisp, Fortran, Cobol, PL/I, C, C++[8], Modula-2, Pascal, Ada, Prolog, roff, and TEX[9] are supported in Emacs. Other distributed packages are interfaces of mail systems, news systems, shells, lisp interpreters, debuggers, a help system, and so on [Stallman 87]. Emacs is the most successful meta-tools and meta-environments.

From '80s, many meta-programming environments based on language-oriented editors have been developed. A meta-programming environment generates a language-oriented environment from a language-description. A generated environment has features in integrated programming environment, such as language-oriented, integrated, interactive, and incremental, and it supports template-based editing, incremental syntactic and semantic checking, and debugging. Then, studies on a meta-programming environment aim to support upper phases and to incorporate new functions, such as a project and resource management, consistency checking, and transformations, into the generated environment.

In order to support various programming activities and to provide various functions, a meta-programming environment should provide a more describable description language for language features and facilities. Emacs-lisp is a typical example of a flexible and describable language. However, Emacs is a text-based system and a text representation of a software object is a low level of software structure. An advanced meta-programming environment should be constructed on a higher level syntactic and semantic structure of software representation. And also, it should be based on a powerful framework for combining software tools.

Meta-programming environments will be described in detail in Chapter 2.

[7]GNUEmacs is distributed from Free Software Foundation.

[8]C++ is distributed from AT&T and Free Software Foundation.

[9]TEX is a trademark of American Mathematical Society.

1.3 Software Modification

A software is modified many times during its life. It is fact that a major part of the software development cost is spent on software modifications. In particular, over 60% of total software development cost is a maintenance cost which is considered as a modification cost.

A programmer modifies a software by various reasons such as a bug fixes and changes of design decisions. In particular, a programmer modifies a software frequently during a prototyping and for a software development in a complex and ill-structured application domain. Furthermore, by modifying a released software, a software for a similar specification will be developed cheaper cost. Usually, a programmer, in particular a novice programmer, starts from modifying program examples in text books, when he/she meets a new programming language or a new software package. A programmer can not avoid software modifications in all software phases. However, unfortunately, there are serious problems related to a software modification as follows.

(1) Side-effects of a modification are occurred and propagated into many parts of various software objects. A programmer must correct these side-effects. However, these corrections are *tedious* and maybe *routine works*. So, these works waste the intelligent ability of a programmer and decrease the productivity. Furthermore, missing side-effects decreases the reliability of the software.

(2) Many modifications change a *well-structured* software into an *ill-structured* software. This decreases the readability and modifiability of the software. In particular, *ad hoc* corrections such as loopholes, goto statements, type conversions, and temporal global variables make impossible further modifications and corrections.

(3) A modification destroys relationships and consistencies between software objects. For example, even if a programmer modifies and corrects a program, the consistencies between the program and other software objects such as specifications, designs, and test cases may be destroyed.

In order to resolve these problems, many approaches such as a *modular programming,* an *object-oriented programming,* and an *automatic programming* are proposed and used in practical. These approaches are effective, however, above problems remain yet.

A most effective facility of a modification support is an *automatic correction of side-effects* caused in software objects. This facility would resolve above problems and increase the productivity and the reliability of a software development. However, there have been a few researches on automatic correcting. This dissertation will present a modification support system, which provides automatic corrections in one program caused by changes of data types, in Chapter 4.

1.4 Organization of Dissertation

This dissertation discusses a meta-programming environment and a modification support system on the environment. The organization of the dissertation is as follows.

In Chapter 2, studies on meta-programming environments are discussed. An approaches taken in the dissertation is proposed.

In Chapter 3, a meta-programming environment, MUSE, is presented. MUSE mainly supports design and coding phases. It has been designed to support multiple kinds of software objects and to provide language-oriented facilities for each software object. In order to realize these design goals, MUSE introduces an *object-oriented concepts* into its *meta-model* and the *meta-description language*. A software object is represented as a collection of objects based on an *extended attributed abstract syntax tree*. A node of the tree is an instance of a class defined in a description of a particular language. A class is defined for an element of software object, such as syntactic elements, declarations, files, modules, and test cases. Two mechanisms, *gate* and *internal-class*, are introduced to realize advanced facilities. Facilities provided by MUSE are template-based and design editing with an incremental static checking, incremental attribute propagation and static semantic checking, management facilities, debugging, and modification support.

In Chapter 4, a modification support system on MUSE is presented. Following discussions of modifications, supporting facilities, and related works, an approach to correct side-effects is proposed. Next, the author presents a system developed on the proposed approach. The target correction of the system is side-effects caused by a type modification in one program. Side-effects are detected by propagating type attributes on an internal representation. When a side-effect is detected, the system replaces the object with another object of another class which can remove the side-effects. The author discusses problems of the presented system and presents some future directions of modification support.

Finally, Chapter 5 summarizes the results of the dissertation and some comments on future research are given.

Chapter 2

Overview of Meta-Programming Environment Studies

A meta-programming environment makes it easier to construct a language-oriented environment and to customize the constructed environment. In this chapter, an overview of a meta-programming environment is given first, then related works are shown. Finally an approach adopted in this dissertation is briefly explained.

2.1 On Meta-Programming Environments

In this section, overview of meta-programming environments is summarized.

2.1.1 Definition

There is no definition for a meta-programming environment which is commonly accepted. The author uses a looser definition, *a meta-programming environment is an environment which can be customized and extended by describing behaviors of the environment in a particular description language provided by the environment.*

In order to create an environment for a particular application domain and a language, an environment builder writes a description about the target language features and the responses for programming activities. This description is one of software knowledge called a *meta-description,* and the language is called *a meta-description language.*

2.1.2 Architecture

Figure 2.1 shows the general architecture of a meta-programming environment. An environment builder describes a meta-description which consists of a *software scheme definition* and *tool definitions.* A software scheme definition defines a syntactic and semantic structure of software objects in a target language. A tool definition specifies the behaviors of a software tool on a software object in the target language.

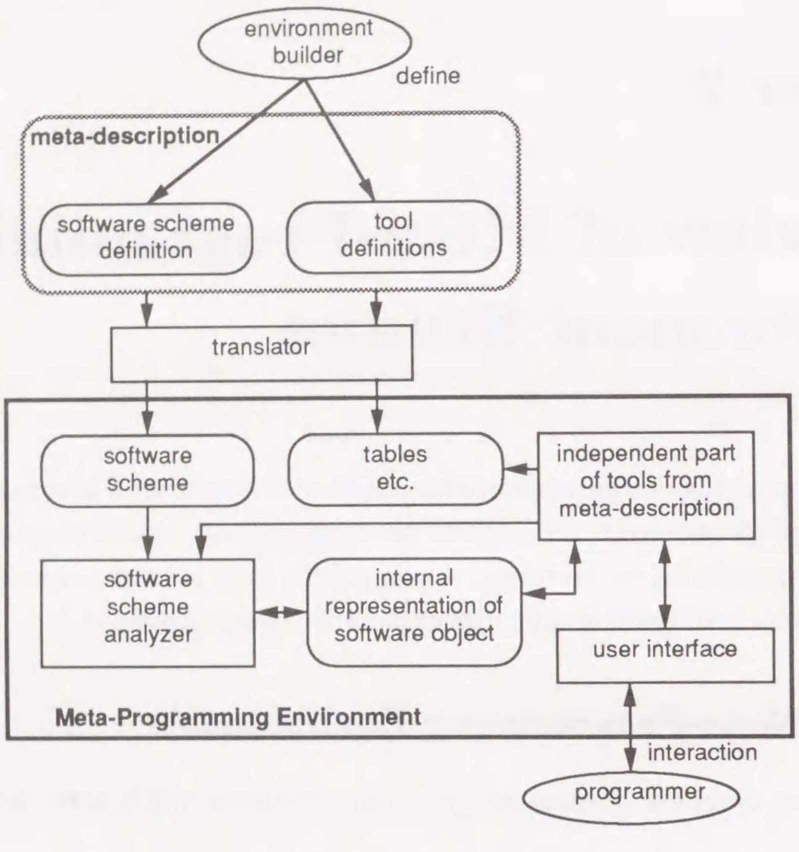


Figure 2.1: Architecture of Meta-Programming Environment

A meta-description is transformed into internal forms, such as event driven tables, decision tables, procedures, and frames. A meta-programming environment is constructed by transformed information and independents part from a meta-description. The following are components of a general meta-programming environment.

(1) Software Scheme Analyzer

In a meta-programing environment and an integrated programming environment, a software object (e.g., a program) is represented as a particular data structure, called *internal representation*. An internal representation represents syntactic and semantic structure of a software object. Most of environments use extensions of intermediate data of compilers, such as a *parse tree* and an *abstract syntax tree*. The tree structure represents syntactic structure and the attributes attached with a node of the tree represent semantic information. A language syntax and semantics are represented as a *(software) schema* on a particular model of an internal representation called *meta-model*. The *software schema analyzer* detects an erroneous updating operation to the internal representation and guarantees the correctness by referring to the software schema.

(2) Internal Representation

This is a software representation which is represented as a particular data structure. This representation is shared and accessed by software tools in the meta-programming environment. An instance of a representation is based on a particular software schema.

(3) Software Tools

These tools are designed to work independent from a meta-description. They work on an internal representation and provide facilities for a particular language or domain by referring a tool definition. Typical software tools are as follows.

- A *editing tool* selects a part of the internal representation and updates it. Most of provided editors are template-based language-oriented editors. Recent works incorporate an *incremental parser* for text-oriented editing into the editor.
- A *pretty-printer* and an *unparser* show an internal representation in user friendly form such as text representation or a graphical representation.
- A *checker* checks correctness of a software object. An incremental syntax checker and static semantic checker check the syntactic and static semantic correctness for each editing operation.
- A *translator* and a *code-generator* translate an internal representation into another representation such as internal representation in other language and object code.
- A *debugger* and an *interpreter* interprets the internal representation and execute it.
- A *management tool* manages software objects and controls versions and configurations.

It is important that the programmer edits and accesses an internal representation only by the provided editor. Other tools are activated through the editor and work on the internal representation.

(4) User Interface

User interface manages *displayed objects* and controls *events* from the programmer. This is heavily dependent on the window system, such as SunView[1], X Window System[2], and NeWS[3].

2.1.3 Meta-Model

In a meta-programming environment, a data structure of an an internal representation specifies the behaviors of software tools and communications between tools. This structure is called *meta-model.* Based on the meta-model, a *meta-description language* which describes a syntactic and semantic structure of a target language and behaviors of software tools is decided.

An internal representation on the meta-model should be able to represent syntactic and semantic structure of software object in a natural form. Since the representation is shared with software tools, it must have all information to work software tools. In particular, a syntactic and static semantic correctness of the representation should be checked incrementally. Of course, the space and the access cost of the representation should be minimized as possible.

Most of meta-programming environments use a *parse tree* and an *abstract syntax tree* as a meta-model. A parse tree is an intermediate format of compilers and it is generated by parser. A node in a parse tree represents a syntactic element, i.e., terminal and non-terminal symbol in *BNF (Backus Normal Form).* A token which is produced by a lexical analyzer places on a leaf node. A meta-environment with an incremental parser uses this representation.

An abstract syntax tree is a more space-efficient representation than a parse tree. An abstract syntax tree reduces and takes out unnecessary intermediate non-terminal nodes and nodes for tokens from a parse tree. For example, a part of a parse tree corresponding to an Ada statement

```
if Z then
      Y := X;
      X := 0;
end if;
```

is shown in Fig. 2.2 and a case of an abstract syntax tree is shown in Fig. 2.3. Intermediate nodes such as `statement` and `expression` are removed. A parse tree and an abstract syntax tree represent only syntactic structure of a software object. In order to represent a semantic information, attributes which represent semantic information

[1] SunView is a trademark of Sun Microsystems.
[2] X Window System is a trademark of MIT.
[3] NeWS is a trademark of Sun Microsystems.

such as types of expressions, names of identifiers and variables, and values of literal are attached to a node of the tree.

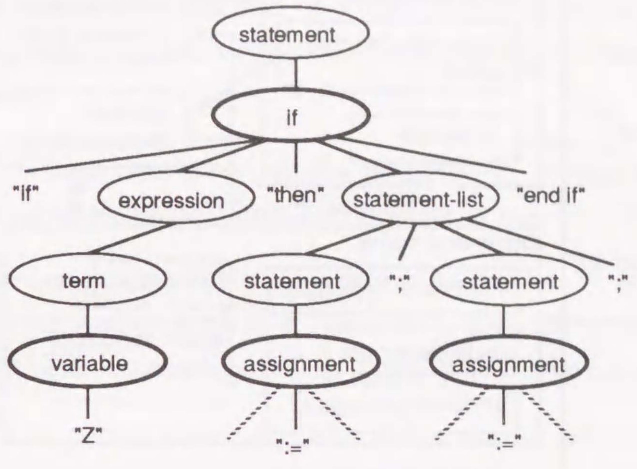


Figure 2.2: A Parse Tree

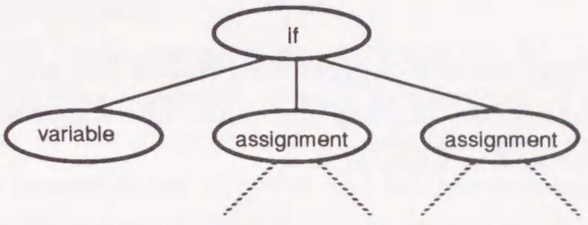


Figure 2.3: An Abstract Syntax Tree

A software schema for a tree based meta-model is described by defining types of a node in the tree, which is shown as a label of node in Fig. 2.2 and Fig. 2.3. Syntax can be represented as links of a node and static semantics is represented as attributes attached to the node. A definition of a type of a node specifies these links and attributes.

It is important to decide better meta-model before constructing meta-programming environments. For example, in order to construct Ada programming environment and supporting software tools, several standard internal representations (e.g., DIANA [Evans 83]) are proposed.

Tree-based representations are used for a text-based programming language. In order to represent a *design* or *diagram-based* language and to specify behaviors of software tools and communications between software tools, more describable and active meta-model is required.

Recent works use an *ER-model*, an *object-oriented model* (e.g., Garden for design environment [Reiss 87], SW2 for accessing and managing software objects [Laff 85]),

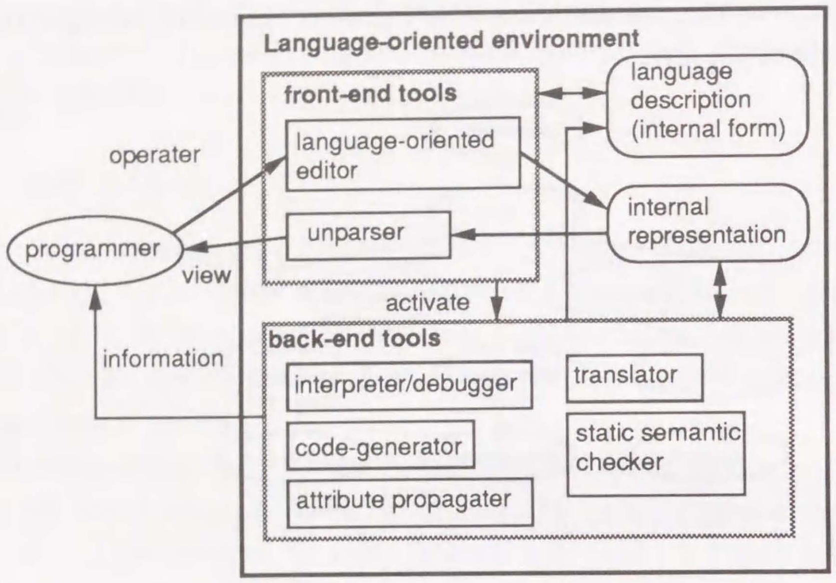


Figure 2.4: A Constructed Language-Oriented Environment

plan representation in Programmers' Apprentice [Waters 82], and others.

An object-oriented meta-model which is introduced into the presented meta-programming environment will be described in Chapter 3.

2.2 Construction of Language-Oriented Environments

Early works of meta-programming environments used to develop construction systems for *language-oriented environments.* At first, the author presents a system organization of a constructed environment. A language-oriented environment is generated from a language-description for text-based programming language used in coding phase. Describing methods for language features and studies on construction systems are presented.

2.2.1 Constructed Environment

Figure 2.4 shows a constructed language-oriented environment. This environment supports typical facilities such as template-based editing, syntax and static semantic checking, debugging, and code generation. An environment builder describes a language description for a target programming language and it is translated into an internal form.

The *front-end* of the environment is a *language-oriented editor* (or *syntax-directed* *editor*) which checks syntactic correctness in each editing operations and removes an illegal operation. A programmer updates and focuses an internal representation through the editor. A *focus* means selecting and moving a cursor on a part of the internal representation. A language-oriented editor can be classified into three classes from the view of editing styles.

Template-Based: The programmer edits a program by using templates except some small elements such as names, identifiers, literals, and expressions.

Text-Only: The programmer edits a program in textually. A text is concerned with an internal representation such as a parse tree. An incremental parser parses the edited text and updates corresponding parts of the internal representation.

Complex-Type: The programmer can edit the program by using template and by textually. An internal representation is updated incrementally.

An unparser creates and shows a pretty-printed text representation from the updated internal representation.

The other software tools place on the *back-end* of the environment. These are activated from the language-oriented editor. An incremental syntactic checker checks syntactic correctness of the program and cancels the error operations. An attribute propagator propagates attributes of nodes and an incremental static semantic checker detects static semantic errors such as type and scope errors by checking attributes. An interpreter/debugger sets *hooks* on the internal representation and interprets the internal representation and the associated hooks. A code-generator or a translator translates the internal representation into other formats such as assemble codes.

2.2.2 Description of Language Features

In this section, methods and studies for describing a language features are shown. Language features are as follows.

(1) Syntax

Syntax of a language is an important feature. Usually, BNF and its extensions are used to represent language syntax, i.e., context free grammar. In order to generate *tokens* through a lexical analysis, a regular expression is used. A template-based editor uses the notation of syntax for checking syntactic correctness. An editor with incremental parser uses the notation for parsing text and for checking syntax errors.

Cornell Program Synthesizer (CPS) [Teitelbaum 81] is a template-based editor, and the Synthesizer Generator which generates CPSs uses BNF-like syntax [Reps 84a].

Mentor uses the abstract syntax for an abstract representation, i.e., an abstract syntax tree, which plays a same role of BNF for a concrete syntax [Donzeau-Gouge 84]. Syend: language-oriented editor generator uses a BNF-like notation which is YACC-compatible LALR(1) grammar for a multiple-entry parser (i.e., incremental parser) [Horgan 84]. SPSG uses XLALR(1) grammar for efficient incremental parsing [Handa 86]

(2) Static Semantics

In addition to the syntax, a language gives static semantic constraints such as *type consistencies* and *scope constraints.* In order to represent static semantics of a program, attributes are used. Static semantic errors are detected by propagating attributes and by checking whether the propagated attribute satisfies the given constraints or not. In order to specify the static semantics, action routines and attribute grammar have been used.

