TransformGen: Automating the Maintenance of Structure-Oriented Environments

DAVID GARLAN
Carnegie Mellon University
and
CHARLES W. KRUEGER
On-Demand Technologies, Inc.
and
BARBARA STAUDT LERNER
University of Massachusetts

A serious problem for programs that use persistent data is that information created and maintained by the program becomes invalid if the persistent types used in the program are modified in a new release. Unfortunately, there has been little systematic treatment of the problem; current approaches are manual, ad hoc, and time consuming both for programmers and users. In this article we present a new approach. Focusing on the special case of managing abstract syntax trees in structure-oriented environments, we show how automatic transformers can be generated in terms of an implementor's changes to the grammar of these environments.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance; D.2.m [**Software Engineering**: Miscellaneous—*rapid prototyping*; H.2.1 [**Database Management**]: Logical Design—*schema and subschema*; H.2.m [**Database Management**]: Miscellaneous

General Terms: Design, Management

Additional Key Words and Phrases: Schema evolution, structure-oriented environments, type evolution

Support for research on Gandalf was provided in part by ZFE F2 KOM of Siemens Corporation, Munich, Germany, and in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, under contract F33615-90-C-1465, ARPA order no. 7597. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, or the U.S. government or Siemens Corporation.
This article is a revision and extension of an earlier version, "A Structural Approach to the Maintenance of Structure-Oriented Environments," which appeared in the *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December, 1986.
Authors' addresses: D. Garlan, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; C. W. Krueger, On-Demand Technologies, Inc.; B. Staudt Lerner, Computer Science Department, University of Massachusetts, Amherst, MA 01003.

1. INTRODUCTION

A serious problem for programs using persistent data is that information created and maintained by the program becomes invalid when the persistent types used in the program are modified in a new release. This problem affects many types of software including programming environments, database systems, and operating systems. For example, in the widespread conversion from Version 4.1 of BSD Unix™ to Version 4.2, file directory structures created under 4.1 were incompatible with 4.2 and therefore required conversion. To install a new release of an environment or system requires, at the very least, design and construction of a conversion process for existing persistent data, and work must partially halt while conversion takes place. Users are burdened with a period of instability and loss of functionality in the case of inadequate conversions. Consequently, users and implementors[1] are faced with a dilemma: stability can be achieved by ignoring successive releases, in which case the environment or system will not meet the evolving needs of its users; change can be allowed, at the cost of a time-consuming process of conversion; or changes can be restricted to allow backward compatibility at the expense of providing optimal type organization in the new release.

Structure-oriented environments,[2] such as those supported by Gandalf [Habermann et al. 1986], Centaur [Borras et al. 1988], Pecan [Reiss 1984], and the Synthesizer Generator [Reps and Teitlebaum 1984], are a class of programming environments for which these problems are particularly severe. A structure-oriented environment is typically generated from a formal description or *grammar*. One of the purposes of the grammar is to define the structure of abstract syntax trees created with such an environment. When the user exits the environment the abstract syntax tree representation of the program, along with attributes and other information maintained by the environment, is made persistent to avoid reparsing and reanalyzing the program when the user accesses it at a later time. When the grammar is changed in any but trivial ways, existing database whose trees are structured according to the old grammar may not be compatible with the new grammar, thus preventing the new environments from accessing the old trees. At early stages of environment prototyping it may be possible to simply discard the old databases. However, in practical settings where users have come to depend on the information and programs created with the old environment, a sudden announcement that existing trees are no longer valid will not be acceptable.

For most database systems, operating systems, and programming environments the data invalidation and conversion problem is mollified by the fact that major evolutionary changes are rare and usually affect only a small amount of the existing information. For structure-oriented environments,

[1] Throughout we refer to the designer, builder, and maintainer of an environment as an *implementor*.

[2] We take the term "structure-oriented environment" to be synonymous with "syntax-directed environment," "language-based environment," "structure editor-based environment," etc.

however, changes tend to be much more frequent and, in general, will significantly affect all existing environment databases. Changes occur more frequently because of the ease with which environment implementors can generate new environments from grammars. Moreover, improvements to the environment—either through the introduction of new tools or the enhancement of existing tools—usually require changes to the grammar for the environment database. (For some examples see Garlan and Miller [1984].)

There has been little research on systematic solutions to this problem. This is as true for general programming environments as it is for structure-oriented environments. Current approaches are either blatantly ad hoc, are based on the idea of reparsing textual representations of existing databases, or severely limit the types of changes that can be made. While the parsing approach may be adequate for some purposes, it has a number of problems, itemized later, that make it unsuitable as a general solution.

In this article we present an alternative approach. Focusing on a solution to the problems for structure-oriented environments, we show how automatic converters can be generated in terms of an implementor's *changes* to the grammars of these environments. In the following sections we describe the design and implementation of an environment, called *TransformGen*, in which an implementor can make structured changes to the grammar of a structure-oriented environment. The output of TransformGen is a new grammar together with a *transformer*, which takes instances of databases built under the old grammar and automatically converts them to instances of databases that are legal under the new grammar. We begin with an overview of the transformational approach. Then we present an example of its use together with a detailed analysis of TransformGen. We compare this work to other related research and evaluate the impact of the key design decisions we made. In conclusion we indicate how these techniques might be applied to solve similar problems for programming environments and systems in general.

2. THE TRANSFORMATIONAL APPROACH

We begin by considering the motivating issues behind this work, the general approach employed by TransformGen, and the difficult design problems implied by such an approach.

2.1 The Need for Automated Transformation

A structure-oriented environment typically represents and stores its data as an attributed abstract syntax tree (AST), often called the environment database.[3] The set of legal trees is determined by a *grammar* for that environment. Like a context-free grammar for a programming language, the grammar of a structure-oriented environment primarily consists of a collec-

[3]The use of the term "database" is reasonable since the AST serves as a central persistent repository for all data manipulated by the tools in the environment. Many environments also support queries over this data, transactions, replication, etc.

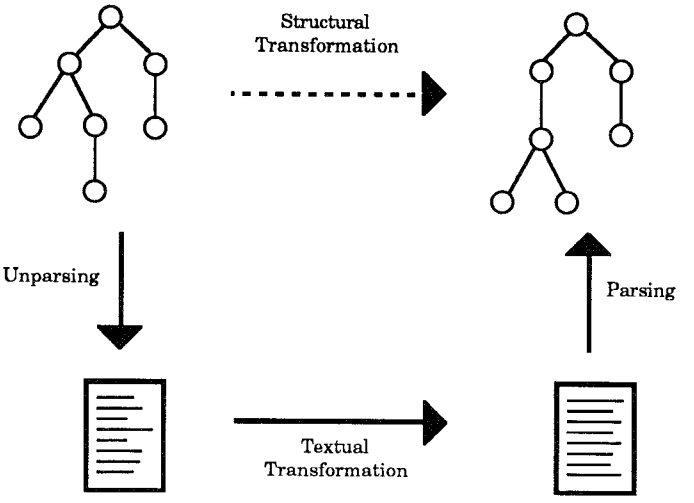

Fig. 1. The transformation process.

tion of productions. Each node in a legal abstract syntax tree is an instance of some production in the grammar for the environment, and the type of each child of a node is determined by the corresponding component of the production.

When changes are made to the grammar of a structure-oriented environment, stored trees are usually rendered invalid. For example, if a production "$X::= ab$" in a grammar is changed to "$X::= abc$," thus adding a new component to the production, any "X" node in an existing AST will no longer be a valid instance of the new "X" production.

When such changes are made, one option is to simply discard all old trees. But this is clearly not a feasible alternative when substantial bodies of information are at stake. The alternative is to convert old trees to a form that is acceptable in the new environment.

A common approach to conversion is illustrated by the three-stage process pictured in Figure 1. First, a tree is "unparsed" to a textual form. Next, changes are made to this textual representation to produce a textual form that is legal for the new environment. Finally, the modified text is converted to a new tree by a parser for the new grammar.

There are a number of problems with this approach:

A Parser Must be Built. One of the attractions of structure-oriented environments is that they can be built without the overhead of producing a parser. Since the data in the environment is represented as an AST, tools such as compilers or interpreters can use these representations directly. While it is true that for a released system of production quality a parser would normally be constructed, construction of a new parser might not be necessary for intermediate internal versions used within a development group.

A Substantial Manual Effort May be Involved in Transforming Text. In addition to the work needed to produce a new parser or modify an existing one, some changes to the textual form must typically be made by hand before the new parser can be applied. One could write a textual transformer, but this itself must be handcoded. While some progress has been made in techniques for producing automatic transformers for text [Nix 1985], currently these techniques are not powerful enough to handle the range of transformations needed in this context.

Information May be Lost. In many structure-oriented environments, parts of the AST may not normally be displayed to a user. For example, a timestamp representing the creation date for a module in a programming environment might be kept for internal bookkeeping. Such information would typically be lost when the old tree is unparsed. Of course, it is possible to write a special unparser that writes out *all* stored information, but this would require the construction of a special parser and unparser just for the transformation process.

Conversion is Ad Hoc. Since there is no direct, enforceable correspondence between the changes made to the grammar and the changes that must be made to the textual form of a program or the translating parser, there is little guarantee that the resulting translated program will faithfully mirror changes intended by the implementor of the grammar changes.

An alternative approach is illustrated by the dashed line in Figure 1: Existing trees produced by the old environment are directly converted to trees that are valid in the new environment. Using this approach, no parser is needed; no human need be involved in the translation process; and the transformation can be quite efficient since no intermediate forms are involved.

But in order for such a solution to be practical it must be possible to generate a tree-to-tree transformer automatically or semiautomatically. Automation reduces the cost of creating a transformer and can provide certain guarantees of completeness and soundness. More importantly, nonautomated solutions require an implementor to have detailed understanding of the representations of trees and how they relate to their grammars. This is contrary to the aim of most structure-oriented environments, which remove the burden of understanding the internal representations by allowing the implementor to create and maintain an environment at the abstract level of the environment's grammar [Habermann et al. 1986].

In the remainder of this article we show how it is possible to automate the generation of transformers for enhancing structure-oriented environments. We describe an environment, called *TransformGen* (for "Transformer Generator"), in which a grammar may be modified. Given a set of grammar changes, TransformGen produces both a new grammar and a transformer that can be automatically applied to convert old trees to new ones.

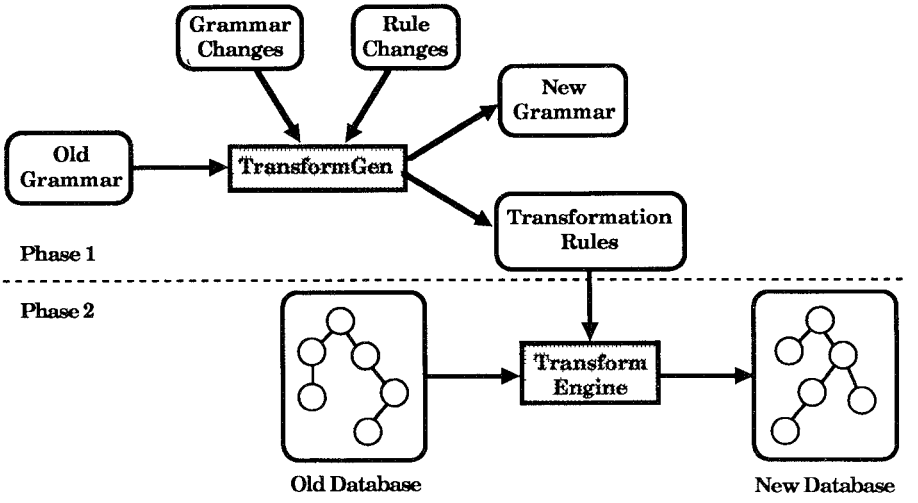

Fig. 2. The TransformGen process.

2.2 Overview of TransformGen

TransformGen is a structure-oriented environment itself. The data manipulated by TransformGen represents the grammars used to generate other structure-oriented environments. Within TransformGen, an implementor modifies existing grammars by applying TransformGen's editing commands. These commands allow an implementor to add and delete productions, change the number of components of a production, change the types of the components of a production, etc.

