MERCURY: Distributed Incremental Attribute Grammar Evaluation

Gail E. Kaiser
Josephine Micallef
Columbia University
Department of Computer Science
New York, NY 10027

Simon M. Kaplan
University of Illinois
Department of Computer Science
Urbana, IL 61801

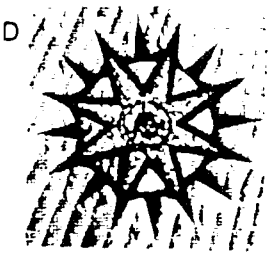
December 1987

CUCS-280-87

Abstract

This technical report consists of the two most recent papers from the MERCURY project. *Multiuser, Distributed Language-Based Environments* explains the application of incremental attribute grammar evaluation algorithms to generation of distributed programming environments and describes the implementation of the MERCURY system. *Version and Configuration Control in Distributed Language-Based Environments* presents new algorithms that permit MERCURY to support multiple versions and configurations of modules and to more efficiently propagate changes to aggregate attributes.

Prof. Kaiser is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and the New York State Center of Advanced Technology — Computer & Information Systems, and in part by a Digital Equipment Corporation Faculty Award. Ms. Micallef was an IBM Fellow when this research project began. Prof. Kaplan is supported in part by a grant from AT&T Corporation.



Multiuser, Distributed Language-Based Environments

Gail E. Kaiser, *Columbia University*
Simon M. Kaplan, *University of Illinois*
Josephine Micallef, *Columbia University*

How do you keep teams of programmers informed of system changes without burying them in mail messages? Make the environment responsible for propagating changes.

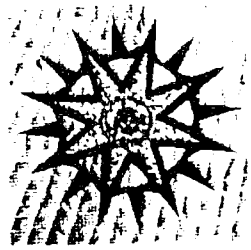
Large software projects involve teams of programmers who cooperate in development and maintenance. Each programmer typically is responsible for part of the system, one or more modules. Each module exports certain facilities to other modules and imports certain facilities from other modules. Communication problems arise when module interfaces change or do not meet what programmers imagine their specifications to be. Networks of workstations aggravate this problem as personnel become distributed.

One common solution is to pass messages among the programmers. When a programmer modifies a module interface, he sends electronic mail describing the change to all other programmers who use the module. However, in real-world software projects, the list of programmers using a particular module changes frequently because of concurrent modifications by other programmers to their modules. So the programmer sends a message to the entire team to be sure of reaching everyone who may be affected by the change.

The result is a mail deluge. Some programmers spend hours reading mail and consequently get little work done, while others ignore their mail and get lots of work done. Unfortunately, sometimes this work must later be undone or redone because of incorrect assumptions about module interfaces.

We propose a better solution: message passing among the programming environments used by the programmers. This can easily be accomplished using language-based environments that automatically identify interdependencies among program parts and immediately inform programmers of static semantic errors in one part of a program caused by changes to another part. Such environments are language-based because the determination of interdependencies and errors is specific to the particular programming language

0740-7459/87/1100/0058 $01.00



Overcoming limitations

Use of language-based environments has been limited primarily to novice programmers working alone on relatively small programs. This is due in part to dissatisfaction with the structure-oriented user interface and in part because these environments could not support multiple programmers working simultaneously on a large software system.

There are several promising approaches to solving the first problem, most of them involving a text-oriented rather than a structure-oriented interface. Incremental parsing technology[1] makes it possible for changes to the text to be reflected immediately in the program's underlying structural representation.

We solve the second problem with Mercury, our prototype of a multiuser, distributed language-based programming environment, where the environment is responsible for propagating changes. Whenever an imported module changes in a way that is incompatible with its use in an importing module, Mercury automatically notifies each programmer of errors in his own module introduced by the change in the imported module. The programmer can go about his business knowing that he will be informed of exactly those changes that affect him.

We generate each language-based environment from a formal specification — an attribute grammar — of the desired programming language. Attribute grammars attach attributes to each program part to summarize the interdependencies and interfaces between it and other parts of the program, and they permit rapid recalculation of these summaries as the program changes. The box on pp. 64-65 explains attribute grammars.

Attributes attached to each module describe its interface. Each interface has two parts: (1) the facilities exported by the module and (2) the names of other modules in the system and the facilities exported by these modules that are available for import. These attributes provide enough information to check if any intermodule inconsistencies are introduced by a particular change to an exported facility. If a change does not involve an exported facility, no intermodule propagation is required and none takes place.

The advantage of attribute grammars over other mechanisms is that there are already incremental attribute-evaluation algorithms that support automatic propagation to exactly those attributes that are affected by a particular change. The propagation occurs as soon as the change occurs.

We have extended the best-known of these algorithms[2] to a parallel form, which makes it possible to propagate among multiple users, in either a single-machine or a distributed programming environment. We have added an attribute-propagation layer that supports many programming environment facilities and reliability of the distributed environment during network and machine failures.

We generate each environment from a formal specification — an attribute grammar — of the desired language.

Mercury supports change simulation[3] in addition to change propagation. Change simulation lets a programmer ask what-if questions about whether a particular change to his module's interface would cause errors in his own or other modules, without making the change visible to other programmers. This is done by performing the attribute propagation on a copy of each relevant module.

We are not advocating that programmers cease to inform each other when they change module interfaces. They must do so to explain the motivation for their changes, since the environment cannot determine this automatically. However, the environment could prompt the programmer for this information after each change to a module interface or, less intrusively, at the end of every session. The environment could treat these explanations as special attributes of the modified program parts, to be propagated with the attribute-evaluation algorithm, or it could simply mail them to the appropriate programmers.

We are also not advocating that the programmer be notified of every error in his module immediately after every change. We describe the mechanisms to do this, but it is not necessary to take full advantage of these capabilities. Instead, each programmer could inform the environment whether he wants to be notified of inconsistencies immediately, only at the end of a session, only on check-in to the version-control system, only on user command, and so on. Mercury can separate intramodule and intermodule propagation, so static semantic errors due to the programmer's own changes can be detected at one granularity and those due to other programmers' changes at another.

Incremental interface checking

Incremental interface checking among modules can be achieved in traditional single-user, language-based environments, like the Cornell Synthesizer Generator.[4] Consider the program in Figure 1. Module *M* exports facility *x* (which could be a procedure or a type, for example) for use in other modules and imports facility *y* from module *N*. Module *N* exports *y* and imports *x*. The bodies of the two modules are omitted.

Now suppose the programmer, using a language-based environment, removes facility *x* from the export list of module *M*

```
MODULE M;
  EXPORT x;
  FROM N IMPORT y;
  ...
END; /* M */

MODULE N;
  EXPORT y;
  FROM M IMPORT x;
  ...
END; /* N */
```

Figure 1. Skeleton of a program with two modules.

Because the omitted portion of the program may cover many screens, the programmer may not remember that module *N* imports facility *x* and so may not realize that his edit causes an error in *N* due to the use of a now undefined facility.

However, the environment *does* remember and immediately warns the user that this small change caused an error elsewhere in the program. The notice can be done unobtrusively, such as by displaying "error" in the corner of the screen. The programmer is free to ignore this error indicator and deal with it later, when he gives a command to scroll to the error. The environment would then display module *N*, as Figure 2 shows. Because the error was detected and presented in context, while he was removing *x* from the export list of module *M*, the programmer is immediately aware of what caused the problem and can restore or fix the problem some other way.

One way the environment can detect such errors is to recompile the entire program after each edit. However, the result would be intolerably poor response time for all except the tiniest programs. Instead, a language-based environment, in effect, recompiles only those parts of the program affected by the edit.

The environment stores the information it needs to check for errors in attributes associated with certain program parts. In particular, the facilities exported by, and available for import into, a module are represented as attributes of the module. Attributes are defined in terms of other attributes, thus capturing interdependencies in the program.