Action Routine: Action routine was used in ALOEGEN[4] (A Language-Oriented Editor Generator) [CMU84]. An action routine is a procedure which is defined in each kinds of node. An action routine is activated by a editing operation and it processes some works, including attribute propagation and error checking. Though an action routine is powerful, the description cost is large.

Attribute Grammar: An attribute grammar was proposed in [Knuth 68]. In an attribute grammar, attributes are defined for terminal and non-terminal symbol of context free grammar, and semantic equations are defined to a production rule of the grammar. There are two types of attributes, a *synthesized* attribute (an attribute propagated from *right* hand side of a production rule to the *left* hand side) and an *inherited attribute* (an attribute propagated from *left* hand side to the *right* hand side). Attribute grammar is introduced into compilers and compiler-compilers. In order to evaluate attributes incrementally, [Reps84b] proposes an incremental attribute propagation algorithm and incorporates it into the Synthesizer Generator. Semantic equations are independent of each other and are modularized, and it is represented by clear and declarative.

An attribute grammar is more formalized than action routines, and this describing burden is smaller than writing action routines. However, runtime evaluation costs for an attribute grammar is larger than action routines.

After an attribute grammar, some methods are proposed to propagate attributes. ARL (Action Routine Language) in Gandalf is one of attribute grammar [Harbermann 86]. For efficient runtime supports, a description written in ARL is translated into action

[4] ALOEGEN is distributed from CMU.

routines and daemons. PSG uses more formal method than an attribute grammar. The method is based on a *context relation* and an *unification* [Bahlke 86]. [Horwitz 86] proposes a method which incorporate relational operations into an incremental attribute propagation. [Tenma 86] uses *message passing* between nodes for static semantic checking and other operations.

(3) Semantics, Interpretation, and Code-Generation

Semantics of a programming language is an execution or an interpretation of a program written in the language. A construction system supports one interpreter and the interpreter traverses an internal representation for executing. When the interpreter visits a node, it accesses the *semantic description* associated with the node and executes it. A code-generator works as similar to an interpreter. CPS provides an assemble language for interpretation. PECAN provides a high level semantic description language for incremental code-generation [Reiss 84a]. PSG supports a functional language to generate language-specific debugging system [Bahlke 87].

Usually, debugging operations are realized in a language-independent way. For setting breakpoints and tracings, a debugging operation attaches a pre-defined language independent *hook* into a node of the internal representation. In addition to evaluate the semantic description, an interpreter/debugger evaluates attached hooks.

(4) Format or Unparsing Scheme

An unparsing scheme is used by an unparser or a pretty-printer to translate an internal representation into a more friendly representation such as a pretty-printed text and a diagram. A behavior of an unparser (or a pretty-printer) is similar to an interpreter.

2.2.3 Construction Systems

Many construction systems for language-oriented environments are developed with a language-oriented editor. Synthesizer Generator generates CPSs for Pascal-like programming languages. This environment supports template-based editing, incremental syntactic checking, and incremental static semantic checking based on an attribute grammar. Mentor can support multi-lingual documents based on an abstract representation and an interactive tree manipulation language Mentol [Donzeau-Gouge 84].

The Gandalf project aims to develop a construction system for generating environments which support (1) program construction, (2) configuration management, and (3) version control. For realizing these facilities, ALOEGEN, ARL, and action routines are developed. Generated ALOEs are used as front-end of the environment. Subprojects of Gandalf create prototype environments; GNOME [Garlan 84] and SMILE.

PECAN provides editing and debugging facilities for Pascal-like languages on an advanced workstation. The main advantage of PECAN is its user interface. By using bitmap display of a workstation, it provides multiple view of programs and status of execution, for example, syntax-directed editor, symbol table display, data type display, expression display, flow graph display, NS-chart display, command history display, and execution view [Reiss 84b].

PSG (Programming System Generator) generates language-oriented programming environments from a formal definition of a programming language. It introduces a context relation, which represent a context of a program, and an unification for propagating attributes.

2.3 Directions of Meta-Programming Environments Studies

In this section, recent studies on meta-programming environments and future directions are summarized.

2.3.1 Supporting Upper Phases

The target phase of construction systems for language-oriented environments is only coding phase. However, a software is developed through various phases. In particular, upper phases such as requirements analysis, requirements specification, system design, and program design phases are more important than coding phase.

In order to support design phases, many design techniques are proposed. These techniques propose several kinds of diagram-based languages, such as dataflow diagram, data structure diagram, finite-state diagram, petri-net, Entity-Relationship (ER), structure chart, state transition diagram, and so on. An environment to support design phases should provide design editors for creating designs and checkers for checking correctness and consistencies of created designs.

A meta-environment to support design phases should be able to define a graphical design language. The definition consists of syntax, semantics, formats, and operations on a design. An interpreter or a simulator which executes a design helps a programmer to design a better software system quickly through a prototyping. A commercial CASE (Computer Aided Software Engineering) environment, StP (Software through Pictures) provides six graphical editors and a project database [Wasserman 86]. Though StP provides a customizing method, it is impossible to create a graphical editor for a new design language. Metaview is a meta system for CASE environment which can be enhanced and changed. It provides a language to specify a specification scheme on ER-model [Sorenson 88].

Garden proposes a *conceptual programming* methodology. The programmer customizes or creates a design language (and a method) for a particular application domain, before he/she designs a software in the domain. Garden provides an object-oriented language to specify a design editor and an execution of a design written in the specified design language [Reiss 87].

For supporting more upper phases, i.e., requirement analysis or requirements acquisition phase, some knowledge-based tools are proposed. By interacting clients, these tools create a formalized and concrete specification from ambiguous client's requirements. However, these tools are not working in practical and the applicable application area is very narrow. These tools need domain specific knowledge gotten by analyzing the application domain.

2.3.2 Supporting between Software Objects

In addition to support each development phases or one software object, supporting facilities between phases and between software objects are needed. Examples of such facilities are management facilities, transformations, and consistency checking.

Management

Several management facilities such as resource management, configuration management, version control, plan management, and project management exist. To realize management facilities, management tools, a languages for describing management information, and a software database are studied on. A project-centered software database has all software objects created in the project and provides functions to project members and software tools. Such functions are *access control*, *version control*, and *dependency controls*. Furthermore, a software database is required to provide active functions, i.e., a manager can define an *action* which is activated automatically when a specific event or a situation is occurred.

A language for management specifies a management information such as software configurations, access lists, version structure, and a development plan. From the view point of a meta-environment, a management language can be realized by a meta-description. In order to realize management facilities in a meta-environment, a mechanism which can relate to different kinds of software objects should be introduced into the meta-model.

Examples of widely used management tools are Make [Feldman 79], SCCS [Allman 86], and RCS [Tichy 86] on UNIX. In particular, Make provides a language to specify dependencies and rules between files and suffixes. Gandalf is a meta-programming environment to provide management facilities.

Process Programming

Concerning plan management, process programming is proposed and environments based on the process programming are researched on. A manager writes a program, which represents a sequence of activities of software development, called *process program*. According to a process program, an environment can activate software tools and checks software objects automatically. Examples of environments based on the process programming are [Huff 88], Arcadia [Tayler 86], and Tame [Bashili 87].

Transformation

A transformation is useful to enhance the productivity of a software development. In a meta-programming environment, transformations change an internal representation of a language into one of the other languages. This is realized by giving *transformation rules* between two representations.

Consistency Checking

A consistency checker detects inconsistencies between different kinds of software objects. PegaSys supports consistency checking between (dataflow) designs and (Ada) programs. In addition to the definitions of elements in a design language and a programming language, PegaSys uses the first-ordered logical formula with the elements which represents semantics of the elements. By deducting logical formula generated from designs and programs, inconsistencies among them are detected [Moriconi 86].

2.3.3 Intelligent Programming Environments

Recent research aims to develop a programming environment with more advanced facilities. In particular, *intelligent programming environment* and an *intelligent programming assistant* are studied on [Barstow 84b]. The widely accepted definition of an intelligent programming environment is *an environment which automates (parts of) programming activities by using software knowledge*. Since an intelligent environment uses software knowledge, it can be customizable and programmable by changing the knowledge. There are four types of software knowledge which are researched in the intelligent environments [Ramanathan 88].

1. *Software Object:* This is a knowledge about software objects. Knowledge about a language is included this type of knowledge. Transformation rules and consistency

rules are also included.

2. *Development Process:* This is a knowledge about a software development process and programming activities. A process program is one of process knowledge.

3. *Software Tools:* Since an intelligent assistant automatically activates software tools, knowledge about usages and functions of software tools is needed.

4. *User:* It is useful for a programmer to provide help messages and checking levels which are specialized to the skill level of the programmer. These facilities needs an information about a particular programmer.

Some of above knowledge are used in intelligent environments. However, in particular, formalization and usage of 2–4 is a current topics of *Software Engineering* and *Artificial Intelligence*. They are not used in an actual environment.

The Programmer's Apprentice project in MIT is a famous research of an intelligent assistant. This project aims to develop theory of programming and to automate the programming activities. The key ideas of the project are as follows.

- *Assistant Approach:* The system assists programmers by automating detailed and routine works rather than a full automatic programming.

- *Cliches:* These are knowledge about fragments of a program.

- *Plan:* This is a knowledge representation of a program on which the assistant works.

The project has been developed several demonstration system, e.g., KBE [Rich 78], [Waters 82], KBEmacs [Waters 85], Requirements Apprentice and Design Apprentice [Rich 88].

PROUST detects bugs in a program by using knowledge about programming theory and bugs [Johnson 85]. CHI project is a research on a knowledge-based software environment based on a wide-spectrum language **V** for describing knowledge about software objects and software processes [Smith 85].

These environments use knowledge about software objects. In particular, knowledge of fragments of software objects, such as code skeletons, design scheme, and algorithms are used. These are called as *intelligent macros*. An intelligent macro consists of a name, an invariable part, variable parts, constraints, and so on. A programmer creates a software objects by combining these macros. According to the constraints and contexts of an intelligent macro, it automatically customizes itself by changing and filling a variable part.

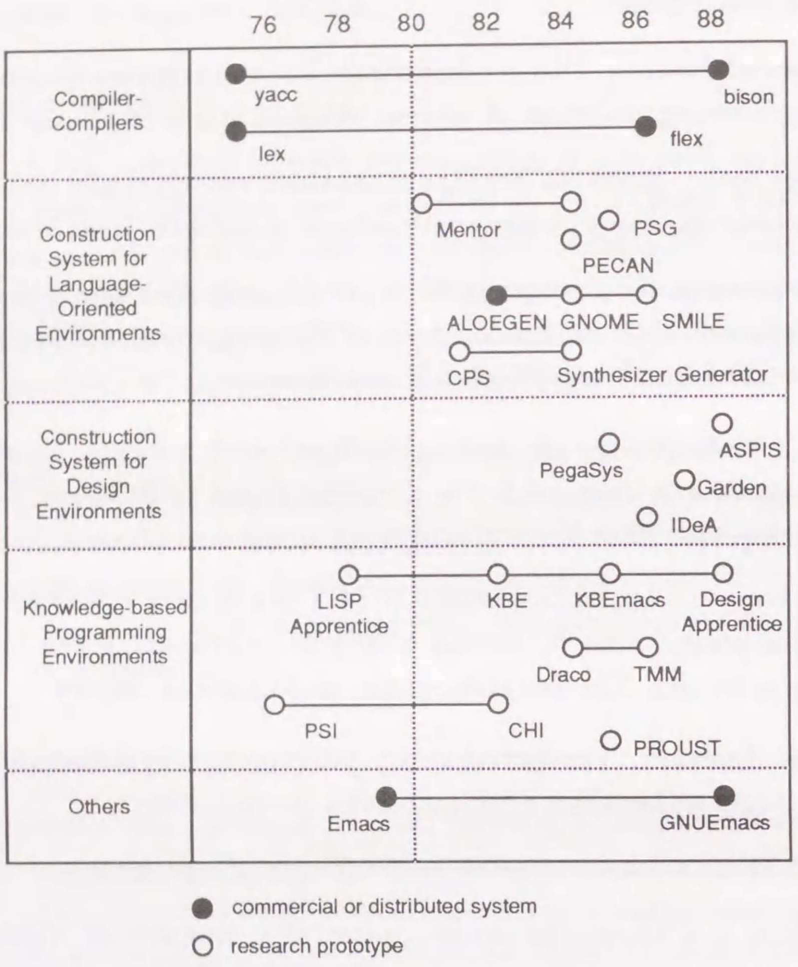


Table 2.1: History of Meta-Tools and Meta-Environments

Table 2.1 summarizes the history of meta-tools and meta-programming environments.

2.4 An Approach to Development Meta-Programming Environment

As discussed in above sections, a meta-programming environment must have benefits of an integrated programming environment and provide editing, incremental syntactic and static semantic checking, incremental attribute propagation, and debugging/execution facilities through a friendly user interface. Furthermore, as shown in Section 2.3, an advanced meta-programming environment should provide facilities to

(1) *support various types of software objects,*

(2) *manage software objects,*

(3) *support design languages,* and

(4) *accept intelligent macros.*

These facilities should be realized on *meta level*, and language dependent parts of them should be written in a highly describable meta-description language. From the view point of enhancing the software development costs,

(5) *modification support facilities*

should be realized. Since a modification causes side-effects on various kinds of software objects, a modification support must be realized on a meta-level and realized language independently.

In this dissertation, a meta-programming environment MUSE (Multiple Language Support Environment) [Tsubotani 87], [Tenma 88] and a modification support system on MUSE [Tenma 90a] are presented.

In order to realize (1)-(5), an active meta-model with highly describable and modularized meta-description language are needed. MUSE introduced a meta-model which is based on an object-oriented concepts. The system architecture is designed to support and to manage multiple kinds of software objects. (1) is realized by changing language knowledge for each type of software objects. (2) needs a mechanism to represent relationship between software objects. MUSE introduces a *context* and a *gate* on the object-oriented meta-model. (3) is realized by supporting design editors and a describing method for defining syntax, semantics, and format of a diagram-based design language. (4) is stated by using the object-oriented meta-model and its meta-description language. MUSE will be described in detail in Chapter 3.

As an advanced facility of MUSE, limited facilities for (5) are realized. The modification support system corrects parts of a program which are side-effected by a modification of data types. The system is based on basic facilities of MUSE, an *attribute propagation* and a *class hierarchy.* The modification support system will be described in detail in Chapter 4.

Chapter 3

A Meta-Programming Environment, MUSE

In this chapter, a meta-programming environment MUSE is presented.

3.1 Design Goals

MUSE is a meta-programming environment which aims to support design and coding phases. The first goal of MUSE is to provide benefits of an integrated programming environment such as language-oriented, integration, interactive, and incremental. In addition to these features, MUSE is design to realize the following goals.

(1) Supporting Various Kinds of Software Objects

During a software development, various kinds of software objects, which are documents and information with formal syntax, such as specifications, designs, programs, test cases, management information, and development plans are created. Each software object requires language-oriented facilities for it. However, most of meta-programming environments can support only one language at one time. For supporting another language, another meta-description and a construction of language-oriented environment is needed. In addition to support language-oriented facilities for one software object, MUSE realizes facilities for supporting multiple software objects.

(2) Supporting Design Phases

In design phases, software objects are represented as diagrams, such as flow chart, dataflow chart, data structure diagram, and petri-net diagram. In order to manipulate these representations, MUSE should be able to support editing and manipulating facilities for these diagrams. The meta-description language of MUSE should provide a method to represent syntax, semantics, and format of a diagram-based design language.

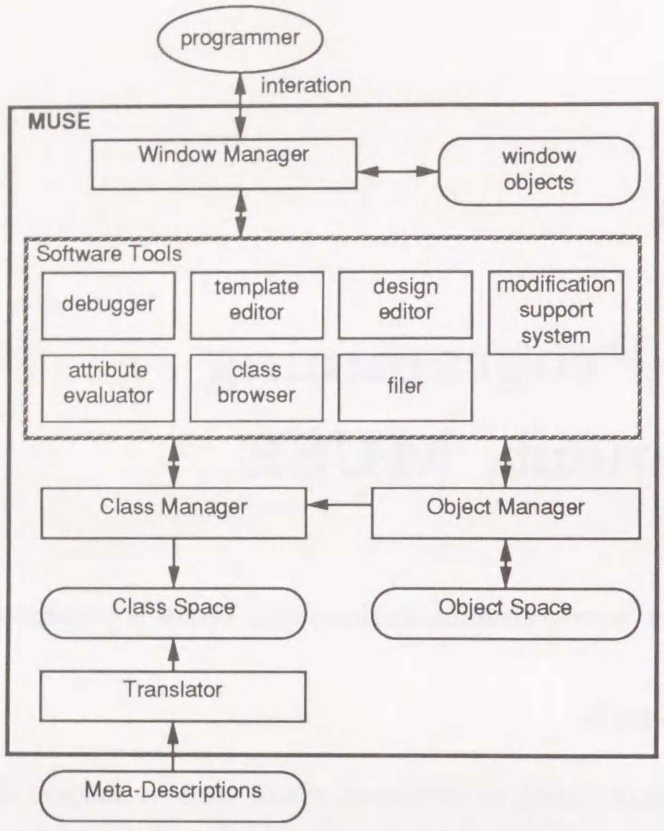


Figure 3.1: System Organization of MUSE

(3) High Level Meta-Description Language

In a meta-programming environment, a development of a meta-description is important. Extensibility, modifiability, and modularity are required to the language. MUSE makes it possible to add higher level facilities by using basic mechanisms of the meta-description language.

3.2 System Architecture

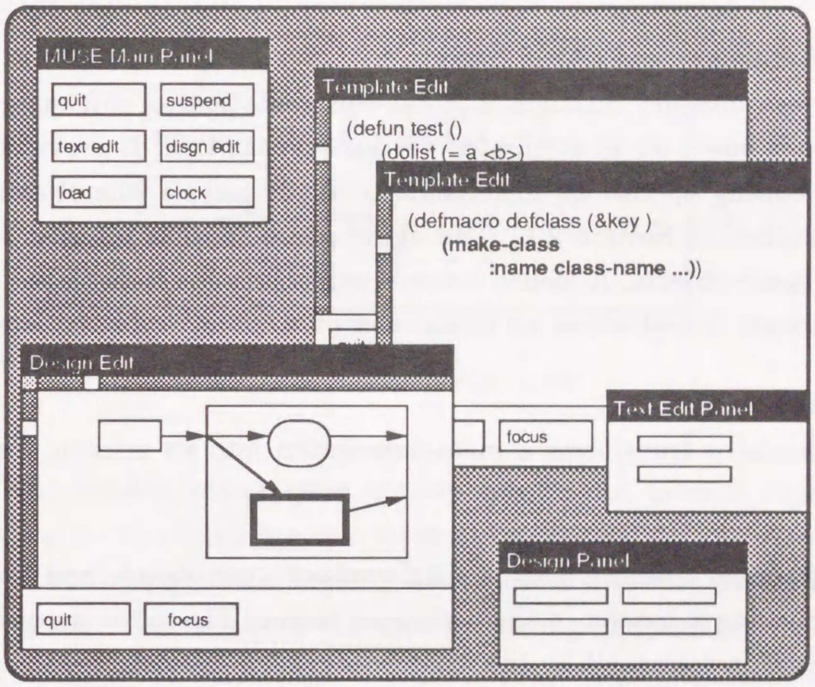
In this section, the system organization of MUSE is described from two points of view such as a *software module view* and a *data management view.*

3.2.1 Software Modules

Figure 3.1 shows the system organization of MUSE. MUSE consists of following software modules.