However, TransformGen not only provides an environment for modifying grammars but also maintains a table of transformation rules. Each transformation rule describes how to map an instance of a production in the old grammar into an instance of a production in the new grammar. As editing commands are applied to an existing grammar, TransformGen produces default transformation rules that correspond to those grammar changes. TransformGen also allows the implementor to augment the default transformation rules, again using a set of built-in editing commands.

The overall process of using TransformGen is pictured in Figure 2. As illustrated, the approach results in a two-phase process. In the first phase an implementor interacts with the TransformGen environment to modify a grammar as well as to augment the default transformation rules. In the second phase, a *transform engine* uses the transformation rules to convert old trees (valid for the original grammar) to new trees (valid for the modified grammar).

Although in principle existing trees can be transformed at any time, in practice the transformations are applied in a lazy fashion. Associated with each revision of an environment is the set of transformation rules to convert trees constructed with the previous incarnation of the environment. When

Initial Entry

```
production:  P
    new self:  P
    component 1:  Transform old component 1
    component 2:  Transform old component 2
    component 3:  Transform old component 3
```

Entry After Swap

```
production:  P
    new self:  P
    component 1:  Transform old component 2
    component 2:  Transform old component 1
    component 3:  Transform old component 3
```

Fig. 3. Table entry before and after swapping components.

the user attempts to read an out-of-date tree into a new environment, the environment automatically applies the transformation rules to convert the tree. If several revisions have occurred since the tree was last read, several transformation passes will be required.

The transformation of a tree is done in a top-down fashion. Starting with the root node of the old tree, the transform engine consults the rules associated with the production corresponding to that node. Each rule indicates what conditions must be true to use the rule, how to construct a new node, and how to recursively build the children of the new node. (The transformation engine algorithm is presented in Section 7.)

TransformGen stores its rules in a table, which has an entry for each production in the old version of the grammar. Each entry contains one or more rules. These determine how instances of that production should be translated. When TransformGen is first invoked on a grammar, every entry in its transformation table initially consists of the identity transformation rule. The result of using such a table is that each node in the old tree is simply copied to the new tree. As the implementor modifies grammar productions, TransformGen updates the table with rules that provide a default interpretation of each change made to the productions.

As a simple example, consider a production "$P::= abc$." Initially its entry in the transformation table would contain the rule pictured in the top portion of Figure 3. This rule simply copies instances of P from the old tree to the new. Suppose an implementor swaps the first and second component of P to produce "$P::= bac$" in the new grammar. The new entry would contain the default rule shown at the bottom of the figure. When applied to an instance of P, this rule swaps its first two children.

Because the implementor can change grammars and transformation rules only by using the commands provided, TransformGen is able to support two important properties. First, it establishes *completeness* in the sense that every production in the old grammar will have at least one transformation rule for converting nodes of that production into nodes of productions in the new grammar. Second, it guarantees *soundness* in the sense that any tree transformation that takes place will, in fact, result in a legal tree in the new grammar. (These properties are elaborated in Section 6.)

However, TransformGen does not attempt to completely eliminate the possibility that transformation errors will be encountered by the transform engine when converting a tree. This is because, as we will see later, certain transformation rules depend on the particular data in the tree being transformed. While it would be possible to limit the repertoire of the implementor to transformations that could be statically checked (and hence eliminate transformation time errors), to do so would significantly restrict the flexibility of TransformGen. Later sections will illustrate this point in detail.)

2.3 Requirements for Automated Transformation

The two fundamental challenges for implementing a transformer generator are *composability* and *coverage*. Composability refers to the problem of representing the effects of arbitrarily many grammar changes within a single transformer.[4] This becomes a significant problem when multiple modifications are made to interacting productions. To take a simple example, if an implementor adds a new component to a production and later reorders these components, the generated transformer must be able to interpret the composition of the two changes in terms of the original component list.

Coverage refers to the ability of a transformer to provide an appropriate transformation rule for each grammar change. In general it is not possible for a transformer generator to provide complete coverage automatically. This is because there may be many reasonable ways to transform existing trees so that they correspond to a given grammar change. It is therefore impossible to pick a single default that will always match the implementor's intentions.

To illustrate, suppose a production goes through the following change:

(1) $X ::= yz$—*the original form*
(2) $X ::= wz$—*implementor changes the type of the first component*

There are several legitimate interpretations of this change. One interpretation treats the change as a deletion followed by an addition. A second interpretation treats the change as a type modification. Adopting the first interpretation we would discard the first component of all X nodes when transforming trees. Adopting the second interpretation, we can exploit the fact that a node can be an instance of more than one (union) type, and keep the first component if it happens to be in the type required by the new grammar. In general, TransformGen chooses interpretations that attempt to maximize the amount of information preserved across transformations, and hence would pick the second interpretation. But it should be clear that TransformGen's choice may not be correct in all situations.

Since complete automatic coverage is not possible, it is essential that an implementor be able to augment the default rules to handle transformations that require alternative interpretations of the grammar-editing commands.

[4] There is also the issue of composing the transformational changes between successive revisions of a grammar. This is easily handled by functional composition of the individual transformers.

TransformGen allows the implementor to augment the transformation tables in two ways: by directly modifying the transformation tables and by writing transformation functions to perform more complex transformations. These are described in Sections 4.1.2 and 4.1.3, respectively.

3. A MOTIVATING EXAMPLE

In this section we illustrate the kinds of changes an implementor might make to a grammar and the ways we would expect a transformer to handle those changes. However, first we need to introduce some terminology and describe the context in which TransformGen is used.

3.1 ALOE Grammars and Databases

TransformGen is implemented as part of the Gandalf environment generation system [Habermann et al. 1991]. Gandalf environments are generated by linking together a language-independent kernel, a syntactic description of the environment, and a semantic description of the environment. An environment's syntax is defined by an ALOE grammar. This grammar determines the form of the data stored in a Gandalf database by defining the set of legal ASTs that a user of the environment can create.

An ALOE grammar is divided into three sections: terminals, nonterminals, and classes. The rules defining terminals and nonterminals are called *productions*. Consider the grammar fragment shown in Figure 4.

Every ALOE grammar has a unique root (or start) production. In this case the root is *MODULE*. A nonterminal production lists the classes of its components. There are two kinds of nonterminals: *fixed arity* and *variable arity*. The right-hand side of a fixed-arity nonterminal contains a list of components representing the classes to which the children of the nonterminal must belong. *MODULE* is a fixed-arity nonterminal with three components: *module-name*, *interface*, and *implementation*.

In contrast, the right-hand side of a variable-arity nonterminal contains a single class, syntactically enclosed in ⟨ ⟩. A variable-arity nonterminal node may have any number of children, but they must all come from the same class. Thus a variable-arity nonterminal represents a homogeneous list. In the example above, *IMPORTS* may have zero or more components, each of the class *import-item*.

A class is a type union of a set of productions.[5] Nonterminals use classes to indicate which set of productions are legal for each component. For example, the *interface* class defines a type that is the union of the *INTERFACE* and *EMPTY-INTERFACE* productions. Thus, an instance of an *INTERFACE* or *EMPTY-INTERFACE* production would be legal as the second child of *MODULE*.

There are two types of terminal productions: *valued* and *static*. A valued production has a lexical routine, which defines the legal values for the production. For instance, the *MODULE-NAME* production has a value that

[5]Our use of the term "class" is therefore quite different from its use in object-oriented languages, where "class" refers to an object type definition.

```
Root Production: MODULE

Nonterminals:

  MODULE = module-name  interface  implementation

  INTERFACE = imports  exports

  IMPORTS = <import-item>

  EXPORTS = <export-item>
  ...

Classes:
  module-name = MODULE-NAME

  interface = INTERFACE | EMPTY-INTERFACE

  imports = IMPORTS

  exports = EXPORTS

  export-item = EXPORT-ITEM

  implementation = SOURCE
  . .

Terminals:
  MODULE-NAME = {valued}
          lex: lexidentifier

  EXPORT-ITEM = {valued}
          lex: lexstring
  ...
```

Fig. 4. Grammar for a module description environment—Version 1

is accepted by the *lexidentifier* lexical routine. Static terminals do not require a value. They are typically used, as *EMPTY-INTERFACE* is here, to define optional components. Another use is to define the values of an enumeration type, such as *TRUE* and *FALSE*.

The grammar of an environment determines how a user constructs abstract syntax trees that are syntactically legal in the environment. Nonterminal and terminal productions in the grammar are *instantiated* to construct *nodes* in an abstract syntax tree. The user interacts with an ALOE environment by constructing a tree in a top-down fashion. Each nonterminal is displayed to the user as a template to be filled in. The template consists of some concrete syntax (also specified in the grammar, but not shown in the above example) and placeholders, or *metanodes*, for each child of the nonterminal. The user is expected to perform a construction at each metanode. The tree is complete when no metanodes remain.

3.2 An Example Grammar Modification and Transformation

To illustrate the problems associated with changing a grammar, consider again Figure 4. Here a *MODULE* is described as an entity having as one of

its components a single *implementation*, which is written in the C programming language.

Suppose that after having used an environment generated from this description, we decide to make the following enhancements to the system:

—allow multiple versions of each module implementation and associate a version number with each version,
—associate documentation with each module,
—allow a module implementation to consist of either C or Lisp source code,
—distinguish the export items to be either procedures, data types, or variables so that we can perform intermodule type checking.

The grammar might then look as pictured in Figure 5.

Any stored instance of a *MODULE* constructed from the environment based on the old grammar will now be obsolete. One reason is that the new environment expects *DOCUMENTATION* in the place that the old environment has an *INTERFACE*. Figure 6 shows the same logical entity (a module) with two different structural representations, one for each version of the grammar. The shaded portions of the figure represent those portions of the tree that are different in the two grammars.

What is needed is a way to transform old instances of *MODULE*s into the new format. A set of changes that will accomplish this transformation is the following:

—The old *SOURCE* of a *MODULE* must be nested as the second component of the first *VERSION* of a sequence of *VERSION*s.
—The new *VERSION* must be given a default *VERSION-NUMBER*. In this case we choose the value 1 as the initial version of existing modules.
—*SOURCE* must be changed to *C-SOURCE*.
—Each element in the list of *EXPORT-ITEM*s must be parsed and classified as either being a *PROCEDURE-SIGNATURE*, *TYPE-DECL*, or *VARIABLE-DECL*.

In Section 5 we present the details of such a transformer and describe how it is constructed by TransformGen. But first we introduce the basic concepts underlying the generation of transformers in TransformGen.

4. THE GENERATION OF A TRANSFORMER

To describe how TransformGen generates a transformer based on modifications to a grammar we first explain how TransformGen covers the full range of possible grammar modifications, and then we show how TransformGen composes interacting modifications within a transformer.

4.1 Achieving Coverage

Achieving coverage simply means that it is possible to express every transformation that an implementor might want to make. Recall from Section 2 that a transformation table contains an *entry* for each production in the old

```
Root Production: MODULE

Nonterminals:

  MODULE = module-name  documentation  interface  versions

  INTERFACE = imports  exports

  IMPORTS = <import-item>

  EXPORTS = <export-item>

  VERSIONS = <version>

  VERSION = version-number  implementation

  ...

Classes:
  module-name = MODULE-NAME

  documentation = DOCUMENTATION

  interface = INTERFACE | EMPTY-INTERFACE

  imports = IMPORTS

  exports = EXPORTS

  versions = VERSIONS

  version = VERSION

  version-number = VERSION-NUMBER

  implementation = C-SOURCE | LISP-SOURCE

  export-item = PROCEDURE-SIGNATURE | TYPE-DECL | VARIABLE-DECL

  ...

Terminals:
  MODULE-NAME = {valued}
          lex: lexidentifier

  DOCUMENTATION = {valued}
          lex: lextext

  VERSION-NUMBER = {valued}
          lex: lexinteger

  ...
```

Fig. 5. Grammar for a module description environment—Version 2.

version of a grammar, and each entry consists of one or more *rules*. Each rule has a condition and an action that describes how to transform old instances of a production. To achieve complete coverage, TransformGen supports three mechanisms for creating transformation rules: generating rules automatically, directly editing the transformation table, and writing transformation functions.

