

Constructing a Toolset for Software Maintenance with OOAG

Chung-Hua Hu¹, Ji-Tzay Yang², Feng-Jian Wang², and William C. Chu³

¹Information Technology Laboratory, Chungwa Telecommunication Laboratories

²Department of Computer Science and Information Engineering, National Chiao Tung University

³Institute of Information Engineering, Feng-Chia University

e-mail: (¹chhu, ²jjyang, ²fjwang)@csie.nctu.edu.tw, ³chu@fcu.edu.tw

Abstract

*This paper presents a model called **object-oriented attribute grammar** (OOAG) that can be used to construct a toolset for software maintenance. The kernel of OOAG consists of two inter-related parts: a **model-view-shape** (MVS) application framework and an AG++, an object-oriented extension to traditional AGs. By combining compositional and generative techniques seamlessly, OOAG preserves both advantages introduced by respective OO and AG models, such as rapid prototyping, reusability, extensibility, and incrementality. So far, a toolset prototype consisting of a number of programming and maintenance tools were implemented using OOAG on the Windows environment. The editors developed can be used to construct programs by specifying the associated flow information in explicit (visual) or implicit (textual) ways, while the (incremental) maintenance tools, such as DU/UD tools and a program slicer, can help analyze incomplete program fragments to locate and inform the user of useful information.*

1. Introduction

Visual representations are helpful for understanding and maintaining existing software [10]. During program maintenance, a variety of diagrams, such as control-flow and data-flow graphs, can be used to present multi-faceted information about software. In the past decade, many studies concerned with toolsets (or tools) for software maintenance and understanding can be found in the literature. However, most of these toolsets, such as [9][17], were focused on functional descriptions of these visual environments, i.e., what functions that the environments provided. Few actually concern how to construct or generate maintenance toolsets in a systematic manner.

In this paper, we present an approach, called the *object-oriented attribute grammar* (OOAG) model, to integrating *compositional* and *generative techniques* [12] seamlessly, and use this approach to systematically construct a maintenance toolset. This hybrid model consists of two inter-related parts: a *model-view-shape* (MVS) application framework and an AG++, an object-oriented extension to traditional AGs. Within the OOAG model, the MVS framework is mainly responsible for handling graphical user interfaces of tools in the toolset, while AG++ for

handling incremental and context-dependent analysis for given semantic specifications of the tools. The OOAG model, which provides an underlying platform for supporting specification generation and framework specialization, may be practical enough to construct domain-specific applications that deal with fine-grained language semantics as well as a mass of graphics-drawing activities.

So far a toolset prototype composed of several tools for handling programming and various maintenance tasks has been implemented. With current editors developed, one can construct programs by specifying the associated flow information in explicit (visual) or implicit (textual) ways. The (incremental) maintenance tools developed, such as DU/UD (Definition-Use/Use-Definition) tools and a program slicer, can help analyze incomplete program fragments to locate and inform the user of useful information during programming and maintenance phases.

The rest of this paper is organized as follows. Section 2 describes the advantages and weaknesses of OO and AG models, respectively. The internal structure and functionality of OOAG are discussed in Section 3. Sections 4 and 5 describe the system architecture of an OOAG-based toolset for software maintenance, and discuss the implementation aspects of several related tools. The conclusions are summarized in Section 6.

2. Advantages and Weaknesses of OO and AG Models

Object-oriented programming languages and their application frameworks (or class libraries) have gained reputations over these years because they provide powerful mechanisms, such as encapsulation, inheritance, and polymorphism, to help develop good-looking and high-quality applications. An object-oriented framework generally provides a family of inter-related classes, i.e., software components, which can be reused, mainly through composition and inheritance, for the rapid construction of domain-specific applications. The main goal of these frameworks is intended to support the systematic and compositional reuse of large-scale software.

Advantages of object-oriented models are summarized as follows. First, object-oriented frameworks facilitate *rapid application development* (RAD) by providing generic application templates that enable developers to stepwise “fill in” the implementation details of new applications. Second, these

frameworks generally provide both a reusable model and greater flexibility; developers are allowed to extend and evolve the frameworks by augmenting customized components. However, developers usually spend more time learning how to use these object-oriented frameworks than to use generation approaches. One main reason is that a framework comprises a variety of domain classes, such as container, I/O, and GUI classes, with complex inheritance and aggregation structures. Developers have difficulties in constructing fancy applications without giving thorough insights into dozens of classes. Moreover, programming methodology based on framework specialization is still *imperative*. That is, it requires specifying source code manually, which may be less abstract and generally involves more effort than using generation approaches.

AGs are a well-known specification method for specifying semantic constraints and computational dependencies based on the program-tree structure [6]. A change, such as a sub-tree replacement, to a program requires reevaluation of attributes attached to the affected tree nodes. An AG specification does not contain any explicit sequencing of the computations apart from attribute dependencies. Instead, an *attribute evaluator* constructed for the AG is responsible for arranging the associated computations in an order that is compatible with those dependencies. Advantages of traditional AGs can be summarized as follows. First, AGs seem to be better than hand-coded methods due to their theoretical simplicity. Applications may be specified and generated quickly using the generation approach than by framework specialization. This approach facilitates generative reuse by enabling extension and customization of AG specifications. Second, AGs provide more abstract support for specifying attribute dependencies and (optimal) incremental evaluation algorithms, so that attribute evaluations in response to specification changes are done *incrementally*.

Traditional AGs has the following weaknesses. First, the strict functional and declarative AG models discourage the use of certain well-known efficient implementations, hence yielding less efficient solutions. This means that generated applications are restricted to the descriptive power offered by the generation language. For example, most of applications generated via AGs are text-based because AGs are awkward in specifying GUIs without the support of external ready-made libraries. The main reason is that GUIs inherently need to deal with a mass of input-handling and graphics-drawing facilities. AG specifications of these facilities may be too primitive and perhaps larger than implementation code. Thus, it is often argued whether AGs are an adequate method for the solution of specific problem domains, such as GUIs. Second, large AGs lack well-defined constructs that can improve comprehension and maintenance of AG specifications [Kastens91]. Third, writing AG specifications based on a language with complicated and ambiguous syntax requires considerable effort. For example, it must prevent from circular dependency of attributes as well as other bothersome restrictions.