(1) Window Manager

The window manager provides the multiple window based user interface. This manages *window objects* and its contents, and handles events given from the programmer. When the window manager receives an event, it activates a particular function associated to it. Figure 3.2 shows an example of user interface of MUSE. There are several kinds of window objects such as editors, panels, menus, message regions, buttons, and formatted text shown in the editors. In this example, boldface parts, i.e., a box in a design editor and a `(make-class ...)` in a template editor means that they are focused and selected now. This is similar to a cursor position in a display-oriented editor.



```
(defun test ()
  (dolist (= a <b>)
```

```
(defmacro defclass (&key )
    (make-class
        :name class-name ...))
```

Figure 3.2: User Interface of MUSE

(2) Software Tools

MUSE has following software tools.

- template editor
- design editor
- attribute evaluator
- class browser

- filer
- debugger
- modification support system

These tools works on an internal representation of software object.

(3) Class Manager and Class Space

The *class manager* manages *language knowledge* in the class space. It provides access functions to the language knowledge for software tools. A language-knowledge is generated from a meta-description.

(4) Object Manager and Object Space

The *object manager* manages internal representations of software objects. The representation is based on an attributed abstract syntax tree. It provides access functions and basic editing operations to software tools. It plays a role of software scheme analyzer described in Section 2.1. The *object space* consists internal representations of loaded software objects. A loaded internal representation is displayed in an editor. The object manager is realized as an interpreter of an object-oriented language.

(5) Translator

The translator transforms a meta-description into an internal format used in the class space.

A programmer selects a part of displayed software object. and then gives an operation by clicking a button in the command menu. The given event is translated into a corresponding command by the window manager. The command sends messages to the focused node through the object manager. The class manager searches a method from the class space and returns it, and the object manager evaluates the returned method. The method achieves the operation by activating functions provided from software tools and by sending messages to other nodes. After finishing the operation, if needed, an unparser updates the window object in an editor.

3.2.2 Data Management View

This subsection describes the data management architecture which realizes multiple language support. Figure 3.3 shows the data management architecture of MUSE. From the view point of the programmer, a software object is shown in a template editor or a design editor. All operations to the software object is invoked through the editor.

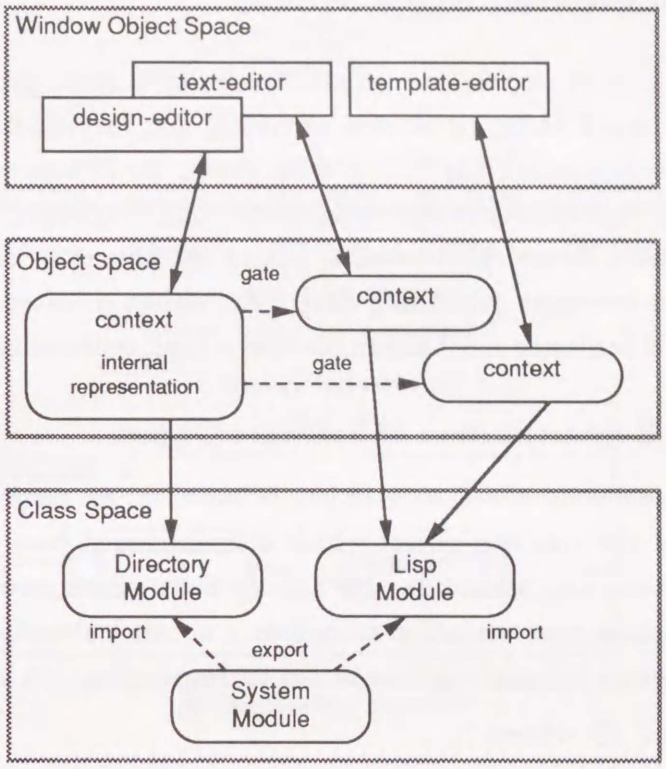


Figure 3.3: Data Management in MUSE

One editor in the *window object space* corresponds to one *context object* in the object space, and a context object has one internal representation and related information about it. A context object is created for a unit of software object such as a module and a program. This is also a file unit for saving and loading. The object space has several context objects. A context object is connected to a node of another internal representation in another context object with *gate* which will be described in Section 3.3.

A context object is associated with one module in the class space. An operation to the software object is processed by referring information in the associated module. A module is an internal form of a meta-description, and it has a set of classes and other language-dependent information. Usually, one module is defined to one language. A module can export its classes to other modules. Other modules can import and use the exported classes from other modules. In this example, `Lisp module` and `directory module` import classes exported from `system` module.

The data management architecture described above realizes multiple language supports and provides language-specific facilities for each software object.

3.3 Object-Oriented Meta-Model

A meta-model is a most important decision to design a meta-programming environment. Since the target software objects of MUSE are various, the meta-description language should have a capability to represent them. To decrease the describing cost, the reusability and modularity of the meta-description language are required.

In order to satisfy above requirements, MUSE introduces a meta-model based on an object-oriented concepts [Goldberg 83a]. An object-oriented language is highly modularized and inheritance mechanism realizes a high reusability.

3.3.1 Internal Representation of Software Object

An internal representation of software object is based on an *attributed abstract syntax tree*. Each *node* of the tree is a object which is instantiated from one of class defined in a meta-description, i.e., a module. By adding links which connects any two nodes of the tree, an internal representation composes a network structure.

Figure 3.4 shows an internal representation corresponding to a part of Lisp program

```
(svref (+ i j) <form>).
```

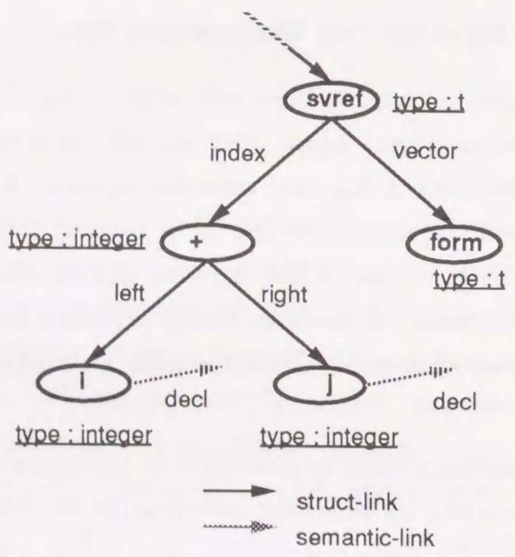
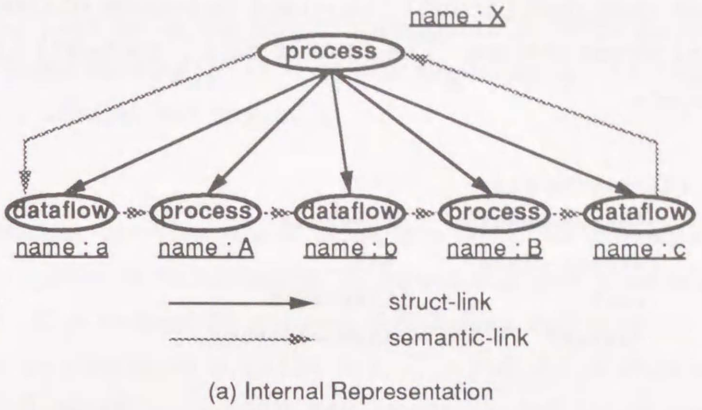


Figure 3.4: Internal Representation of Lisp Program

Figure 3.5 shows (a) an part of internal representation of dataflow design and (b) its diagram representation.

A node in an internal representation represents one of software units which compose a software object such as syntactic elements, program fragments, design elements,



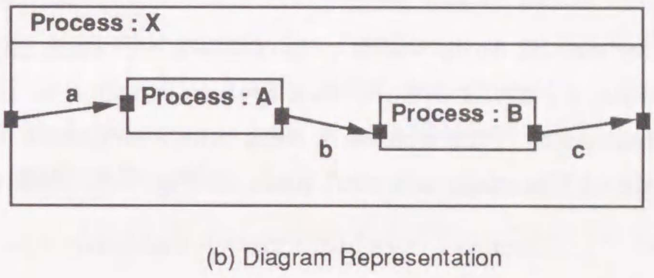


Figure 3.5: Internal Representation of Dataflow Design

skeletons, and algorithm units. A label of a node represents a name of the class which the node belongs to. A node has struct links, semantic links, and attributes. A struct link connects two nodes and represents a parent/child relationship. Struct links compose the tree structure, i.e., an abstract syntax tree. A semantic link connects two nodes in any position. An attribute, shown as underlined text in Fig. 3.4 and Fig. 3.5, represents a properties of a node such as type of expressions, name of identifiers and variables, and value of the literal.

This representation can applicable to text-based context free language, diagram based design language, and other formalized complex information.

3.3.2 Meta-Description Language

A meta-description for a particular language defines a *module*. A module consists of a *module declaration*, *class definitions*, and some descriptions specified for some software tools. Class definitions define the syntax and semantics of the target language.

Figure 3.6 is an example of module declaration for a `Lisp` module. In this example, name of the module is `Lisp`, and the module imports classes which are exported from `system` module (`:import-module`). Imported classes are visible in this module. The export field (`:export-class`) `all` shows that all classes defined in this module are exported. The root class part (`:root`) `Lisp-root` is a name of class which is a root node of an internal representation. The context field (`:context`) `Lisp-context` is a class of context object.

```
(defmodule Lisp
  :import-module     (system)
  :export-class      all
  :root              Lisp-root
  :context           lisp-context)
```

Figure 3.6: An Example of Module Definition

A class is defined for one of units which constructs a software object in the target language. In this module, a class is defined to a system function in Lisp. Figure 3.7 is an example of class definition. This defines a class which defines a function **svref** in Lisp module. The node of this class is a root node in Fig. 3.4. Following are contents of a class definition.

```
(defclass svref
  :super      (form)
  :struct     ((index :class form :range (1 1))
               (vector :class form :range (1 1)))
  :attribute  ((type :default t)
               (type-const :default t))
  :class-attribute  ((format ...)
                     (relation ...) ...)
  :method     ((run () ...)(trace () ...)
               (stop-at () ...) ...)
  :probe      (((vector type)() ...) ...))
```

Figure 3.7: Definition of `svref` Class

(1) Name

The name of a class.

(2) Superclasses

When an OR syntax rule

```
L ::= R1 | R2
```

exits in the *abstract syntax* of the target language, L must be defined as a superclass of R1 and R2. A class inherits (3)-(8) from its superclasses. An override is allowed to restrict constraints of inherited properties.

(3) Struct links

A link is a *instance variable* which value is a node. In particular, a struct link is used to represent syntactic relationship. A struct link and a semantic link is defined as a same format. It is defined by a *name*, a *linkclass*, and a *range*. A linkclass limits classes which can be connected with the link. The connected class must be a subclass of the linkclass. A range gives lower and upper limitations of number of links. In Fig. 3.7, a node which is connected with `index` link of `svref` class must be an instance of subclass of `form` class. Of course, the `form` class is included. The number of `index` link must be over and equal to 1 and under and equal to 1, so it is just 1. A struct link is decided from a *production rule* in the abstract syntax of the target language. For example, an production rule

```
<dolist> ::= (dolist (<var> <list> [<result>]) {<form>})
```

creates four struct links of `dolist` class as shown in Table 3.1. Ranges `0..1` and `1..infi` (infinity) correspond to the optional [] and repetition {} in BNF meta-rules, respectively. (2) and (3) decides the syntax of the target language and they are used to guarantee the syntactic correctness.

name	linkclass	range
var	variable	1..1
list	form	1..1
result	form	0..1
body	form	1..infi

Table 3.1: Struct-Links of `dolist` Class

(4) Semantic links

A semantic link connects a node with another node in any position. A semantic link represents any relationship between nodes except for syntactic relationships. Examples of semantic links are links from a variable to its declarations and its reverse directions, i.e.,

```
(decl :class declaration :range (1 1)) and
(use :class variable :range (0 infi)).
```

In Fig. 3.7, there is no semantic links.

(5) Attributes

An attribute definition defines an instance variable which value is not a node. This is used to represents properties of a node. In Fig. 3.7, a type attribute with default value t is defined. (3)-(5) correspond to instance variables of Smalltalk-80.

(6) Class Attribute

A class attribute corresponds to a *class variable* of Smalltalk-80. It is defined for giving information which is static an unique to the class. In particular, information which is accessed by software tools is defined as a class attribute. Examples of class attributes are *format*, *attribute relation*, *copy-rule*, etc. These will be described in detail in Section 3.4. In Fig. 3.7, class attributes are omitted.

(7) Methods

A method is a procedure invoked by a message from the programmer and from other nodes. A message is sent through a link. A message expression

```
(<- 'index 'evaluate)
```

is sent to a node which is connected to a index link and the message activates the method evaluate of the node.

(8) Probes

A probe is a kind of *daemon* in *frame* representation, and it is similar to the *active value* of LOOPS[1]. A probe consists of *(a) target-link:* a name of link, *(b) target-items:* names of links and attributes which are defined in linkclass of the target-link, and *(c) body:* a body of procedure. The body is activated automatically when one of the target-items of a node connected with the target-link is updated. When a probe is activated, three arguments, which are node which has the probe, target node connected to the target-link, and old value of the target item, are passed to the probe. A probe defined in Fig. 3.7 is activated when a type attribute of node connected with the vector link is updated.

3.3.3 Gate

In order to manage different kinds of software objects, a *gate* is introduced into the basic meta-model. A gate is used first in Mentor [Donzeau-Gouge 84]. Figure 3.8 shows an example of gate. A gate itself is a node of subclass of gate class. A gate

[1]LOOPS is a object-oriented system on Interlisp-D.

has a special semantic link which is connected to another software object. A language of connected software object may be different from the language which the gate node belongs to. This example shows a gate node which is a second version of `module B` in a `project management information` written in project management language. The gate is connected with a Lisp program written in Lisp.

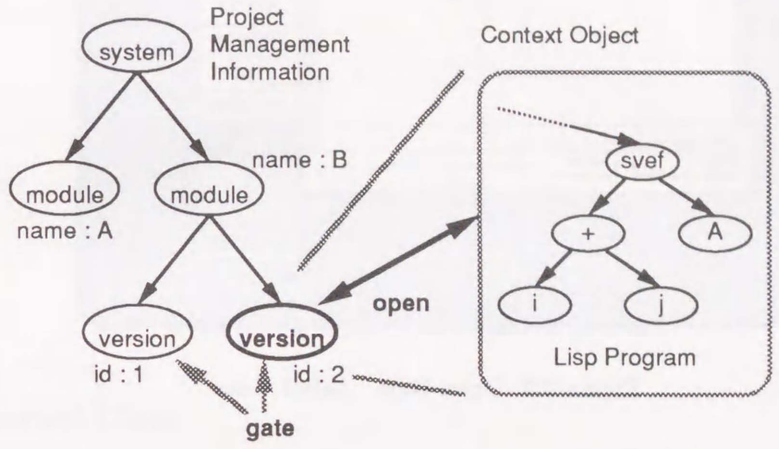


Figure 3.8: An Example of Gate

By selecting a gate node and sending open message, an editor for the connected software object is created as following steps.

(1) The programmer focuses a gate node.

(2) The programmer sends a message open by selecting menu item which represents method list of the focused node (Fig. 3.9).

(3) The object manager loads an internal representation of software object and its *context object* from the file system into the object space by a *filer.*

(4) If a module of the loaded software object does not exist, the class manager loads the module from the file system into the class space by a filer (Fig. 3.10). Then, the class manager associates the context object and the module.

(5) An editor corresponding to the loaded software object is created and displayed (Fig. 3.11).

By introducing gate, the environment builder can creates a language to manage software objects and realizes management facilities on MUSE. A gate and a context are implemented as classes defined in the `system` module.

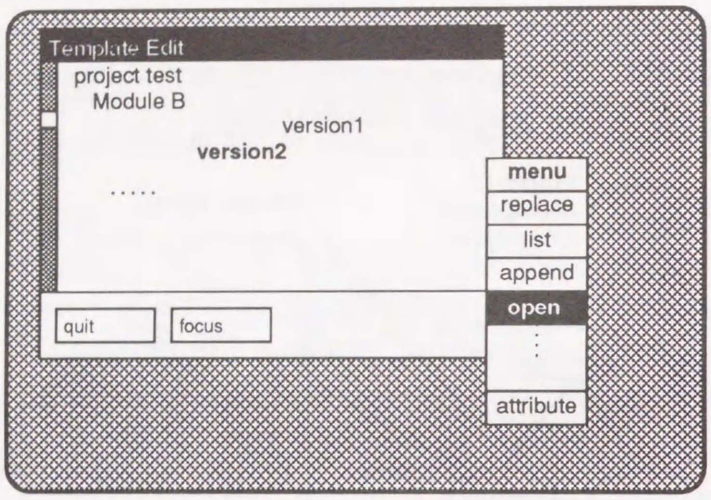


Figure 3.9: Open Gate - Select Gate -

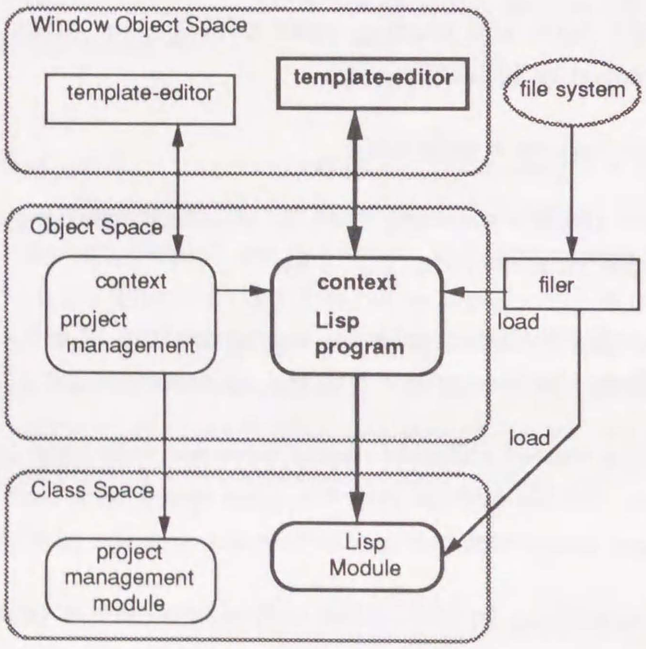
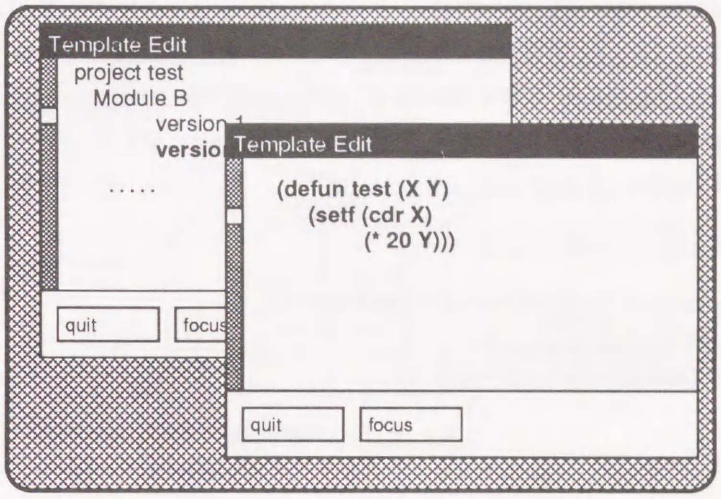


Figure 3.10: Open Gate - Load Internal Representation and Module -



```
project test
  Module B
      version 1
      versio
      .....
```

```
(defun test (X Y)
    (setf (cdr X)
          (* 20 Y)))
```

Figure 3.11: Open Gate - Open New Editor -

3.3.4 Internal-Class

In a software development, many items such as processes, data structures, modules, functions, procedures, and variables are declared. There is no difference between an access to a declared item and a language-defined syntactic element. For example, function calls of language-defined functions and user-defined functions are same in Lisp.