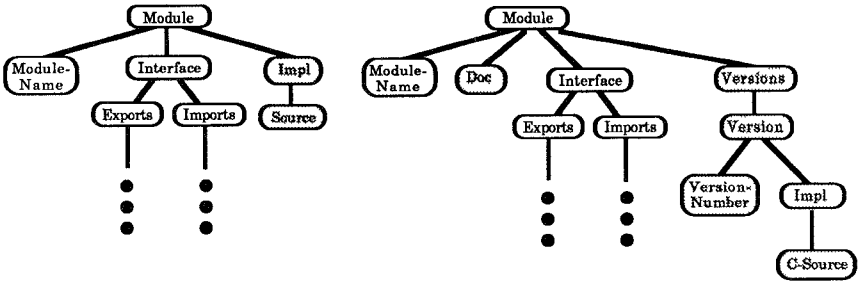

Fig. 6. Transforming an old MODULE to a new MODULE.

4.1.1 *Generating Transformation Rules Automatically.* As outlined in Section 2, TransformGen provides a set of commands for changing a grammar. In addition to modifying a grammar, each command also modifies the rules in the transformation table to reflect the grammar changes that it makes. We can classify grammar-editing commands into four categories based on their effects on existing databases. The categories are no-ops, syntactic-legality modification, minor restructuring, and major redefinition. As we describe these categories we will provide one example command from each. Complete descriptions of all TransformGen commands can be found in Staudt et al. [1988].

No-ops. As the name suggests, commands that are no-ops have no effect on existing databases. Since they cannot affect existing databases, they are also no-ops with respect to the transformation tables. Grammar-editing commands in this category include:

—Creating a production or class
—Adding a production to a class.

To illustrate why these are no-ops, consider the effects that creating a production should have on existing databases. Since the new production did not exist in the old version of the grammar, it is impossible for existing databases to have a node instantiated by that production. Since no instances of that production will be encountered when transforming databases, we do not require an entry in the transformation table.

Syntactic-Legality Modification. The syntactic-legality commands do not change the structure of the database that is being transformed, but *do* change the typing constraints of the grammar. In most cases when an editing command in this category is used, it will result in a situation where some nodes in the old database will be syntactically legal, but others will not. To deal with this situation the generated transformer must check syntactic legality when it performs the database transformations. Editing commands in this category include:

—Changing a lexical routine

Initial Table Entry

```
production:  OP1
    new self:  OP1
    value:  Use old value
```

Entry After Changing Lexical Routine

```
production:  OP1
    new self:  OP1
    value:  Use old value        { Warning:  Lexical routine was changed. }
```

Fig. 7. Table entry before and after changing a lexical routine.

—Deleting a production from a class
—Changing a component's class.

Consider the command that changes a lexical routine. Suppose a terminal production that uses the lexical routine *lexnumber* (which accepts both real numbers and integers) is changed so that it uses *lexinteger*. Then an instance of this terminal production that has a real number for its value in the old database will no longer be legal using the new grammar.

Since these commands do not affect the structure of the database, the transformation tables do not need to be changed. However, to help the implementor recognize that some databases may now be syntactically illegal, we annotate the transformation table with hints that indicate what type of change was performed. For the example of modifying a lexical routine, the two versions of the table entry are shown in Figure 7. The implementator then can scan the transformation table for these trouble spots and deal with them appropriately. As described later in this section, the implementor can directly edit the transformation table or add transformation functions to deal with the syntactically illegal cases. For the lexical-routine example the implementor might write a transformation function to round the illegal real number to create a valid integer.

Structural Modification. Structural-modification commands result in simple structural changes to a database. They always involve some manipulation of the components of a nonterminal. The structural-modification commands include:

—Deleting a component
—Creating a component
—Reordering components.

Suppose we delete component 1 of a production *FIXEDOP*. The transformation table must reflect the fact that the production now has one less component, and when performing the transformation it is the first child that should be skipped. Figure 8 shows the transformation table before and after the component deletion. The modified table says that component 1 of *FIXEDOP* in the new database should be a recursive transformation of component 2 of the old database. Component 1 from the old database is skipped, thereby

Initial Table Entry

```
production:  FIXEDOP
    new self:  FIXEDOP
    component 1:  Transform old component 1
    component 2:  Transform old component 2
    component 3:  Transform old component 3
```

Entry After Deleting a Component

```
production:  FIXEDOP
    new self:  FIXEDOP
    component 1:  Transform old component 2
    component 2:  Transform old component 3
```

Fig. 8. Table entry before and after deleting the first component of a fixed-arity nonterminal.

achieving the desired deletion. (Section 4.1.2 explains how the implementor can change the transformation table entries to move the old value of component 1 to another location in the database, rather than deleting the value.)

Major Redefinition. The major-redefinition commands change the basic nature of a production (e.g., changing a nonterminal to a terminal). Each requires that we completely replace at least one portion of a rule (e.g., delete all the component transformations). The major-redefinition commands include:

—Deleting a production
—Changing a terminal to a nonterminal, and vice versa
—Changing a static terminal to a valued terminal, and vice versa.

As an example of a major-redefinition command, consider the deletion of a production. When a production is deleted, all nodes built using that production will be illegal in the new database since the production for the node does not exist in the new grammar. Unless the implementor directs TransformGen to do otherwise, all nodes constructed with the deleted production will be discarded when transforming the database. The transformation table entry must then encode the fact that the node should be replaced by a metanode, and any information about recursively transforming components or values can be removed from the table. Figure 9 shows a transformation table entry before and after deletion of a production. This entry states that all *FIXEDOP* nodes should be replaced with a metanode. The effect of this transformation is that all subtrees rooted at *FIXEDOP* are discarded.

4.1.2 *Editing Transformation Tables.* Automatic generation of transformation table entries does not provide complete coverage. Transformations that are not automatically generated include:

—Nonlocal restructuring of the database
—Constructing default values or default children
—Making context-dependent changes
—Dealing with changes to the environment's semantics.

Initial Table Entry

```
production:  FIXEDOP
    new self:  FIXEDOP
    component 1.  Transform old component 1
    component 2:  Transform old component 2
    component 3:  Transform old component 3
```

Entry After Deleting the Production

```
production:  FIXEDOP
    new self:  META
```

Fig. 9. Table entry before and after deleting a production.

Despite the fact that these transformations cannot be automatically generated, they *can* be incorporated into a transformer if they are provided by the implementor. The implementor can describe the necessary transformations by modifying the transformation table or writing auxiliary functions to assist in the transformation. The fact that the implementor must provide the transformations is not a fundamental limitation of the transformation approach, but rather reflects the fact that many different transformations are possible for a single grammar modification. As we gain more experience with TransformGen we should be able to generate more of the common transformations automatically, but we believe that direct editing will always be necessary to capture arbitrarily complex transformations.

When editing the transformation tables, the implementor can define more powerful transformations than those that are automatically generated. These transformations are expressed in a language that extends transformations we have outlined with the following constructs:

—Tree expressions
—Production instantiations for component transformations
—Default values for valued-terminal nodes
—Function calls (to be discussed in the next section).

Tree expressions allow nonlocal restructuring of the database. They may appear in component transformation rules. They describe a path through the old database starting at the node being transformed and ending at the node that should be recursively transformed for the given component. A tree expression is a sequence of terms, evaluated from left to right. Each term uniquely identifies the next node in the path. Legal values for a term are: *component*, *parent*, *right* (sibling), and *left* (sibling). For example, a tree expression such as "Transform old parent, left" means that the node to be transformed is the old node's parent's left sibling.

Production instantiations supply default constructions for component transformations. These are most useful for adding components to an existing nonterminal and for nonlocal restructuring. When specifying a production instantiation, the implementor must also indicate how to compute the components or the value of the new production, using any technique desired (tree expressions, default values, production instantiations, or function calls).

```
production:  A
    new self:  A
        component 1:  Transform oldnode, right
        component 2:  Construct 3
                        Value:  1
```

Fig. 10. Example of an edited transformation table entry.

Default values are used to provide initial values to valued terminals created with production instantiation rules. The implementor can supply this value either by specifying a default value directly in the transformation table or by specifying the name of a function that will compute the value at transformation time.

Figure 10 illustrates how these constructs might be used. The value for component 1 is specified as a tree expression. The node to transform is the right sibling of the node currently being transformed. Component 2 uses a production instantiation rule to create a valued terminal with a default value of 1.

The TransformGen environment assures that a transformation table is syntactically correct. It also performs static analysis to determine whether or not the transformation table modifications made by the implementor correspond to a valid transformation for the old and new grammar by identifying potential sources of transformation time errors. (The details of this checking are given in Section 6.)

4.1.3 *Transformation Functions.* Thus far we have discussed how transformation tables are used to develop a transformer. The transformation tables use a declarative language to define nonterminal and terminal transformation rules. While allowing the implementor to modify the transformation tables has greatly enhanced the capabilities of TransformGen, there is still some functionality an implementor may need that cannot be expressed directly in the declarative language used for the transformation tables. We can effectively extend the expressiveness of the transformation tables by allowing implementors to call functions written in ARL, a powerful tree-oriented programming language [Staudt and Ambriola 1986]. In particular, context-dependent changes and changes resulting from new semantic restrictions often cannot be expressed directly in the transformation table language, but can be expressed easily in ARL.

There are four ways in which functions can be used in the transformation process:

—To compute boolean conditions
—To compute values for valued terminals
—To construct subtrees
—To locate a node in the old tree that is to be recursively transformed.

Boolean conditions are used to incorporate context dependency into the transformation process. The implementor can write boolean functions to check the state of the old or new database. If the function returns true, the attached transformation will proceed. Another use for boolean conditions is to avoid transforming a node that would violate the new semantics of the environment.

A function may be used to compute a value for a valued-terminal node. This value may depend on the contents of the old database (such as the value of a particular terminal in the old database), the contents of the new database, or even external information (such as the date).

An implementor may also write a transformation function that will return a complete subtree. The function may itself recursively invoke the transformer to complete the subtree. In general, we discourage use of this type of function since an implementor may make assumptions about the grammar when writing the function. If later grammar modifications violate these assumptions, the transformation function will fail at transformation time. We believe that this type of function is rarely needed to achieve complete coverage. (In Section 5 we show how a subtree construction function can be replaced by a combination of other techniques.)

A final use of transformation functions is to locate nodes in the old tree that should be recursively transformed. This class of functions is needed to handle the situations in which a path determined by a static tree expression is insufficient.

4.2 Achieving Composability

As the implementor issues each grammar-editing command, TransformGen updates the transformation table as described in Section 4.1. During this process the implementor can modify the transformation table at any time: either the intermediate versions of the table between grammar-editing commands or the version of the transformation table after all of the grammar-editing commands have been issued.

In any case, it is important that the final transformation table correspond to the entire sequence of grammar changes. We refer to the problem of encoding a sequence of changes in a single table as the *composability* problem. Composability is important because it allows TransformGen to generate a single-pass transformer for the complete sequence of changes.

Composability can be achieved if each grammar- or table-editing command can be interpreted in the context of any previous sequence of changes. Moreover, theoretical considerations aside, it is important that the meaning of a sequence of composed modifications be clear so that an implementor can understand the collective behavior. This is particularly important in the presence of automatic generation of transformation rules.

4.2.1 *Transformation Table Mappings.* It is useful to think of the grammar modification process as being in a *start* state, proceeding through a series of operations that modify the *current* state, until the *goal* state is reached. The start state is the old version of the grammar and the *identity*

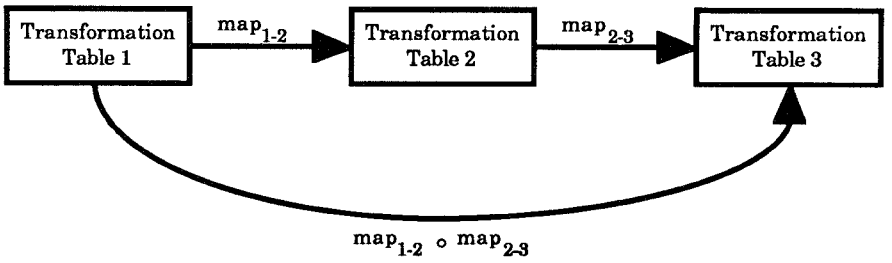

Fig. 11. A sequence of table revisions by table mappings.

transformation table, which transforms old databases into identical new databases. The current state is the most recent version of the grammar at a particular point in time and the corresponding transformation table. The goal state is the final version of the grammar and transformation table.