After an edit, those attributes associated with the program part that changed and any other parts that depend on them must be reevaluated. Using the dependency information among the attributes, the environment evaluates the minimum number of attributes necessary to detect and report any errors caused by the change. The environment uses an incremental attribute-evaluation algorithm to perform this minimal recalculation.

The algorithm works by propagating information along dependency links in the program's internal representation. Figure 3 illustrates the flow of information along the dependency links of the program shown in Figure 1. One attribute associated with the entire program contains all the facilities exported by all the modules; intermodule propagation to check the consistency between exported and imported facilities passes through this attribute.

Two dependency links cut across a module's boundary: The first connects the attribute associated with the module's export statement to the attribute associated with the entire program; the second connects the program attribute with the module's import-statement attribute. In a module, there are dependency links from the import-statement attribute to the attribute for the statements in the module's body (to check that the imported variable is used correctly), and from the attribute for the module's local declarations to the export-statement attribute (to check that the exported variable is declared). If a program edit changes any attribute value, all attributes that depend on this value are recalculated.

To perform this recalculation, the algorithm first constructs a model, a special dependency graph of the attributes associated with the part of the program that was changed. The model contains a directed arc from each attribute to every other attribute that depends (directly or indirectly) on its value.

Reevaluation starts with attributes that have no incoming arcs. For those attributes that change in value, the model is expanded to include all attributes that depend on these attributes directly, together with arcs between these attributes and those already in the model to represent all direct and indirect dependencies. Then the original attribute and all its arcs are removed, leaving a new set of attributes with no incoming arcs. The process repeats until the model is empty. This approach works whenever the attribute grammar is noncircular, which is normally the case.

In our example, there is a chain of dependency links from the export-statement attribute of *M* to the import-statement attribute of *N*. If *M*'s export statement is changed, the model will eventually expand to include *N*'s import attribute. When *x* is removed from the export list in module *M*, the export attribute for *M* changes accordingly. This triggers recalculation of all dependent attributes, including the import attribute for *N*. This calculation detects an inconsistency between *N*'s import attribute and *N*'s actual import list, which contains a facility not included in the import attribute. The result is the error message in Figure 2.

```
MODULE M;
  FROM N IMPORT y;
  ...
END; /* M */

MODULE N;
  EXPORT y;
  FROM M IMPORT x; <-- cannot import this identifier
  ...
END; /* N */
```

Figure 2. Error notification after program change.

Parallel interface checking

We have developed a parallel version[5] of this algorithm. Our version spawns a new process for every attribute in the model that has no incoming arcs, so they can be evaluated in parallel. When an attribute's value changes, its process expands the model, removes the attribute and all its arcs from the model, finds all the attributes previously at the ends of these arcs that now have no incoming arcs, and spawns new processes to evaluate these attributes. This manipulation of the model must be atomic, so synchronization among the concurrent processes is done by locking the model. It is not necessary to lock attributes, because either there is one writer and no readers (during reevaluation) or no writers and perhaps multiple readers (after reevaluation).

Multiuser interface checking

We expand the traditional programming-in-the-small, language-based environment paradigm (a single user editing monolithic programs) to programming-in-the-many (many users editing the same program asynchronously). We assume that two programmers cannot edit the same part of the program; that is, there is some division of the program among programmers. The obvious division is for each programmer to be responsible for one or more modules. Therefore, we propose a model of editing where many programmers access a common, internal representation of the program, but are each given an area of this representation that only they can modify.

Suppose Dick and Jane are editing our sample program. Dick can edit module M only and Jane can edit N only. Suppose that Dick deletes x from the export list of M. An error message appears immediately on Jane's screen, and she can scroll to the actual location of the error in module N.

It might seem that this could be accomplished with the parallel version of the algorithm. However, consider what happens when the attributes affected by Dick's change are being propagated and at the same time Jane deletes y from the export list of N, setting off a new set of propagations. The internal representation of the program is now being asynchronously modified by two processes.

Previous algorithms for incremental evaluation either assume a single change to the program (where exactly one point in the program has inconsistent attributes) or require that multiple changes be synchronized[6] (a model consisting of the union of the dependency graphs of those program parts with inconsistent attributes is formed before evaluation begins). The latter algorithm is useful for efficiently recalculating attribute values when a single editing operation causes multiple nodes of the program to have inconsistent attributes but is not applicable to multiple, asynchronous editing operations.

Our algorithm[5] performs incremental attribute evaluation for multiple, asynchronous edits. Our algorithm associates a single model with each program segment (in our example, with each module). When there is more than one user, an asynchronous change by one programmer in a module can propagate into another module through a dependency link that crosses module boundaries. This adds a new component to the model of the second module.

These two components can be vertex-disjoint if the intermodule propagation and the original propagation affected different areas of the module. Two disjoint components of the model may become joined if an expansion of one component adds an attribute that is already part of the other component. At this point, it is necessary to add arcs from this attribute to all direct and indirect dependencies in the combined piece.

A difficulty arises if module M is modified and the change is propagated to module N at just the moment that N is itself being edited (while the internal data structure is being modified). Our algorithm cannot permit an attribute propagation to arrive in N when the syntactic structure of the module is changing.

To solve this problem, we introduce fire

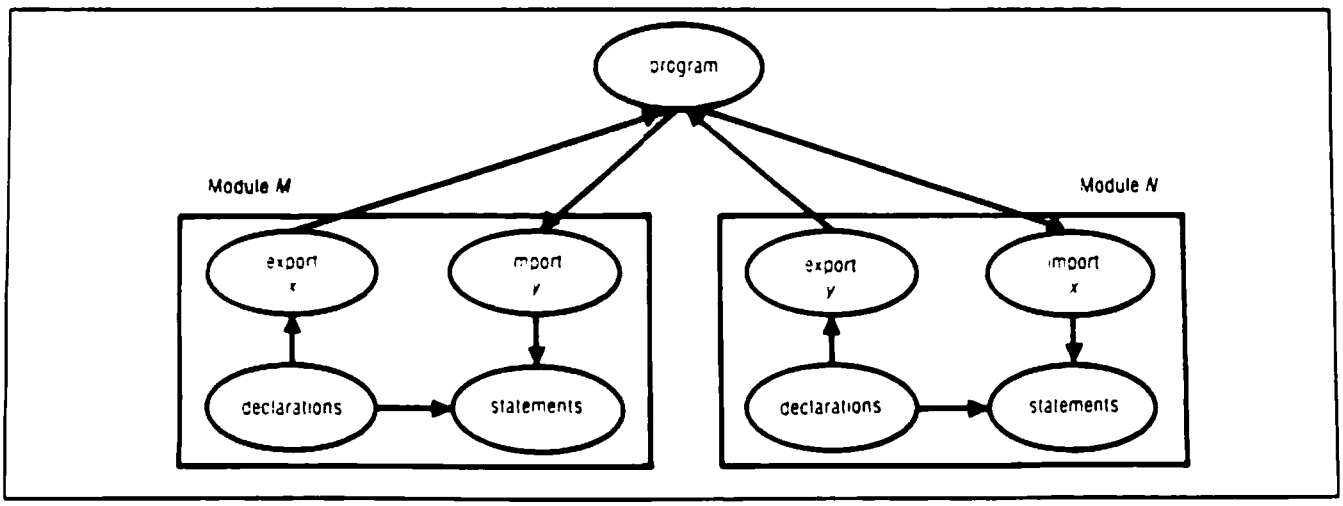


Figure 3. Logical representation of a program.

walls. A fire wall is a barrier that shelters a program segment while it is being modified. A fire wall can be "up," in which case any attribute propagation attempting to cross the boundary is delayed, or "down," in which case the fire wall is entirely invisible. The fire wall need be up only when the internal representation of the program segment is actually being changed. This is a minimal amount of time compared to the time spent by the environment performing attribute evaluations and the time the programmers spend browsing the program.