However, in most of meta-programming environments, an access to a declared item is realized as a syntactic element with a `name` attribute which has name of the declared item. For example, a call for function `X` represents a node of `function-call` class with name attribute `X`. However, a function call for system defined function `svref` is realized as a node of `svref` class. This realization is not a natural.

To realize these accesses naturally, MUSE introduces the concepts of *internal-class*. A declaration creates a class with its name of declared item called an internal-class. In this case, the declaration of function `X` creates an internal class `X` dynamically. An access to `X` (function call of `X`) is represented as an instantiation of `X` class. An internal class is memorized into a `symbol-table` attribute of `scope` class or its subclasses. Syntactic elements with an unique scope such as modules, procedures, functions, and blocks are defined as subclasses of `scope` class.

Figure 3.12 shows (a) a declaration of function `test`, (b) a function call of `test`, and (c) a definitions of its internal class. An internal class is defined as a subclass of `function-call` class.

Because some declared items such as procedures and functions have a structure, an internal class for them should have struct links. In this case, `test` class has two struct links corresponding to arguments of the function. An internal-class is generated by a probe of `declaration` class. This probe is activated when the programmer sets a `name` attribute to an identifier node. In this case, when a `name` attribute of `identifier` node is set on `test`, a probe of `defun` class,

```
:probe (((id name) ...) ...)
```

is activated and creates the internal-class `test`.

```
(defun test (a &key b) ...)
```

(a) Definition of `test`

```
(test <form> :key <form>)
```

(b) Function Call of `test`

```
(defclass test
  :super     (function-call)
  :struct    ((a :class form)
              (b :class form :range (0 1))
  ...)
```

(c) Internal Class of `test`

Figure 3.12: An Example of Internal-Class

By introducing an internal-class, a syntactic checking and a static semantic checking become more easily, because information for checking is declared into an internal-class clearly and the checking mechanism for an internal-class is same as for other cases. From the view point of software tools and the object manager, an internal class is same as a class defined in a module. However, the running costs and description costs to write a probe consider to increase.

3.4 Facilities Provided by MUSE

This section describes facilities provided by software tools in MUSE and presents the definition formats for each software tools.

3.4.1 Template-Based Editing

MUSE provides a pure template-based editor. The editor activates basic editing operations directly. Figure 3.13 shows a organization of the editor. The basic editing

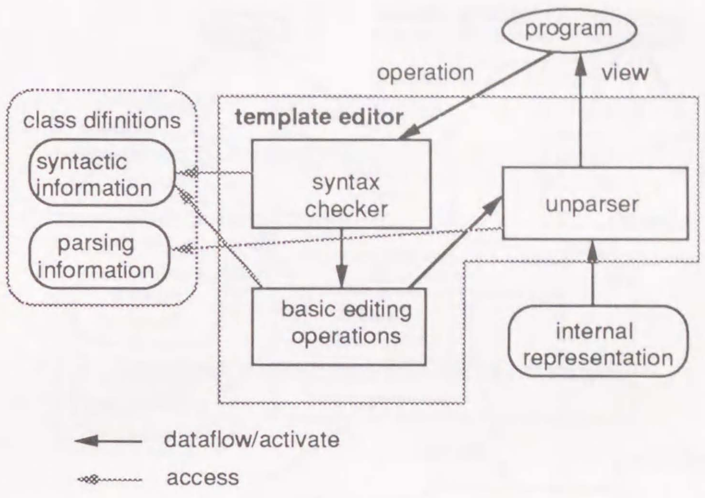


Figure 3.13: Configuration of Template-Editor

operations are as follows.

Replace: This operator takes one argument, class name. This operation replaces a class of the focused node with a node of the given class. After replacement, attributes and links of a replaced node which has the same name with the given class's are copied into the new node (Fig. 3.14).

Insert/Append: Insert and append take no argument. These create a new link and a connected node before (insert) or after (append) the focused node.

Create: This operation takes one argument, a name of struct link. This operation creates a link of the given argument on the focused node and a connected node.

Delete: This operation deletes a subtree of focused node.

Set-Attribute: This operation takes two arguments, a name of attribute and a value. This operation sets the given value into the given attribute of the focused node.

Before activating these operators, the editor checks syntactic correctness of the operation. If editing operation violates the constrains of struct links, it causes a syntax error. When an error is detected, an error message is shown to the programmer and the operation is canceled. This checking guarantees the syntactic correctness of software object to other software tools. This is important to work software tools safety, because software tools is designed under the syntactic correctness of an internal representation.

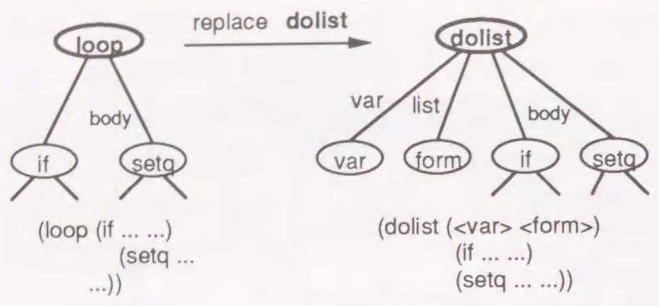


Figure 3.14: Editing Operation - `Replace` -

3.4.2 Design Editing

Figure 3.15 shows the organization of a design editor in MUSE. Editing commands of the design editor, such as `move`, `add/delete node`, `connect/disconnect`, and `set label` are associated to the basic editing operations as described in the previous subsection.

When the programmer inputs a design editing operation, the *design command analyzer* activates basic editing operations corresponding to the command. This correspondency is defined in the *design editor description* and the *design format.* Some commands such as `move` and `resize` do not call basic operations. Before calling basic operations, the analyzer checks the syntactic correctness. If the operation is not correct syntactically, the operation is canceled. Otherwise, basic operations are activated and they updates the internal representation.

After updating the internal representation, the design command updates *design objects* in the window object space. A design object corresponds to a graphical object displayed in the design editor. There are three types of design object, *node, connector,* and *port.* Figure 3.16 shows design objects in Fig. 3.5. Process `X`, process `A`, and process `B` are nodes. `a`, `b`, and `c` are connectors. A connector connects two nodes. A port places on the edge of a node and attaches a connector to the node.

A *design editor description* is defined in one module. It gives basic relationships between an internal representation and a design object. It is used to interpret a design editing operation into basic editing operations. Because correspondency between a node in an internal representation and a design object depends on a design language, this description is required. For example, in some cases, a node corresponds to a design object *node*, but in other cases, a node may correspond to a *connector.*

Figure 3.17 is an example of design editor description in a `dataflow` language. A struct description (`:struct`) shows that a parent/child relationship between two nodes in the internal representation corresponds to a `contain` relationship between

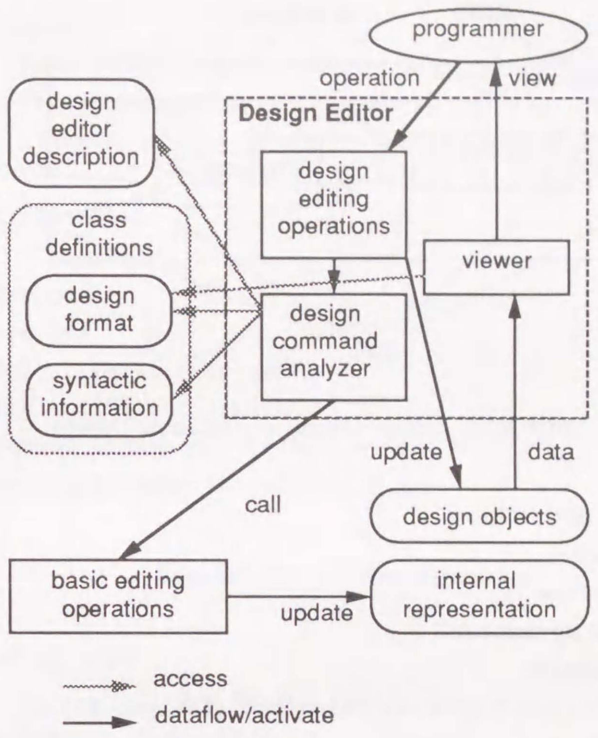


Figure 3.15: Configuration of Design Editor

two nodes in design objects. A node description (:node) shows that a node in an internal representation corresponds to a node or a connector. A command description (:command) decides a semantic-links which are created by design editor command connect.

A design format is defined for each class definition. This information decides relationships between an instance of the class and a corresponding design object. This information is used to interpret the design command to the class. Figure 3.18 is an example of design format for process class in dataflow language. The design format shows that a node of process class corresponds to a node in design object. The shape of the design object is box and its label is a name attribute of the node in an internal representation. Two port definitions :in-port and :out-port represent that these ports correspond semantic link input and output, respectively.

3.4.3 Attribute Propagation and Static Semantic Checking

An attribute evaluation and propagation is important to represent *context sensitive information* and *static semantics*. In particular, *type checking* is a most typical static semantic checking. Some meta-programming environments use an attribute grammar

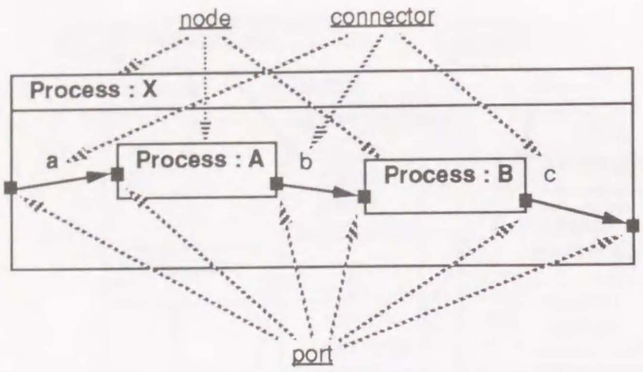


Figure 3.16: Design Object in Dataflow Design

```
(defdesigneditor data-flow
  :struct   contain
  :semantic connector
  :object   (node connector)
  :command  ((connect
             (($1 :class process :semantic ((output $3)))
              ($2 :class process :semantic ((input $3)))
              ($3 :class dataflow :semantic ((from $1)
                                              (to $2))))))
```

Figure 3.17: Design Editor Description

or its extensions. Others are an action routine, a context relation, and a message passing.

To reduce the description costs, MUSE uses an attribute relation which is similar to the context relation in PSG. The attribute relation propagates both real value of an attribute and its constraint. A constraint propagation is mainly used in the modification support system. An attribute relation is defined as a class attribute and it is parsed into probes and methods. An example of attribute propagation and type checking is described as follows.

Figure 3.19 is an attribute relation for type attribute in svref class and Table 3.2 is its table format. A *propagate equation* (:propagate) represents a propagation scheme of type attributes. An element of the equation corresponds to a node connected to a link in *link description* (:link). A *constraint description* (:constraint) represents constraints for type attributes of connected nodes. The pseudo link *self* indicates the node itself.

The second elements of the *propagation equation* (simple-vector $x $y) shows

```
(defclass process
  :super     box
  :struct    ((sub :class element :range (0 infi)))
  :semantic  ((input  :class dataflow :range (0 infi))
              (output :class dataflow :range (0 infi)))
  :attribute ((name)))

(defdesign process
  (node :shape box
        :label    (attribute name)
        :node     ((struct sub))
        :in-port  ((semantic input))
        :out-port ((semantic output))))
```

Figure 3.18: A Design Format

```
(relation type
  :link      (index vector self)
  :propagate ((mod $x))
             (simple-vector $x $y)
             $y)
  :constraint ((mod infi)
             (simple-vector (mod infi) t)
             t))
```

Figure 3.19: Attribute Relation of `svref` Class - Lisp -

that the type is `simple-vector` and its dimension and element type are same as variable `$x` and `$y`, respectively. Two variables `$x` and `$y` show `index` and `element` type of a node connected to the `vector` link are same as `type` attributes of nodes connected to `index` link and the node itself, respectively. And also, these variables represent bidirectional propagation of `type` attributes.

A *constraint* part gives constraints which the attribute must satisfy. A `type` attribute of a node connected to `vector` link must be a `simple-vector` type and its `index` type must be a positive value `(mod)`. The constraint `t` accepts all types. Attributes are updated for each editing operation incrementally. When a constraint error is detected, all propagated attributes and the editing operation is canceled.

In order to check the type correctness, information about the *type hierarchy* of the target language is defined in a module. In particular, a programming language which has a type hierarchy, such as Ada and CommonLisp, requires information for checking

link	index	vector	self
propagation	(mod $x)	(simple-vector $x $y)	$y
constraint	(mod infi)	(simple-vector (mod infi) t)	t

Table 3.2: Attribute Relation of svref Class - Table format -

type constraints.

Figure 3.20 is a part of a definition of type hierarchy for CommonLisp. A type X satisfy a type constraint Y which is a superclass of the type X. For example, a type simple-vector satisfies the constraint vector and the type vector satisfies the constraint array and t. A subtype satisfies a type constraint which is a subtypes of the type in CommonLisp. But, this is not allowed in Ada.

The following describes a simple example of type attribute propagation. At first,

```
<form>
```

is replaced by svref class. The result is

```
(svref <form> <form>).
```

```
(def-type-hierarchy Lisp
    (t
       (list
           (a-list)
           (cons))
       (array
           (vector
                (simple-vector)
                ...)
           ...)
       ...))
```

Figure 3.20: Definition of Type Hierarchy in Lisp

The corresponding internal representation is shown in Fig.3.21. Type-const attributes of nodes connected to index and vector links are set to constraints field of attribute relation of svref. A type-const attribute is a constraint for a type attribute given from its context. If a type attribute does not satisfy a type-const attribute, this type causes a type constraint error. Next, the programmer replaces the second

`<form>`, which is a node connected to `vector` link, with a node of internal-class `A`, which has a type `(simple-vector (mod 8) fixnum)`. The result program is

```
    (svref <form> A)
```

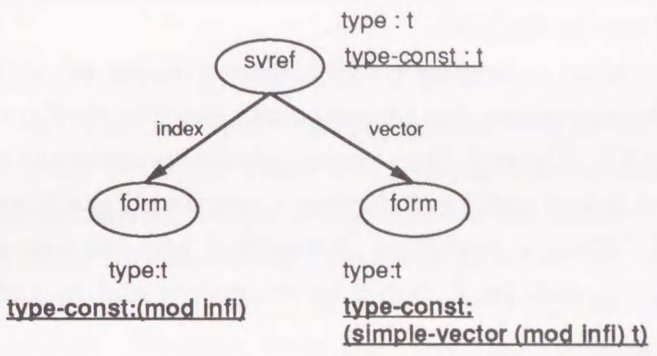


Figure 3.21: Example of Attribute Propagation

Since, type of A satisfies the type-const `(simple-vector (mod infi) t)`, this replacement is acceptable. The corresponding internal representation is shown in Fig. 3.22. By matching `type` attribute of node `A` and the second part of propagation equation `(simple-vector $x $y)`, variables `$x` and `$y` bind values `(mod 8)` and `fixnum`, respectively. The value of `$x` is propagated into a `type-const` attribute of a node connected to the `index` link. The value of `$y` is propagated into the `type` attribute of `svref` node. Since a `type` attribute is a synthesized attribute and the `type` attribute of `svref` node is decided, the value of `$y` is set into the `type` attribute. Vice versa, the value of `$x` is set into a `type-const` attribute. The propagated attributes satisfy their constraint.

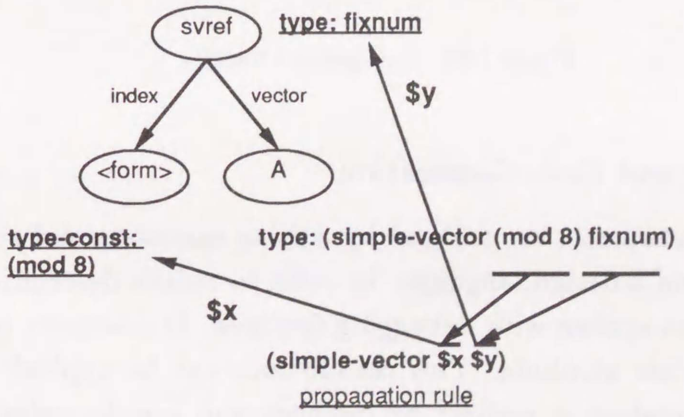


Figure 3.22: Example of Attribute Propagation

3.4.4 Software Management

A software development project requires the management facilities for managing the many software objects. By defining a language to manage software objects and by using gates, management facilities such as *project management*, *configuration management*, and *version control* can be realized.

For example, to define a language which defines nodes as compilation units and semantic links as dependencies, the language can provide configuration management facilities such as MAKE. Figure 3.23 shows an internal representation of a management information. `Module B` and `module C` depend a `module A`, and they refer the exported information from `A`. When a module `A` is modified and the exported information is updated, `module B` and `module C` should be re-checked and re-compiled. A probe

```
((depend export) ...)
```

of the `module` class can achieve this process automatically.

Management facilities realized in a language-oriented environment can avoid unnecessary operations. For example, MAKE recompiles a program in which a comment is modified. Management on a language-oriented system can avoid this unnecessary works, because it can recognize comments.

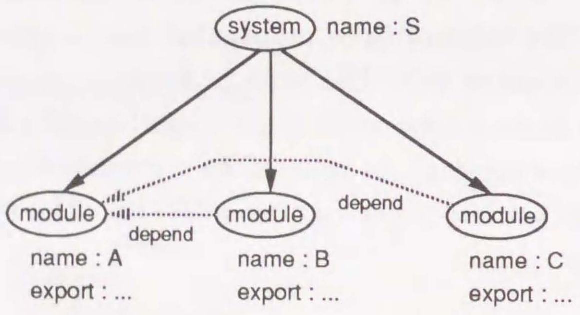


Figure 3.23: Management Facility

3.4.5 Debugging and Code-Generation

The debugger and interpreter are realized by writing executing and simulating methods for a program and a design language. In order to reduce description costs, MUSE supports an execution system with debugging facilities. It interprets pseudo assemble codes written in a class attribute. This pseudo code can be applied to a Pascal-like language. Code-generation is realized by methods and pseudo codes in class definitions, and the code-generator on MUSE generates SOA codes [Tsubotani 86]. These are described in detail in [Tsubotani 88].