3. An Overview of OOAG

Several studies, such as TOOLS[7], Ag[2], OOAG[13], Door[3], were devoted to extending the AG formalism with features supporting object-orientation. The main goal of these approaches is to employ object-orientation to promote the practical usefulness of AGs. That is, object-orientation turns AGs from a theoretical, static, and rather primitive formalism into a descriptive, dynamic, and unified specification language covering all aspects of semantics on a high level. OOAG presented in this paper differs from the above approaches in that it not only proposes another feasible object-oriented extension to traditional AGs, but also an improved mechanism to integrate AG++ specifications with application-dependent, object-oriented software components such as symbol tables and GUI libraries[4]. Importing these software components helps simplify the construction of AG++ specifications.

3.1 The Model-View-Shape Application

Framework

In our previous work[4], a model-view-shape (MVS) architecture, adapted from the *model-view-controller* (MVC) architecture[8], is used to construct a flow-based editor that can be used to depict the control-flow graphs of programs. This architecture, which can be viewed as a micro-architecture for language models, enforces a layered and loosely-coupled structure, so that the user-interface components may be more independent, maintainable, and reusable than those proposed in the original MVC architecture. On the basis of the MVS architecture, the associated framework was systematically constructed. In our approach, the grammar of a target language is modeled as a suite of well-classified software components embodying fine-grained language semantics and presentation information. Fig. 3.1 shows a class hierarchy for a C subset language. The **Model**, **View**, and **Shape** class hierarchies correspond to model, view, and shape classes, respectively. The following paragraphs only briefly discuss the functionality of the class hierarchy, and the associated implementation details can be found in [4].

A model and a view classes are constructed for each kind of language construct defined in the target language. These model and view classes are classified into hierarchies based on the functionality of language constructs. For example, if-then and if-then-else statements are selection statements, which are also kinds of structured statements. The model, a representation of the application domain, contains attributes and operations for maintaining the semantics of a program being edited. The view, which is used to handle the user interface, contains attributes and operations for managing the program's display and input-event interactions. A view object usually consists of a single or a set of shape objects for graphical presentations. These shape objects are application-independent, i.e., they are treated as reusable drawing elements. For example, Fig. 5.2 shows a sample program fragment consisting of some predefined graphical templates. These graphical templates, which are the building blocks for program construction, can be treated as graphical extensions to conventional text-based programming language constructs.

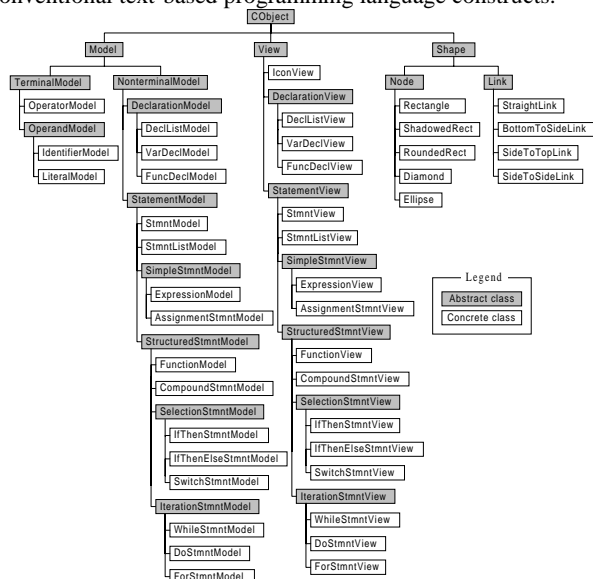


Fig. 3.1: An MVS application framework.

3.2 AG++

AG++ [4][18] is a specification language that extends traditional AGs with object-oriented features, such as *abstraction* and *modularity*, and other innovative features, such as *domain-specific patterns*. These features employ high-level constructs to extend basic AG concepts and decompose AG specifications into a set of well-structured modules, so that clear and comprehensible AG specifications can be written easily using

these constructs. For example, abstraction and modularity features are modeled by class and module constructs (in AG++), while domain-specific patterns are referred to as remote access and collective computing constructs. Moreover, the protocol construct in AG++ can be viewed as an *integration mechanism* for incorporating ready-made software components into AG++ specifications.

Class and module constructs. Although AGs are a good model for specifying semantic specifications and computations of attributed trees, most traditional AG specifications do not decompose well so reusing semantic specifications remains difficult [6]. On the other hand, partitioning semantic specifications hierarchically into a set of clusters, i.e., *modules*, helps manage massive production contexts and associated *semantic classes* effectively. To address the abstraction and modularity issues, AG++ provides a two-level abstraction model, which is referred to as class-level and module-level abstractions, for semantic specifications. The integrated decomposition method of performing a systematic classification of semantic specifications is referred to [4]:

