A Structural Approach to the Maintenance of Structure-Oriented Environments

David Garlan
Charles W. Krueger
Barbara J. Staudt

Carnegie-Mellon University
Department of Computer Science
Pittsburgh, PA 15213

Abstract

A serious problem for programming environments and operating systems is that existing software becomes invalid when the environment or operating system is replaced by a new release. Unfortunately, there has been no systematic treatment of the problem; current approaches are manual, ad hoc, and time consuming both for implementors of programs and for their users. In this paper we present a new 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 formal descriptions of these environments.

1. Introduction

A serious problem for programming environments and operating systems is that existing software becomes invalid when the environment or operating system is replaced by a new release. (In the widespread conversion from Version 4.1 of BSD Unix™ to Version 4.2, for example, any program that made use of the file directory structure became obsolete.) At the very least, work must partially halt while conversion takes place: programs are modified and recompiled and old data representations are converted to new. Users are burdened with a period of instability and loss of functionality for inadequate conversions. Implementors of programs are burdened with the tasks of locating all programs to be changed and then, typically, manually modifying old code on a case-by-case basis. Consequently, users and implementors are faced with a dilemma: stability can be achieved by ignoring successive releases, in which case the environment will not meet the evolving needs of its users; or change can be allowed, at the cost of a time consuming process of conversion.

Structure-oriented environments,[2] such as [8, 4, 9, 10], are a class of programming environment for which these problems are particularly severe. Structure-oriented environments are usually generated from a formal description that is processed and linked with a collection of common facilities. The formal description, or grammar, is typically a variant of BNF. It characterizes the form of programs, which are represented as abstract syntax trees. The common facilities typically support the creation, modification, and storage of programs in the running environment. When the grammar of a structure-oriented environment is changed in any but trivial ways, existing trees representing valid abstract syntax under the old grammar may not be correct under the new grammar. At early stages of environment prototyping it may be possible to simply discard the old trees. However, in practical settings where users have come to depend on the programs created in the old environment, a sudden announcement that all existing trees are no longer valid will not be greeted with enthusiasm.

For most operating systems and programming environments the situation is mollified by the fact that major evolutionary changes may be rare and may

Research on Gandalf is supported in part by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ. and in part by ZTI-SOF of Siemens Corporation, Munich, Germany.

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

affect only a small number of programs. For structure-oriented environments, however, changes tend to be much more frequent and, as we have mentioned, affect virtually all existing program trees. Changes occur more frequently because users wish to take advantage of the ability to easily generate new environments from formal descriptions. Improvements to the environment - either through the introduction of new tools or the enhancement of existing tools - usually require changes to the grammar that characterizes program trees. (For some examples see [5].)

There has been virtually no research on systematic solutions to these problems. This is as true for general programming environments as it is for structure-oriented environments. Current approaches are either blatantly ad hoc, or are based on the idea of reparsing textual representations of existing program trees. While the latter 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 paper we present a new 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[3] *changes* to the formal descriptions 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 formal description of a structure-oriented environment. The output of TransformGen is a new grammar together with a *transformer*, which takes instances of trees built under the old grammar and automatically converts them to instances of trees that are legal under the new grammar. We then discuss the practical effect this approach has had in coping with changes to a large structure editor environment in use at a number of educational and industrial sites. In conclusion we briefly indicate how these techniques might be applied to solve similar problems for programming environments and systems in general.

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

2. The Transformational Approach

To illustrate the problems associated with changing the grammar that describes a structure-oriented environment, consider Figure 2-1. Here a MODULE is described as an entity having as one of its components a single IMPLEMENTATION. Suppose that after having used an environment generated from this description, we decide to enhance the system by allowing a module to have a collection of implementations, each implementation distinguished by a version number. The grammar might then look as pictured in Figure 2-2.

```
MODULE ::= MOD-NAME INTERFACE IMPLEMENTATION
           attribute is-used: boolean

MOD-NAME ::= identifier

INTERFACE ::= list of EXPORT-ITEM

IMPLEMENTATION ::= ...
...
```

A typical grammar consists of a collection of BNF-like productions that specifies the abstract syntax of a programming environment. Here, for example, MODULE has three components whose types are given by the productions MOD-NAME, INTERFACE, and IMPLEMENTATION. Productions may have associated attributes such as the 'is-used' attribute of MODULE.