To formalize our description of transformation table modifications, we introduce an abstract model of *transformation table mappings*. For each editing command, we present a transformation table mapping that takes the transformation table from the current state to the next. Under this model, the composability problem reduces to the problem of composing transformation table mappings.

For example, Figure 11 shows a sequence of two revisions to a transformation table, made by applying the two mappings $map_{1\text{-}2}$ and $map_{2\text{-}3}$, and a single mapping that is equivalent to the composition of the sequence (the lower arc). After making a sequence of modifications to a transformation table, the implementor can reason about the effect of this sequence on the initial transformation table by thinking in terms of the equivalent composed mapping.

We express transformation table mappings as pairs of patterns: *match patterns* and *substitution patterns*. A match pattern denotes a production that was modified, and the substitution pattern denotes how the transformation table should be modified. When a mapping is applied to a table, every production instantiation for the production specified in the match pattern will be replaced by using the corresponding substitution pattern.

Figure 12 shows a simple mapping with a match/substitution pair and its effect when applied to a transformation table entry. The match pattern is a template for all instantiations of production A. A_1 and A_2 are *pattern variables* and denote the first and second component transformations in a match. This pattern matches the *new self* construction in the example table. A_1 is bound to "Transform old component 1," and A_2 is bound to "Transform old component 2." The substitution pattern describes how to modify the existing instantiation rules for production A. The instantiation of A is replaced by an instantiation of B, and the component transformations are swapped.

Mappings are applied to all production instantiation rules, which may occur in the *new self* portion of a rule or the component transformations.

Transformation Table Mapping

match pattern:		substitution pattern:
A	⇒	B
component 1. A_1		component 1. A_2
component 2. A_2		component 2. A_1

Transformation Table Entry Before Applying the Mapping

```
production:  A
    new self:  A
    component 1:  Transform old component 1
    component 2:  Transform old component 2
```

Transformation Table Entry After Applying the Mapping

```
production:  A
    new self:  B
    component 1:  Transform old component 2
    component 2:  Transform old component 1
```

Fig. 12. A transformation table mapping and its effect.

Note that the match pattern corresponds to a production instantiation in the current grammar version, and the substitution pattern corresponds to a production instantiation in the next grammar version.

Figure 13 shows the effect of applying a second mapping to the final transformation table of Figure 12. In this mapping, production B is replaced by production C, which has a new first component. To demonstrate the composition of mappings, Figure 14 shows a single composed mapping that is equivalent to the two sequential mappings.

4.2.2 *Transformation Table Mappings for Grammar-Editing Commands.* We now reexamine the categories of grammar-editing commands that were introduced in Section 4.1.1, define the transformation table mappings for some representative commands, and explore the implications on the composability problem.

No-ops. The commands that TransformGen treats as no-ops do not modify the table. These mappings are therefore trivial to compose with others.

Syntactic-Legality Modification. Commands that modify syntactic legality result in an annotation being added to some part of the table. Since annotations are simply hints to the implementor, they do not otherwise affect the table, and composition is trivial. Annotations are simply collected in a set for each *new self* and *component* slot in the table. The set insertion operation prevents duplicate annotations. For example, if a lexical routine is changed twice, the table entry would still have just one annotation.

Structural Modification The three TransformGen commands in this category all modify the component transformation portion of a transformation table rule. As the following three examples show, each structural-modification command composes with previous commands since the corresponding

Second Transformation Table Mapping

match pattern:		**substitution pattern:**
B	⇒	C
component 1. B_1		component 1. META
component 2. B_2		component 2. B_1
		component 3. B_2

Transformation Table Entry Before Applying Second Mapping

```
production:  A
    new self:  B
    component 1:  Transform old component 2
    component 2:  Transform old component 1
```

Transformation Table Entry After Applying Second Mapping

```
production:  A
    new self:  C
    component 1:  META
    component 2:  Transform old component 2
    component 3:  Transform old component 1
```

Fig. 13. A second mapping and its effect.

Composed Transformation Table Mapping

match pattern:		**substitution pattern:**
A	⇒	C
component 1. A_1		component 1. META
component 2. A_2		component 2. A_2
		component 3. A_1

Transformation Table Entry Before Applying Composed Mapping

```
production:  A
    new self:  A
    component 1:  Transform old component 1
    component 2:  Transform old component 2
```

Transformation Table Entry After Applying Composed Mapping

```
production:  A
    new self:  C
    component 1:  META
    component 2:  Transform old component 2
    component 3:  Transform old component 1
```

Fig. 14. The composed mapping and its equivalent effect.

transformation table mappings are based solely on the state of the transformation table before the command is applied and the particular command that was applied.

The command to delete a component from a grammar production has a corresponding table mapping to remove the component transformation from the transformation table. For example, Figure 15 shows a mapping that deletes the second component of production *A*. It is easy to see that this

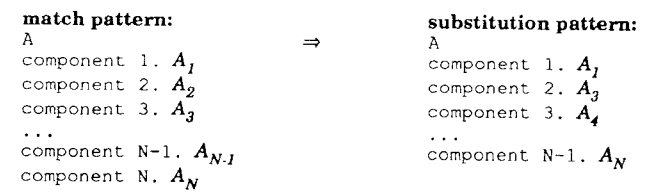

Fig. 15. Deleting the second component of production A.

mapping composes with any other mappings that produced the current state of production A since (1) the match pattern only depends on the current state of A and (2) the substitution pattern simply discards information to produce the next state of A.

The command to add a component to a grammar production has a corresponding table mapping to add the new component transformation in the transformation table. Without additional information from the implementor, TransformGen can only construct a *META* node for the new component. Our previous example in Figure 13 showed a mapping that adds a new component in the first position. This mapping also composes with other mappings that produce the current state of production A since (1) the match pattern only depends on the current state of A and (2) the substitution pattern simply adds a *META* node construction to produce the next state of A.

The command to reorder components in a grammar production has a corresponding table mapping to reorder component constructions in the transformation table. Our previous example in Figure 12 shows a mapping that swaps component constructions in positions 1 and 2. This mapping also composes with other mappings that produce the current state of production A since (1) the match pattern only depends on the current state of A and (2) the substitution pattern simply reorders the component transformations to produce the next state of A.

Major Redefinition. Major-redefinition commands require that we overwrite some portion of a transformation table entry. Overwriting is independent of the previous changes that have been made to the production. Therefore, the mappings associated with these commands trivially compose with previous mappings. Figure 16 shows examples of some major-redefinition mappings.

4.2.3 *Interactions between Transformation Table Mappings and Direct Editing.* In addition to the table modifications that TransformGen automatically applies as part of the grammar-editing commands, the model of transformation table mapping can be extended to handle the more extensive modifications that can be made by an implementor directly editing the transformation tables. This extension takes into account the fact that the implementor can add new information about how to construct components. This is expressed through extensions in the substitution patterns of a mapping.

Deleting a Production

match pattern:		**substitution pattern:**
A component 1. A_1 component 2. A_2 ... component N. A_N	⇒	META {Warning: all components deleted. }

Changing a Valued Terminal to a Nonterminal

match pattern:		**substitution pattern:**
A value: Old value ...	⇒	A {Warning: Old value discarded.} component 1. META component 2. META component 3. META ... component N. META

Changing a Static Terminal to a Valued Terminal

match pattern:		**substitution pattern:**
A	⇒	A value: UNDEFINED

Fig. 16. Example mappings for major-redefinition commands.

In previous mappings we have used only *META* and pattern variables as component transformations in substitution patterns. Additionally, an implementor can use the following:

—Production instantiations of productions defined in the next version of the grammar
—Tree expressions which follow the structure defined by the start grammar version
—Context-dependent transformations, whose boolean conditions may depend on the old or new database, but no intermediate states
—Default values for valued terminals in the goal version of the grammar, and
—Subtree transformation functions that return subtrees instantiated using the goal version of the grammar.

An example of a mapping that uses all of these extensions is shown in Figure 17. Note that in addition to pattern variables A_1 through A_N and *META* constructions, the substitution pattern now contains literal strings that correspond to the direct editing extensions. The mapping adds a new production instantiation for the second component of production A and a conditional transformation for the third component.

This example demonstrates how the implementor's commands to edit the transformation table directly compose with previous transformation tables, by selectively overwriting portions of the table, and reusing information via the pattern variables. It is interesting to note that a complicated table

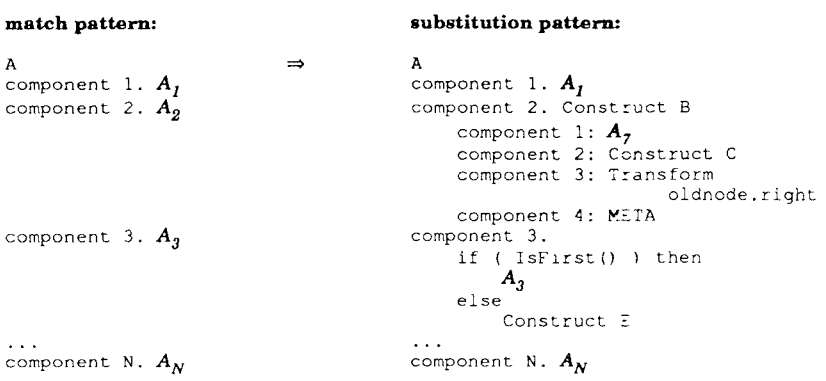

Fig. 17. Example extensions to the table mappings.

produced by an implementor may also be matched by a match pattern. Match variables can be bound to the table entries as before and reused in substitution patterns. Therefore, editing commands issued after the implementor has edited the transformation table also compose.

4.2.4 *Summary of Composing Transformation Table Mappings.* The previous examples have demonstrated that we use only a few simple techniques to ensure composability of TransformGen operations on transformation tables:

—The match patterns in a transformation table mapping correspond to production instantiations in the current state of the transformation table.
—The substitution patterns correspond to production instantiations in the next state of the transformation table.
—The pattern variables in the match pattern correspond to component transformations in the current state of the transformation table.
—The pattern variables in the substitution pattern depend only on pattern variables in the match pattern.

By making sure that these conditions are satisfied in each of our mappings, we can be sure that every possible sequence of transformation table mappings in TransformGen will compose. Furthermore, this model allows the implementor to understand and reason about the effects of any sequence of grammar-editing commands.

5. A SOLUTION TO THE MOTIVATING EXAMPLE

In this section we step through the generation of a transformer using the example introduced in Section 3.1. The original version of the grammar is shown in Figure 18. Recall that we would like to change the grammar so that:

—Documentation is associated with each module.
—A module implementation can be written in either C or Lisp.

```
Root Production: MODULE

Nonterminals:

  MODULE = module-name  interface  implementation

  INTERFACE = imports  exports

  IMPORTS = <import-item>

  EXPORTS = <export-item>

  ...

Classes:
  module-name = MODULE-NAME

  interface = INTERFACE | EMPTY-INTERFACE

  imports = IMPORTS

  exports = EXPORTS

  export-item = EXPORT-ITEM

  implementation = SOURCE

  ...

Terminals:
  MODULE-NAME = {valued}
          lex: lexidentifier

  EXPORT-ITEM = {valued}
          lex: lexstring

  ...
```

Fig. 18. Grammar for a module description environment—Version 1.

—Export items are distinguished as being either procedures, data types, or variables so that we can perform intermodule type checking.

—Each module may contain multiple versions, and a version number is associated with each version.