3.5 Remarks

This chapter described a meta-programming environment MUSE which aims to support design and coding phases. By writing meta-descriptions in the meta-description language, the environment builder can built an integrated environment for a particular language.

In the design goals of MUSE, language-specific facilities and the design editing functions are realized. And also, features of an integrated programming environment are provided. Software tools in MUSE share the internal representation of software objects. The template-based and design editor provide the uniform user interface and play a front-end of other software tools. Facilities provided by MUSE are editing facilities, incremental syntactic and static semantic checking, debugging, management, and modification support. However, from the view point of inter-phases supporting, transformations and inconsistency checking between different kinds of software objects are necessary. The transformation may be achieved by providing transformation rules. Inconsistency checking will be realized by adding a logical notation into the meta-description language and creating its interpreter.

The object-oriented meta-model is better for describing various kinds of software objects than other static models. Furthermore, the modularity and reusability is higher. However, the description cost of methods and probes is not so high. So, some formal describing notations for particular software tools such as an attribute relation, design formats, and unparsing scheme are supported. These description formats can be created only by analyzing and formalizing behavior model of software tools.

In order to increase the usability of MUSE, more quick response of the environment is needed and an incremental parser should be incorporated into.

Chapter 4

A Modification Support System on MUSE

This chapter presents a modification support system on MUSE. At first, software modification and supporting facilities are discussed. Next, the author presents an approach and the system behaviors. Problems and some future directions of a modification support studies are discussed.

4.1 On Software Modification Supporting Facilities

A software modification is that the programmer updates a part of a software. In this chapter, the author considers a software modification and supporting facilities for a modification.

4.1.1 Problems in Software Modification

As discussed in Section 1.3, a software object is modified many time during its life. A software is modified by various reasons, such as

- changes of clients' requirements,
- bug fix,
- changes of design decisions,
- enhancements of capabilities of software system, and
- porting the system,

in the development and the maintenance.

Before discussing the problems of software modification, programming activities needed after a software modification are shown in Fig. 4.1. When a client of a system offers new requirements, a programmer must add functions to designs. Creations and updates of designs require corrections of related designs and programs. These works cause side-effects and need further corrections on them.

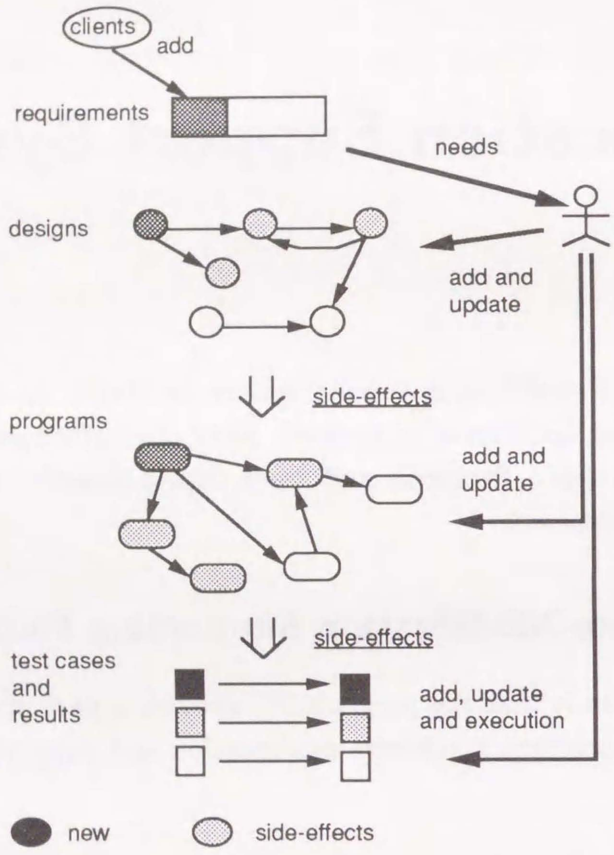


Figure 4.1: Activities after a Software Modification

In addition to these corrections, the programmer should check and correct documents, manuals, and test cases. In particular, in order to check the correctness of the modification, new test cases and executions of them are necessary.

The programmer must deal with these works after each software modification. A modification support aims to reduce burdens of programming activities caused by a software modification.

Concerning a software modification, there are following problems.

1. Side-Effects

A modification causes a lot of side-effects on the modified software object and other software objects. So, the programmer should correct these side-effects. However, these corrections are tedious and may be routine works, and waste the intelligent ability of the programmer. A correction is not an essential work. Unfortunately, moreover, a correction itself may cause side-effects. Furthermore, a programmer misses to correct side-effects frequently. This decreases the reliability of the software.

2. Breaking Software Structure

   Many modifications on a software object breaks a structure of it. Modifications change *well-structured* software object into *ill-structured one*. This decreases the reliability and the productivity of a software development. In particular, an *ad hoc* correction on a small piece of a software object such as loopholes, type conversions, and unnecessary global variables, and a breaking visibilities make further modifications difficult. Furthermore, such corrections may cause unrecognizable errors.

3. Breaking Consistency

   A modification may break relationships and consistencies between software objects. For example, even if a programmer corrects only one program, the correction may break consistencies between the program and related designs. Usually, a software is developed by a project team which consists of many persons, and the developer and the maintenance person of the software are different. These facts make corrections more difficult.

These problems are serious, and decrease the productivity and reliability of a software development. In order to remove above problems, a systematic approach to support a software modification should be studied.

4.1.2 Reducing Modification Cost

A modification needs many programming activities. In order to reduce the costs for a modification, several approaches are proposed and used. There are

- *object-oriented programming*,
- *modular programming*,
- *automatic programming*,
- *configuration management*, and

• *supporting facilities to correct side-effects.*

These are effective and cooperative approaches.

An object-oriented programming limits the area of side-effects narrow. A modification in a method does not propagate side-effects outside of the method. The dynamic binding of an activated method realizes the ***polymorphism*** of a message passing. Recently, some languages introduce object-oriented facilities such as Objective-C, C++, new Flavors, LOOPS, and CLOS (CommonLisp Object System).

In a modular programming, a programmer creates a program as a set of modules. A module has an *interface*, which specifies an exported information and an imported information between other modules, and a *body* which specifies the implementation of the module. A *package* in Ada and a *module* in Modula-2 are examples of module. A module does not propagate side-effects caused by a modification of the body to outside of the module.

An automatic programming aims to create an executable code from a non-executable description automatically. Because programs are generated from a specification, corrections of programs are not necessary. However, an application domain of automatic programming is very narrow. There is no complete general purpose automatic programming system. An example of configuration management, MAKE, removes works for recompilation of programs after a modification. It works on according to the description of dependencies between programs.

An automatic correction of side-effects caused by a modification will reduce costs of software modification rapidly. This approach resolves the problem (1) and maybe (2) and (3). However, this has not been researched except for a few works. A modification support system presented in this dissertation aims to realize the automatic correction of side-effects. There are several facilities to help corrections of side-effects, such as

(a) detecting side-effects,

(b) advice for corrections,

(c) automatic corrections, and

(d) checking correctness of corrections.

Furthermore, from another points of view, these facilities are divided into supporting

(i) in one software object,

(ii) between same kinds of software objects, and

(iii) between different kinds of software objects.

The targets of the presented system are (a) and (c) in (i).

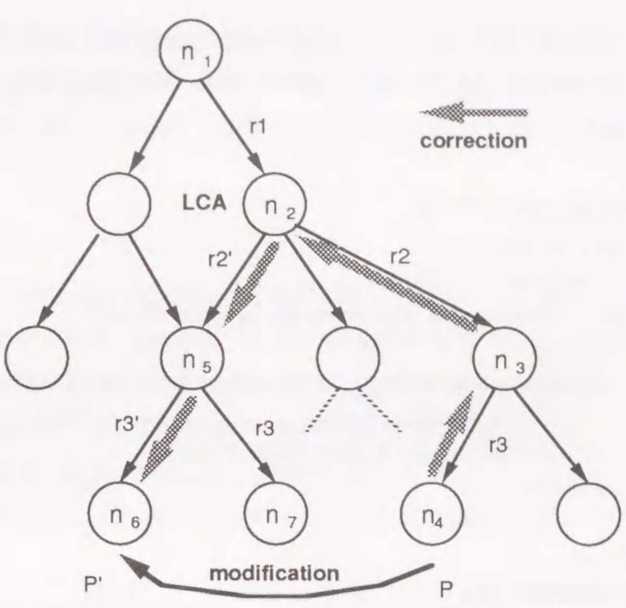


Figure 4.2: DAG in TMM

4.1.3 Related Works

A few works try to automate corrections caused by a modification. TMM [Arango 86] in Draco project [Freeman 87] is a maintenance support system. TMM uses a DAG (Directed Acyclic Graph) which represents possible decisions in a particular domain as shown in Fig. 4.2. A node in DAG represents an abstract level of a program. This example is a DAG in a *sort algorithm* domain. An edge represents a *refinement* from an abstract level to a concrete level. The program *P* uses *r1, r2 (Use-HeapSort)*, and *r3*. The programmer changes a design decision *r2* into the *r2' (Use-QuickSort)*. TMM reverses design decisions to the least common abstraction (LCA). In this example, the decisions go up from node *n4* to the node *n2*. Next, selects another design decisions from LCA. The result program is *P'* on node *n6* which uses *r1*, *r2'*, and *r3*. These processes are executed automatically. TMM reduces modification costs, however a complete domain analysis and construction of DAG are needed. Cost of these works will be large.

KBEmacs (Knowledge-Based Emacs) in Programmer's Apprentice project uses a *cliche* and a *plan*. Figure 4.3 is a cliche for `file_enumeration`. A role (enclosed by {}) is a kind of *placeholder* and *constraints filed* gives constraints to roles. A cliche represents a fragment of a program code. A plan is a knowledge representation of a program based on dataflow and controlflow. A modification to text representation is reflected into the corresponding plan. KBEmacs evaluates properties on the plan, and by using constraints in a cliche, some roles are filled and replaced automatically. This function realizes the automatic corrections, but they are limited and in *ad hoc* realization [Waters 85].

```
clich FILE_ENUMERATION is
     primary roles FILE;
     described roles FILE;
     comment "enumerates the records in {the file}";
     constraints
          RENAME("DATA_RECORD", SINGULAR_FORM({the file});
          DEFAULT({the file_name},
          CORRESPONDING_FILE_NAME({the file});
     end constraints;

     FILE: {};
     DATA_RECORD: {};
begin
     FILE := {the input file};
     OPEN(FILE, IN_FILE, {the file_name});
     while not {END_OF_FILE, the empty_test} (FILE) loop
          {READ, the element_accessor} (FILE, DATA_RECORD);
          {DATA_RECORD, the output data_record};
     end loop;
     CLOSE(FILE);
exception
     when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
          CLOSE(FILE); PUT("Data Base Inconsistent");
     when others => CLOSE(FILE); raise;
end FILE_ENUMERATION;
```

Figure 4.3: The cliche `file_enumeration`

4.1.4 Treatment of Modification

In order to treat a modification well, formalization of a software modification and properties of a modification is needed. The key feature of a modification is a programmer's intention of a modification, *"Why he/she modifies a software object?"* and design decisions on the software object . Following are view points of treatment of a modification.

(1) Software Representation

A software representation, on which a modification support system works, should be able to represent semantics of a program well. Text representation is unconventional for a modification support system. An attributed abstract syntax tree can represent syntax, but it is not sufficiently. A better representation is needed. Knowledge-based software representations are studied, but widely accepted representation is not developed yet.

(2) Software Components

This is a target unit by a modification operation. This unit is also the unit treated by supporting system. In order to realize better supporting, sizes of software components should be a higher abstraction level. A support system can not treat a modification in character level. But, it will be able to treat a modification in algorithmic level. This suggests that a programmer makes a software object by combining semantical software components. A modification is performed on such components.

(3) Side-Effects

In order to detect side-effects of a modification, the system should represent the side-effects. A context and static semantics can be represented as attributes. An attribute propagation will be able to realize computing side-effects in a software object.

(4) Modification Operations, Direction of Modification

A semantical editing operation helps a high level of treatment of a modification. For example, an abstraction command

```
add a new member AGE to record type PERSON
```

is higher than editing commands in characters. By defining detailed corrections specialized to each higher level operations, a system will provide better supporting facilities. In particular, it is important that an operation represents a direction of a modification such as inserting guards, enhancing speed/space, dividing cases, and changing *magnitude* to *set*. However, a tradeoff between the description costs and the capability of the system exists. In the presented system, from the view point of generality, it provides and supports only few basic template-based editing commands.

4.2 An Approach to Correction of Side-Effects

This section presents a method to correct side-effects caused by a modification. Figure 4.4 shows an overview of the approach. This approach works on an internal representation in MUSE. Each node represents one of software component such as algorithmic fragments, design scheme, functions, and declared items. A node has properties to represent syntax and semantics of the component, and also a node has constraints from neighboring components, such as *type-const attributes* and *linkclass constraints.* Before a modification, all node satisfy its constraints.

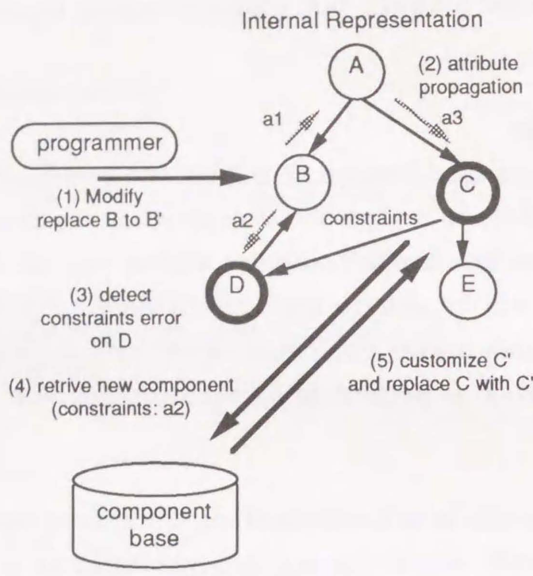

Internal Representation
A
(2) attribute propagation
a1
a3
programmer
B
C
(1) Modify replace B to B'
a2
constraints
D
E
(3) detect constraints error on D
(5) customize C' and replace C with C'
(4) retrive new component (constraints: a2)
component base

Figure 4.4: An Approach to Correct Side-Effects

When a programmer modifies a software object by changing a component in the internal representation, the modification destroys the consistency of the software by changing properties of a component. Changed properties are propagated to other components. When a property of a component breaks its constraints, the component which caused a side-effect should be corrected.

The correction is achieved by replacing the side-effected component with another component which removes all inconsistencies on the component. This correction problem is divided into two subproblems, (1) searching a new software component and (2) replacing an old component with new one.

To search a component from a *component base*, following information is used.

(i) constraints given from neighbor components,

(ii) decisions on an old component, and

(iii) directions of the modification.

The searched component should satisfy (i) for removing inconsistencies. The author thinks that a correction is achieved by customizing the previous programmers' decisions to meet the intention of a modification. So, a founded component should have a role which is moved to (iii) from (ii).

There are two types of corrections as shown in Fig. 4.5. A correction should be achieved to meet the updated new properties or constraints by the modification. A correction must not change the new property or new constraint. Because, they can be regarded as to represent an intention of modification. In case Fig. 4.5 (a), the propagated or changed new properties of component A do not satisfy constraints from components B and C. In this case, a component which gives constraints to A should be replaced. The new component must give constraints which the component A can accept.

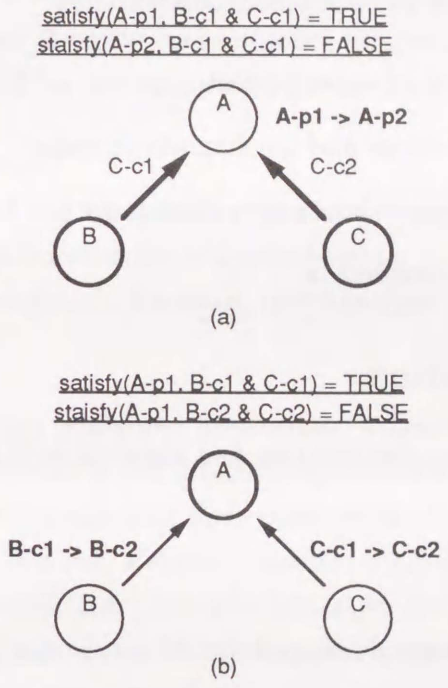

satisfy(A-p1, B-c1 & C-c1) = TRUE
staisfy(A-p2, B-c1 & C-c1) = FALSE
A
A-p1 -> A-p2
C-c1
C-c2
B
C
(a)
satisfy(A-p1, B-c1 & C-c1) = TRUE
staisfy(A-p1, B-c2 & C-c2) = FALSE
A
B-c1 -> B-c2
C-c1 -> C-c2
B
C
(b)

Figure 4.5: Two Types of Side-Effects

Otherwise, case (b), constraints given from B and C are changed and an inconsistency occurs. This case requires the replacement of component A.

After finding a new component, the approach replaces the old component with the new one. However, the old component has many properties. The replacement must not remove these properties. They are copies indirectly through transformations according to the correspondences between the old component and the new one as shown in Fig. 4.6. This process is called *component customization* in this dissertation.

To summarize this approach, following key points should be considered and decided to develop a system based on this approach.

- software representation

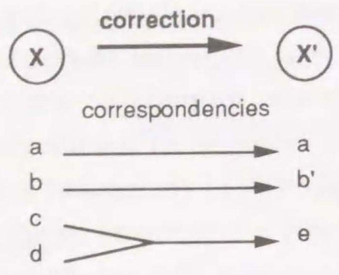


Figure 4.6: Component Customization

- representation of properties of a component
- method to propagate properties and constraints of node
- method to search a component in a component base
- method of component customization

4.3 System Organization

This section presents the system organization and algorithms of a modification support system developed on MUSE.

4.3.1 Design Goals

The modification support system developed on MUSE is designed for the following design goals and limitations.

(1) Automatic Correction

The system detects and corrects side-effects automatically. The system does not provide the advice facility and does not check the correctness of the modification.

(2) Language Independence

A modification and a correction are caused in various languages. The system separates the language-dependent part of the system from other language-independent parts. The language-dependent part is given from a meta-description. This enhances the extendability and the customizability of the system.

(3) Description Costs

Minimize an increase of a meta-description costs by introducing the modification support.

(4) Target Modification and Constraints

The target modification of the presented system is a type modification in a program written in a programming language. Because a type is an important semantical property of a program, and a type and its propagation is well formalized as shown in Section 2.2. In addition to the type constraints, the system treats syntactic inconsistencies, i.e., linkclass constraints.