Fig. 3.2 shows an example AG++ module construct for describing semantic specifications of variable declarations. A module usually contains five parts: an import list, an export list, a lexicon part, a syntax part containing a set of productions, and a semantics part containing specifications of associated semantic classes. The module construct allows AG++ specifications to import (and utilize) pre-built software components written in conventional programming languages. Moreover, symbols introduced in a module can be visible from outside only if they are exported.

```

%MODULE Declaration-Module
%{
#include "model.h"
// Definitions for model classes in the MVS framework
#include "symbol_table.h"
%}
%IMPORT
%EXPORT Declaration-List, Declaration, Type, Identifier
%LEXICON
.....
%SYNTAX // grammar rules are specified in ONF
declaration-list ::= { Declaration-List = $1; }
|
| declaration-list-branch
| declaration ';'
| { @Decl = ($1 -> $2); };
declaration ::= { Declaration = $1; }
|
| type name-list
| { @Type = ($1 -> $2); @Name-List = ($1 -> $3); };
name-list ::= { Name-List = $1; }
|
| name-list-branch
| identifier { Identifier = $2; }
| { @Identifier = ($1 -> $2); };
.....
%SEMANTICS
CLASS Tree-Node // Tree-Node is the root semantic class
ATTRIBUTE
Tree-Node* parent;
Model* model // Model is the root model class in the MVS framework
BOOL marked;
String error_message;
PROTOCOL
Tree-Node(Tree-Node* parent_node)
{ parent = parent_node; Marked = FALSE; error_message = ""; }
void Mark() { marked = TRUE; model->Highlight(); }
void ShowUI() { model->Display(); }
void ShowErrorMessage(String msg) { model->ShowMessage(msg); }
// xxx: Incorporating MVS framework (GUIs) into the AG++
// specification
END
//-----
CLASS Name-List INHERIT Tree-Node
COMPONENT
Identifier* id = SINGULAR @Identifier;
Name-List* name_list = AT_MOST_ONE @Name-List;
ATTRIBUTE
StringList declared_names;
STATIC
SEQUENCE {
declared_names.ADDTAIL(id->GetName());
if (name_list != NULL) declared_names += name_list->declared_names;
};
.....
END
%SEMANTICS

```

```

CLASS Declaration-List INHERIT Tree-Node
COMPONENT
Declaration* var_decl = SINGULAR @Decl;
Declaration-List* decl_list = AT_MOST_ONE @Decl-List;
ATTRIBUTE
StringList declared_variables;
STATIC
SEQUENCE {
declared_variables = var_decl->declared_variables;
if (decl_list != NULL) declared_variables += decl_list->declared_variables;
};
PROTOCOL
Declaration-List(Tree-Node* parent) : Tree-Node(parent)
{ model = new DeclListModel(this); ... };
String IsDeclared(String name) {
switch (declared_variables.NUMBEROFFIND(name)) {
case 0: return "FALSE";
case 1: return "TRUE";
default: return "Semantics error: " + name + " is a redeclared variable!";
}
}
.....
END
//-----
CLASS Declaration INHERIT Declaration-List
COMPONENT
Type* var_type = SINGULAR @Type;
Name-List* name_list = SINGULAR @Name-List;
REMOTE
Declaration* prev_var_decl =
AT_MOST_ONE INCLUDING(Declaration-List) @Prev;
Declaration* next_var_decl = AT_MOST_ONE @Next @Decl;
NodeList_Ref identifier_list =
AT_LEAST_ONE PRE_ORDER CONSTITUTES(Identifier);
STATIC
declared_variables = name_list->declared_names;
PROTOCOL
Declaration(Tree-Node* parent) : Tree-Node(parent)
{ model = new VarDeclModel(this); ... };
BOOL IsDeclared(String name) {
if (declared_variables.FIND(name) == TRUE) return TRUE;
else return FALSE; };
void Parse() { ..... };
.....
END
//-----
CLASS Identifier INHERIT Name-List
PRIVATE
String name, result;
REMOTE
Declaration-List* decl_list = SINGULAR INCLUDING(Function) @Decl-List;
ATTRIBUTE
String type;
STATIC
SEQUENCE {
result = decl_list->IsDeclared(name);
if (result == "FALSE") {
error_message = "Semantics error: " + name + " is an undeclared identifier!";
ShowErrorMessage(error_message);
} else if (result == "TRUE") {
type = decl_list->LookupVarType(name);
} else { error_message = result; ShowErrorMessage(error_message); }
}
.....
END
%END Declaration-Module

```

Fig. 3.2: An AG++ specification of a variable-declaration module (partial).

In the syntax part, “{ **ClassName** = \$X; }”, which is referred to as *class assignments* [Wu95], specifies a mapping process between a (abstract) non-terminal symbol and its (concrete) semantic class. Note that X indicates its appearance position with respect to all non-terminal symbols in a production. For example, “**Declaration-List** = \$1” means that the non-terminal symbol “**declaration-list**” is assigned to (i.e., represented by) a semantic class called **Declaration-List**. Moreover, each production may be attached with *tag functions*, which make virtual connections of parent-child and sibling nodes (enclosed in a production context) by annotating some direction *tags* on the tree arcs. Fig. 3.3 shows some of tags specified in Fig 3.2.

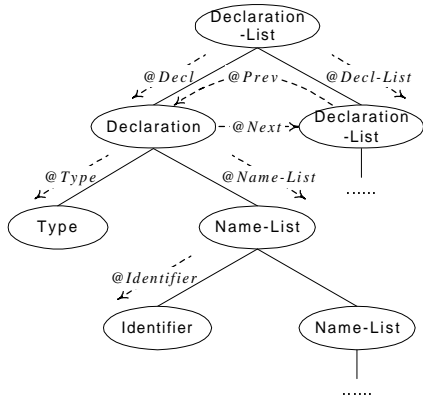


Fig. 3.3: A program tree annotated with direction tags on the arcs.

Remote access and collective computing constructs. The basic AG concepts associate computations of attribute dependencies to a rather small context given by a single production. This means that attribute dependencies are only allowed within adjacent tree nodes, and the value of an attribute cannot be accessed outside the production context. However, sometimes the computations may depend on attributes at the remote tree nodes, i.e., those that are not accessible directly in a production context. A typical example for long range dependencies is the declaration and use of an identifier. With the traditional AG approach, remote attribute dependencies may be specified using a lot of trivial attribution rules, called *copy rules* or *transfer rules* [6], for information propagation. Such attribute dependency specifications can be simplified by introducing remote access constructs, such as INCLUDING, CONSTITUTES, and CHAINING [6]. These existing constructs, however, have limited expressive power because they are only able to describe remote access through straight paths, i.e., paths that follow tree nodes upward or downward. This motivated us to create a more generalized remote access construct, called *path expressions*, for AG++ to access non-straight remote attributes.