Figure 2-1: Grammar for a Module Description Environment - Version 1

Any stored instance of a MODULE constructed from the environment based on the old grammar will now be obsolete since the new environment expects VERSIONS in the place that the old environment has an IMPLEMENTATION. What is needed is a way to transform old instances of MODULEs into the new format. In the example of Figures 2-1 and 2-2 we would like to add an additional level of structure to the grammar to represent the module's sequence of versions. To transform existing trees we would insert the old IMPLEMENTATION of a MODULE as the first component of a sequence of VERSIONS, perhaps giving it a default VERSION-NUMBER in the process. Additionally, we would move the old 'is-used' attribute of the old MODULE to be an attribute of the new IMPLEMENTATION.

```
MODULE ::= MOD-NAME INTERFACE VERSIONS

VERSIONS ::= list of VERSION

VERSION ::= VERSION-NUMBER IMPLEMENTATION

VERSION-NUMBER ::= integer

IMPLEMENTATION ::= ....
                   attribute is-used: boolean
....
```

In this version of the grammar the description of MODULE has been changed to allow it to have multiple versions. Each version has an associated version number. The old 'is-used' attribute of MODULE is now an attribute of the IMPLEMENTATION.

Figure 2-2: Grammar for a Module Description Environment - Version 2

One common approach to the problem is illustrated by the three stage process pictured in Figure 2-3. 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 reparsed into a new tree by a parser that can convert text to abstract syntax trees that are valid under the new grammar.

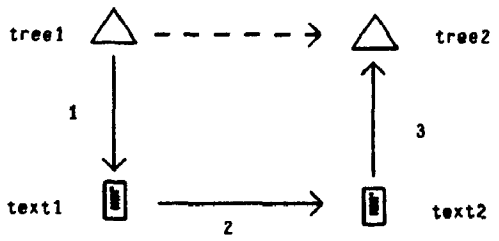

Figure 2-3: The Transformation Process

There are a number of problems with this approach:

- A parser must be built. This voids one of the primary attractions of structure-oriented environments, namely that they can be built without the overhead of producing a parser.
- A substantial manual effort may be involved in transforming text. While some progress has been made in techniques for producing automatic transformers for text [7], currently these techniques are not powerful enough to handle the range of transformations needed in this context.
- Information may be lost. This typically occurs when attributes are used to store information that cannot be directly regenerated from a canonical textual representation. For example, in systems such as [4, 1, 3] an attribute can be used to store such things as a "change log".
- Conversion is ad hoc. There is no direct, enforceable correspondence between the changes made to a grammar and changes that occur in the translation process.

Our alternative approach is illustrated by the dotted line in Figure 2-3. In this approach existing trees produced by the old environment are directly converted to trees that are valid in the new environment. 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 (a) to generate such a tree-to-tree transformer automatically or semi-automatically, and (b) to augment the automatic transformation methods to take care of special cases. The first requirement is necessary to reduce the cost of producing such a transformer, and the second because there are classes of transformations that cannot be automatically generated or that may not be handled efficiently by automatic techniques.

In the following sections we will show how these two requirements can be met. The basic idea is that we provide an environment (called *TransformGen* for "Transformer Generator") in which an implementor makes *structured* changes to an existing grammar. Any number of changes can be made and in any order. When all the changes have been made, TransformGen produces a table of transformation rules that can be interpreted to convert old representations to new. The implementor can then augment the automatic mechanisms in three ways: by directly modifying the tables, which are written in a human-readable format, by writing transformation routines to perform tricky transformations, or by adding a set of *action routines* that are invoked

in a post-processing phase when the transformer has completed its work.

While little has been written about the maintenance of structure-oriented environments, the approach described here most closely resembles the work of Balzer dealing with the maintenance of knowledge representation systems [2]. However, unlike Balzer's work, TransformGen allows multiple changes to be composed into one transformation pass of the database, deals with the more tightly constrained type systems of structure-oriented environments, can potentially incorporate a much wider range of high-level changes, contains hooks for arbitrary implementor extensions, and does not require the overhead of associating with each production type a set of pointers to all instances of that production.

3. Requirements for a Tree Transformer

The approach outlined above requires the ability to interpret changes to a grammar in terms of the effects of these changes on existing trees. Some changes are trivial to interpret. For example, if the implementor adds a new production to a grammar, no transformation is required because existing trees are not affected. Only slightly more complex is a grammar change that adds a new component to a non-terminal production. The required transformation must map components of the old production to the appropriate locations in the new production, leaving an empty placeholder for the new component.