Distributed interface checking

Once we allow multiple edits on a program using fire walls, the next step is to split the programs across machines, since the advent of inexpensive workstations is rapidly making distributed program development with cooperation among the programmers the preferred mode of software development.

We split a program so each fire wall-protected segment (in most cases this is a module) is assigned to a workstation. One workstation may be the home of many modules, typically those under development by the same programmer. In Figure 3, modules M and N would be assigned to different workstations. To avoid the need for centralized storage, the part of the program representing the root of the information-flow tree is replicated on every workstation. However, certain root information is impossible to replicate in a distributed environment and must be handled differently, as explained below.

Each machine handles attribute propagation as if it were the only machine in the network — as long as the propagation remains within the bounds of the fire wall, and thus within its model. When an attribute reaches the fire wall, it must deal with remote machines. This is handled by an attribute-propagation layer. The APL constructs a packet containing the attribute's value and sends it across the network to the other modules. The APL waits until the target module's fire wall is down and then propagates the attribute into the module by simulating an edit at its fire wall. Attribute propagation then proceeds in the target module.

Thus, the export attribute from M is bundled and passed across the network to N, where it is unpacked and inspected. If different from its previous value, the export attribute is propagated into N as soon as the fire wall is down.

The APL performs the packing, unpacking, and dissemination of attribute packets to the target modules. It also maintains a cache of the attribute values that have arrived from each remote module. The cache is assumed to be up to date on the grounds that if more recent information was available it would have arrived. When a new attribute arrives, the APL compares the attribute to its cached value. If the two values are different, the new value is copied into the cache.

In the real world, messages can arrive late and out of order; networks and machines can fail.

The APL also supports dormant modules, those not currently being edited. Attribute propagations are stored in the cache until that module is next edited. This strategy allows a simple optimization that is implemented in our prototype: When a module reawakens, it is passed only the most recent values of changed attributes. Alternatively, attribute propagations to dormant modules could be performed in a background process, with the environment mailing any error messages to the responsible programmer.

Certain kinds of static semantics checking, such as duplicate module names, depend on an ordering of the modules: If a program contains two modules with the same name, the one that comes later in the program is flagged. This works fine in a single-user, nondistributed environment, where only one programmer can add (delete) modules to (from) the program. In that case, the environment can maintain an internal representation of the program that reflects the correct ordering of the modules.

However, if multiple programmers can create modules independently, the environment cannot guarantee that all programmers see the same ordering. For example, suppose the program has a single module, M, and Dick and Jane at the same moment create modules N and O, respectively. Dick may think the program is made up of modules M, N, and O in that order while Jane may think it is ordered M, O, and N. Moreover, if Dick and Jane happen to create modules with the same name simultaneously, say N, it is unclear which module should be considered the original and which the erroneous duplicate.

We solve this problem by restricting programs to consist of an unordered set of modules. Programmers create modules independently. If a module with the same name as an existing module is created, both are flagged with the error "module name declared twice." Duplicate modules are not considered part of the program, and no propagations are sent to or received from such modules.

The APL keeps track of all the modules in the program and stores two pieces of information for each: (1) if the module resides on that machine and (2) if the module is uniquely identified. If a new module is created with the same name as an existing module, the APL adds the name (again) to the list of modules in the system, marks both as erroneous, and propagates an error message to both.

The real world

Our discussion has assumed that all interface changes are propagated to all modules instantaneously. This is unrealistic: In the real world, messages can arrive late and out of order; networks and machines can fail.

Message passing. Fortunately, late and out-of-order messages are not a problem because of the nature of incremental attribute evaluation. In particular, the time a message arrives does not matter — once it does arrive and propagation terminates, the result is the same. If two mes-

sages arrive from distinct modules, the order of arrival also does not matter — the final result is again the same.

Multiple messages from the same module that arrive out of order are handled by comparing time stamps. Every message contains an attribute's name, its new value, and a time stamp indicating when the new value was calculated. The time stamp could be taken from the clock of the machine where the value was calculated, or it could be an integer incremented each time the reevaluation algorithm assigns the attribute a new value (making it possible to move the module among machines). When a message arrives from an APL, its time stamp is compared to the time stamp of the corresponding attribute in the cache. If the time stamp of the message is earlier, the message is discarded; if later, the cached time stamp is updated and the two values compared. If the values differ, the new value overwrites the cached value and is propagated to the target module, triggering attribute reevaluation. A global clock is not necessary — only time stamps of attributes originating from the same module are compared.

Failures. We have developed a special algorithm to deal with failures. Our algorithm repropagates changes to those modules that did not receive the original propagation because they were inaccessible due to machine or network failure. Programmers can continue working on machines that are separated from part or all of the network, knowing that local changes will be propagated and remote changes will be received as soon as the network is restored. Because late and out-of-order messages either do not matter or are handled by time stamps, this approach sufficiently guarantees availability and reliability.

The easiest way to explain this algorithm is through an example. Suppose Dick, Jane, and Sally are working together on a system, editing modules *M*, *N*, and *O*, respectively. Figure 4 shows the interfaces among the modules. Dick's machine is currently unreachable from the rest of the network. Sally changes module *O* to remove the export of *z*. This change is broadcast throughout the network, and is received by the APL for Jane's workstation, where the cache for module *O* is updated. However, since Jane's module *N* does not import *O*, Sally's change does not introduce any errors into module *N*.

Now Sally's workstation crashes and Dick's is restored to the network. Dick's APL broadcasts a special update signal, indicating that network-wide consistency must be reestablished. This prompts Jane's APL to send all the information in its caches for *M*, *N*, and *O* to Dick's APL. The new cache for *N* is quickly discarded, since the time stamps are the same. The new information for *O* replaces the old, and Sally's change is propagated into module *M*, causing the appropriate error message to be displayed. This happens even though Sally's machine is not currently accessible.

What if Dick had made several changes while in isolation? These would be reflected by updated time stamps in the local cache for module *M*. When the older cache *M* arrives from Jane's APL, a comparison of the time stamps causes Dick's APL to send Jane's APL the new cache. Sally's APL will be updated similarly when it eventually broadcasts an update signal.

```
MODULE M ;
  EXPORT x ;
  FROM N IMPORT y ; FROM
O IMPORT z ;
  ...
END; /* M */

MODULE N ;
  EXPORT y ;
  FROM M IMPORT x ;
  ...
END; /* N */

MODULE O ;
  EXPORT z ;
  FROM M IMPORT x ;
  ...
END; /* O */
```

Figure 4. Skeleton of a program with three modules.

Mercury

Mercury is implemented in C and runs under 4.3 BSD Unix. It provides a distributed editing environment for an arbitrary number of Digital Equipment Corp. VAX computers connected by an Ethernet network. We have generated environments for subsets of Modula-2 and Ada.

Mercury has two parts, an editor generator and an APL. The editor generator takes as input an attribute grammar for the desired language and produces a language-based editor tailored to that language. Copies of this editor are installed on each machine. Each invocation is known as a local editor; the entire system of all local editors and the APL is called the distributed editor.

The APL is responsible for propagating changes in attribute information among the local editors. It is language-independent: Distributed editors for several languages can be simultaneously supported by the same APL. The current implementation, however, does not handle the transmission of changes between modules written in different languages. The APL is implemented as a special process on each machine, and each is called a local APL.

Figure 5 shows the structure of a local editor. A local editor is generated in two phases: (1) translation of the attribute grammar and (2) linking the language tables produced in the first phase with a language-independent editor kernel to produce an editor for a specific language.

Our editor generator is built on top of the Cornell Synthesizer Generator, which generates language-based editors for single-user environments. We reused all the code common to both multiuser and single-user cases, including pretty-printing the program on the screen and interpreting user commands. Our major modifications were to extend the attribute-grammar notation with new classes of attributes that specify interface information and to add our new incremental attribute-evaluation algorithm to handle asynchronous edits.