To allow the attachment of documentation to modules we perform three related grammar changes. First we add a new component to the *MODULE* production. Since there are no existing classes that define documentation, we define a new class, called *documentation*, to be the class of the new component. Finally, we define a new valued-terminal production called *DOCUMENTATION* to serve as the sole member of the new class. The first modification requires that we add a new child to all *MODULE* nodes when we transform them. The last two changes are no-ops with respect to existing databases since no existing databases depend on the *documentation* class or *DOCUMENTATION* production. It is therefore not necessary to modify the transformation table for these last two changes. The new definition of the *MODULE* production is:

MODULE = module-name documentation interface implementation

```
production:  MODULE
    new self:  MODULE
    component 1:  Transform old component 1
    component 2:  META
    component 3:  Transform old component 2
    component 4:  Transform old component 3
```

Fig. 19. Transformation table after adding *documentation* component to *Module* production.

The updated transformation table entry resulting from the "add component" table mapping is shown in Figure 19. Using this entry the first component of *MODULE*, *module-name*, is recursively transformed. The second component is the new component, which will be a metanode. The remaining two components are transformations of the corresponding components in the old database. However, notice that their indices differ to account for the new second component.

At this point we can decide whether the transformation entry created by TransformGen is appropriate for this situation. An obvious alternative for the implementor would be to construct a *DOCUMENTATION* node with some default value, perhaps just the name of the module. However, it seems likely that the implementor would leave the new node as a metanode so that the user of the environment can supply documentation. Therefore we leave the automatically generated entry as is.

The next step is to allow either C or Lisp source code. We know that existing databases contain only C code. To achieve this goal we introduce a new production called *LISP-SOURCE*. Since we want to allow Lisp code for any module implementation, we add *LISP-SOURCE* to the *implementation* class. For the sake of improving the naming, we rename the existing *SOURCE* production to *C-SOURCE*. The rename command changes the name in its definition in the production as well as all uses in class definitions.

Neither adding a new production nor adding a production to a class can affect existing databases. Therefore, those two changes have no effect on the transformation table. When we change the name of a production, TransformGen also changes its name in every production instantiation in which it appears in the transformation table. For example, the transformation table entry after changing *SOURCE* to *C-SOURCE* is shown in Figure 20.

The next modification allows us to distinguish among various types of exported items. In the original version of the grammar, exported items were represented simply as strings with no internal structure. If we wish to add intermodule type checking, it would be useful to have more structure associated with these exported items. We therefore need to define new productions to represent procedure signatures, type declarations, and variable declarations. The transformer will need to convert these strings into pieces of structure.

The grammar changes are simple. We define three new productions, *PROCEDURE-SIGNATURE*, *TYPE-DECL*, and *VARIABLE-DECL*. We also change the *export-item* class, so that it contains the three new productions in its class, and remove the use of the production *EXPORT-ITEM*. We also

```
production:  SOURCE
    new self:  C-SOURCE
    ...
```

Fig. 20. Transformation table after renaming *SOURCE* to *C-SOURCE*.

```
production:  EXPORTS
    new self:  EXPORTS
    list:  Transform old list  { Warning:  Class has been modified. }
```

Fig. 21. Transformation table after modifying the *export-item* class.

delete the production *EXPORT-ITEM* since we will no longer be using that production in the new version of the grammar.[6]

This modification requires the transformer to perform *context-dependent* transformations. Depending on the value of the *EXPORT-ITEM* in the old database, we decide which production to construct in the transformed database. As noted in Section 4.1.2, context-dependent changes are beyond the scope of what TransformGen can automatically create entries for. Nevertheless, let us see how TransformGen has changed the table.

Defining the new productions and adding them to a class do not affect the tables. However, deleting a production from a class and deleting a production do. When a production is deleted from a class, each production that uses the modified class receives an annotation reminding the implementor of the change. As a result the *EXPORTS* table entry is modified as shown in Figure 21. When a production is deleted, the transformation table is changed so that all occurrences of the deleted production are replaced by metanodes. The entry for *EXPORT-ITEM* is shown in Figure 22.

The transformation table generated by TransformGen does not do what we want, but this should not be a surprise because of the context-dependent nature of the change. In order for the transformer to implement the desired change correctly, the implementor must write some transformation functions. One solution is to write a function to construct the subtree required by the new grammar. This function would parse the value of the *EXPORT-ITEM* and construct a different subtree depending on the value. The transformation table would be changed as shown in Figure 23.[7]

The advantage to performing the transformation as above is that we only need to parse the value once. However, there is a major disadvantage if later changes are made to the productions used by *parse_export*. If the structure of exported items is changed, the implementor must modify this function to reflect those changes. However, if the implementor describes the transformation using production instantiations in the table, TransformGen will be able

[6] In reality we would probably define more new productions and classes to create more detailed structure, but we ignore those details for this example.

[7] Note that all transformation functions are implicitly passed two parameters: the node from the old tree that we are trying to transform and the metanode in the new tree where we are trying to apply the transformation. The syntax for function calls does not list these parameters because the environment implementor has no control over them and cannot change them.

Fig. 22. Transformation table after deleting *EXPORT-ITEM* production.

```
production:  EXPORT-ITEM
    new self:  META
```

```
production:  EXPORT-ITEM
    new self:  Replace with tree defined by parse_export()
```

Fig. 23. Transformation table after adding call to transformation function

to compose later changes automatically. Figure 24 shows how the transformation could be done directly in the transformation table.

The disadvantage of the second approach is that we may need to attempt to parse a value three times before we know how to perform the transformation. This disadvantage is due to the computational weakness of the transformation table language, rather than an inherent weakness of TransformGen's approach. The advantage is that if the implementor modifies one of the productions mentioned in this entry, TransformGen will use the composition rules to correctly update the entry.

The final set of grammar changes will allow us to have more than one version of each module. Implementation of this change requires the addition of three productions: *VERSIONS*, *VERSION*, and *VERSION-NUMBER*. We also need to add three classes corresponding to these productions: *versions*, *version*, and *version-number*. Finally, we need to modify the *MODULE* production so that it has *versions* as a component rather than *implementation*. The revised portions of the grammar are shown in Figure 25.

As with the earlier modifications, adding productions and adding classes do not change the transformation tables. The only change that modifies the table is the change to the *implementation* component of *MODULE*. The new entry for *MODULE* is shown in the top of Figure 26.

Unfortunately, this does not quite produce the effect we want. What is required is a nonlocal restructuring of the database. Specifically, we need to add two levels to the database. The implementor can accomplish this by editing the transformation table entry for *MODULE* to look as shown in the bottom of Figure 26. This entry says that the fourth child of a *MODULE* node should be *VERSIONS*. Its first child should be a *VERSION*. The first component of *VERSION* should be *VERSION-NUMBER* with a default value of 1. The second child of *VERSION* should be the recursive transformation of the third child of the original *MODULE* node, which is the *IMPLEMENTATION* node. While the automatically generated mechanism is not sufficiently powerful to support this nonlocal restructuring, the declarative language used by the transformation tables allows the implementor to get the correct effect easily.

6. STATIC ANALYSIS OF TRANSFORMATION TABLES

Since TransformGen may be used to transform databases that contain valuable information that cannot be reproduced, errors and unpredictable behavior are serious concerns. Therefore, we must be able to check statically

```
production:  EXPORT-ITEM
    if IsProcSignature() then
        new self:  PROCEDURE-SIGNATURE
        component 1:  ...
        ...
    else if IsTypeDecl() then
        new self:  TYPE-DECL
        ...
    else if IsVarDecl() then
        new self:  VARIABLE-DECL
        ...
    else
        new self:  META
```

Fig. 24. Transformation table with conditional transformations.

Nonterminals:

MODULE = module-name documentation interface versions

VERSIONS = <version>

VERSION = version-number implementation

Classes:

versions = VERSIONS

version = VERSION

version-number = VERSION-NUMBER

Terminals:

VERSION-NUMBER = {valued}
lex: lexinteger

Fig. 25. Grammar changes to support multiple module versions.

Generated Table Entry

```
production:  MODULE
    new self:  MODULE
    component 1:  Transform old component 1
    component 2:  META
    component 3:  Transform old component 2
    component 4:  Transform old component 3
                  { Warning:  The class was changed. }
```

Table Entry after Implementor Modification

```
production:  MODULE
    new self:  MODULE
    component 1:  Transform old component 1
    component 2:  META
    component 3:  Transform old component 2
    component 4:  VERSIONS
        component 1:  VERSION
            component 1:  VERSION-NUMBER
                value:  1
            component 2:  Transform old component 3
```

Fig. 26. Table entry for adding multiple versions before and after implementor modification.

whether or not a transformation table will successfully transform all possible databases as intended, and if not why and where it might fail. In this section we describe static analysis algorithms that warn implementors about potential transformation time errors.

It is helpful to first review the aspects of database transformations that TransformGen addresses in order to understand exactly what static checking it can be expected to accomplish. TransformGen performs syntactic database transformations, but does not address the environment semantics. The set of databases that are both *syntactically and semantically* legal in the *domain* and *range* of a database transformation is clearly a subset of those that are simply syntactically legal. In the case of a semantically restricted domain (i.e., databases that are syntactically correct under the old grammar but cannot exist due to semantic constraints), our algorithm may warn that the transformation table does not adequately handle the semantically illegal databases. It is the implementor's responsibility to ensure that these cases will never be encountered. In the case of a semantically restricted range, our static checking algorithm will not warn that if semantically illegal databases may be produced by a transformation. The implementor can use context-dependent transformations to prevent them from being created during transformation. (We describe the four types of static analysis performed by TransformGen next.)

Old-Grammar Check. TransformGen guarantees that every terminal and nonterminal production in the old grammar has an entry in the transformation table. TransformGen ensures that there is an entry for each old production by creating the initial identity table with all the entries, and then preventing the implementor from deleting table entries.

Root Check. TransformGen checks that the root production from the old grammar version is transformed into the root production of the new grammar version.

New-Grammar Check. TransformGen checks that all production instantiations in the transformation table are consistent with the new grammar. For example, it guarantees that nonterminal transformations have the correct number of component transformations, and that each production instantiation that occurs in a component transformation is syntactically legal for the component.

Tree Expression Check. TransformGen checks to find which tree expressions may not evaluate to nodes in the old database. Furthermore, it checks that if the resulting nodes are transformed, they will be legal for the given component.

The first three checks are straightforward. An error encountered while doing these checks must be corrected before a transformer can be generated. However, the last check generally results in warnings rather than errors, since it is possible that a tree expression can be successfully evaluated for some databases, but not for others. A transformer can be generated if a

warning is found by the Tree Expression Check, but an error may occur at transformation time depending on the state of the database being transformed.

The Tree Expression Check algorithm takes the tree expression specifying which node in the old database to transform, determines all possible productions from the old grammar that the tree expression may evaluate to, finds all possible productions in the new grammar that the old productions may transform to, and then checks that all of these productions are legal components of their parents.

The *TreeExpressionCheck* algorithm is shown in Figure 27. It is passed a start production in the old grammar and a tree expression to evaluate with respect to the start production. The algorithm constructs the minimal set of productions in the new grammar such that, if we evaluate the tree expression with respect to the start production, and then transform the resulting node, we will get a member of the set.

The first loop of the algorithm iterates over each term in the tree expression (left to right) to find all possible productions in the old database that are reachable by evaluating the expression up to that term. We use two sets, *currentset* and *nextset*, in this loop. The set of productions that are reachable by evaluating the expression up to and including the term from the previous iteration are stored in *currentset*. The set *nextset* is empty at the beginning of each iteration and is used to collect the productions reachable by the next term. The productions to be inserted into *nextset* are computed by evaluating the term (selected by the **switch** statement) with respect to each production in *currentset*. Warnings are reported if an attempt is made to access the parent of the root production or a component of a production that is out of range (e.g., trying to access the fourth component for a production that only has three).

After all terms in the tree expression have been evaluated, *currentset* contains all productions in the old grammar that are reachable from the *startproduction* by the tree expression. The algorithm then finds all productions in the new grammar that can result from transforming productions in *currentset*. This set contains all productions that may result from a transformation rule on the start production and tree expression. We then call *ConstructionCheck* to check that each of these new productions is legal at the given component of the start production.

The *ConstructionCheck* algorithm is shown in Figure 28. It is passed a parent production in the new grammar, a component location for that parent production, and a production to be instantiated at that component location. It first checks to see if the production to be instantiated is actually in the new grammar. Then, it checks to see if the production to be instantiated is legal in the specified component location of the parent. We do this by determining the class of the component location for the parent and then checking to see if the new production is a member of this class.