In the general case, grammar changes are not so easy to interpret. 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. 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 those 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 appropriate transformations for a type of change in the environment description. In general it is not possible for a transformer generator to provide complete coverage. This is true for two reasons. First, given any grammar change, there may be several possible ways in which to transform existing trees so that they are valid in the modified grammar. Choosing the correct interpretation requires the ability to infer the implementor's intentions. As a simple example, suppose a production goes through the following sequence of changes:

```
(1) X ::= Y Z      -- the original form

(2) X ::= Z        -- implementor deletes
                      first component

(3) X ::= Z Y      -- implementor inserts
                      new second component
```

Is this sequence of operations really a deletion-addition pair or does it represent a single reorder operation? The choice will have a large effect on the type of transformation generated. Second, a transformer generator can only capture modifications that are made to the grammar itself. If semantic processing is implemented by procedures outside the grammar (such as daemons), changes to the semantics cannot be recognized by the transformer generator.

Since complete automatic coverage is not possible, it is essential to allow an implementor to supplement the automatic mechanism to handle complex transformations and transformations associated with semantic processing.

TransformGen is capable of automatically handling the following types of grammar changes.

- Addition or deletion of productions
- Renaming of productions
- Addition or deletion of components or attributes of a production
- Reordering productions, or components or attributes of a production
- Changing the type of the value associated with a terminal node
- Changing nonterminals into terminals

- Changing terminals into nonterminals

An implementor can supplement the automatic mechanisms to deal with more complicated changes such as:

- Addition of structural levels
- Deletion of structural levels
- Changing semantic processing
- Complex tree restructuring.

As complex transformations become well understood, they can be incrementally added to the automatic repertoire of TransformGen.

4. The Generation of a Transformer

There are two major steps involved in the transformation process. First, a transformer is generated based on grammar changes made by an implementor. Second, the transformer is applied to existing trees created with the old version of the grammar in order to update them to the new version. This section describes the process of generating a transformer using the example introduced in Section 2. In order to understand this process, the basic algorithm used by a transformer is described first.

The transformation of existing trees involves a top-down construction of a new tree. The contents of the new tree are derived from the contents of the old tree by applying derivation rules encoded in the transformer. These derivation rules indicate how to translate each production in the old grammar into a production in the new grammar. An individual rule indicates what production to use to construct a new node, and how to recursively build each component and attribute of the new node.[4] The choice of rule to apply is determined primarily by the production used to construct the node in the old tree. Thus as each node in the old tree is being transformed, the rule base is consulted in order to perform the correct construction in the new tree.

[4]Since attributes are described via attribute grammars, the same transformation mechanism can be applied to transform attribute trees.

4.1. Transformation Tables

The editing commands of TransformGen not only modify the grammar specification, but also generate a tree transformation table consisting of derivation rules that reflect grammar changes. The table contains one or more entries for each production in the original grammar. Each entry consists of three parts: a transformation on the production operator itself, a transformation on each component of the production, and a transformation on each attribute of the production. For instance a production that has not been modified would have an entry as shown in Figure 4-1.

```
production operator: OP1
  new self: OP1
    component 1: Transform old component 1
    component 2: Transform old component 2
    ...
    attribute a: Transform old attribute a
    attribute b: Transform old attribute b
    ...
```

Figure 4-1: Table Entry for an Unmodified Production

This would have the effect of translating every instance of OP1 to itself.

As in Figure 4-1, transformations often simply specify a production in the new grammar and a recursive transformation for each old component and attribute. However in order to provide transformations that add structural levels to a tree, it is necessary to specify a more complex subtree to build. Consider the grammar changes between Figures 2-1 and 4-2.

```
MODULE     ::= MOD-NAME INTERFACE VERSIONS

VERSIONS   ::= list of IMPLEMENTATION

IMPLEMENTATION ::= ...
                   attribute is-used: boolean
    ......
```

Figure 4-2: Grammar for a Module Description Environment - Intermediate Version

These grammar changes require the addition of a structural level to existing trees allowing for the list

of IMPLEMENTATIONs. The transformation table entry is shown in Figure 4-3.

```
production operator: MODULE
  new self: MODULE
    component 1: Transform old component 1
    component 2: Transform old component 2
    component 3: VERSIONS  -- Use the new
                              VERSIONS production
      element 1: Transform old component 3
                 -- Old IMPLEMENTATION node
        attribute is-used: Transform old
                           attribute is-used
```