We designate certain attributes — those that capture the flow of information among modules — as interface attributes. These attributes are defined by semantic equations in the language specification, as described in the box on pp. 64-65.

We provide a union operator for defining attributes at the root of the program

Background on attribute grammars

An attribute grammar is a set of rules giving the context-free syntax of the language, like YACC specifications. Each rule is associated with a set of semantic equations that specify context-sensitive information, such as symbol resolution and type checking, which cannot be expressed directly in the syntax part of the rules. These equations are similar to simultaneous equations in algebra, and their variables are called attributes, thus the name attribute grammars. Attribute grammars were first proposed by Knuth[1] and the technology for generating language-based environments from attribute grammars was developed by Reps et al.[2]

Figure A contains a small attribute grammar for a simple module-interface language. The example language is meant to illustrate attribute grammars and is not intended to be realistic. Real attribute grammars can be very large; our attribute grammar for only a small subset of Modula-2 is 1063 lines.

In our example language, a program consists of several modules, each of which can import and export exactly one variable. A module can import any variable that has been exported by any other module. To check that imported variables are used correctly, the attributes exportsin and exportsout are used to build a list of all exported variables. This global exported variables list is passed to the import statement in each module using the attribute allimports.

The attribute grammar in Figure A consists of five rules:

Rule 1. The first rule states that a program consists of modules. It is followed by three semantic equations for program: The first defines the values of the allexports attribute of program to be equal to the exportsout attribute of modules (exportsout is itself defined later on in the figure); the second initializes the exportsin attribute of modules to empty, and the third defines the allimports attribute of modules to be the same value as allexports.

Rule 2. This rule specifies that modules is an ordered list of zero or more components, each of which is a module. Exportsin contains the exported variables of all modules preceding this one, and exportsout contains the exported variables of all modules up to and including this one. The allimports attribute is passed to both components of modules.

Rule 3. This rule states that a module has an identifier and export, import, declarations, and statements components. (In the figure, MODULE, IS, END, and ; are reserved words or symbols in the programming language and must be placed as indicated by the rule.) This rule is followed by five semantic equations.

The first defines an error attribute associated with each module. The error attribute specifies what text string to display. In this case, the possible error is that the identifier used to name the module is the name of some other module. If the module name is in fact unique, the error string is null. The namesof function extracts the set of module names from the list of facilities available for import so this check can be made.

The second equation defines the exportsout attribute of the module as the union of the exportsin attribute of that module and an entry composed of the idname attribute of the module name and the exportid attribute of the export list. If the name of the module is not unique, however, the exportsout attribute is assigned the exportsin value.

The third equation passes declarations.locals, the list of variables declared in this module, to the possibleexports attribute of the export statement. One of these variables may be exported by this module.

The fourth equation equates the allimports attribute of the module's import list to the value of the module's own allimports attribute.

The fifth equation assigns to statements.variables the union of the imported variable and the local declarations, specified by the attributes importid and locals respectively. This attribute indicates the variables that can be used in the statements part of a module.

```
program ::= modules
        {program.allexports = modules.exportsout;
         modules.exportsin = Null;
         modules.allimports = program.allexports;}

modules1 ::= /*Empty production*/
        {modules1.exportsout =
           modules1.exportsin;}
      | module modules2
        {module.exportsin = modules1.exportsin;
         modules2.exportsin = module.exportsout;
         modules1.exportsout = modules2.exportsout;
         module.allimports = modules1.allimports;
         modules2.allimports = modules1.allimports;}

module ::= MODULE id IS export; import; declarations;
        statements; END;
        {module.error =
           if id.idname not in
              namesof (module.exportsin)
           then ""
           else "<--module name
                   declared twice";
         module.exportsout =
           if id.idname not in
              namesof (module.exportsin)
           then
              union (module.exportsin,
              entry(id.idname, export.exportid))
           else module.exportsin;
         export.possibleexports = declarations.locals;
         import.allimports = module.allimports;
         statements.variables = union(import.importid,
                                declarations.locals);}

export ::= EXPORT id
        {export.exportid = id.idname;
         export.error = if id.idname in
                           export.possibleexports
                        then ""
                        else "<--cannot export this
                           identifier";}

import ::= FROM id1 IMPORT id2
        {import.importid = id2.idname;
         import.error =
           if (id1.idname in namesof(import.allimports))
           and (id2.idname =
                exportsfrom(import.allimports,
                  id1.idname))
           then ""
           else "<--cannot import this identifier";}
```

Figure A. An example attribute grammar. The rules stating the program syntax are in plain text, reserved words are in small capitals, the attribute-grammar syntax is in boldface, and semantic equations are in italics. The grammar's form has been modified slightly to aid readability.

Rule 4. The export rule indicates the variable that is exported by the module and is followed by two semantic equations. The first equation assigns the exported variable name to the exportid attribute; this is used to build the export list of the module, as mentioned above. If the exported variable has not been declared (and therefore it is not in possibleexports), the second equation assigns the error attribute an appropriate message.

Rule 5. The import rule indicates a module (id_1) from which a variable is imported (id_2) and is also followed by two semantic equations. The first copies the variable's name to the importid attribute, from where it can be propagated for use in the module's statements. The second equation defines the error attribute; it states that the module id_1 must exist and must actually export the variable id_2, or the indicated error message is displayed. The function exportsfrom takes the list of available imports and the name of a module as arguments and returns the identifier, if any, exported by that module. The rules for declarations and statements are omitted for the sake of brevity.

Decorated trees. Besides being a formal way of describing program constraints, attribute grammars can be seen as decorated trees: The rules make up the structure of a tree, and the attributes are the decorations at the nodes of the tree. Each attribute in an attribute grammar is attached to a symbol in the rules; when a grammar is used to construct a tree, the symbols in the rules become the nodes, the components of the symbols (the bodies of the rules) become children nodes, and the attributes are decorations of the nodes corresponding to the symbols to which they are attached in the grammar. This tree structure is the internal representation for programs in language-based editors.

Figure B shows the tree representation of the example program in Figure 1 in the main article. This program has two modules, *M* and *N*. *M* exports a variable *x*, and *N* exports a variable *y*. Each module imports the other's exported variable. The tree in Figure B represents the complete structure for module *M*, and represents the module node for module *N*. The syntactic structures are shown as ellipses; the attribute instances associated with them are shown as rectangles. To conserve space we omit the terminals of the grammar in the diagram of the tree (such as the keywords MODULE and IS) and the diagram for module *N*, which is similar to that for module *M*.

Now suppose the user removes facility *x* from the export list of module *M*, thus modifying the internal tree representation of the program. The semantic equations must now be reapplied so that the decorations remain consistent with the tree.

The edit is at a point specified by the fourth syntax rule in Figure A — export — so we first consider the semantic equations associated with this rule. The value of id.idname becomes null and this new value is copied to export.exportid. This causes the second equation under module (Rule 3) to propagate this change by recalculating the attribute module.exportsout for *M* to contain only a null identifier in the exported variable field. In turn, this causes a recalculation of the exportsout attribute in modules (Rule 2) and the allexports attribute in program (Rule 1), so the entry for module *M* contains no exported variable.

The new list of variables that can be imported is passed to each of the modules, including *N*, by the attribute allimports. In *N*, the fourth equation for module (Rule 3) reassigns import.allimports, and then import's (Rule 5) second equation finds that the imported variable is no longer available and the error attribute is changed from the null string to the error message "← cannot import this identifier."

A similar series of recalculations is also done in module *M*, but because there are no errors there the effect will be invisible to the user. This attribute reevaluation in response to program changes is the basis for incremental interface checking in language-based environments.