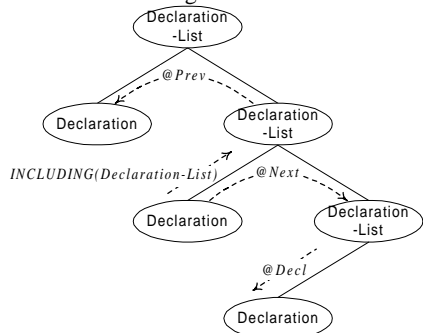


Fig. 3.4: Remote access using a path expression.

A path expression enforces modular use and explicitly specified paths for remote node access. It defines a virtual path pattern that matches one (or more than one) remote tree node with a *regular expression* of tags. For example, as shown in Fig. 3.2, class Declaration declares two attributes, prev_var_decl and next_var_decl, to reference two remote nodes, i.e., previous and next variable declarations, via path expressions, “AT_MOST_ONE INCLUDING(Declaration-List) @Prev” and “AT_MOST_ONE @Next @Decl”. Fig. 3.4 shows graphical notations for accessing remote nodes via the above path expressions. Moreover, the remote node(s) collected via a path expression can be constrained using *cardinality clauses*, such as SINGULAR, AT_MOST_ONE, and AT_LEAST_ONE.

Attribution rules and protocols. As shown in Fig. 3.2, attribution rules that are originally attached to the end of each

production in traditional AGs are now specified in a higher-level style in the STATIC sections of respective AG++ semantic classes. These attribution rules may be evaluated in parallel by the AG++ attribute evaluator. On the other hand, sometimes attribute dependencies are not represented via value dependencies but side effects, which are invisible by the AG++ code generator. In this case an *explicit dependency construct*, called SEQUENCE, is introduced to explicitly specify the evaluation sequence of attribution rules and inter-rule attribute dependencies by guiding the AG++ code generator to add dependency relations on attribute occurrences.

Protocols act as a gateway for communicating with remote tree nodes and external software components. A protocol, which may be implemented as a C++ member function, provides an entry point for external software components to request AG++-specific services, such as incremental attribute evaluations, by sending a message to the protocol provider directly. Also, protocols can be viewed as a mechanism to access external services, such as management of GUIs and symbol tables, written in imperative languages such as C++. Protocol ShowUI() defined in class Tree-Node shows such an example. By those means efficient implementations can be achieved without loss of the declarative characteristics of traditional AG specifications.

4. An OOAG-Based Toolset for Software Maintenance

4.1 The System Architecture

Fig. 4.1 shows the system architecture of an OOAG-based toolset for software maintenance. Tools in the toolset can be generally divided into two classes: user-defined and built-in tools. In the user-defined tools, a flow-based editor and a language-based text editor usually provide (graphical) user interfaces to display a wide variety of program information that users can interact with. For example, a flow-based editor may be used to display control-flow, data-flow, or certain kinds of program-dependency relationships in graphical layouts. These editors are inherently interactive; that is, they have the ability to receive user-input events, interpret and handle these events, and respond to users with some feedback.

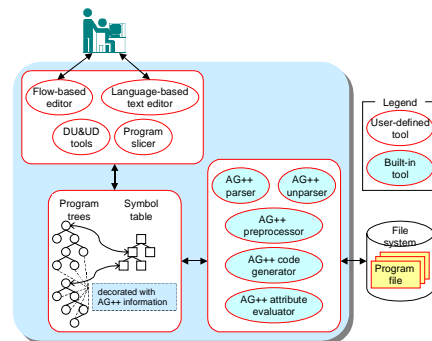


Fig. 4.1: The system architecture.

In contrast to the above editors, DU/UD tools and a program slicer, used to analyze program-flow information during programming and maintenance phrases, can be viewed as back-end tools. These flow analyzers do not interact with the user directly, but can be activated by the AG++ attribute evaluator when it is evaluating the associated attribution rules. When a change to the internal program representations, such as program trees and the symbol table, is effected, the AG++ attribute evaluator starts to evaluate affected attribution rules to ensure that the AG++ attributes are consistent with the tree. Moreover, in the built-in tools, an AG++ parser is responsible for parsing and transforming program text into the corresponding program tree decorated with AG++ information, while an AG++ unparser performs the reverse operation. The functionality of the AG++

preprocessor and AG++ code generator will be described in the next section.

4.2 A Systematic Tool-Construction Process

Fig. 4.2 shows a high-level process for constructing tools in a maintenance toolset with OOAG. This process is generally divided into four steps in order to construct the kernel and (graphical) user-interface part of a maintenance toolset with AG++ and the MVS framework, respectively. In step 1, for a target language that tool in the toolset are based on, the associated grammar rules are reorganized and represented via a set of semantic classes by following the classification scheme, i.e., decomposition method, mentioned in [4]. The specifications of these semantic classes specified in this step are mainly used to describe the internal structure of the program tree. In step 2, the semantic specifications are translated into intermediate language tables, which are briefly called the AG++ model knowledge, by an AG++ preprocessor. The AG++ model knowledge is intended to contain information commonly used by all tools in the toolset.

In step 3, the core functionality of a tool is specified declaratively via a set of attributes, attribution rules, and protocols in AG++ semantic classes to model interactions among tree nodes. That is, AG++ is applied to specify how each node in the program tree computes the associated attributes and communicate with each other in order to perform the desired functionality of the new tool. The AG++ code generator first analyzes attribute dependencies specified in attribution rules in the tool specification, and schedules the attribute evaluation sequence. It then resolves path expressions by generating additional attributes and methods that are used to identify remote tree node(s) matched in the path expression. Next, the AG++ code generator integrates the translated specification with the AG++ model knowledge to generate source code in the target language, i.e., C++. In step 4, the generated code is then compiled using a C++ language compiler, linked with external software components (e.g., the MVS framework) and a built-in AG++ incremental evaluation engine to generate an executable maintenance toolset. The AG++ incremental evaluation engine is a system-supported library for controlling the order of attribute evaluation in static or dynamic ways.