Figure 4-3: Adding a Structural Level to a Production

This entry will transform every old MODULE into a new MODULE. The MOD-NAME and INTERFACE are placed in the same locations as in the old tree. The third component of a new MODULE gets its value by building a VERSIONS node and making the IMPLEMENTATION of the old MODULE be the first element of the VERSIONS list. Note also that the *is–used* attribute now becomes an attribute of the IMPLEMENTATION rather than the MODULE.

```
production operator: MODULE
  new self:  if oldnode.is-used then MODULE
                -- Checks that the module is used
    component 1: Transform old component 1
    component 2: Transform old component 2
    component 3: VERSIONS
      element 1: Transform old component 3
        attribute is-used:  Transform old
                            attribute is-used
    endif
```

Figure 4-4: Conditional Transformation

In some cases the implementor will not want every instance of a production to be transformed into the same new production. To allow for this possibility, a condition may be attached to each transformation table entry. The condition might be used to compare the value of a terminal node to a specific string, make queries concerning other nodes in the original tree, or test attribute values. For instance suppose in the example of converting MODULEs to contain multiple IMPLEMENTATIONs we do not want to waste time transforming MODULEs that are not used. Then we only want to do the conversion when the *is–used* attribute of a MODULE is true. This transformation table entry is shown in Figure 4-4.

4.2. Composability

The composability problem described in Section 3 is overcome by encoding transformation table entries in such a way that grammar changes can be easily combined. As a grammar is being modified there will typically be a sequence of changes that can be made in a somewhat arbitrary order. At each intermediate step during the grammar modification the transformation table must also be updated appropriately. However, the final transformation table should be the same, independent of the order in which the grammar changes were made. To better understand the problem and solution it is useful to think of the grammar modification process as beginning 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. The current state is any intermediate version of the grammar. The goal state is the new version of the grammar. The key to the solution is in recognizing that the implementor is always editing an intermediate version of the grammar. The productions of this intermediate grammar version are always encoded in the current state of the transformation table. Therefore whenever a change is made to a grammar production, the transformation entries must be searched for all occurrences of the production operator name. The effects of the production modification must be replicated at each occurrence of the production operator name.[5] Note that as in Figure 4-4 a production operator name may occur within a component description. By replicating the effects we are ensuring that the actual transformation of existing trees can be performed in a single pass.

[5] It is important to emphasize that a transformation entry consists of a rule to transform an operator, a set of rules to transform the components, and a set of rules to transform the attributes of the production. The production operator name from the old grammar is specifically not a part of the entry. It is simply the index into the transformation table. The result is that the old production operator names never change. Only the entries associated with those names are modified.

Consider the grammar changes between Figures 4-2 and 2-2. The associated change to the transformation table must compose with the table shown in Figure 4-4. The grammar change requires that we change our current understanding of VERSIONS. To do this we search the transformation table for all occurrences of the VERSIONS production operator. We want to modify the first element to be a VERSION production. We also want to give a default value of 1 to the VERSION-NUMBER component. Now the transformation entry for a MODULE is as shown in Figure 4-5.

```
production operator: MODULE
  new self:  if oldnode.is-used then MODULE
    component 1: Transform old component 1
    component 2: Transform old component 2
    component 3: VERSIONS
      element 1: VERSION  -- Use the new
                             VERSION production
        component 1: VERSION-NUMBER with
                     value = 1
                         -- Use a default value of 1
        component 2: Transform old
                     component 3
          attribute is-used: Transform old
                             attribute is-used
```

Figure 4-5: Composition of Modifications to Related Productions

4.3. Coverage

Complete coverage of grammar editing requires the assistance of the implementor. This assistance can come in three forms: editing the transformation table directly, writing transformation routines that are invoked during tree transformation, and writing action routines that are called during a postprocessing phase. Direct editing of the transformation table is most useful for grammar changes that involve moving information to a different level of the tree, supplying values for terminal nodes and attributes, etc. These types of changes are expected to be common, but difficult to describe in a way that the transformer generator can understand. By editing the tables directly the implementor will still receive the benefits of composition with later grammar changes.

The major limitation of the transformation table is that all constructions in the new tree must be statically expressed as constant production names. If a transformation cannot be expressed in terms of boolean conditions and constant productions names in the table, then a more powerful mechanism must be invoked to assist in the transformation. This extra power is needed for cases where, for example, the user must be asked to interactively choose how to transform a node, or where the information needed to select a production to construct is dependent on parts of the new tree that have not yet been constructed. Such transformations can be described in ARL, an imperative tree-oriented programming language [11]. This language is capable of examining both the old and new trees and querying the user for advice. It is thus capable of performing any conceivable tree transformation. The disadvantage of a procedural approach is that composability can no longer be guaranteed. It is therefore the implementor's responsibility to ensure that the transformation routines correctly map from the start state to the goal state, taking into account all editing changes that affect the node in question. It is possible to recursively call the transformer from a transformation routine. Thus a tricky transformation can be applied to a node, and then its components can be constructed using the table as before. The chief advantage of action routines invoked during a postprocessing phase is that they have access to the completely transformed tree, and thus can make changes to the tree based on its transformed state. This should be most useful for recomputing attributes, such as symbol tables, that depend directly on the tree.