Note that if this check fails, the algorithm issues a *Warning* rather than an *Error*. This is because there may be no existing database that would actually have a node that would be transformed to the one in *componentproduction*,

```
TreeExpressionCheck (startproduction, whichcomponent, treeexpression)
{
  currentset = { startproduction }
  nextset = { }

  for each term T in treeexpression
  {
    switch (T)
    {
      case component:
        for each production P in currentset
        {
          if (P is a terminal production)
            Warn(treeexpression, "has component access for a terminal", P, "in term", T);
          else if (P is a variable arity production)
            Warn(treeexpression, "has component access on a list", P, "in term", T);
          else if (component number of T is out of range for P)
            Warn(treeexpression, "has out of range access", T, "for production", P);
          for each production N in component class T of P
            nextset = nextset ∪ { N };
        }
        break;
      case parent:
        for each production P in currentset
        {
          if (P is the root production)
            Warn(treeexpression, "parent access for root production", P, "in term", T);
          for each class C where P is a member
            for each production P with C as a component
              nextset = nextset ∪ { P };
        }
        break;
      case right:
        -- similar to "parent" followed by "component"
        ...
        break;
      case left:
        -- similar to "parent" followed by "component"
        ...
        break;
    }
    if (nextset == { })
    {
      Error(treeexpression, "will never evaluate to a node");
      return ;
    }
    currentset = nextset;
    nextset = { };
  }

  resultset = { };
  for each production P in currentset
    for each rule R in table entry for P
      resultset = resultset ∪ {new self of R };

  for each production P in set resultset
    ConstructionCheck(startproduction, whichcomponent, P);
}
```

Fig. 27. The TreeExpressionCheck algorithm.

due to the semantic constraints of the environment. Since TransformGen cannot statically evaluate environment semantics, it is the responsibility of the implementor to verify that the *Warning* condition does not occur.

The static-analysis algorithms presented here have several limitations. The most obvious is the inability to check the behavior of implementor-defined

```
ConstructionCheck(parentproduction, whichcomponent, componentproduction)
{
  let newgrammar be the new grammar version
  if (componentproduction is not a production of newgrammar)
    Error(componentproduction, "is not a production of ", newgrammar);

  let newclass be the class for whichcomponent of production parentproduction in newgrammar
  if (componentproduction is not a member of newclass)
    Warning(componentproduction, "is not a member of ", newclass);
}
```

Fig. 28. The ConstructionCheck algorithm.

transformation functions called from a transformation table. It is, therefore, the responsibility of the implementor to check that transformation functions are correctly implemented.

A related issue is that the boolean conditions for selecting rules in a table entry or a component transformation are currently expressed through implementor-defined transformation functions. Therefore our algorithm treats alternative rules guarded by boolean conditions as nondeterministic alternatives. That is, our algorithm does not attempt to analyze the conditions that must hold for each rule to be selected, but rather assumes that any rule within a group of alternatives may be selected under any condition. Again, it is up to the implementor to determine whether or not the boolean conditions guarding the rule will prevent an error at transformation time. In any event, the transformer will not allow implementor errors to corrupt a transformed database. The transformer guarantees that only syntactically legal databases are built, and implementor errors in the transformation functions that are encountered at transformation time will be detected and reported. It is, however, the implementor's responsibility to ensure that transformed databases correspond to the semantic rules of the new environment.

It is possible that, due to an error in a transformation table, an infinite mutual recursion may occur during the transformation of a database. For example, if the transformation of one node calls for the recursive transformation of a sibling in a component transformation, and if the sibling calls for the recursive transformation of the first node in one of its component transformations, then the transformer will attempt to build an infinite branch in a new database tree. As part of the future work on TransformGen, we need to fully characterize the problem and determine if there is a means of detecting infinite recursion in transformation tables.

Despite these drawbacks, static analysis is an essential tool to be used when generating transformers. Since the new tree created by the transformer is constructed top-down, all nodes except the root will be constructed as a component of another node. The transformer guarantees that a new root will be created (using *Root Check*) and that each component will be correctly constructed, or a warning will be produced by the static analysis. Therefore, if the errors and warnings identified by static analysis are examined and corrected as necessary, and no subtree transformation functions are used,

```
TransformEngine (database) returns tree
{
   let thetree be the tree constructed when reading database
   let theversion be the grammar version used by thetree
   while (theversion is not the current version of the grammar)
   {
      let newtree be a new empty tree for version theversion + 1
      TransformNode (root of thetree, root of newtree);
      let thetree be newtree
      let theversion be theversion + 1
   }
   return thetree;
}
```

Fig. 29. The TransformEngine driver.

then the generated transformer is guaranteed to construct a syntactically correct tree consistent with the new version of the grammar without reporting any transformation time errors.

7. THE TRANSFORM ENGINE ALGORITHM

We now turn our attention to the second phase of the TransformGen process, where the transformation engine interprets the table provided by TransformGen to convert a database from one version of a grammar to another. In this phase, a given transformer is simply used as a front-end to a Gandalf structure editor.

When a user invokes an editor on a database, the database header is examined to determine which version of the grammar the database uses. This information determines whether transformation is required, and, if so, which transformer(s) to apply. If the database does not use the most current version, we perform a multipass transformation. Each pass updates the database to the subsequent version. Thus if a database is three versions out of date, we use three passes of the transformation algorithm. For small databases, we transform the entire database. However, ALOE environments typically divide large databases into segments [Krueger et al. 1989]. (A small database is typically the size of a procedure, while a large database may represent all the modules composing a large program.) For segmented databases, we transform individual segments as intersegment references are followed, rather than the entire database at once.

A single pass of the transformer constructs a new database in a top-down fashion using information from the old database, the transformation table, and a new grammar. We start by first building a new database tree with a metanode at the root, as shown in the *TransformEngine* algorithm of Figure 29. We then consult the transformation table entry for the root production of the old database tree to determine the root production in the new database. *TransformNode* transforms a node and recursively transforms the node's components. Therefore the entire database is transformed when the call to *TransformNode* completes.

```
TransformNode (oldnode, newmeta)
{
  let rule be the first transformation rule in the entry for oldnode's production whose condition
    evaluates to true

  if (rule is an instantiation rule)
    ApplyInstantiationRule (oldnode, newmeta, rule);
  else if (rule is an implementor-defined function)
    ApplyFunctionRule (oldnode, newmeta, rule);
  else
    ReportErrorAtNode ("No applicable rule for node.", newmeta);
}
```

Fig. 30. The TransformNode algorithm.

```
ApplyInstantiationRule (oldnode, newmeta, rule)
{
  let newproduction be the production specified by rule
  let newvalue be the value specified by rule

  if (instantiation of newproduction with value newvalue is syntactically legal at newmeta)
  {
    let newnode be a new node with production newproduction and value newvalue
    place newnode in the database at newmeta

    TransformComponents (oldnode, newnode, rule);
  }

  else
    ReportErrorAtNode ("Syntactically illegal instantiation attempted at node.", newmeta);
}
```

Fig. 31. The ApplyInstantiationRule algorithm.

The heart of the transformation algorithm is the recursive-descent algorithm, which relies on three subroutines: *TransformNode*, *ApplyInstantiationRule*, and *TransformComponents*.

TransformNode is shown in Figure 30. To transform an individual node, we need to determine which rule in the transformation table entry for the node's production to apply. The entries are indexed by productions from the old version of the grammar. For each entry there may be multiple rules. We apply the first rule whose condition evaluates to true.

There are only two types of rules possible. If the rule specifies a specific production to instantiate, we call *ApplyInstantiationRule* (shown in Figure 31). If it specifies an implementor-defined function to call, we call *ApplyFunctionRule* (not shown due to its simplicity). If no rules evaluate to true, then we do not transform *oldnode*. Instead the metanode passed in as *newmeta* remains unchanged, and the user receives an error message.

ApplyInstantiationRule, shown in Figure 31, interprets production instantiation rules. A production instantiation rule consists of the name of a production from the new version of the grammar, a rule for providing a value for the production (if required), and rules for constructing each child of the node in a new database. We construct a node with the new production and

value (if required) in the new database. Then we apply the rules for the components to get values for the new node's children. Note that the test for syntactic legality is actually unnecessary in most cases since TransformGen checks syntactic legality as the transformation table is constructed. However, it is required in situations in which TransformGen issued warnings that a transformation might be syntactically illegal depending on the state of the trees encountered, such as when a tree expression is used to identify a node to recursively transform.

TransformComponents, shown in Figure 32, creates a value for each component of a newly created nonterminal. The component transformation algorithm is very similar to the node transformation algorithm already shown in *TransformNode*. The major exception is that there are three potential ways of getting a value for a component. As with node transformation, the implementor can specify a particular production to instantiate or an implementor-defined function to call. Additionally the implementor can request the recursive transformation of some node from the old database using a tree expression. To interpret a recursive transformation rule, we need to find the rule associated with the production of the node we wish to transform, in this case the third component of *oldparent*. To do this, we simply call *TransformNode* recursively to transform the desired node from the old database.

The expected performance of the transformation algorithm is $O(n)$ where n is the number of nodes in the new database. It is linear since we evaluate at most two rules for each node in the new database: one in *TransformComponents*, and if the node is created by a recursive transformation, we also apply a rule in *TransformNode*. We cannot place an upper bound on the complexity of transforming a database, since the implementor can write arbitrarily complex transformation functions.

8. CASE STUDY

Since TransformGen's creation in 1986, it has been used to create new versions of each environment used within the Gandalf project. To provide a more concrete understanding about how TransformGen is used in practice, we present a case study describing the transformation of the ARL environment [Ambriola et al. 1984] from version 1 to version 2. ARL is a tree-oriented language for describing the semantic processing in Gandalf environments. It is distributed with the current Gandalf distribution and is used at dozens of academic and industrial institutions around the world.

The ARL grammar contains over 150 productions and is now in its fifth version. Changing from version 1 to version 2 of ARL required 10 changes to the ARL grammar. These grammar changes affected 72 productions and 9 classes. One and one-half hours were spent making changes to the ARL grammar and editing the transformation tables. The transformation tables contained 36 modified entries. Of these, TransformGen correctly generated 26 entries. While it was necessary to write 50 functions to assist in the transformation, all of these functions were extremely simple. In particular, 48 of them were single-line boolean functions that examined one node in the old

```
TransformComponents (oldparent, newparent, rule)
{
  for each component C of newparent
  {
    let componentrule be the first component rule of rule for component C whose condition evaluates
      to true

    if (componentrule is a recursive transformation)
      let oldnode be the node referenced by the expression in the transform statement
      TransformNode (oldnode, C);
    else if (componentrule is an instantiation rule)
      ApplyInstantiationRule (oldparent, C, componentrule);
    else if (componentrule is an implementor-defined function)
      ApplyFunctionRule (oldparent, C, rule);
    else
      ReportErrorAtNode ("No applicable rule for component.", C);
  }
}
```

Fig. 32. The TransformComponents algorithm.

database and could have easily been eliminated by some simple improvements in the transformation table language.

It is useful to consider each of the ARL modifications in which the tables generated by TransformGen were not correct. Each of these modifications present a different type of grammar change that exceeds the capabilities that are currently automated and requires implementor intervention. While implementor intervention is required, the actual table modifications made by the implementor are quite simple. Thus the entire process of generating a transformer remains a simple one.

The ARL grammar changes for which TransformGen could not automatically generate transformations are the following:

—Replacing a production with a set of productions
—Creating a default value
—Adding a structural level to the database
—Merging multiple nodes into a single node.

The difficulty in replacing a production, call it A, with a set of productions, $\{B, C, D\}$, is that the transformer must decide whether to construct B, C, or D in the new database when it encounters production A in the old database. Essentially, the transformation table must identify the conditions under which each new production should be chosen. Since TransformGen generates tables solely based on the grammar modifications, it cannot add conditions to the tables, since that requires understanding the meaning of the productions.