(5) Directions of Modification

The system does not treat the direction of a modification. Because, it is difficult to formalize directions and it needs large number of modification operations. Directions of a modification should be incorporated into the future system.

(6) Range of Propagation

Further limitation of the system is range of a propagation of side-effects. Of course, side-effects are propagated to other software objects. However, the system treats propagation in only one program. Because, propagations between software objects need heavy costs.

Based on above design goals and limitations, following design decision are decided. The software representation is the internal representation of MUSE. This is important for cooperating and communicating the system with other software tools in MUSE such as editors, a syntactic checker, a static semantic checker, and an attribute propagator.

For representing properties, type attributes on nodes are used. For representing constraints, *type-const attributes* and *link constraints* are used. Type attributes and type-const attributes are propagated according to the attribute relation.

In order to realize component customization, a *copy-rule* is introduced into a class (component) definition. This shows correspondecies between properties of the class and properties of its superclass.

4.3.2 Algorithmic Behaviors

Figure 4.7 shows the organization of the modification support system. Software modules enclosed by a dotted lines are components of a modification support system.

- An *attribute propagator* propagates attributes from the modified node.
- A *type checker* checks the correctness of propagated attributes. These two tools are also used for checking static semantics of a program. When the checker detects an inconsistency, it activates an *automatic corrector*.

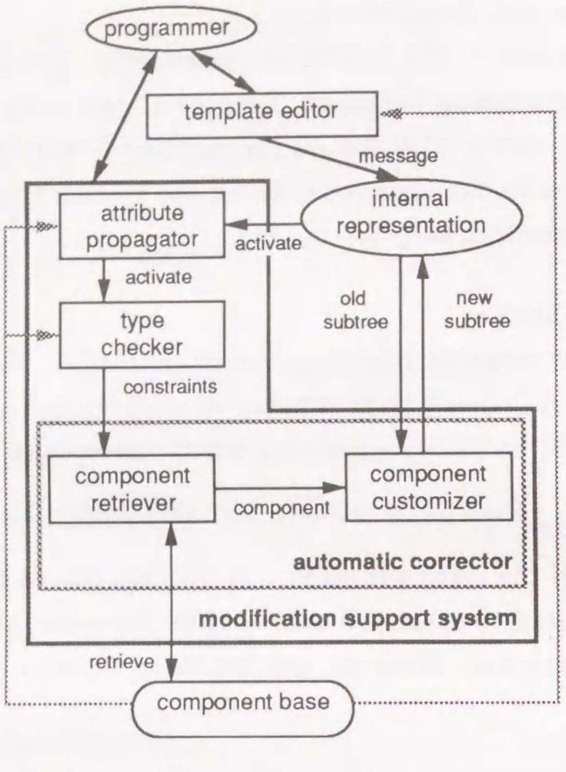
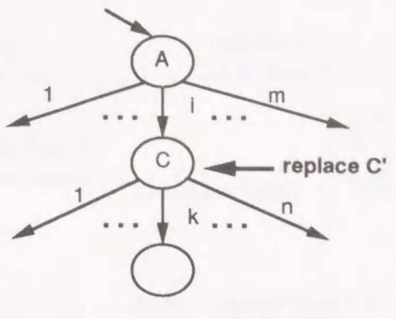


Figure 4.7: Organization of Modification Support System

- A *component retriever* decides the replaced node at first. Next, it searches a class with constraints in the component base, i.e., class hierarchy, for the target programming language.
- A *component customizer* instantiates a node from the searched class and it customizes the node by copying properties of the replaced node.

Before a modification, all nodes of an internal representation satisfy linkclass constraints and type constraints. Figure 4.8 shows an example of internal representation and its related attribute relation. In this case, a programmer replaces node `C` with node `C'`. The command sends a message to a `replace` method defined in class `C'`. The activated method replaces the node and propagates `type` and `type-const` attributes. A side-effect is detected by checking type constraints. A node on which a side-effect is detected should be corrected.

As described in Section 4.2, when an inconsistency occurs between a new attributes and its constraints, a node which gives the constraint is corrected. Otherwise, when an inconsistency occurs between an old attribute and its new constraints, the node which has an old attribute is corrected. The following is the steps of the activated method.



(a) Internal Representation

link name	self	1	...	n
propagation				
constraint				

(b) Attribute Relation of C'

Figure 4.8: An Example of Modification

Figure 4.9 shows an algorithm of a general `replace` method.

Step 1: Replace `C` with `C'`.

Step 2: An attribute propagator sets type constraint attributes to nodes connected with `C'` according to the attribute relation of `C'` shown in Fig. 4.8. In Fig. 4.9, `link(k)` means that a node connected with the `k-th` link in an attribute relation. A `type-const` is a constraint attribute for type attribute and `constraint(C', k)` is a value of `k-th` constraint field in the attribute relation of class `C'`.

Step 3: A type checker checks type consistencies on the children nodes. When an inconsistency is detected, it calls an automatic corrector and corrects the node. Since a correcting operation itself contains modification operations, the correction will cause further automatic corrections.

These three steps update and correct children nodes of `C`, and types of them are elaborated. Next, type attribute of `C'` is decided according to the propagation field of the attribute relation.

After deciding the type attribute of `C'`, a probe which looks at the `type` attribute of `C'` is activated automatically. The probe computes side-effects and corrects nodes. In Fig. 4.9, a probe defined in `A` is activated. It propagates type attributes and corrects itself and neighbor nodes except to `C'`. Figure 4.10 shows the algorithm of general probe which is activated by updating a type of a target node. If necessary, an automatic

```
replace method
    replace-node(C, C')

    for i := 1 ~ n
        link(i).type-const := constraint(C', k)

    for i := 1 ~ n
        if update(link(i).type-const) then
            if not satisfy(link(i).type, link(i).type-const) then
                correct(link(i, link(i).type-const))

    for i := 1 ~ n
        env := unify(propagate(C', i), link(i).type)
        self.type := assign(propagate(C', self), env)
```

Figure 4.9: Algorithm of Replace Method

```
probe(target, type)
    if update(target.type) then
        if not satisfy(target.type, constraint(A, target)) then
            correct(self, target.type)
```

Figure 4.10: Algorithm of Probe

corrector replaces `A` with `A'` which can accepts a type of `C'`. This correction propagates type attributes to nodes connected with `A'` and corrects the connected nodes.

By replacing an old node on which an inconsistency occurs with a new node which resolves the inconsistency error, the old node is corrected. The term *old node* is used to represent a node on which an inconsistency occurs. There are four types of inconsistencies as shown in Fig. 4.11.

(a) Node A is new, and its link constraints (linkclass) of l is not a superclass of X. X should be replaced with a subclass of the linkclass.

(b) Node B is new and it does not satisfy the linkclass of l. X should be replaced with a class which linkclass accepts B.

(c) Node C has new type constraint *C-const* and the node X has type *X-type* which does not satisfy the constraint *C-const*. The node X should be replaced with a node which has a type satisfying the constraint *C-const*.

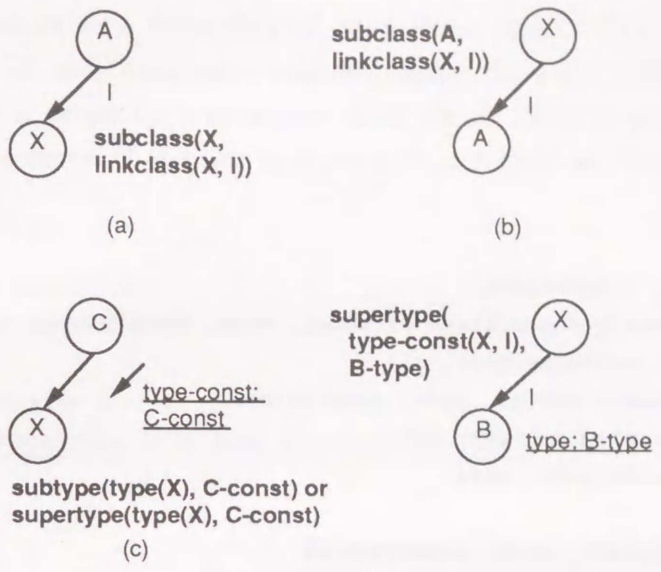


Figure 4.11: Inconsistency and Retrieve Constraints

(d) Node *D* has new type *D-type*, a node *X* which gives a constraint through a link *l* should be replaced with a node which gives constraints on which a node *D* can accept.

Since a programmer selects a class in a class hierarchy for creating a program, the position of the class in the class hierarchy can be regarded as the role of the class. So, the component retriever searches a class which satisfies constraints toward superclasses of the class of old node which should be replaced. Then, it searches a class, which fits the constraints at most, toward the subclasses of the founded superclass. For saving the previous programmer's decisions on the old node, the component customizer copies properties of the old node into the new node instantiated from the best fit subclass.

Steps of an automatic correction is described as follows and Fig. 4.12 shows algorithms of the correction.

Step 1: A component retriever searches a class which satisfies the given constraint from the old class to its superclasses. In Fig. 4.12, the function `resolve` returns `TRUE`, when the class `c` satisfies the constraints.

Step 2: A component customizer instantiated a node from the founded superclass and copies properties of old node into the newly instantiated node according to the *copy-rule*. A copy-rule is defined as one of class-attribute. A copy-rule represent correspondences between properties of the node and properties of its superclass. For example, a copy-rule shown in Fig. 4.13 represents `a-list` link of `push-a-list`

```
correct(node, constraints)
     c1 := search-super(class-of(node), node, constraints)
     temp := customize(node, c1)
     c2 := search-sub(c1, node, constraints)
     new := customize(temp, c2)
    replace-node(node, new)

search-super(class, node, constraints)
      for c := supers(class)
      if resolve(node, c, constraints) then
           return c
      else
           search-super(c, node, constraints)

search-sub(class, node, constraints)
     if not resolve(class, node, constraints) then
          return FALSE
     for c := subs(class)
     if cs := search-sub(c, node, constraints) then
          return cs
     return class

customize(old, class)
     new := instantiate(class)
     for k := properties(old)
          if i := same-name(class, k) or i := copy-rule(class, k) then
               new.i := old.k
```

Figure 4.12: Algorithm of Correction

class corresponds to a `data` link of `push-data` class. This mapping is used to change the old node into more abstract node. In Fig. 4.12, `same-name` shows that there is a property `i` in a new class which name is same as a name of `k`, and copy-rule shows that there is a property `i` which can be mapped into `k` by a copy-rule.

```
(defclass add-a-list
    :super (add-data)
    ...
    :class-attr (
        (copy-rule (add-data ((link data)(link a-list))
                ...)
            ...)
        ...)
    ...)
```

Figure 4.13: Copy-Rule of `add-a-list` Class

Step 3: A component retriever searches a class which satisfy constraints more accurate from the superclass to its subclasses. This step finds a more efficient and a more concrete class than the superclass.

Step 4: The component customizer instantiates a node of the found class in Step 3, and copies properties from a node customized in Step 2 to the newly instantiated node in Step 4.

These steps changes an inconsistent part of a program into a corrected part of a program.

4.4 System Behaviors

In this section, behaviors of the modification support system is described.

4.4.1 Corrections for a Type Modification

Figure 4.14 shows an example of a CommonLisp program. The function `def-a-list` defines a subtype of `a-list` (associate list) named `relation`. A *cons pair* in the `relation` type consists of a `symbol` type (*key* and *car* part) and `t` type (*value* and *cdr* part).

```
(def-a-list <data> nil
    :key 'symbol
    :value 't)

(defvar <data> nil
    :type 'relation)

(defun add-data (key value)
    (push (cons key value) <data>))

(defun get-data (key)
    (cdr (assoc key <data>)))
```

Figure 4.14: An Example of Lisp Program

In order to enhance access speed to `<data>`, a programmer replaces `a-list` type with `hash` type. He/she moves the cursor in the template-editor onto `def-a-list` and inputs a command: `replace def-hash`. This modification requires corrections as follows.

- `nil` is the initial value of `a-list`, but an initialization of `hash` is `(make-hash-table)`.
- In the function definition `get-data`, an access to an `a-list` data, `(cdr (assoc key <data>))` must be replaced with an access to a `hash` data, `(gethash key <data>)`.
- In the function definition `add-data`, a store to an `a-list` data, `(push (cons key value) <data>)` must be replaced with an store to a `hash` data, `(setf (gethash key <data>) value)`.

Figure 4.15 shows the result of corrected program after above three corrections. A modified part and modified part and corrected parts are represented as underlined texts.

Editing Operation

Figure 4.16 shows parts of an internal representation which corresponds to Fig. 4.14. Each node has a `type` attribute and a `type-const` attribute. The editing command activates a `replace` method of `def-hash` and it replaces `def-a-list` node with `def-hash` node.

```
(def-hash 'relation
    :key 'symbol
    :value t)

(defvar <data> (make-hash-table)
    :type 'relation)

(defun add-data (key value)
    (setf (gethash key <data>) value))

(defun get-data (key)
    (gethash key <data>))
```

Figure 4.15: A Corrected Program

Customization of Subtree

Next, the component customizer is activated. It copies attributes and links from `def-a-list` to `def-hash` according to the copy-rule. In this case, `def-hash` and `def-a-list` have three links with the same name, i.e., `id`, `key-type`, and `value-type`. By copying properties, syntactic errors and type inconsistencies are fixed. A correction of the modified node changes children nodes and it causes further corrections. These processes corrects all nodes in the subtree of the modified node.

Attribute Propagation

After the corrections of the subtree, side-effects on parent nodes and nodes connected by semantic links are detected and corrected. Table 4.1 shows the attribute relation of `def-hash`. The type attribute of `def-hash` is set to the result of

```
(hash "evaluation of \$x" "evaluation of \$y")}.
```

In this case, `$x` binds a `type` attribute of the node connected to the `key` link, and `$y` binds a `type` attribute of the node connected to the `value` link. These variables represent bidirectional propagation of attributes by pattern matching.

The attribute propagator propagates `type` attributes and `type-const` attributes by referring attribute relations. Figure 4.17 shows propagation of type attributes.

1. The type attribute of `def-hash` node is set to `(hash symbol t)` and `type-const` is set to `t`. `t` is the most super type in CommonLisp type hierarchy. So, the type attribute satisfies the `type-const` attribute.

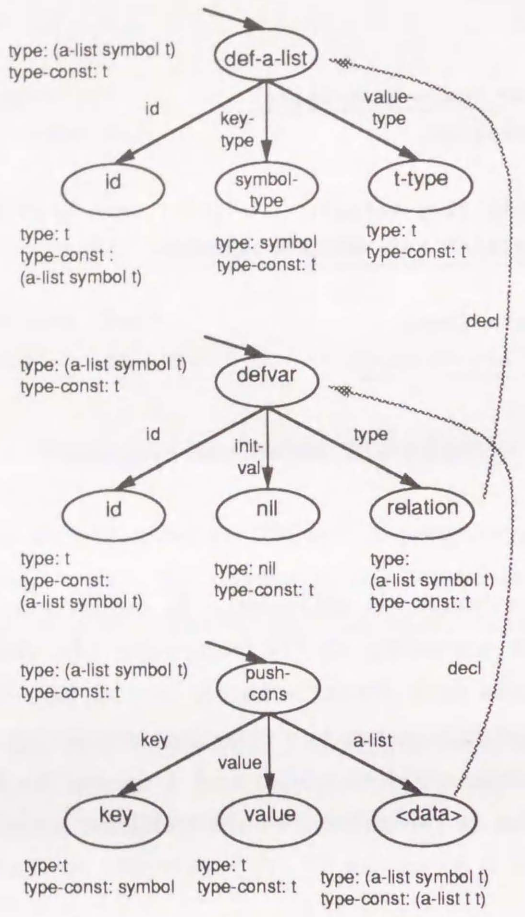


Figure 4.16: Internal Representation of Original Program

link class	self	id	key-type	value-type
propagation	(hash $x $y)	(hash $x $y)	$x	$y
constraints	(hash t t)	t	t	t

Table 4.1: Attribute Relation of `def-hash`

2. In the same way, type attributes of id, key-type, value-type are set to t, symbol, and t, respectively.
3. By changing type attribute of def-hash, the relation node which connects def-hash node by a semantic link updates a type attribute of itself.
4. This updating activates a probe of defvar node and it updates type attributes of itself and id node. These occur no inconsistency.
5. This updating updates type attributes of <data> node in the function definitions add-data and get-data. Figure 4.17 shows only a part of add-data.
6. The type and type-const attributes of <data> in the function add-data are set to (hash symbol t) and (a-list t t), respectively. This causes a type inconsistency, because hash is not a subtype or supertype of a-list. This requires replacement of push-cons which gives constraints to the type of <data> node.

Retrieve Component to Superclass

To remove the inconsistency, the component retriever searches a class with the condition, that is, which can store a data into a hash with symbol key. This can be represented as retrieve constraint explicitly. In this case, there is a *retrieve constraint* given from relationships between a type and a type-const attributes,

```
(sub-or-super ((link a-list)(attribute type-const))
       ((link a-list)(attribute type))).
```

This means that a type-const attribute of the node connected the a-list link must be a super or subtype of a type attribute of the node which is connected the a-list link. In this case, ((link a-list)(attribute type)) is evaluated and replaced with (hash t t). So, this constraint is evaluated into

```
(sub-or-super ((link a-list)(attribute type-const)) (hash t t)).
```

The retrieving is started from push-cons class to its superclasses, because a superclass gives looser constraints. Figure 4.18 shows a part of the class hierarchy of CommonLisp.