5. Current Implementation

TransformGen has been implemented as an extension of the Gandalf structure-oriented environment generator system [8]. As it turned out, it has been relatively easy to build the necessary tools as simple extensions to existing facilities. The implementation consists of two main components. The first is the TransformGen environment itself. The second is a grammar-independent transformation engine that is linked with the generated transformation tables to produce a tree transformer for a particular set of grammar changes.

Within the TransformGen environment the implementor updates a grammar by making structural

changes to an existing grammar. In fact, TransformGen is simply an extension to the Gandalf System grammar editor AloeGen [6] that is used for constructing grammars from scratch. The implementation of TransformGen has changed AloeGen from a simple structured editor into a knowledge-based editor. In AloeGen an editor command resulted in a simple change to the grammar maintained by AloeGen. In TransformGen an editor command results not only in the same simple grammar change, but also in changes to the transformation table.

For the most part, the user of TransformGen does not need to be aware that a transformer is being constructed. The user interface to TransformGen is virtually identical to the user interface for AloeGen. In fact, as long as the user applies commands for which TransformGen knows how to build the tables, the user interfaces are identical. If the user applies a command which TransformGen cannot fully interpret, the user is warned of the need to manually edit the transformation tables before proceeding with the change. In order to manually edit the transformation tables, the user must change from the normal AloeGen grammar editing view to the TransformGen view. In this view, the transformation tables that have been constructed by TransformGen are presented to the user. The formatting of those tables is exactly as shown throughout this paper. The user is now free to manually edit those tables. While the user is performing this editing, TransformGen watches over the user's shoulder and prevents him from performing changes which are inconsistent with the grammar. For instance, TransformGen ensures that the correct number of children are supplied for each nonterminal, that each operator has the proper attributes, etc.

The transformation engine for a particular set of grammar changes resembles a typical structure editor that might have been produced by any structure-oriented environment generator. However, there are two important differences. First the "editor" must simultaneously access two grammars,[6] one describing the old trees and the other describing the ones to be built. Second, it must know how to interpret a transformation table. When an old tree is to be transformed it is first read using the old grammar. Then, as described in Section 4, a new tree is constructed in a top-down fashion using information from the old tree, the transformation table, and the new grammar. The process is started by consulting the transformation table entry for the root operator of the old tree to determine the root operator in the new tree. The component description in that table entry contains the information that recursively invokes the transformer for each component of the new root. Finally, if necessary, it performs a post processing walk of the new tree, invoking any implementor-supplied routines that augment the transformation tables.

In our implementation, existing trees are transformed only as needed. Each tree is tagged with a version stamp that indicates the grammar version with which it was produced. When a user attempts to access a tree, the environment checks that the version stamp is consistent with the current version of the grammar. If not, the appropriate transformer is invoked to bring the tree up-to-date. If several versions of the grammar have been released since a tree was last read then the transformation proceeds incrementally from one version to the next until it becomes current.

6. Evaluation

The current implementation has been used to make a substantial number of changes to the grammar of ARL, the language we use for writing semantic routines [1]. The grammar for the environment is a large one, containing over 150 productions, and is currently being used in research projects at a number of academic and industrial institutions around the world. Using TransformGen we have been able to generate transformation tables for 67 grammar changes to ARL. It took a single day to generate the tables. One hour was spent making changes to the ARL grammar. TransformGen generated the proper tables for 48 of the 67 changes made. It was also necessary to write 30 routines to assist in the transformation. The vast majority of these routines were extremely simple. Over 20 of them were boolean functions that only needed to examine a single node in the tree.

[6] Actually, there may be more than two grammars since attributes may have their own grammars as well.

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

- Replacement of a production with a set of productions.
- Addition of structural levels to the tree.
- Relocating pieces of the tree to new locations.
- Merging of multiple nodes into a single node.

It is useful to consider each of these in more detail to understand why they failed. 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 tree when it encounters production A in the old tree. The decision is likely to depend on some part of the state of the tree, which is what happened in this case. As a result it was necessary to write transformation routines to examine the state and decide on a particular production to use.