Figure C illustrates how an attribute grammar is translated into an editor. An editor generator (a program similar in concept to the YACC parser generating system) takes as input an attribute grammar specification of a language. This specification is translated into intermediate language tables that, when compiled together with an editor kernel, produce an editor tailored to the specific language. In this figure, the rectangles represent data and the ellipses the processing functions. The editor kernel consists of several parts, including a user interface, the incremental attribute-evaluation engine, a tree-manipulation engine, and systems-support utilities.

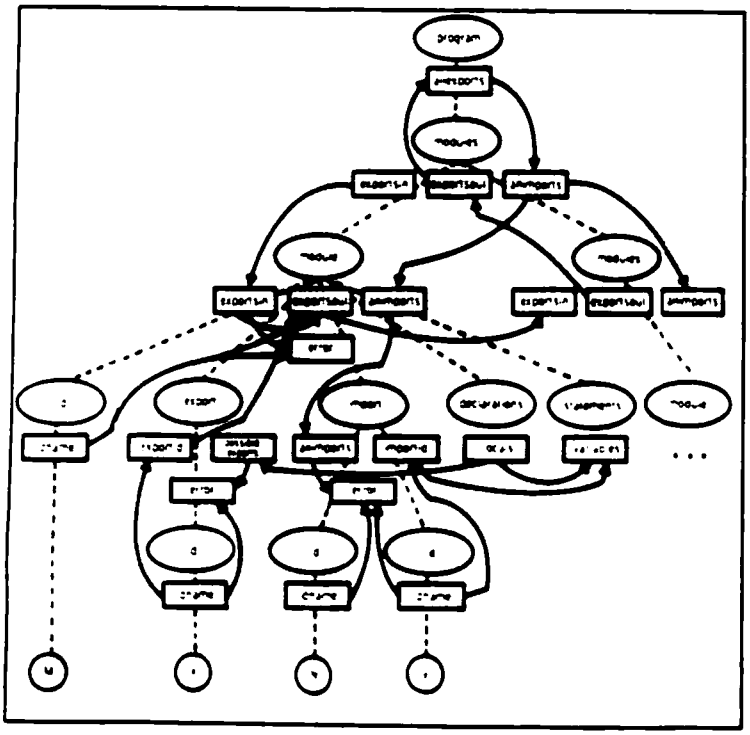


Figure B. A tree representation of a program.

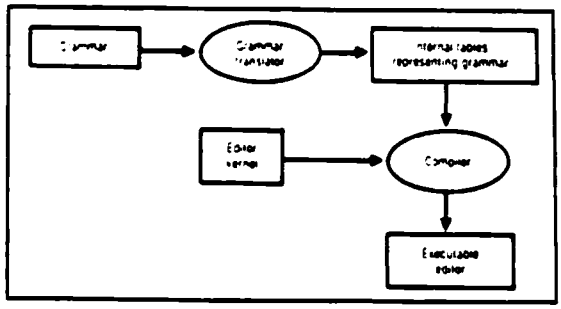


Figure C. A diagram of the generator.

References

1. D.E. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory*, June 1968, pp. 127-145.
2. T. Reps, T. Teitelbaum, and A. Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Trans. Programming Languages and Systems*, July 1983, pp. 449-477.

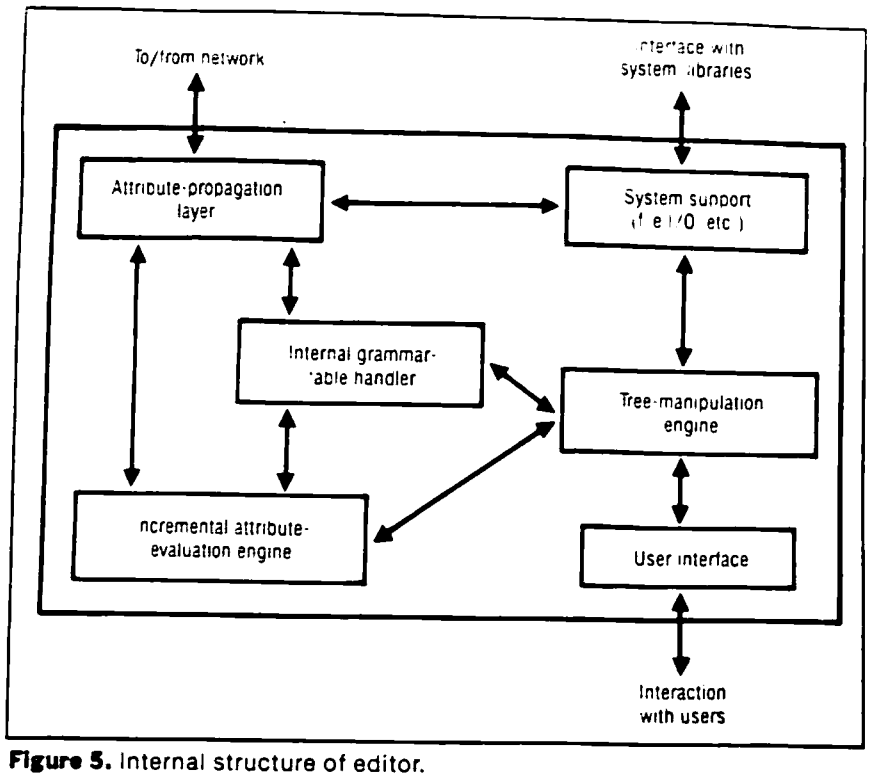


Figure 5. Internal structure of editor.

from interface attributes of each module. (The current implementation does not support defining attributes at the root of the program that are arbitrary functions of the interface attributes of the modules, but this has not been a problem in practice since union is sufficient for interface attributes used for change propagation, notably the external symbol table.)

When an edit of a module changes the value of an interface attribute, the new value is propagated by the APL to all the workstations. The attribute at the root of the program (which is replicated on all workstations) is recomputed, and propagated to the attributes of the modules that are dependent on it.

In a multiuser environment, a local editor can receive input from two sources. The programmer using the local editor can perform arbitrary editing operations on the module being developed. The local editor might also receive a new value for an attribute as a result of a change in some other module. The second input (which does not change the program itself) should never happen while the program is being modified by the user. The fire-wall effect is obtained by using the Select system call to check for pending requests. The local editor performs these operations, one at a time, as they occur. However, we do not serialize the attribute evaluations because that might result in repeated recalculation of the same attribute.

Our implementation uses a single process for the local editor and uses Unix software signals to handle the attribute evaluation of multiple, asynchronous edits. The editor process is responsible for performing the incremental attribute-evaluation algorithm. Input from the user or the APL causes the Sigio signal (which indicates I/O is possible on a file descriptor) to be generated. This signal is trapped and the Sigio signal handler we provide is invoked. The handler processes the input, creates the model for this change, and merges it with the current model — which could be empty if no attribute propagation is in progress. After returning from the handler, the editor process continues with the attribute-evaluation algorithm at the point it was interrupted, but the model now also reflects the attributes that need to be reevaluated as a result of the new edit.

By supporting cooperation among many language-based environments, we achieve programming-in-the-many.

Mercury does not maximize parallelism by concurrently evaluating independent attributes in a local editor. There is no nice mechanism for sharing data among Berkeley Unix's heavyweight processes, and the advantages of such parallelism cannot be realized on the VAX uniprocessor. However, the fully parallel algorithm described above would be suitable for a multiprocessor with lightweight processes.

The distribution component of the incremental attribute-evaluation algorithm works as follows. When a module's interface attribute gets a new value, the local editor sends a message containing this new value to the local APL. The local APL broadcasts this message to all other machines on the network. The local APL on each of these remote machines updates the corresponding attribute at the root and sends it to all the local editors whose modules belong to the same program, thus informing them of the change. Whenever a new value is computed for an interface attribute, a time stamp derived from the local clock is attached to the value. The time stamp is used in reestablishing consistency among the local APLs in the case of machine or network failure.