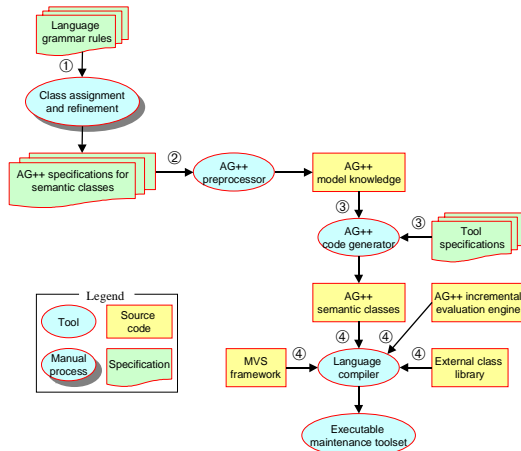


Fig. 4.2: Procedures for constructing a maintenance toolset using OOAG.

So far we have used Visual C++ and the Microsoft foundation class (MFC) library as an underlying platform for developing a toolset prototype for software maintenance and the MVS framework on the Windows environment. The AG++ parser for the C language subset was constructed using Lex and YACC, while the AG++ unparser, AG++ preprocessor, AG++ code generator, and AG++ attribute evaluator are custom-made C++ programs.

5. Examples of Generating Tools

5.1 The Flow-Based Editor

Figs. 3.2 and 5.1 show a partial AG++ specification of a flow-based editor for a C language subset. This specification is briefly divided into three modules including variable-declaration, expression, and statement modules. As the user constructs a program by inserting language constructs, appropriate new nodes are created and added to the internal program tree. With the OOAG approach, each tree node is modeled by an AG++ semantic object, which may contain two kinds of AG++ attributes: inherited and synthesized attributes. The attributes held by a semantic class come from two sources: the attributes originally defined in the class and the attributes inherited from base class(es) of the class. Each semantic class has a *construction protocol*, similar to the *constructor* in a typical C++ class, for setting up the initial state of the instantiated semantic object. For example, the construction protocol of the If-Then-Else-Stmnt semantic class, i.e., `IfThenElseStmnt(Tree-Node* parent)`, is responsible for (1) creating a sub-tree for representing the if-then-else statement and attaching the sub-tree to the base program tree, and (2) creating a reference to the associated model object, i.e., `IfThenElseStmntModel` object, in the MVS framework. The model object in this case serves as an interface between an AG++ semantic object and a corresponding user-interface component, i.e., view object, which determines how the language construct is displayed on the screen.

```
%SEMANTICS
CLASS Statement INHERIT Statement-List
PRIVATE
    Tree-Node* child;
STATIC
    declared_variables = parent->declared_variables;
PROTOCOL
    Statement(Tree-Node* parent) : Tree-Node(parent)
        { model = new StmtModel(this);
          model->AppendTemplateTransformationMenu("Assignment Stmtnt".
          &InsertAssignmentStmntTemplate);
          model->AppendTemplateTransformationMenu("If-Then Stmtnt".
          &InsertIfThenStmntTemplate); ... };
    void InsertAssignmentStmntTemplate() { child = new Assignment-Stmnt(this); ... };
    void InsertIfThenStmntTemplate() { child = new If-Then-Stmnt(this); ... };
    .....
END
//-----
CLASS Expression INHERIT Tree-Node
COMPONENT
    Scalar-Expression* scalar_expr = SINGULAR @Scalar-Expr;
    Rational-Op* rational_op = AT_MOST_ONE @Rational-Op;
    Expression* expr = AT_MOST_ONE @Expr;
ATTRIBUTE
    StringList declared_variables, used_variables;
    Type type;
STATIC
    if ( ((rational_op == NULL) && (expr != NULL)) ||
        ((rational_op != NULL) && (expr == NULL)) )
        { error_message = "Syntax error"; ShowErrorMessage(error_message); };
    declared_variables = parent->declared_variables;
SEQUENCE {
    used_variables = scalar_expr->used_variables;
    if (expr != NULL) {
        used_variables += expr->used_variables;
        if (scalar_expr->type.IsTypeCompatible(expr->GetType()))
            type.SetType("bool");
        else error_message = "Semantics error: type incompatibility!";
    } else type.SetType(scalar_expr->GetType());
}
PROTOCOL
    Expression(Tree-Node* parent) : Tree-Node(parent)
        { model = new ExpressionModel(this); ... };
    int Parse() { ... };
END
//-----
CLASS If-Then-Else-Stmnt INHERIT Statement
COMPONENT
    Expression* expr = SINGULAR @Expr;
    Statement* then_stmnt = SINGULAR @Then-Stmnt;
    Statement* else_stmnt = SINGULAR @Else-Stmnt;
STATIC
    declared_variables = parent->declared_variables;
PROTOCOL
    If-Then-Else-Stmnt(Tree-Node* parent) : Tree-Node(parent) {
        expr = new Expression (this);
        then_stmnt = new Statement (this);
        else_stmnt = new Statement (this); ...
    }
```

```

}
END
//-----
CLASS Function, Stmtnt-List, If-Then-Stmtnt, While-Stmtnt, Do-Stmtnt, Compound-Stmtnt,
Assignment-Stmtnt, Function-Call-Stmtnt, Scalar-Expression, Term, Factor, .....

```

Fig. 5.1: An AG++ specification of statement and expression modules (partial).

Our current flow-based editor provides a syntax-directed editing facility [15][11] that enable the user to construct programs by depicting the associated control-flow graphs. For a placeholder of structured statements, the editors guide the user to replace it with an instance of some (valid) structured statement. The replacement operation is performed when the user selects a template from a *template-transformation menu*, which is maintained by a view class in the MVS framework. For example, Figs. 5.2 and 5.4 show a control-flow graph and a corresponding program tree representing the `ComputerMax()` function before and after the user inserts an if-then statement template into a statement placeholder. Constructing a template-transformation menu with respect to different language grammars is simple using OOAG; the construction protocol of the *Statement semantic class* shown in Fig. 5.1 gives a feasible solution. For example, a message in the form of `model->AppendTemplateTransformationMenu("If-Then Stmtnt", &InsertIfThenStmtntTemplate);` is used to inform the model, i.e., `StmtntModel`, object to add a menu item, "If-Then Stmtnt", and to invoke a *call-back protocol*, `InsertIfThenStmtntTemplate()`, when the user selects such an item. Note that the code underlined in Fig. 5.1, e.g., `model->Highlight()`, specifies the actual interactions between AG++ semantic objects and model objects in the MVS framework.