Adding structural levels to the tree and relocating pieces of the tree are more difficult problems. The difficulty is that arbitrary changes are being made to the *structure* of the tree, while the *contents* of the tree should remain unchanged. This type of change is easy to implement by directly editing the transformation table. Unfortunately, we do not yet know how to express such modifications in a way that is easy both for the person modifying the grammar to understand and for TransformGen to interpret unambiguously. We expect that with experience using TransformGen this type of change will become understood well enough that we can automate it in the future.

Merging multiple nodes into a single node is really a special case of the relocation problem discussed above. In the old version a COMMENT operator was defined to be a single line comment. In the new version a COMMENT was allowed to be multiple lines. The goal was to combine consecutive single-line comments in old trees into single multiple-line comments in the new tree. This involved writing a transformation routine to find the consecutive comments and combine them.

Initial experiments using TransformGen indicate that it 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 must either 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.

7. From Syntax to Semantics

The transformations performed by TransformGen are strictly structural. That is, the system currently provides no assistance in modifying the semantic processing and other tool functionality associated with a changed environment. If, for example, an implementor deletes a son from a production in a grammar, then any existing semantic processing routine that attempts to access that son will be in error. Currently it is up to the implementor to discover this fact and change such routines appropriately.[7]

We are investigating the possibility of extending TransformGen to provide some form of semantic assistance. While no transformer can completely automate the propagation of structural modifications into associated semantic changes, it may be possible for the transformation environment to warn the implementor of potential and actual 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 dependency on logical structure explicit. Daemons written in a high level tree manipulation language (such as ours [1]) and attribute grammars (such as the Synthesizer Generator's [10]) both have this property. The extent to which we will be able to provide automatic support for semantic changes, however, remains an open issue.

[7] By using the Gandalf programming environment and symbolic addresses to navigate through trees in the semantic routines, it is easy to find all the uses of a production. Modification is still up to the implementor, however.

8. Conclusion

While the techniques described in this paper 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 would indicate 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 an environment in which monitored changes can be made to this notation. This environment must be able to translate these changes to consequent actions that can be performed during the object transformation process. Third, any resulting transformation scheme must be extensible along two dimensions: it must be possible to augment the repertoire of transformations automatically handled by the transformer as new classes of transformation become better understood, and it must be possible for the person who is making the changes to augment the automatic mechanisms with routines for handling 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.

Acknowledgements

We are greatly indebted to many people who helped to shape the ideas that have emerged in this work. In particular, Benjamin Pierce, Mark Tucker, and Nico Habermann contributed substantially to the design of TransformGen and provided valuable feedback throughout. We would also like to thank those who provided constructive criticism of earlier versions of this paper: John Wenn, Richard Lerner, Nico Habermann and Benjamin Pierce.

References

[1] Vincenzo Ambriola, Gail E. Kaiser, and Robert J. Ellison.
An Action Routine Model for ALOE.
Technical Report CMU-CS-84-156, Carnegie-Mellon University, Computer Science Department, August, 1984.

[2] Robert Balzer.
Automated Enhancement of Knowledge Representations.
In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 203-207. 1985.

[3] Ravinder Chandhok, David Garlan, Dennis Goldenson, Mark Tucker, and Phillip Miller.
Structure Editing-Based Programming Environments: The GNOME Approach.
In *Proceedings of NCC85.* AFIPS, July, 1985.

[4] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang.
Programming Environments Based on Structured Editors: The Mentor Experience.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[5] David B. Garlan and Phillip L. Miller.
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, April, 1984.

[6] Charles W. Krueger.
The GANDALF Editor Generator Reference Manuals.
In *The GANDALF System Reference Manuals.* Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA., 1986.

[7] Robert Nix.
Editing by Example.
In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 186-195. ACM, January, 1984.

[8] David S. Notkin and A.Nico Habermann.
Gandalf Software Developments.
IEEE Transactions on Software Engineering, 1986.
To appear.

[9] Steven P. Reiss.
Graphical Program Development with PECAN Program Development Systems.
In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments.* ACM-SIGSOFT/SIGPLAN, April, 1984.

[10] Thomas Reps and Tim Teitlebaum.
The Synthesizer Generator.
In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments.* ACM-SIGSOFT/SIGPLAN, April, 1984.

[11] Barbara J. Staudt and Vincenzo Ambriola.
The ALOE Action Routine Language Manual.
In *The GANDALF System Reference Manuals.* Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA., 1986.