First, add-a-list is examined, but it does not satisfy retrieve constraints. Therefore, the retriever tries to examine add-data which is more superclass of add-a-list. The retriever applies the copy-rule to the retrieve constraint. Because add-a-list class has a copy-rule to the superclass add-data,

```
(add-data ((link data))(link a-list)))
```

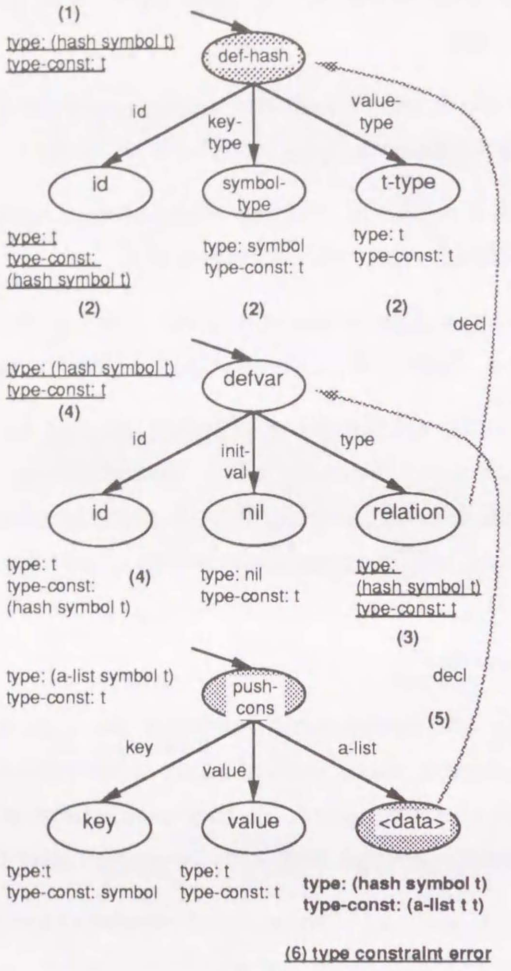


Figure 4.17: Attribute Propagation after a Modification

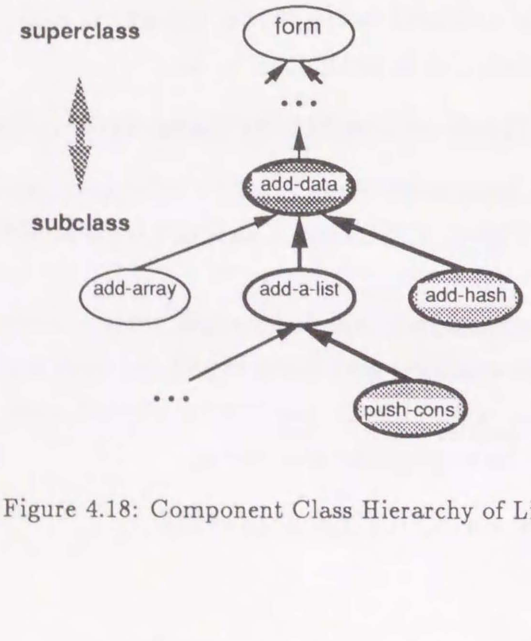


Figure 4.18: Component Class Hierarchy of Lisp

between a-list link and data-list. This represents a mapping from a a-list link of add-a-list class to a data link of add-data class. The new retrieve constraint to add-data which is applied by the above copy-rule is

```
(sub-or-super ((link data)(attribute type-const))
              (hash t t)).
```

The add-data class satisfies this condition, because hash which is given from constraint field of the attribute relation is a subtype of t type. These relationships are given in the language description as shown in Fig. 3.20. When a class is retrieved, the retriever calls the customizer.

Component Customization to Superclass

The component customizer transforms the push-cons node and its properties into the retrieved add-data node and its properties. This process is shown in Fig. 4.19.

First, the customizer transforms push-cons into add-a-list. Attributes and links with the same name, such as type, type-const, key, value and a-list are copied from push-cons to add-a-list directly. Second, add-a-list is transformed into the add-data. Attributes and struct links key and value are copied directly, but a link a-list can not be copied. The customizer transforms this link into data link according to the copy rule of add-a-list,

```
(add-data ((link data))(link a-list))).
```

Retrieve Component to Subclass

The class add-data is a common and more abstract class which add something into the data with key. But, this is not executable and not concrete. In order to get more concrete and executable class, the retriever is activated again. The retriever searches a class which satisfies the retrieve constraints more *accurate* from the superclass to its subclasses. In this case, at first, the retriever examines all direct subclasses of add-data. The add-hash satisfies the retrieve condition more accurately. The constraint for add-hash class is

```
(sub-or-super ((link hash) (attribute type-const)) (hash t t)).
```

This is already applied to a copy rule of add-hash class,

```
(add-data ((link data)) ((link hash))).
```

An *accurately* means that the distance of two types in type-hierarchy is smaller. For example, the distance of hash and hash is 0 and the distance of hash and t is 1.

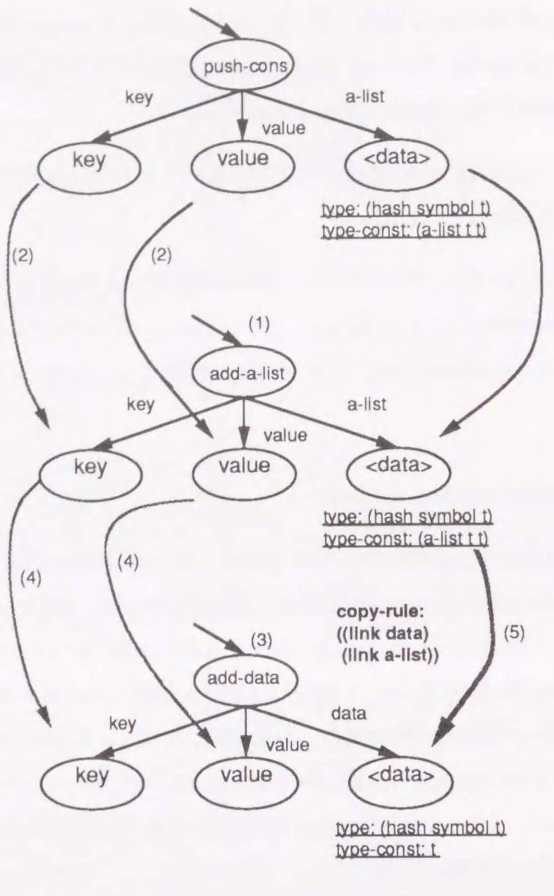


Figure 4.19: Customization from add-cons to add-data

After all examines, the retriever selects the most accurate class, and then continues retrieving in subclasses of the selected class. In this case, add-hash is selected, and it has no subclasses. The retriever calls the customizer and gives add-hash class to the customizer.

Component Customization to Subclass

The customizer copies properties of add-data node into add-hash node. Figure 4.20 shows this process. As the same processing, the function definition of get-data in Fig. 4.14 is corrected into the function definition in Fig. 4.15. By replacing a node as above steps, attributes of the newly placed node are computed, and if necessary, side-effects are propagated to the neighbor nodes, except to a node which is corrected already, i.e., <data> nodes. Of course, some of them cause inconsistencies and automatic corrections.

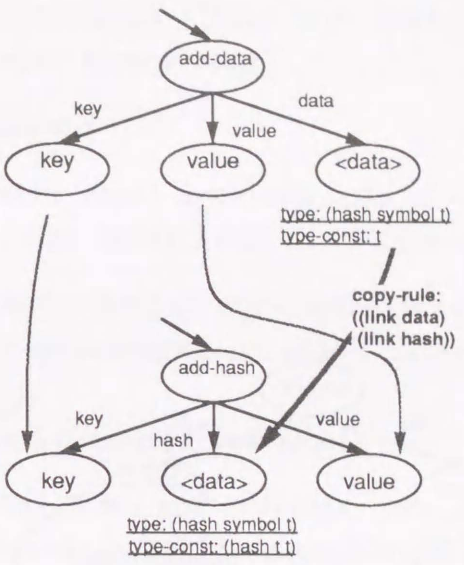


Figure 4.20: Customization from `add-data` to `add-hash`

4.4.2 A Correction in More Complex Case

The author shows a little more complex case of component customization. An original program consists

```
(cdr (assoc 'age <data>)).
```

`a-list` which is a type of variable `<data>` is changed into the `struct-person` type. The above form must be corrected to the next form

```
(person-age <data>).
```

Figure 4.21 shows a customization in this correction. The `get-a-list` node is replaced with `access-key` node which is retrieved with a retrieve constraint

```
(sub-or-super ((link data) (attribute type-const)) struct-person)
```

and a copy-rule

```
(access-key ((link data)) ((link a-list))).
```

Next, the retriever examines the subclasses of `access-key` class. A `person-age` class has a copy-rule

```
(access-key ((link key) (attribute name)) ((class-attr key)))
```

and retrieve constraint which is defined in the class definition of `person-age` explicitly. This is

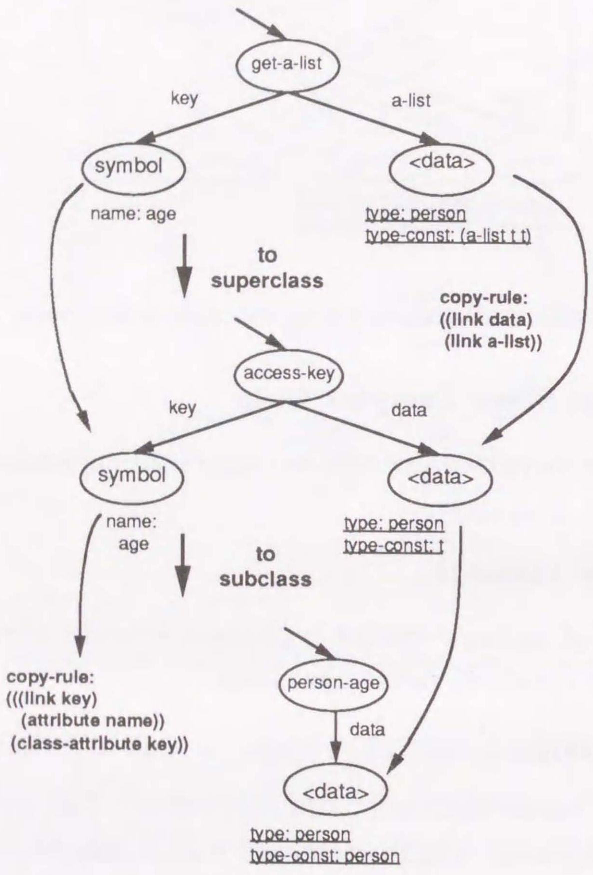


Figure 4.21: Customization from `get-a-list` to `person-key`

```
(equal ((class-attr key)) ((link key) (attribute name))
       :when (super access-key))
```

So, the retrieve constraints are

```
(equal ((class-attr key)) age) and
(sub-or-super ((link data) (attribute type-const)) struct-person).
```

The `person-age` class satisfies these retrieve constraints and the customizer transforms `access-key` node into the `person-age` node. In this customization, `key` link is reduced.

4.4.3 Corrections for Syntactic Constraints

Above examples shows side-effects and corrections related to types. This subsection shows corrections for syntactic errors, i.e., inconsistencies of linkclass constraints. An example of an original program is

```
(when <cond>
      <body1>
      <body2>),
```

and a programmer replaces `when` node with `if` node. Then, `body` links which are `<body1>` and `<body2>` are copied into `then` part of `if` node by using copy-rules from `when` class to `boolean-cond` class and `boolean-cond` class to `if` class. The result program is

```
(if <cond>
    (progn  <body1> <body2>))
    nil).
```

The next example is an inconsistency between a new node and its parent node, i.e., a child node breaks a linkclass constraint given from its parent node. Figure 4.22 shows a process of this correction. A part of an original program is

```
(setq X (list ...))
```

and a programmer replaces the symbol `X` with `(cdr X)`. The function `setq` takes a `symbol` class as the first argument. So, the customizer replaces this fragment with an abstract class `<set>`. Next, the abstract fragment is transformed into the `setf` node. This class can take one of *place* functions as a first argument, such functions are `car`, `cdr`, `svref`, `aref`, and `gethash`. The corrected program is

```
(setf (cdr X) (list ...)).
```

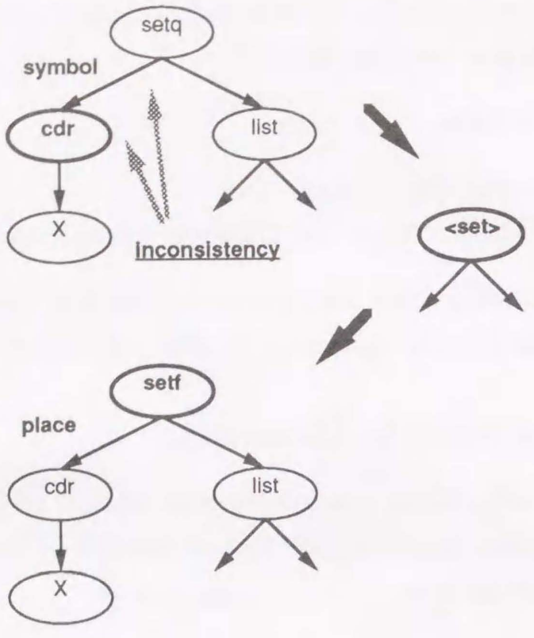


Figure 4.22: Automatic Correction for Link Constraints

4.5 Remarks

Since a cost for modification is very large, a modification support system increases the productivity of software development rapidly. In particular, an automatic correction of side-effects frees the programmer from routine works after a modification.

This chapter presents an approach which automates corrections of side-effects and a modification support system based on the approach. This approach bases on a software representation which is composed by software components and an attribute propagation. An automatic correction is realized by retrieving component in a class hierarchy with a retrieve constraints. The retrieved component is customized according to copy-rules and it replaces the original component.

Since the system is realized language independent, a builder can realize the modification support facilities for a particular language by describing a meta-description. An increase of description cost to introduce the modification support is not so large. Only, copy-rules and retrieve constraints are required in addition to an original meta-description.

The approach and the system are applicable to wide domains. The presented approach can treat a modification and work on any size of components.

However, some problems remain as follows.

1. This system works for type modifications, but this is not completely. For example,

in Fig. 4.15, a correction of initialization is impossible in the current system. The reason of this impossibility is a lack of initializing components. If `nil` component and `make-hash-table` component are defined as subclasses of an `initialize` component, this correction will become possible.

2. The current system can not treat side-effects of a modification which does not change type attributes. In this case, the system supports only customization and corrections for nodes in the subtree of the modified node.
3. A correctness of automatic correction is important, but it is not guaranteed. The current system notifies the programmer *before* and *after* the correction for each correction.
4. The system can not avoid a *loop dependency* of corrections.

Some future directions of modification support are discussed as follows.

Target Modification

The current target is type modification only. To treat other types of modifications on the presented approach, other attributes and constraints such as assertions, data flow, and control flow are needed. Furthermore, more semantical representation of a program, such as a *plan representation* of Programmer's Apprentice, should be developed for more advanced facilities.

Support between Different Kinds of Software Objects

During a software development, various kinds of software objects are created and modified. In order to keep consistencies between various software objects, supporting facilities between different kinds of software objects should be provided. This will be realized to memorize decisions between two software objects in form of refinement rules. The modification support system should customize such memorized decisions to meet the programmer's intention of a modification.

Support between Software Objects

The current system can treat only one software object. Of course, a software system consists of many software objects. However, there is a problem concerning a modification support between software objects, *"When does the system propagate side-effects?"* The current system propagates side-effects *immediately*. But, in a large software system, this approach is not practical, because the propagation costs becomes very large.

To solve this problem, a ***delayed propagation*** should be supported. This consists two solutions as follows.

1. Propagation before Editing:

   Before a programmer edits a software object, the system searches all depending software objects and propagates side-effects from them. Then, the modification support system corrects the target module.

2. Runtime Propagation:

   The system collects side-effects before the software object is executed or edited. Then, the system corrects the module according to the corrected side-effects. This approach corrects executing data in addition to program codes. The author calls these facilities as a runtime modification support. This facility is useful for debugging and should be supported in an interactive environment such as Smalltalk-80 and Lisp environment. Because these environments are always in active, a programmer will not want to stop the environment and to remove lots of objects in it.

These solutions need a formalization of side-effects and a formalization of direction of a modification. A modified software object remembers a list of modifications, side-effects, and directions of modifications. Furthermore, a method to reduce and to remove overridden side-effects in the list is needed.

Based on the above discussions, our research group is now developing a next modification support system, an *object-oriented environment with runtime modification support.* It is characterized as follows.

At first, the target language is *typed* and *constrained* object-oriented language. Since side-effects are limited in an object-oriented language, the system can concentrate only important side-effects between classes. Since a software unit is only class, the system considers only constraints and relationships between classes and objects.

Second, the system aims to provide a runtime modification support facilities. The next system does not immediately propagate side-effects outside of a class. When an object receives a message and its time stamp is older than one of its class, the system corrects the object before evaluating a method. But, if superclasses of the class or its related classes are updated, the class is corrected before correcting the object.

Third, to treat a direction of a modification, the system limits modification operations to a class. A class is updated only by sending a message defined in a meta-class of the target class.

Lastly, a programmer can define any constraints between objects in the class definition. This will help to correct objects. Now, the research group design and implement the next system. However, many problems remain yet.

Chapter 5

Conclusions

In order to increase the productivity of software development, an integrated programming environment supports programming activities. To construct an integrated environment for a particular language and to adapt the environment for a particular application domain, meta-programming environments have been studied on. In addition to support coding activities, recent works on meta-programming environment aim to realize facilities for various activities on meta level. In particular, a modification support will increase the productivity rapidly and frees programmers from non-creative works.

An approach to develop a meta-programming environment and an approach to correct side-effects caused by a modification have been discussed in this dissertation. A meta-programming environment MUSE which is based on an object-oriented meta-model is proposed. A modification support system to correct side-effects for type modifications is proposed and designed on MUSE.

In Chapter 1, researches on software tools and environments based on the software engineering technologies and backgrounds of this research was discussed. In particular, integrated programming environments, meta-approaches, and problems of software modifications are discussed.

In Chapter 2, studies on meta-programming environments are outlined. A language-oriented environment and description method for language features are presented. Following the discussions on recent studies on meta-environments, an approach to construct a meta-programming environment is proposed.

A meta-programming environment MUSE based on the object-oriented meta-model was presented in Chapter 3. MUSE can provide language-oriented facilities for each various kinds of software object by switching language knowledge. A software object is represented as a collection of objects, and an object is instantiated from a class defined in a meta-description. A class is defined for one of various software units. Attributes of an object and their constraints are propagated according to the declarative description, attribute relation. A gate makes it possible to define a language for management facilities. By defining languages for formatted documents and information written in the MUSE meta-description language, the environment builder can provide various advanced facilities which work on the document.

A modification support system, presented in Chapter 4, corrects type inconsistencies and linkclass inconsistencies caused by a modification automatically. A correction is realized by replacing a node on which an inconsistency occurs into another component which removes inconsistencies. This is achieved by retrieving a new component in the class hierarchy by using retrieve conditions. Component customization is done by copying the original properties into the new component.

The design and implementation of the meta-programming environment has been described in detail. By using the prototype system, language-oriented facilities for Pascal, Lisp, dataflow, simple management, and simple spread-sheet are implemented. In particular, for Lisp, a modification support facilities are realized by adding copy-rules and retrieve constraints. Through these implementations, the effectiveness of this approach, and the advantage of the object-oriented meta-model, was confirmed.

Further researches are needed on the following issues.

(1) Supporting Facilities between Software Objects

These facilities such as inconsistency checking and transformation can increase the productivity and the reliability. These are realized by transformation and consistency rules and its evaluators.

(2) Incorporating Other Kinds of Software Knowledge

In particular, knowledge about software process will make it possible to activate software tools automatically. In order to realize a modification support between different phases and different kinds of software objects, a process knowledge and transformation tools will be useful for representing programmer's decisions between different kinds of software objects.

(3) Automatic Correction between Software Objects

To realize modification support between software objects, a method for delayed propagation of side-effects should be developed. In particular, a runtime modification support mechanism, which propagates side-effects and corrects executing data at runtime, makes programmers possible to update the developing and executing environment itself without suspending it.

Appendix A
A Prototype System

MUSE can work on workstations which has KCl (Kyoto CommonLisp), C, and X Window Version 11 Release 2 and 3. Current MUSE works on Sun3[1] with SunOS4.0[2] and Luna with UniOS-B[3]. Major parts of MUSE is written in KCl and window interface is written in C. A meta-description language is realized as a surface language of KCl. Some system functions are realized in system-defined class written in the meta-description language. Gate, context, declaration, and scope are realized as classes definitions.