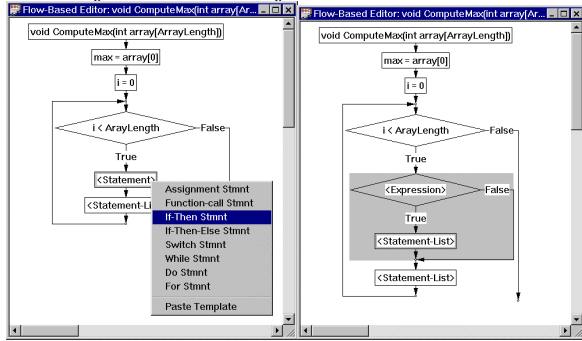


Fig. 5.2: Menu-driven template selection.

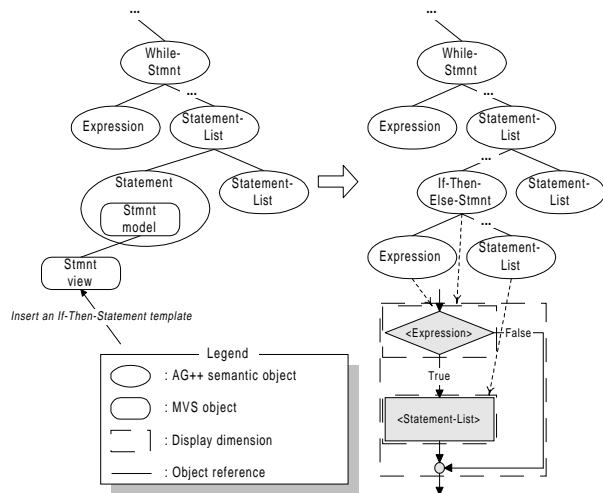


Fig. 5.3: Inserting an if-then statement into a statement placeholder (Fig. 5.2).

Incremental semantic analysis [5] would be performed by calculating and updating attribute values throughout the program

tree in response to program modifications. So far the incremental semantic analysis, whose functionality is partially specified in Figs. 3.2 and 5.1, can check at least three kinds of errors: undeclared variables, redeclared variables, and type incompatibility.

5.2 The Flow Analyzers

5.2.1 Design Rationales

In the past decade, a number of flow-analysis techniques based on the program-tree manipulation have been continuously explored. Attribute grammars and *action routines* [11] are two well-known examples. The common features of these two techniques are that the program semantics are represented as semantic attributes attached to tree nodes, and the flow analysis is performed by traversing the program tree and evaluating the associated attributes' values. Constructing a flow analyzer with OOAG is flexible and extensible; designers can specify the associated tool specification using a pure AG approach, i.e., via attributes and attribution rules, or using a hybrid approach by incorporating protocols, which can be viewed as action routines. Reusing existing attributes, attribution rules, and protocols may reduce the construction cost for new flow analyzers.

Our current flow-analysis model, which combines both features of AGs and action routines, acts as the *node-marking process* [14] operating on the program tree. The whole analysis is performed via message-passing between AG++ semantic objects in the program tree. When a semantic object receives a message or gets a return value of the message it sends, it has the best local knowledge to do whatever next action it deems appropriate. That is, a semantic object may evaluate the attributes' values, send another message to other semantic object(s), or just return a specific value. During the flow analysis, those language constructs evaluated to be the analysis results are indicated by marking the corresponding semantic objects. Moreover, the user interfaces of new flow analyzers are of no need to be constructed from scratch because existing user-interface components, supported in the MVS framework, can be (re)used to manage and display the analysis results. This entails that our flow-based tools provide uniform and consistent user interfaces to interact with users.

5.2.2 DU and UD Tools

One of the major tasks performed by (incremental) data-flow analyzers is the computation of data-flow dependencies, such as *definition-use (DU)* and *use-definition (UD) chains*, with respect to specific variables. This subsection gives an example of constructing DU and UD tools with the ability to compute and display data-flow information. Fig. 5.5 shows the semantics part of an AG++ specification of intraprocedural DU and UD tools. (The interprocedural version is available in [4].) Here the specification only points out the key attributes and protocols implemented in a number of semantic classes, while information specified in `STATIC` and `REMOTE` sections is omitted for clarity.

```

%SEMANTICS
CLASS Expression INHERIT Tree-Node
PROTOCOL
void ComputeUDChain
(String var_name, NodeList_Ref marked_nodes) {
parent->GetDefinedVariablesBackwardUp(var_name, this, marked_nodes);
}
// Initiate a UD analysis w.r.t. variable 'var_name' by sending
// GetDefinedVariablesBackwardUp() to its parent node. After ComputeUDChain()
// completes execution, marked_nodes collects a list of nodes constituting a UD chain.
int GetUsedVariablesForwardDown
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes)
{
// If this expression is a use of variable 'var_name'
if (used_variables.FIND(var_name) == TRUE)
// If this expression is still un-marked
if (marked_nodes.FIND(this) == FALSE) {
Mark(); // Mark and highlight this expression
// Add this expression to the list of marked model objects;
marked_nodes.ADDTAIL(this);
}
return 0; // Report that var_name's value is not re-defined yet;
}
}