In the conversion of ARL, we were replacing a valued terminal with a set of static terminals. The choice of which static to build depends on the value of the original terminal. As a result it was necessary to write boolean functions to examine the state and decide on a particular production to use. The original production was:

FIELD-NAME = {valued}

```
ARITY = { static }
CLS = { static }
...
```

Fig. 33. The new static productions.

```
production:  FIELD-NAME
    if IsArity() then
        new self:  ARITY
    else if IsCls() then
        new self:  CLS
    ...
    else
        Report "Unexpected value for a FIELD-NAME."
```

Fig. 34. Converting *FIELD-NAME* to one of a set of static terminals.

Figure 33 shows a subset of the static productions that replaced it. Figure 34 shows the corresponding transformation table entry. The entry consists of a list of rules that each look for a different value. If the value is something other than what we expect, an error is reported at transformation time using the *Report* command. This command prints an error message and highlights the node in the new database that we were trying to transform.[8] Figure 35 shows the body of one of the boolean functions.

Another change that required editing of the transformation table involved providing an initial value for a new component of a production. By looking at the grammar modifications, TransformGen knows that a component has been added. However, it has no way of knowing what a reasonable initial value for the component should be. It therefore leaves the component as a metanode and expects the user to provide the value. The implementor can provide the initial value instead of the user by a simple modification to the transformation table.

In ARL, the original version of the production was:

DAEMON = daemon-name parameters event-part

The new version of the production is:

DAEMON = daemon-name parameters declarations event-part

TransformGen generated a table entry that would leave the new component, *declarations*, as a metanode, shown in Figure 36. The implementor made the modification shown in the bottom of Figure 36 in order to provide an initial value. The value specified is an empty list of declarations.

A third type of grammar change that is not currently automated in TransformGen is one in which an existing structure is nested inside another structure. This is an example of a more general problem of nonlocal restruc-

[8] In reality, the implementor would probably want to call a function to report the error, so that an error message could be constructed that would indicate what the value was in the old database.

```
boolean function IsArity (node oldnode, newmeta)
  /* Return true if the value of oldnode is the string "arity" */
  return StringEq (oldnode.value, "arity");
```

Fig. 35. Simple boolean function.

Generated Table Entry

```
production:  DAEMON
    new self:  DAEMON
    component 1:  Transform old component 1
    component 2:  Transform old component 2
    component 3:  META
    component 4:  Transform old component 3
```

Table After Implementor Modification

```
production:  DAEMON
    new self:  DAEMON
    component 1:  Transform old component 1
    component 2:  Transform old component 2
    component 3:  DECLARATIONS
        List:  empty
    component 4:  Transform old component 3
```

Fig. 36. Table entry for a new component before and after implementor modification.

turing of the data. In the general case, moving data around might require arbitrary computation and cannot be simply derived by examining the grammar changes. In the current version of TransformGen, no nonlocal restructuring is automated.

In the ARL conversion, we were allowing a list of nodes where previously only a single node was legal. As a result, the transformation tables needed to be modified to build the extra level for the new list production, and transform the old node as an element of the list. Originally a *CASE-STATEMENT* allowed each part (*CASE-ELEM*) to test only one value as below:

CASE-ELEM = value statement

The new version of CASE-ELEM allows us to attach multiple values to each statement.

Nonterminals:
  CASE-ELEM = values statement
  VALUES = ⟨value⟩
Classes:
  values = VALUES

The transformation table entry generated by TransformGen is shown in the top half of Figure 37. This entry specifies that the old node from the *value* class should be recursively transformed into one from the *values* class. It also annotates the entry with a warning concerning the class change. A simple editing change, as shown in the bottom half of Figure 37, adds the list structure that is desired.

Generated Table Entry

```
production:  CASE-ELEM
    new self:  CASE-ELEM
    component 1:  Transform old component 1
        { Warning. Class has been modified  }
    component 2:  Transform old component 2
```

Table Entry After Implementor Modification

```
production:  CASE-ELEM
    new self:  CASE-ELEM
    component 1:  VALUES
        component 1:  Transform old component 1
    component 2:  Transform old component 2
```

Fig. 37. Table entry for embedding a value into a list before and after implementor modification.

The ARL conversion presents us with a second example of nonlocal restructuring. In this case, we wanted to merge multiple nodes into a single node. In the old version a *COMMENT* production was defined to be a single-line comment. In the new version a *COMMENT* was allowed to be multiple lines. This change was achieved by modifying the lexical routine attached to the *COMMENT* production. This transformation actually required no grammar changes. Therefore, the default transformation provides a 1-1 identity mapping between old comments and new comments, ignoring the new functionality supported by comments. The default entry is shown in the top of Figure 38.

While it would be syntactically legal to leave all existing comments as single-line comments, it would be inconvenient from the user's point of view. Therefore, we wanted to combine consecutive single-line comments in old databases into single multiple-line comments in the new database. This involved writing a transformation function to find the consecutive comments and combine them. The transformation table as modified by the implementor is shown in the bottom of Figure 38. If we are at the first comment, we create a *COMMENT* node whose value is the concatenation of the values of all consecutive comments, separated by new lines. When we try to transform the remaining comments, we want to replace them with nothing. This is achieved by specifying *nil* in the table. If *nil* is encountered when transforming a list child (as will be true for comments), it means that there should be no corresponding child in the new database. If it is encountered when transforming a fixed-arity component, it is equivalent to *META*.

Our experience with TransformGen has shown that TransformGen is a powerful tool for alleviating the problems of structure modifications. In the past, changes such as those made to the ARL grammar would have invalidated virtually every one of the hundreds of programs written using the earlier version. Now instead of apologetically informing our colleagues that they either must throw away their existing code or make do with the old inferior release, we can send them the new grammar together with the transformer produced by TransformGen.

Generated Table Entry

```
production:  COMMENT
    new self:  COMMENT with old value
```

Table Entry After Implementor Modification

```
production:  COMMENT
    if IsFirstComment() then
        new self:  COMMENT with value defined by MergeComments
    else
        new self:  nil
```

Fig. 38. Table entry for merging consecutive comments before and after implementor modification.

9. EVALUATION

We now reflect on some of the key design decisions. In many cases there are reasonable alternatives to the decisions we made: some of these were explicitly rejected; others set the stage for future extensions of TransformGen.

The Usefulness of an Environment. Fundamental to our approach is the idea that an implementor modifies a grammar in the context of an environment that can monitor those changes and incrementally build a transformer. In adopting this approach we explicitly rejected the alternative of attempting to infer the changes (made, say, with a text editor) by comparing the initial and final versions of the grammar. As we have illustrated, the ability to monitor and control the process of grammar modification provides many benefits that would be difficult to achieve with the alternative. First, the implementor can use the commands of TransformGen to disambiguate different interpretations of a change that would lead to quite different tree transformations. Second, the environment guides the implementor by providing a specific repertoire of commands. Third, the environment provides incremental feedback and consistency checking, so that errors are detected and reported early in the modification cycle.

Lazy Transformation. TransformGen adopts a lazy approach by transforming trees when they are first accessed by a modified environment. There are, however, other plausible times when old trees could be transformed into new ones. An eager approach might transform *all* trees as soon as the new environment has been formed (or even earlier, as the environment is being modified). We rejected this alternative as being both impractical and undesirable. It is impractical since each Gandalf tree is stored as a separate database. The database associated with a particular environment can be dispersed across many directories and machines. In practice it would be impossible to locate all the databases associated with a modified environment. It is undesirable since some users may prefer to use the old version of the environment: their trees should not be converted at all. On the other hand, using our approach we do run the risk that a transformation may encounter an error years after the changes have been made.

At the other extreme, a tree could be transformed incrementally as a user accesses nodes. This approach has the advantage that costs of conversion are amortized across the use of a database, rather than being paid up front on entry. This is particularly attractive for large databases where transformation time is not negligible. However, the extreme case of node-by-node transformation is impractical—since transformations need not be local—and costly—because of the additional bookkeeping overhead in time and space. On the other hand, as we described earlier, in practice Gandalf databases are naturally structured in "segments," each of which may be defined by a different grammar [Krueger et al. 1989]. Since transformations are typically local to a segment, and the bookkeeping overhead is small (relative to the size of the database), we can transform databases a segment at a time. (Interactions between segments cause no problem in this scheme because whenever a node within a segment is accessed the entire segment is transformed.)

An additional possibility that we have considered is to combine segmented transformation with opportunistic processing. Accessed segments would then be transformed as needed; however, the environment would process remaining segments in the background during slack periods of computation.

It is worth noting that some care must be taken to ensure that the transformation interacts appropriately with other database functions such as concurrent access and transactions. In particular, long-term transactions can be problematic if it is possible to change a grammar when some user is in the middle of a transaction.

Linear Ancestry. One of the basic assumptions that we made in the design of TransformGen was that environments evolve in a linear fashion. This was motivated by the typical use of Gandalf environments in which an implementor augments an environment that is shared by a group of software engineers. Given this assumption, updating of trees based on multiple environment revisions is relatively straightforward, since the transformers can simply be applied in succession.

It turns out, however, that there are cases in which the changes to an environment diverge as separate branches of a revision tree. In such cases a database that is transformed to be valid in one branch cannot be transformed into a grammar that is correct in another branch. To accomplish such a transformation would require the ability to *merge* the transformation tables from several sources. This appears to be a fruitful avenue for future research.

High-Level Commands. Our original goal was to develop a number of high-level commands for modifying grammars, together with the associated powerful transformation rules. However, we began with a collection of relatively low-level commands (itemized in Section 4). Surprisingly, these have been largely sufficient for most purposes. This has been true, in part, because the implementor can augment the transformation rules, and hence improve on the simple default transformations. But it may also be the case that a large repertoire of commands is simply not needed. On the other hand, we do feel the need for certain commands that would result in nonlocal restructur-

ing of the tree. In particular, it would be useful to have commands that would add or remove levels from a tree.

Another useful extension of TransformGen would be to improve the expressive capabilities of the transformation tables in order to reduce the need for writing transformation functions. The most promising extension would be to allow simple conditions to be written inline. For example, if this capability had been available when creating the ARL transformer, we would only have needed to write 2 of the 50 transformation functions, since the others were all simple conditionals.

Treatment of Errors. As discussed earlier, TransformGen allows the implementor to include transformation rules that may not be valid in all circumstances. This is typically done for two reasons. First, the implementor may know that certain syntactically valid trees will never occur in practice, and hence the transformation will succeed on any encountered tree. (This is possible since tools and other semantic-processing routines may prohibit certain syntactic constructions from an environment.) Second, certain information may best be obtained from the user of the environment, rather than the implementor. This might be the case for default values of some nodes, or in the case when information would otherwise be thrown away by the transformation.

Our experience with this decision is that transformation time errors rarely occur, and that the added flexibility is well worth the potential inconvenience for users of the environment. However, it may be that there are circumstances in which stronger guarantees should be made by TransformGen. For example, in a novice programming environment the transformation process should be completely transparent. A useful—albeit simple—extension of TransformGen would be to allow the implementor to set the level of static checking to exercise more control over the number and kind of transformation time interactions that could take place for a given set of transformation rules. Another extension would include a facility for exception handling, again an area for future research.

Semantic Processing. Currently TransformGen provides no assistance in modifying semantic processing and other tool functionality associated with a changed environment. If, for example, an implementor deletes a component from a production in a grammar, then any existing routine that attempts to access that component will be in error. It is up to the implementor to discover this fact and change such routines appropriately. We view this as an important limitation in the current approach.

It may be possible to extend TransformGen to provide some form of semantic assistance. While no transformer can completely automate the propagation of syntactic modifications into associated semantic changes, it may be possible for the transformation environment to warn the implementor about inconsistencies introduced by a grammar change. The ability to do this relies on the fact that semantic processing and other tool functionality is described in a notation that makes its dependence on structure explicit. Semantic routines written in ARL typically use class and production names

to navigate through the database. By analyzing such expressions, TransformGen could identify at least some uses of modified productions and classes. Attribute equations used by other environment generation systems also have this property.

10. RELATED WORK

While there has been little research devoted to solving the problems of maintaining structure-oriented environments, there are two areas of related work to discuss. The first deals with the tree transformations used by compilers. The second area of interest is the transformations supported by other types of databases.