The APL has two parts: (1) a transport layer that handles the actual transmission of messages, both among the local editor and its APL and among local APLs, and (2) an attribute cache that contains the latest value of all attributes that passed through the APL. The attribute cache is identical on all machines, except when some part of the network is down; then the local APLs might temporarily have old attribute information, but consistency will eventually be reestablished.

The transport layer uses sockets for all interprocess communication. We chose datagram communication over stream communication because the restriction on the number of open streams would have limited the number of local editors an APL could support. Datagram communication does not guarantee reliable transmission of messages — messages can be transmitted out of order or may be lost. But old or out-of-order messages are not a problem for this application. We prevent messages from being lost by defining a reliable distributed environment on top of the datagram that provides acknowledgments and retransmission, among other things.[8]

Mercury supports teams of programmers collaborating on the development and maintenance of large software systems. The environment supports incremental checking of interdependencies among modules, whether the modules reside on the same machine or are distributed among multiple machines connected by a network. Each module is edited in a language-based programming environment, previously suited only to programming-in-the-small. By supporting cooperation among many such environments, we achieve programming-in-the-many.

Mercury has a serious limitation, however, with respect to programming-in-the-large — it assumes there is only a single version of each module and a single way of composing the modules into a system. For example, when a programmer changes an interface of module *M*, this change is propagated to all modules that import *M*, even though some of the programmers responsible for these modules prefer to continue using some previous version of *M*. We are working on system-modeling and version-control facilities for Mercury. One such facility will maintain a system model that specifies the module interconnections — which modules make up the program and how they are ordered; this will be replicated on every workstation. If the model is changed, then the new model will be broadcast throughout the APL, suspending normal execution of the APL algorithms until every machine has the new model. A related facility will add a version map for each module to its local APL, so that change propagation (at the granularity requested by the user) will be applied to the version of each imported module selected in the system model or by the programmer. ⊹

Acknowledgments

Wenwey Hseush implemented the APL with the help of Su-Chuan Tsai, Chen Yen, and Bulent Yener. Joe Adipietro, Katherine Feldman, Luke McCormick, Linda Mischel, and Michael Tavis worked with Josephine Micallef on the implementation of the editor generator, the incremental attribute-evaluation algorithm for asynchronous edits, and other extensions to the Cornell Synthesizer Generator. Rowan Maclaren and Michal Melamede developed the attribute grammars for subsets of Modula-2 and Ada, respectively. Michael Tavis also helped Micallef and Hseush with the demonstration for the Columbia University industrial associates in February 1987. The Synthesizer Generator was developed at Cornell University by Tom Reps and Tim Teitelbaum; we acknowledge their effort in its development, and also thank Reps for his assistance in converting the generator to run as described here.

Kaiser is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and New York State Center of Advanced Technology — Computer and Information Systems, and in part by a Digital Equipment Corp. faculty award. Kaplan is supported in part by a grant from AT&T Corp.

References

1. G.E. Kaiser and E. Kant, "Incremental Parsing without a Parser," *J. Systems and Software*, May 1985, pp. 121-144.
2. T.W. Reps, *Generating Language-Based Environments*, MIT Press, Cambridge, Mass., 1984.
3. G.E. Kaiser and D.E. Perry, "Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution," *Proc Conf. Software Maintenance*, CS Press, Los Alamitos, Calif., 1987.
4. T. Reps and T. Teitelbaum, "The Synthesizer Generator," *Proc. SIGSoft/SIGPlan Software Eng. Symp.*, ACM, New York, 1984, pp. 41-48.
5. S.M. Kaplan and G.E. Kaiser, "Incremental Attribute Evaluation in Distributed Language-Based Environments," *Proc. Symp. Princ. Distributed Computing*, ACM, New York, 1986, pp. 121-130.
6. T. Reps, C. Marceau, and T. Teitelbaum, "Remote Attribute Updating for Language-Based Editors," *Proc. Symp. Princ. Programming Languages*, ACM, New York, 1986, pp. 1-13.
7. G.E. Kaiser and S.M. Kaplan, "Reliability in Distributed Programming Environments," *Proc. Symp. Reliability in Distributed Software and Database Systems*, ACM, New York, 1987, pp. 45-55.
8. W. Hseush and G.E. Kaiser, "A Network Architecture for Reliable Distributed Computing," *Proc. Symp. Simulation of Computer Networks*, ACM, New York, 1987, pp. 11-22.

Gail Kaiser is an assistant professor of computer science at Columbia University, where she received a Digital Equipment Corp. faculty award. Her research interests include programming environments, evolution of large software systems, application of artificial intelligence technology to software development and maintenance, software reusability, object-oriented languages and databases, and distributed systems.

Kaiser received the MS and PhD in computer science from Carnegie Mellon University, where she was a Hertz Fellow, and the ScB from the Massachusetts Institute of Technology.

Simon Kaplan is an assistant professor of computer science at the University of Illinois at Urbana-Champaign. His research interests are programming languages and programming environments, with special interest in concurrent computation, distributed programming environments, and building support for the design phase.

Kaplan received the PhD from the University of Cape Town, South Africa, and is a member of the ACM.

Josephine Micallef is a candidate for the PhD in computer science at Columbia University, where she was an IBM Fellow. Her research interests are software development environments, distributed computing, and attribute grammars. Her thesis research combines her work in distributed programming environments, version control, and configuration mangagement.

Micallef received the SC (summa cum laude) and MS in computer science from Columbia University. She is a member of the Computer Society of the IEEE, the ACM, and Phi Beta Kappa.

Questions about this article can be addressed to the authors at the Department of Computer Science, 450 Computer Science Building, Columbia University, New York, NY 10027.

IEEE Software

NOVEMBER 1987

THEME ARTICLES

SPECIAL FEATURES

To appear in International Workshop on Software Version and Configuration Control, Grassau, West Germany, January 1988.

Version and Configuration Control in Distributed Language-Based Environments

Josephine Micallef
Gail E. Kaiser

Department of Computer Science
Columbia University
New York, NY

Abstract

We discuss a set of algorithms for supporting change propagation in distributed programming environments. Our previous papers presented algorithms suitable only for the over-simplified scenario of a single program made up of a collection of modules, each with only a current version where every change to the interface of any module was propagated to all the other modules. This paper describes new algorithms that handle realistic software development and maintenance by large teams of programmers, using real programming languages that permit nested modules, where each system evolves over time through multiple configurations specifying a particular version of each module. Furthermore, the new algorithms are dramatically more efficient in that they propagate exactly those changes that affect each module version.

No scene from prehistory is quite as vivid as that of the mortal struggles of great beasts in the tar pits . . . Large-system programming has over the past decade been such a tar pit . . . [as] the accumulation of simultaneous and interacting factors brings slower and slower motion.

— Frederick P. Brooks, Jr., *The Mythical Man Month*, 1982.

1. Introduction

Building large software systems is hard. Several factors contribute to this, most of which are directly related to the size and complexity of the system. The task of building a large system is usually divided among several people, each working on a different part of the system. Since the pieces are interrelated, ensuring consistency of the software system requires interaction among the programmers. This leads to a *communication* and *coordination* problem [29]. Another characteristic of large systems is that they evolve over time. At any point in time, there are released versions of the system, experimental versions of the system, versions of the system targeted for different hardware, and so on. This results in a proliferation of versions of the components of the system. An additional burden is placed on the programmer — he must select versions for each component in the configuration he is working in, and maintain consistency within that configuration. We call this the *version/configuration control* problem.