Program size of MUSE is over 11,000 lines of KCl and 1,500 lines of C and C++. The total size of executable module is over 2M bytes. Number of lines and classes of meta-descriptions are shown in Table A. 1.

name	classes	lines
Ada	123	1835
CommonLisp	64	1673
DataFlow	13	157
Pascal	96	1697
Project Management	7	96
Root	16	236
Spread-Sheet	3	99
system	4	109

Table A. 1: Sizes of Meta-Descriptions

[1] Sun3 is a trademark of Sun Microsystems.

[2] SunOS is a trademark of Sun Microsystems.

[3] UniOS is a trademark of OMRON Corporation.

Appendix B
Syntax of Meta-Description Language

The context-free syntax of the meta-description language presented in Chapter 3 is described using a simple extension of Backus Normal Form.

```
module ::= module_definition class_definition_part

module_definition ::=
          (defmodule module_identifier
            [:import (module_name {module_name})]
            [:export (class_name {class_name})]
            [:context class_name]
            [:root class_name])

class_definition_part ::= class_definition {class_definition}

class_definition ::=
          (defclass  class_identifier
            [:super (class_name {class_name})]
            [:struct (link_specification {link_specification}
                                          [:inherit nil])]
            [:semantic (link_specification {link_specification}
                   [:inherit nil])]
            [:attribute (attribute_specification
  {attribute_specification}
  [:inherit nil])]
```

```
          [:class-attr class_attribute_body]
          [:method (method_definition {method-definition})]
          [:probe (probe_definition {probe_definition})] )

link_specification ::= (link_identifier :class class_name
                [:range range_constraint])

range_constraint ::= (integer integer) | :0-1 | :0-i | :1-i

attribute_specification ::=
          (attribute_identifier [:default lisp_expression])

class_attribute_body ::=
          ([(format format_definition)]
           [(relation relation_definition)]
           [(condition condition_definition)]
           [(copy-rule copy_rule_definition)]
           {class_attribute_specification})

format_definition ::= format_header format_body

format_header ::= (simple format_type) | (complex format_type)

format_type ::= typeA | typeB | typeC | typeD

format_body ::= (format_item {format_item})

format_item ::= string | (struct link_name) |
                (attribute attribute_name)


relation_definition ::= (relation {relation})

relation ::= (attribute_name
         (:link (link_name {link_name})
          :prop (prop_var {prop_var})
          :const (attribute_value {attribute_value})))
```

```
condition_definition ::= (condition {condition})

condition ::= (predicate location location)

location ::= ({node_location} location_end)

node_location ::= (link link_name) {(link link_name)}

location_end ::= (attribute attribute_name) |
                 (class class_name)


copy_rule_definition ::= (copy_rule {copy_rule})

copy_rule ::= (class_name node_location node_location)

class_attribute_specification ::=
          (class_attribute_identifier lisp_expression)


method_definition ::= (method_identifier lambda_expression)

probe_definition ::=
          ((link_name attribute_name) lambda_expression)
```

Bibliography

[Aho 77] A. Aho and J. Ullman, *Principles of Compilers,* Addison-Wesley, 1977.

[Aho 86] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1986.

[Allman 86] E. Allman, "An Introduction to the Source Code Control System," *in UNIX Programmer's Supplementary Documents,* vol. 1, 1986.

[Arango 86] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon, "TMM: Software Maintenance by Transformation," *IEEE Software,* vol. 3, no. 3, pp. 27–39, May 1986.

[Bahlke 85] R. Bahlke and G. Snelting, "The PSG - Programming System Generator," *in Proceedings of the ACM SIGPLAN Symposium Language Issues in Programming Environments,* pp. 23–33, Jun. 1985.

[Bahlke 86] R. Bahlke and G. Snelting, "The PSG System: From Formal Language Definitions to Interactive Programming Environments," *ACM Transactions on Programming Language and Systems,* vol. 8, no. 4, pp. 547–576, Oct. 1986.

[Bahlke 87] R. Bahlke, B. Moritz and G. Snelting, "A Generator for Language-Specific Debugging Systems," *in Proceedings ACM Proceedings of the SIGPLAN Symposium on Interpreters and Interpretive Techniques,* pp. 38–44, Jun. 1987.

[Barstow 84a] D. Barstow, "A Display-Oriented Editor for INTERLISP," *in Interactive Programming Environments,* D. Barstow, H. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, pp. 288–299, 1984.

[Barstow 84b] D. Barstow and H. Shrobe, "From Interactive to Intelligent Programming Environments," *in Interactive Programming Environments,* D. Barstow, H. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, pp. 558–570, 1984.

[Bashili 87] V. Bashili and H. Rombach, "Tailoring the Software Process to Project Goals and Environments," *in Proceedings of the IEEE International Conference on Software Engineering,* pp. 345–357, 1987.

[BSI 87] British Standards Institution, *Specification for Computer language Pascal,* BSI BS-6192, 1982.

[Buxton 80] J. Buxton and L. Druffel, "Rationale for STONEMAN," *in Proceedings of the IEEE 4th International Computer Software and Applications Conference,* pp. 66–72, Oct. 1980.

[CMU84] Carnegie-Mellon University, *ALOE Users' and Implementors' Guide (Fourth Edition),* 1984.

[Dandwall 78] E. Dandwall, "Programming in an Interactive Environment : The LISP Experience," *Communications of the ACM,* vol. 10, no. 8, Mar. 1978.

[Dart 87] S. Dart, R. Ellison, P. Feiler, and A. Harbermann, "Software Development Environments," *IEEE Computer,* vol. 20, no. 11, pp. 18–28, Nov. 1987.

[Delisle 84] N. Delisle, D. Menicosy, and M. D. Schwartz, "Viewing a Programming Environment as a Single Tool," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 49–56, Apr. 1984.

[DoD83] United States Department of Defense, *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD-1815A, Jan. 1983.

[Dolotta 78] T. Dolotta, R. Haight, and J. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *The Bell System Technical Journal,* vol. 57, no. 6, July-August 1978.

[Donnelly 88] C. Donnelly and R. Stallman, "BISON: The YACC-compatible Parser Generator," *Free Software Foundation,*1988.

[Donzeau-Gouge 80] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, " Programming Environments Based on Structured Editors: The MENTOR Experience," *INRIA Technical Report,* no. 26, May 1980.

[Donzeau-Gouge 84] V. Donzeau-Gouge, G. Kahn, and B. Lang, and B. Melese, "Document structure and modularity in Mentor," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 141–148, Apr. 1984.

[Dowson 86] M. Dowson, "ISTAR–An Integrated Project Support Environment," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 27–33, Dec. 1986.

[Evans 83] A. Evans, K. Butler, G. Goos, and W. Wulf, *DIANA Reference Manual,* Pittsburgh: Tartan Laboratories Inc., 1983.

[Feldman 79] S. Feldman, "Make-A Program for Maintaining Computer Programs," *Software Practice and Experience,* vol. 9, Apr. 1979.

[Fischer 84] C. Fischer, G. Johnson, J. Mauney, A. Pal, and D. Stock, "The POE Language-Based Editor Project," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 21–29, Apr. 1984.

[Freeman 87] P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," *IEEE Transactions on Software Engineering,* vol. SE-13, no. 7, pp. 830–844, July 1987.

[Fritzson 84] P. Fritzson, "Preliminary experience from the DICE system, a distributed incremental compiling environment," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 113–123, Apr. 1984.

[Garlan 84] D. Garlan and P. Miller, "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 65–72, Apr. 1984.

[Goldberg 83a] A. Goldberg and D. Robson, *Smalltalk-80: The Languages and Its Implementation,* Addison-Wesley, 1983.

[Goldberg 83b] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, 1983.

[Handa 86] T. Handa and M. Kori, "A Generator for Syntax-Directed Programming System," *Transactions on Institute of Electronics, Information and Communication Engineering Japan, Part ,* D vol. J69-D, no. 11, pp. 1605–1616, Nov. 1986.

[Harbermann 86] A. Harbermann and D. Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering,* vol. SE-12, no. 12, pp. 1117–1125, Dec. 1986.

[Horgan 84] J. Horgan and D. Moore, "Techniques for Improving Language-Based Editors," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 7–14, Apr. 1984.

[Horwitz 85] S. Horwitz and T. Teitelbaum, "Relations and Attributes: A Symbolic Basis for Editing Environments," *in Proceedings of the ACM SIGPLAN Symposium Language Issues in Programming Environments,* pp. 96–106, Jun. 1985.

[Horwitz 86] S. Horwitz and T. Teitelbaum, "Generating Editing Environments Based on Relations and Attributes," *ACM Transactions on Programming Language and Systems,* vol. 8, no. 4, pp. 577–608, Oct. 1986.

[Huff 88] K. Huff and V. Lerser, "A Plan-based Intelligent Assistant That Supports the Software Development Process," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 97–106, 1988.

[Johnson 75] S. Johnson, "Yacc: Yet Another Compiler Compiler," *Computing Science Technical Report 32, AT&T Bell Laboratories,* 1975.

[Johnson 85] W. Johonson and E. Soloway, "PROUST: Knowledge-Based Program Understanding," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 3, pp. 267–275, Mar. 1985.

[Karinthi 87] R. Karinthi and M. Weiser, "Incremental Re-Execution of Programs," *in Proceedings of the ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques,* pp. 38–44, Jun. 1987.

[Kernighan 78] B. Kernighan and D. Ritchie, *The C Programming Language,* Prentice-Hall Software Series, 1978.

[Kernighan 81] B. Kernighan and J. Maskey, "The UNIX Programming Environment," *IEEE Computer,* vol. 14, no. 4, pp. 25–34, Apr. 1981.

[Knuth 68] E. Knuth, "Semantics of Context-Free Languages," *Math. Syst. Theory,* vol. 2, no. 2, pp. 127–145, Jun. 1968.

[Laff 85] M. Laff and B. Hailpern, "SW 2 - An Object-based Programming Environment," *in Proceedings of the ACM SIGPLAN Symposium Language Issues in Programming Environments,* pp. 1–11, Jun. 1985.

[Lesk 75] M Lesk, "Lex: A Lexical Analyzer Generator," *Computing Science Technical Report 39, AT&T Bell Laboratories,* 1975.

[Lichtman 86] Z. Lichtman, "Generation and Consistency Checking of Design and Program Structures," *IEEE Transactions on Software Engineering,* vol. SE-12, no. 1, pp. 172–181, Jan. 1986.

[Medina-Mora 81] R. Medina-Mora and P. Feiler, "An Incremental Programming Environment," *IEEE Transactions on Software Engineering,* vol. SE-7, no. 9, pp. 472–482, Sep. 1981.

[Moriconi 85] M. Moriconi and D. Have, "Pegasys: A System for Graphical Explanation of Program Design," *in Proceedings of the ACM SIGPLAN Symposium Language Issues in Programming Environments,* pp. 148–160, Jun. 1985.

[Moriconi 86] M. Moriconi and D. Hare, "The Pegasys: Pictures as Formal Documentation of Large Programs," *ACM Transactions on Programming Language and Systems,* vol. 8, no. 4, pp. 524–546, Oct. 1986.

[Ramanathan 88] J. Ramanathan and S. Savlar, "Providing Assistant for Software Lifecycle Approaches," *IEEE Transactions on Software Engineering,* vol. SE-14, no. 6, pp. 749–757, Jun. 1988.

[Penedo 88] M. H. Penedo and W. E. Riddle, "Guest Editors' Introduction Software Engineering Environment Architectures," *IEEE Transactions on Software Engineering,* vol. SE-14, no. 6, pp. 689–696, Jan. 1988.

[Reiss 84a] S. P. Reiss, "An Approach to Incremental Compilation," *in Proceedings of the ACM SIGPLAN Compiler Construction,* pp. 144–156, Jun. 1984.

[Reiss 84b] S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 30–41, Apr. 1984.

[Reiss 85] S. P. Reiss, "PECAN: Program Development System that Support Multiple Views," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 3, pp. 276–285, Mar. 1985.

[Reiss 87] S. Reiss, "Working in the Garden Environment for Conceptual Programming," *IEEE Software,* vol. 4, no. 6, pp. 16–27, Nov. 1987.

[Reps 84a] T. Reps and T. Teitelbaum, "The Synthesizer Generator," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 41–48, Apr. 1984.

[Reps84b] T. Reps, *Generating Language-Based Environment,* MIT Press, 1984.

[Rich 78] C. Rich and H. Shrobe, "Initial Report on a LISP Programmer's Apprentice," *IEEE Transactions on Software Engineering,* vol. SE-4, no. 6, Nov. 1978.

[Rich 88] C. Rich and R. Waters, "The Programmer's Apprentice: Research Overview," *IEEE Computer,* vol. 21, no. 12, pp. 10–25, Nov. 1988.

[Sandewall 78] E. Sandewall, "Programming in an Interactive Environment: The LISP Experience," *ACM Computing Survey,* vol. 10, no. 1, pp. 35–71, Mar. 1978.

[Schwartz 84] M. Schwartz, N. Delisle, and V. Begwani, "Incremental Compilation in Magpie," *in Proceedings of the ACM SIGPLAN Compiler Construction,* Jun. 1984.

[Sheil 83] B. Sheil, "Power Tools for Programmers," *Datamation Magazine,* 1983.

[Smith 85] D. Smith, G. Kotik, and S. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 11, pp. 1278–1295, Nov. 1985.

[Sorenson 88] P. Sorenson and J. Trenblay, "The Metaview System for Many Specification Environment," *IEEE Software,* vol. 6, no. 6, pp. 30–38, Nov. 1988.

[Stallman 81] R. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," *in Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation,* pp. 147–156, Jun. 1981.

[Stallman 87] R. Stallman, *GNU Emacs Manual (Sixth Edition, Version 18), Free Software Foundation,*1987.

[Standish 84] T. Standish and R. Taylor, "Arcturus: a Prototype Advanced Ada Programming Environment," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* pp. 57–64, Apr. 1984.

[Stroustrup 86] B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

[Sweet 85] R. Sweet, "The Mesa Programming Environment," *in Proceedings of the ACM SIGPLAN Symposium Language Issues in Programming Environments,* pp. 216–229, June 1985.

[Swinehart 85] D. Swinehart, P. Zellweger, and R. Hagmann, "The Structure of Cedar," *in Proceedings of the ACM SIGPLAN Symposium Language Issues in Programming Environments,* pp. 230–244, Jun. 1985.

[Swinehart 86] D. Swinehart, R. Zellweger, R. Beach, and R. Hagmann, "A Structural View of the Cedar Programming Environment," *ACM Transactions on Programming Language and Systems,* vol. 8, no. 4, pp. 419–490, Oct. 1986.

[Tayler 86] R. Tayler, L. Clarke, L. Osterweil, J. Wileden, and M. Young, "Arcadia: A Software Development Environment Research project," *2nd ICAAE,* pp. 137–149, 1986.

[Teitelbaum 81] T. Teitelbaum and T. Reps, " The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM,* vol. 24, no. 9, pp. 563–573, Sep. 1981.

[Teitelman 81] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *IEEE Computer,* vol. 14, no. 4, pp. 25–34, Apr. 1981.

[Teitelman 84a] W. Teitelman, "A Display-Oriented Programmer's Assistant," *Interactive Programming Environments,* pp. 240–287, 1984.

[Teitelman 84b] W. Teitelman, "A Tour Through Cedar," *IEEE Software,* vol. 1, no. 2, pp. 44–73, Apr. 1984.

[Tenma 86] T. Tenma, H. Tsubotani, M. Tanaka, and T. Ichikawa, "A Generation System for Language-Oriented Editors," *in Proceedings of the IEEE Computer Software & Applications,* pp. 105–112, Oct. 1986.

[Tenma 88] T. Tenma, H. Tsubotani, M. Tanaka, and T. Ichikawa, "A System for Generating Language-Oriented Editors," *IEEE Transactions on Software Engineering,* vol. SE-14, no. 8, pp. 1098–1109, Aug. 1988.

[Tenma 89] T. Tenma, Y. Sato, M. Tanaka, and T. Ichikawa, "Development of a Meta-Programming Environment MUSE," *Transactions on Institute of Electronics, Information and Communication Engineering Japan, Part D-I,* vol. J72-D-I, no. 22, pp. 864–873, Dec. 1989, in Japanese.

[Tenma 90a] T. Tenma, Y. Sato, Y. Morimoto, M. Tanaka, and T. Ichikawa, "A Modification Support System -Automatic Correction of Side-Effects caused by a Type Modification-," *in Proceedings ACM Computer Science Conference,* Feb. 1990.

[Tenma 90b] T. Tenma, Y. Sato, Y. Morimoto, M. Tanaka, and T. Ichikawa, "A Modification Support System -Automatic Correction of Side-Effects caused by a Type Modification-," *Transactions on Institute of Electronics, Information and Communication Engineering Japan, Part D-I,* May 1990.

[Tsubotani 86] H. Tsubotani, N. Monden, M. Tanaka, and T. Ichikawa, "A High Level Language-based Computing Environment to Support Production and Execution of Reliable Programs," *IEEE Transactions on Software Engineering,* vol. SE-12, no. 1, pp. 134–146, Jan. 1986.

[Tsubotani 87] H. Tsubotani, T. Tenma, M. Tanaka, and T. Ichikawa, "A Generation System for Language-Oriented Editors," *Transactions on Institute of Electronics, Information and Communication Engineering Japan, Part ,* vol. J70-D, no. 4, pp. 711–719, Apr. 1987.

[Tsubotani 88] H. Tsubotani, *Study on Language-Oriented Environments,* Dissertation of Doctoral Candidate in Hiroshima University, Jan. 1988.

[Tichy 86] W. Tichy, "An Introduction of the Revision Control System," *in UNIX Programmers Supplementary Documents,* vol. 1, 1986.

[Walker 87] J. Walker, D. Moon, D. Weinreb, and M. McMahon, "The Symbolics Genera Programming Environment," *IEEE Software,* vol. 4, no. 6, pp. 35–45, Nov. 1987

[Wasserman 86] A. Wasserman and P. Pircher, "A Graphical, Extensible Integrated Environment for Software Development," *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 131–142, Dec. 1986.

[Waters 82] R. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 1, Jan. 1982.

[Waters 85] R. Waters, "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, Nov. 1985.

[Wilander 80] J. Wilander, "An Interactive Programming for Pascal," *Behandling Informationtidskrift for Nordisk*, no. 20, pp. 163–174, 1980.

[Winograd 73] T. Winograd, "Breaking the complexity barrier," *in Proceedings of the ACM SIGPLAN/SIGIR Interface Meeting on Programming Languages - Information Retrieval*, pp. 13–30, Nov. 1973.

[Winograd 79] T. Winograd, "Beyond Programming Languages," *Communications of the ACM*, vol. 22, no. 7, pp. 391–401,.

[Wirth 83] N. Wirth, *Programming in Modula-2 (Second Edition)*, New York: Springer-Verlag, 1983.

[Yuasa 85] T. Yuasa and R. Nakajima, "IOTA: A Modular Programming System," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 179–187, Feb. 1985.