10.1 Tree Transformation in Compilers

Compilation can be thought of as a series of transformations on languages, where each transformation typically augments the previous with a set of attribute-value pairs at each node. Attribute grammars have been used to describe the transformations that modify the syntactic form of an abstract tree and thereby convert a tree defined by one context-free grammar to a tree defined by another [Drossopoulou et al. 1982; Keller et al. 1984; Leverett et al. 1980; Lewis et al. 1984; Monke et al. 1984].

These approaches to tree transformations differ from our work on TransformGen in two important ways. First, our focus has been on *automating* the definition of a transformer. Thus automatic composition and coverage have been important considerations in our work. In contrast, compiler techniques require an implementor to define a set of pattern-matching associates and correspondences between input and output grammars. The implementor must incorporate and integrate all changes by hand, usually with complete and detailed knowledge of the grammar definitions. This manual approach may be reasonable in the context of building a compiler, which must be carefully crafted to produce both efficient and semantically correct grammar transformations. However, it is impractical in our setting in which an implementor is encouraged to make numerous modifications to program grammars while developing a programming environment.

The second major difference between our work and compiler techniques is the emphasis on environment support. Given that we provide an environment for the implementor to use in the generation of programming environments, we believe it is essential that we also provide support within the environment for changing environment descriptions.

10.2 Transformation of Databases

Databases have existed for a long time, and so also has the need for database transformation. TransformGen can therefore be reasonably compared to similar work in the transformation of relational databases, knowledge bases, and object-oriented databases.

Sockut and Goldberg [1979] survey relational database systems from the perspective of the types of reorganization they support. Our notion of trans-

formation correspond to two levels of reorganization in their taxonomy: *infological reorganization* and *string reorganization*. Infological reorganization involves changes to schema. String reorganization involves changes to relationship membership. For example, adding an attribute to a schema is an example of infological reorganization. Changing a relationship from 1-1 to 1-many is a string reorganization. The systems surveyed typically offer support for a few simple changes, such as adding and deleting attributes. Most systems allow the database implementor to write custom programs on an ad hoc basis to perform more interesting transformations. While they allow implementor extensions they do not provide any real support for the development of those extensions as we do.

Balzer [1985] has addressed database transformation in the context of knowledge representation systems. His system provides a language for making structural changes to the description of a knowledge representation system (or "domain model"). It also provides tools for mapping those changes into corresponding transformations on existing data. Balzer uses an eager transformation approach. As soon as a change occurs, his system makes the appropriate modifications to the knowledge base. Unlike his approach, TransformGen allows multiple changes to be composed into one transformation pass of the database. Also, TransformGen can potentially incorporate a much wider range of high-level changes than the fixed repertoire provided by Balzer's system. Additionally, TransformGen contains hooks for arbitrary implementor extensions. Finally, since TransformGen performs lazy transformation, it does not require the overhead of maintaining links between a production and all of its instances as Balzer's system does.

One final class of databases to consider is object-oriented databases. Orion [Banerjee et al. 1987], GemStone [Penney and Stein 1987], and O_2 [Barbedette 1991] have taken a fairly traditional approach to the role of database transformation. When a class definition is changed, objects of that class need to be transformed to correspond to the new definition. These systems provide automation for certain types of changes, such as deletion of instance variables from a class. Of these, O_2 is the only system to support extension of the transformation process to more interesting types of changes, such as moving an instance variable from one class to another. O_2 also analyzes changes with respect to their effects on methods. In particular, it identifies methods that reference deleted classes or attributes/methods of classes, methods that must be type-checked following a change, and methods that are still type-safe but whose behavior may have changed due to changes to methods they call. OTGen [Lerner and Haberman 1990] is a system we designed based on the concepts developed in TransformGen to support flexible transformation of object-oriented databases. As such it has many of the features of TransformGen, but is aimed at a different class of databases.

Another approach to evolution of object-oriented databases relies on the simultaneous maintenance of multiple versions of classes and objects. This approach was first proposed by Skarra and Zdonik [1986]. Rather than transform objects whose class definition has changed, they support multiple versions of the same class within a single database. The advantage of this

approach is that the database does not need to be transformed, and existing code can operate on existing objects without being changed. Each class has an interface that is a union of the interfaces of all the versions of that class. Compile-time type-checking is performed against this union. However, any individual object will represent only some subset of the interface, defined by the particular class version it is an instance of. Therefore, they allow a programmar to associate error handlers with a class to handle application of operations to objects of the wrong version. In effect, some of the type-checking is deferred until run-time, since an object's version cannot be known until then. Bratsberg [1992] uses views to support class evolution. In his model, a class consists of an intent (its definition) and an extent (the set of objects). Evolution is modeled by allowing an object to belong to more than one extent. To be a member of more than one extent, there must be attribute and operation consistency relations to map between the corresponding intents. These multiversioning approaches are more flexible than the one proposed here, since they allow the coexistence of both old and new data and tools. On the other hand this flexibility comes at a cost: it requires the additional overhead (in both space and time) for maintaining and accessing multiple class and object versions. Furthermore, there is little support for automatic development of functions to maintain consistency between versions of objects.

One final comparison should be made between all the database systems described in this section and our structured databases. With traditional databases, there is typically a 1-1 mapping between a database description and a database. Therefore, it is reasonable to require a database administrator to take responsibility for transforming the database and handle any errors that occur during transformation. As databases become larger and more distributed, this approach may no longer be feasible. With our structured databases, there is typically a 1-many mapping between a database description and databases. There can easily be thousands of databases for a single database description, and they can be located at many sites. Without a metadatabase to keep track of all these databases, it is not possible for a database adminstrator to perform all the transformations and address the problems that might arise. Instead we must provide an environment that supports the development of robust transformers that can handle problems directly rather than require the user to become involved in the transformation process. We believe that TransformGen provides the support necessary to build such robust transformers.

11. CONCLUSION

While the techniques described in this article were developed specifically to solve problems of grammar evolution for structure-oriented environments, many of the results carry over to other systems. In particular, our experience indicates that there are three essential ingredients to a successful approach to maintenance based on structural transformation. First, the objects to be transformed must be represented in a structured form as described by some formal notation. Second, it is important to provide automated support for the

development of the transformers. This support must be able to translate type changes to consequent actions that can be performed during the object transformation process. Third, it must be possible for the person who is making the change to augment the automatic mechanisms to handle special cases.

The results of this approach, at least within the domain of structure-oriented environments, have been encouraging. We have been able to make substantial improvements to existing environments that would have been infeasible using the manual, ad hoc techniques available before TransformGen. The generator for structural transformations is a powerful tool that can be built relatively easily by extending existing environment generators. Having long promoted the use of structure-oriented environments for rapid prototyping, we can now do so with the confidence that these environments can also be maintained.

ACKNOWLEDGMENTS

We are indebted to many people who helped to shape the ideas in this work. First and foremost, Nico Habermann provided guidance and inspiration in all our research endeavors, including this one. Benjamin Pierce and Mark Tucker contributed substantially to the design of TransformGen and provided valuable feedback throughout. Robert Stockton and Benjamin Pierce played a central role in turning the design into a working system. We would also like to thank those who provided constructive criticism of earlier versions of this article: Roberto Bisiani, Stewart Clamen, Lori Clarke, Richard Green, Nico Habermann, Richard Lerner, Anund Lie, David Notkin, Benjamin Pierce, Robert Stockton, John Wenn, and the anonymous referees.

REFERENCES

AMBRIOLA, V., KAISER, G. E., AND ELLISON, R. J. 1984. An action routine module for ALOE. Tech. Rep. CMU-CS-84-156, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, Pa.

BALZER, R. 1985. Automated enhancement of knowledge representations. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 203–207.

BANERJEE, J., KIM, W., KIM, H.-J., AND KORTH, H. F. 1987. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*. ACM, New York, 311–322.

BARBEDETTE, G. 1991. Schema modifications in the $LISPO_2$ persistent object-oriented language. In *Proceedings of ECOOP'91, the 5th European Conference on Object-Oriented Programming* (Geneva, Switzerland, July).

BORRAS, P., CLEMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. 1988. CENTAUR: The system. In *SIGSOFT 88: 3rd Symposium on Software Development Environments*. ACM, New York, 14–24.

BRATSBERG, S. E. 1992. Unified class evolution by object-oriented views. In *Proceedings of the 11th International Conference on the Entity-Relationship Approach*. Lecture Notes in Computer Science, vol. 645. Springer-Verlag, Berlin, 423–439.

DROSSOPOULOU, S., UHL, J., PERSCH, G., GOOS, G., DAUSMANN, M., AND WINTERSTEIN, G. 1982. An attribute grammar for Ada. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*. ACM, New York, 334–349. Also *SIGPLAN Not. 17*, 6.

GARLAN, D. B. AND MILLER, P. L. 1984. GNOME: An introductory programming environment based on a family of structure editors. In *Proceedings of the Software Engineering Symposium*

on Practical Software Development Environments. ACM-SIGSOFT/SIGPLAN, New York, 65–72.

HABERMANN, A. N., GARLAN, D., AND NOTKIN, D. 1991 Generation of integrated task-specific software environments. In *CMU Computer Science: A 25th Anniversary Commemorative.* ACM Press, New York.

HABERMANN, A. N. AND NOTKIN, D. S. 1986. Gandalf: Software development environments. *IEEE Trans. Softw. Eng. SE-12,* 12 (Dec.), 1117–1127.

KELLER, S. E., PERKINS, J. A., PAYTON, T. F., AND MARDINLY, S. P. 1984. Tree transformation techniques and experiences. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction.* ACM, New York, 190–201. Also *SIGPLAN Not. 19,* 6.

KRUEGER, C. W., STAUDT, B. J., AND HABERMANN, A. N. 1989 Scaling up integrated software development environment databases. In *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases.* ACM, New York, 74–78.

LERNER, B. S. AND HABERMAN, A. N. 1990 Beyond schema evolution to database reorganization. In *Proceedings of OOPSLA'90* ACM, New York, 67–76.

LEVERETT, B. W., CATTELL, R. G., HOBBS, S. O., NEWCOMER, J. M., REINER, A. H., SCHATZ, B. R., AND WULF, W. A. 1980. An overview of the production-quality compiler-compiler project. *IEEE Comput. 13,* 8 (Aug.), 38–49.

LEWIS, P. M., ROSENKRANTZ, D J., AND STEARNS, R. E. 1984. Attributed translations. *Comput. Syst. Sci. 9,* 3 (Dec.), 279–307.

MONKE, U., WEISBERGER, B., AND WILHELM. R. 1984. How to implement a system for manipulation of attributed trees. In *GI-Workshop on Programming Languages and Program Development.* (Zurich, Mar.).

NIX, R. P. 1985. Editing by example. *ACM Trans. Program. Lang. Syst. 7,* 4 (Oct.), 600–621.

PENNEY, D. J. AND STEIN, J. 1987. Class modification in the GenStone object-oriented DBMS. In *Proceedings of OOPSLA'87.* ACM, New York, 111–117.

REISS, S. P. 1984. Graphical program development with PECAN program development systems. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments.* ACM-SIGSOFT/SIGPLAN, New York, 30–41.

REPS, T. AND TEITLEBAUM, T. 1984. The synthesizer generator. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments.* ACM-SIGSOFT/SIGPLAN, New York, 42–48.

SKARRA, A. H. AND ZDONIK, S. B. 1986 The management of changing types in an object-oriented database. In *Proceedings of OOPSLA'86.* ACM, New York, 483–493. *Published as SIGPLAN Not. 21,* 11.

SOCKUT, G. H. AND GOLDBERG, R. P. 1979. Database reorganization—Principles and practice. *ACM Comput. Surv. 11,* 4 (Dec.), 371–395.

STAUDT, B. J. AND AMBRIOLA, V. 1986. The ALOE action routine language manual. In *The GANDALF System Reference Manuals.* Tech. Rep., Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.

STAUDT, B., KRUEGER, C., AND GARLAN, D. 1988. TransformGen: Automating the maintenance of structure-oriented environments. Tech. Rep. CMU-CS-88-186, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa

Received April 1987; revised November 1988 and June 1991; accepted June 1992