```

```

END
//-----
CLASS Assignment-Stmnt INHERIT Statement
PROTOCOL
void ComputedUChain(String var_name, NodeList_Ref marked_nodes)
{ parent->GetUsedVariablesForwardUp(var_name, this, marked_nodes); }

int GetUsedVariablesForwardDown
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes) {
if (used_variables.FIND(var_name) == TRUE)
if (marked_nodes.FIND(this) == FALSE)
Mark();
Marked_nodes.ADDTAIL(this);
}
// If this assignment statement is a definition of variable 'var_name'
// report that var_name's value has been re-defined
// otherwise, report that var_name's value is not re-defined yet
if (defined_variable == var_name) return 1; else return 0;
}
END
//-----
CLASS If-Then-Else-Stmnt INHERIT Statement
PROTOCOL
void GetUsedVariablesForwardUp
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes)
{ Parent->GetUsedVariablesForwardUp(var_name, this, marked_nodes); }

int GetUsedVariablesForwardDown
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes) {
int i, j;
Expr->GetUsedVariablesForwardDown(var_name, this, marked_nodes);
i = thenStmnt->GetUsedVariablesForwardDown(var_name, this, marked_nodes);
j = elseStmnt->GetUsedVariablesForwardDown(var_name, this, marked_nodes);
// If var_name's value has been re-defined through both then- and else-stmnt paths
// report that var_name's value has been re-defined;
// Otherwise, at least one execution path prevents from re-defining var_name's value
if ((i == 1) && (j == 1)) return 1; else return 0;
}

void GetDefinedVariablesBackwardUp
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes)
{ parent->GetDefinedVariablesBackwardUp(var_name, this, marked_nodes); }

int GetDefinedVariablesBackwardDown
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes) {
int i, j;
i = thenStmnt->GetDefinedVariablesBackwardDown(var_name, this, marked_nodes);
j = elseStmnt->GetDefinedVariablesBackwardDown(var_name, this, marked_nodes);
if ((i == 1) && (j == 1)) return 1; else return 0;
}
END
//-----
CLASS While-Stmnt INHERIT Statement
PROTOCOL
void GetUsedVariablesForwardUp
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes)
{
expr->GetUsedVariablesForwardDown(var_name, this, marked_nodes);
stmnt->GetUsedVariablesForwardUp(var_name, this, marked_nodes);
parent->GetUsedVariablesForwardUp(var_name, this, marked_nodes);
}

int GetUsedVariablesForwardDown
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes)
{
expr->GetUsedVariablesForwardDown(var_name, this, marked_nodes);
stmnt->GetUsedVariablesForwardDown(var_name, this, marked_nodes);
return 0;
}

void GetDefinedVariablesBackwardUp
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes)
{
stmnt->GetDefinedVariablesBackwardDown(var_name, this, marked_nodes);
parent->GetDefinedVariablesBackwardUp(var_name, this, marked_nodes);
}

int GetDefinedVariablesBackwardDown
(String var_name, Tree-Node* from, NodeList_Ref marked_nodes)
{
stmnt->GetDefinedVariablesBackwardDown(var_name, this, marked_nodes);
return 0;
}
END
//-----
CLASS Function, Stmnt-List, Compound-Stmnt, If-Then-Stmnt, Do-Stmnt, Statement, ...

```

Fig. 5.5: An AG++ specification of DU and UD tools (partial).

As the above specification shows, the functionality of DU and UD tools is systematically handled by the following protocols: `GetUsedVariablesForwardUp()`, `GetUsedVariablesForwardDown()`, `GetDefinedVariablesBackwardUp()`, and `GetDefinedVariablesBackwardDown()`. The first two are responsible for computing DU chains with respect to a variable defined, and the rest for computing UD chains with respect to a variable used. The term “forward” (or “backward”) shown in protocols’ names denotes that the computation sequence would generally follow (or reverse) the control flow of a program. In addition to the above protocols, `ComputedUChain()` and `ComputeUDChain()` serve as the “triggers” initiating the DU and UD analyses, respectively.

Fig. 5.6 shows an example of computing a DU chain with respect to variable `a` after the user issued a “show DU chain” command on the assignment statement “`a=c`”. This command invokes protocol `ComputedUChain()` (defined in the `Assignment-Stmnt` semantic class) to initiate the DU analysis. For those expressions and assignment statements that are evaluated to be part of the DU chain, they will receive a message called `GetUsedVariablesForwardDown()` and then inform the associated user-interface components to highlight the visual layouts by invoking protocol `Mark()`, which is defined in class `Tree-Node`. Likewise, Fig. 5.7 shows an example of computing a UD chain with respect to variable `a` after the user issued a “show UD chain” command on the assignment statement “`f=a`”.

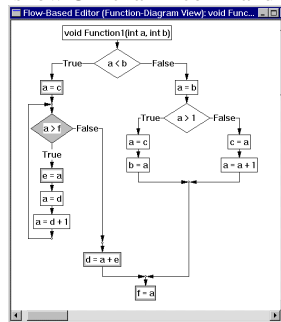


Fig. 5.6: A DU chain w.r.t. variable `a` in “`a=c`”.

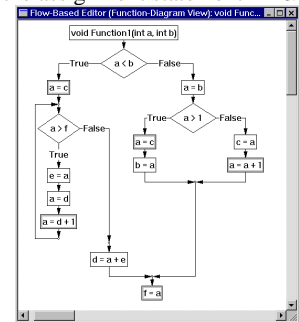


Fig. 5.7: A UD chain w.r.t. variable `a` in “`f=a`”

5.2.3 The Program Slicer

Program slicing [16], an automatic technique determining the statements that may potentially affect (or be affected by) a specific variable at a given statement, aids program understanding by reducing the amount of code a programmer must examine, and presenting only a relevant program subset of interest. Computation of program slices involves examining both data-flow and control-flow dependencies of a program. The data-flow dependencies have been described in Section 5.2.2. On the other hand, statement `S` is *control dependent* on statement `r` if `r` is a predicate (e.g., an expression) that can control whether `S` executes or not. One typical approach to computing program slices is to summarize and symbolize each of control-flow and data-flow dependencies as an edge of a directed graph, called the *program dependence graph* [1], in which the vertices are the statements of a program. In this approach, a forward or backward slice is computed by identifying the set of statements in the slice through the forward or backward transitive closure in this graph. However, construction and maintenance of a program dependence graph during programming is not easy due to two major factors. First, the tool designer has to additionally program some, perhaps complicated, data structures (e.g., reference links) to record the dependence relationships among tree nodes. Second, program modification, such as inserting or deleting a statement, would take the program slicer (much) time to re-calculate the program dependencies in an incremental or a batch way.