In this paper, we describe a class of programming environments that assist programmers with both problems. The environment is centered around a language-based editor, which allows

programmers to enter only syntactically correct constructs. More importantly, the environment is able to analyze the components of the system incrementally after each change to check for static semantic errors, both within each component and among the components. If a change results in errors in some of the components of the system, these components are flagged with error messages. Code is generated for error-free components. The environments described in this paper work with a version control system that stores, retrieves, protects and merges different versions of each component, such as RCS [30] or SCCS [27].

The environments we describe are generated from an attribute grammar (AG) description of the programming language for which the environment is tailored. Using attribute grammars as the basis of programming environment generation offers many advantages, the most obvious of which is the reduction in cost and time for developing a new environment than if the environment was handcoded. In addition, optimal algorithms for incremental analysis are well known, an undo mechanism is automatically supplied by the attribute evaluation algorithm, and there is a large body of theory on the AG formalism.

Software development environments are adapting to a change in the hardware base from large timesharing systems supporting the entire project team to computing environments consisting of workstations connected by local area networks. This trend stems from the advantages offered by personal workstations, including more predictable response time, increased reliability of the system as a whole, and incremental expandability. The programming environments we describe accommodate distributed computing environments: The programming environment is distributed among the workstations. Each programmer interacts with a local programming environment running on his machine. The programming environment handles all communication and coordination with the rest of the system.

We start by introducing language-based environments in section 2. We present two new ideas, both of which extend our previous results in distributed attribute evaluation algorithms [14]. First, we describe algorithms that propagate changes only to modules that are affected by the change; this is the topic of section 3. Second, in section 4, we describe modifications to the algorithms to handle multiple versions of the system components. We conclude by summarizing the contributions of this paper.

2. Background

2.1. Language-Based Environments

A programming environment is called language-based if the support it offers to a programmer is specific to a particular programming language. The most ubiquitous example of language-based environments are the structure-oriented editors, which allow programmers to enter only syntactically correct program fragments. Several of these editors also check for static semantic errors or anomalies in the programs being edited [28], such as declaration of variables before use, type-checking, and use of uninitialized variables. Syntactic and semantic

analysis are dependent on the particular programming language; the part of the environment that deals with these aspects must therefore be written anew for each programming language. However, large parts of the environment are language-independent; this includes, for instance, the user interface routines, the command interpreter, and interfaces to operating systems utilities. One important contribution of the initial work in language-based environments was the capability to generate such environments from a language specification and a language-independent editor kernel. These environments are interactive, that is, the user is notified of (syntactic or static semantic) errors in his program as soon as he enters an incorrect piece of code. This places an additional requirement on these environments: The algorithms used to analyze the program must be fast enough to be executed at every editing change.

An approach that has been used very successfully relies on an attribute grammar specification of the programming language. An attribute grammar extends a context-free grammar (which describes the syntax of the programming language) with attributes that give a "meaning" to strings of that language [17]. Attribute grammars have been used in compiler-compilers to describe the translation of programming languages [8, 6]. Reps *et al.* pioneered the use of attribute grammars for programming environments [23].

Attribute grammars are declarative specifications of the semantics of a programming language. Attributes are associated with symbols of the underlying context-free grammar. The value of an attribute associated with a symbol X is defined by a *semantic equation* on other attributes associated with symbols in the two productions in which X appears, on the left and right hand sides of the productions, respectively. Attributes are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes. A semantic equation defines a value for a synthesized attribute of the left-hand side symbol of a production or an inherited attribute of a right-hand side symbol.

The attributes associated with a symbol decorate the symbol's node in the parse tree representing the program. If an attribute b appears in the semantic equation defining attribute a, then a is *dependent* on b. An edit operation corresponds to a *subtree replacement*, which replaces one subtree in the parse tree with another. After the replacement, the values of the attributes associated with the symbol at the root of the new subtree may be inconsistent. These attributes are reevaluated, and the chain of dependencies induced by the semantic equations is used to reevaluate all those attributes whose value might be changed.

Reps [24] describes an optimal incremental attribute evaluation algorithm whose complexity is proportional to $|AFFECTED|$, where the set *AFFECTED* contains all attributes whose values change as a result of the edit. The algorithm performs a topological sort on a dependency graph emanating from the point of the change, to ensure that an attribute is reevaluated only after the attributes that it depends on have received their final value. This optimality result is one of the main advantages of using attribute grammars as the formalism on which to base incremental language-based environments.

2.2. Distributed Environments

The attribute evaluation algorithm described above was extended by Kaplan and Kaiser to handle asynchronous edits [16]. The basic idea is that if the program is changed while attribute evaluations from the previous change are still proceeding, the dependency chains resulting from the previous change are merged with those due to the new change, and the evaluation algorithm continues with the merged dependency graph. If the two changes initially affect different parts of the program, the dependencies arising from the two changes start out as disjoint pieces and might or might not eventually overlap. While the two pieces are disjoint, no transitive dependencies between the pieces are considered; this means an attribute might be evaluated again if the pieces are merged after it has been evaluated in the context of its previously separate piece. An improved algorithm by Gietz that maintains transitive dependencies between the disjoint dependency graphs, thus avoiding this reevaluation, was described to the authors by Reps [25].

This ability to handle asynchronous edits on a program provides the mechanism that allows language-based environments to assist not simply an individual programmer but an entire project team. Each programmer is responsible for a segment of the system under development. When a programmer makes a change to his segment, the environment propagates the changed attribute values resulting from the edit both to dependent attributes within the same segment and also to other segments whose attributes depend on the changed attributes.

The languages supported by our distributed environments are *modular languages*. A modular language provides a construct for structuring a program, typically called a module. Each module explicitly specifies which facilities it imports from other modules and which facilities it exports to make available for use by other modules. Examples of such languages include Ada™ [1], Modula-2 [31] and Mesa [20]. These modules can be developed independently by different programmers, and correspond to the segments supported in our distributed environments.

Extending the asynchronous attribute evaluation algorithm to work in a distributed computing environment requires that the algorithms be robust enough to withstand machine and network failure without bringing the entire environment to a standstill. The part of the distributed environment that handles the actual transmission of information between the different machines in the distributed environment is called the *attribute propagation layer* (APL). This layer is a continuously-running process on each machine. When an edit in one program segment affects other segments (determined by the dependencies in the attribute grammar specifications), the attribute evaluation algorithm passes control to the local APL, which multicasts the information to the APLs running on the machines where the affected segments reside. These APLs, in turn, propagate the information to the segments in question, where the attribute evaluation algorithm takes over to perform the consistency analysis as usual.

Attributes that pass information between program segments are cached in each APL. The

reason for replicating this global information on each machine in the environment is to ensure high availability and reliability. When a program segment needs an attribute value from a remote machine, the value stored in the cache can be used. This must be the latest value available for this attribute; if there was any newer value, it would have been propagated to this machine and therefore replaced the cached value. This allows quick access of remote information, even if the other machine happens to be down or partitioned from the rest of the network at the time the information is needed. Algorithms for regaining consistency in the caches after machine or network failure are described in Kaiser and Kaplan [15].

One common theme that recurs in our distributed algorithms (both the previously published ones and the new ones presented here) is that we avoid any synchronization among the programming environments running on each machine. The main reason for this bias is that synchronization necessarily means waiting, antithetical to the interactive environments we are addressing. Even without synchronization, we attain a level of consistency of the shared attribute information sufficient for our application. This is an example of the principle described by Cheriton as "problem-oriented shared memory" [4].

Our previously published algorithms for distributed language-based environments have two serious limitations that make them impractical for supporting implementation of real software systems. The first arises from how information is communicated between program segments. Each segment exports facilities that can be used by others, such as types, variables, constants, procedures and so on. The entire collection of exported facilities from all the segments in the system is replicated on each workstation. This information is applied to checking the use of imported facilities within a segment — each imported facility must have been exported by some other segment, and its use must be consistent with the definition in the segment that exported it. This global information is stored as an *aggregate* attribute. An aggregate attribute consists of many components; for instance, a symbol table is usually defined by an aggregate attribute where each component corresponds to an entry for one symbol.