From the above descriptions about program slicing, the most intuitive yet effective way to construct a program slicer is to

reuse data-flow analysis facilities, i.e., the AG++ specifications for computing DU and UD chains, and incorporate control-flow analysis facilities into the slicer. The construction cost, compared with the effort based on the building-from-scratch approach, is reasonably low because the tool designer only needs to concern how to reuse the existing AG++ specification and augment some new functionality to the specification. A concrete AG++ specification for the program slicer can be found in [4].

6. Conclusion and Future Work

This paper presents an approach, called OOAG, to constructing a toolset for software maintenance. By combining generative and compositional techniques, OOAG integrates the flexible modeling capability of object-orientation with the support for specification generation. By extending AGs to incorporate modularity, remote attribute access, collective computing, and object-oriented views on program-tree nodes, OOAG makes semantic specifications more high-level, and therefore easier to construct, understand, and maintain than that using traditional AGs. Moreover, OOAG promotes specification reuse by allowing users to extend existing AG++ specifications to construct new tools with similar functionality. In this approach common attributes and attribution rules are “factored out”, so that duplicated construction effort can be eliminated effectively. The resulting attributions are space-efficient and allow efficient incremental attribute evaluation in these tools. By employing reusable components in the MVS framework to handle the graphical user interface of the maintenance toolset, the specification difficulties due to theoretical limitations of AGs may be removed.

At present, the MVS framework has been designed with respect to programming language aspects. Designers can reuse fine-grained as well as coarse-grained language constructs, and extend them by creating the derived classes in the MVS framework to redefine and augment the user-interface functionality for a new language. To support another class of visual languages, such as object-oriented languages, new model and view classes can be identified, constructed, and added to the MVS framework by locating suitable base classes. Once the addition is completed, the user interfaces of applications defined in the new visual language can be prototyped by instantiating user-interface components in the MVS framework.

Most studies employ flow analysis to facilitate program understanding during the maintenance phase. In their approaches, well-structured (i.e., syntactically and semantically correct) programs are parsed and translated into the corresponding flow graphs, and flow analyzers then traverse these flow graphs to report analysis results to the user. Compared with them, the flow analysis presented in this paper is based on the underlying program tree. Our tree-based flow analysis techniques have the following advantages. First, a complex flow-analysis task can be decomposed into manageable subtasks that are handled by passing messages between associated tree nodes. These subtasks may be reused to construct new flow analyzers. Second, the flow analyzer can directly work on the program tree, without the need to create and maintain redundant data structures, such as program dependence graphs. Third, the user can request flow analysis during programming, and the flow analyzer can deal with incomplete program fragments incrementally as well as executable programs. It is very helpful for program

understanding during programming and maintenance phases.

References

- [1] Ferrante, J., Ottenstein, K., and Warren, J., “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 5, July 1987, pp. 319-349.
- [2] Grosch, J., “Object-oriented attribute grammars,” *Proceedings of the 5th International Symposium on Computer and Information Sciences, 1990*, pp. 807-816.
- [3] Hedin, G., “An object-oriented notation for attribute grammars,” *Proceedings of ECOOP’89, 1989*, pp. 329-345.
- [4] Hu, C. H., *The Study of an Integrated Visual Programming Environment, Ph.D. thesis, National Chiao Tung University, May 1998*.
- [5] Hu, C. H., Wang, F. J., and Wang, J. C., “Constructing a language-based editor with object-oriented techniques,” *Journal of Information Science and Engineering*, Vol. 11, No. 4, Nov. 1995, pp. 1-25.
- [6] Kastens, U., “Attribute grammars as a specification method,” in *Lecture Notes in Computer Science - Attribute Grammars, Applications and Systems*, Alblas, H. and Melichar, B. (eds.), Springer-Verlag, 1991, pp. 16-47.
- [7] Koskimies, K., “Object-orientation in attribute grammars,” in *Lecture Notes in Computer Science - Attribute Grammars, Applications and Systems*, Alblas, H. and Melichar, B. (eds.), Springer-Verlag, 1991, pp. 297-329.
- [8] Krasner, G. E. and Rope, S. T., “A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80,” *Journal of Object-Oriented Programming*, Vol. 1, No. 3, Aug. 1988, pp. 26-49.
- [9] Linos, P. K. and Courtois, V., “A toolset for maintaining hybrid C++ programs,” *Journal of Software Maintenance: Research and Practice*, Vol. 8, Dec. 1996, pp. 389-419.
- [10] Linos, P. K. and Tulula, P., “Visualizing program dependencies: an experimental study,” *Software - Practice and Experience*, Vol. 24, No. 4, Apr. 1994, pp. 387-403.
- [11] Medina-Mora, R. and Feiler, P. H., “An incremental programming environment,” *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, Sep. 1981, pp. 472-481.
- [12] Prieto-Díaz, R., “Status report: software reusability,” *IEEE Software*, Vol. 10, No. 3, May 1993, pp. 61-66.
- [13] Shinoda, Y. and Katayama, T., “Object-oriented extension of attribute grammars and its implementation,” in *Lecture Notes in Computer Science No. 461, Deransart, P. and Jourdan, M. (eds.), Springer-Verlag, 1982*, pp. 177-191.
- [14] Sloane, A. M. and Holdsworth, J., “Beyond traditional program slicing,” *Proceedings of the 1996 International Symposium on Software Testing and Analysis, 1996*, pp.180-186.
- [15] Teitelbaum, T. and Reps, T., “The Cornell program synthesizer: a syntax-directed programming environment,” *Communications of the ACM*, Vol. 24, No. 9, Sep. 1981, pp. 563-573.
- [16] Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, July 1984, pp. 352-357.
- [17] Whitney, M., et al., “Using an integrated toolset for program understanding,” *Proceedings of the CAS Conference (CASCON 95), 1995*, pp. 262-274.
- [18] Wu, P. C. and Wang, F. J., “The evolution of an object-oriented specification for compilers,” *Journal of Information Science and Engineering*, Vol. 11, No. 4, Nov. 1995, pp. 433-452.