A well-known problem with aggregate attributes is that a change to one component of the aggregate results in the reevaluation of all attributes that depend on any component of the aggregate. For instance, a new variable declaration results in reevaluation of all variable references in the scope of the changed declaration. In the distributed environments, a change to an exported facility in one segment is propagated to all segments, including those that do not import the facility.

The second restriction imposed by our previous algorithms for distributed environments is that they assume only one version of each program segment. This is clearly inappropriate for large software systems. We present solutions to these two problems in the following two sections. The end result is a class of distributed language-based environments which are practical candidates for real software development and maintenance.

3. Selective Propagation of Attributes

In this section we describe a refinement of the algorithms presented in section 2.2 where a change in one program segment is propagated to a second segment only if the latter actually uses the changed information. Our work is based on *finite functions*, a new type for aggregate attributes proposed by Hoover which, together with a modified attribute evaluation algorithm, reduces the overhead caused by aggregate attributes in a single-user environment [9]. This is one of several mechanisms proposed in the literature to solve the aggregate problem [12, 13, 5]. We base our approach on Hoover's work because, unlike the others, it solves the problem in the single-user environment within the framework of the attribute grammar formalism.

The asynchronous nature of changes in a distributed environment is the root of a fundamental difference between our work and that of Hoover. We use *finite relations*[1] to represent inter-segment aggregate attributes, rather than functions. The reason is that without synchronization (which as we said before requires too high a price), it cannot be guaranteed that the same component of the aggregate will not be defined simultaneously by more than one programmer. This is true in any multiple-user environment, whether running in a distributed or time-sharing system. To simplify the exposition of the new ideas in this section, we discuss only the changes to the attribute evaluation algorithm to handle attributes whose types are finite relations. Hoover's work can be applied directly to attributes whose propagation is fully contained within a segment, and the combination of his work with ours to reduce the aggregate overhead both within and among the segments is straightforward.

3.1. Definition of Interface Aggregates

An *interface* aggregate attribute is a collection of components from various program segments. These attributes capture the flow of information among the segments, and therefore among machines in the network. We introduce a new attribute type for interface aggregates: the *finite binary relation*. A binary relation on two sets D and R is a subset of $D \times R$, read the cross-product of D and R. Every finite relation type declaration must specify one element of R as the **bottom** element. A binary relation is finite if and only if the set $C = \{(d,r) \mid d \in D, r \in R,$ and r is not **bottom**$\}$ is finite.

We refer to finite binary relations simply as finite relations, since all aggregate values of interest in a programming environment are keyed lists that are binary mappings from a domain (the type of the key) to a range (the information stored for this key). Typically, each module that is part of a (sub)system contributes one component to the aggregate attribute of its parent. The domain D of the relation is the set of module names. The range R is the set of symbol tables for the modules' exported facilities, needed to check consistency between the definition

---

[1]Throughout this paper, the term "relation" denotes the mathematical concept, and not the relations of the database world. This point is noted to distinguish our work from previous research in programming environments where the attribute grammar formalism is augmented with relational database constructs [10].

and uses of these facilities.

The following operations are defined on a finite relation R:

- MAKENULL(R): Makes a null aggregate value. A declaration of an attribute of finite relation type implicitly calls this operation to initialize the attribute to the null value. Typically used to initialize an empty symbol table for a new scope.
- ASSIGN(R, d, r, *<attribute name>*, *<error string>*): Assigns $R \cup \{(d,r)\}$ to R. If the number of components in the aggregate R whose key is equal to d becomes greater than one, then the attribute instance denoted by *<attribute name>* for each program segment defining these duplicate components is set to *<error string>*. A change in a segment's error attribute is propagated to the segment by the usual propagation algorithm. The error string is displayed within the segment's text as indicated in the language specification.[2] Typically adds a new module to the symbol table.
- COMPUTE(R, d): If $(d,r) \in R$ and there is only one component in R whose key is d, returns r. If there is more than one component with the same key d, returns special value **multiple**. If $(d,r) \notin R$, returns **bottom**. Typically looks up a module name in the symbol table.

These are the only operations by which attributes of finite relation type may be manipulated. The reason for this restriction is that the set of segments that use a particular component in an interface aggregate, which is exactly the set of segments that should be informed of a change in this component, is derived automatically from these operations. This is explained in the next section.

Some new notation is needed for specifying distributed language-based environments. We extend the attribute grammar notation as follows:

- Non-terminal symbols in the grammar that can derive segments of the program for separate editing, on the same or different machines, are marked with the keyword **distributable**.
- The "**set of** <non-terminal symbol>" construct is provided to allow a symbol (the left-hand side of such a production) to derive an unordered set of elements. The non-terminal symbol on the right-hand side of the production must be distributable. This rule is more appropriate for describing lists whose elements are distributed than the usual tail-recursive method.

Aggregate attributes whose types are finite relations can only be associated with symbols of the grammar that derive productions by the **set of** construct. If the attribute grammar contains the production "X ::= **set of** Y", then an aggregate attribute associated with grammar symbol X is constructed by means of the ASSIGN operation with two synthesized attributes (one attribute for d and one for r) and one inherited attribute (for the error attribute) from each member of the set derived from the grammar symbol Y.

Figure 3-1 gives an example of the specification of a simple modular language. A program in

[2]Note that even though these segments have the same name, the APL can distinguish between them by means of the channel through which the editor and APL communicate, which is unique for each executing editor.

this language consists of a set of modules. The exported facilities of a module are stored in the attribute *exports* associated with each module. The facilities exported by all the modules are collected in the interface aggregate attribute, *allexports*, associated with the entire program. This is accomplished by means of the ASSIGN operation. A module references facilities exported by other modules through the import statement. An import statement names the module from which the facility is imported, and the facility itself. The import statement creates a use of the component of the *allexports* aggregate identified by the imported module in the analysis to check the legality of the import statement (the imported module must exist and must be unique, and the imported facility must be exported). The COMPUTE operation finds the appropriate component.

Interface aggregate attributes are cached in the APL layer on each machine. The structure of the attribute cache follows the hierarchical relationships among segments. Aggregate attributes are represented by AVL trees, ordered by the key of the components of the aggregate. This is typically the module name. An AVL tree is a height-balanced binary tree representation of a linear list that has $O(\log n)$ worst-case time complexity for list operations (such as insert, delete, member) on a list of n items.

3.2. Construction of Use Lists

Other attribute instances refer to components of aggregates defined by finite relations by means of the COMPUTE operation. The first argument of this operation indicates the aggregate from which the component is to be selected; this aggregate is accessed via an *upward remote reference*. An upward remote reference allows a non-local reference to an attribute of a different production p that necessarily occurs above the production where the reference is made in any parse tree derived from the grammar. The concept of "upward remote references" originated in the Cornell Synthesizer Generator [26]. We use the same notation for upward remote references: $\{id.attr\}$, where *id* is the name of a grammar symbol of the production p, and *attr* is an attribute name associated with this symbol. For example, the operation COMPUTE($\{X.a\}$, d) returns the value r of the component (d,r) in the aggregate a associated with the non-terminal symbol X. This is typically used to access the symbol table associated with the enclosing scope.

In each program segment, the set of references to interface aggregate components can be built from the COMPUTE operations within that segment by the attribute evaluation algorithm, as follows. If an attribute evaluation contains a COMPUTE operation, a *demand* is placed on the component of the aggregate identified by the second argument. As mentioned previously, the entire aggregate is stored in the APL in each machine. It is not desirable to copy the entire aggregate to each segment that has access to this aggregate (that is, the enclosed scopes) because a change to a component in the aggregate would trigger an attribute evaluation for each segment, independent of whether the segment references the changed component or not. This is one of the shortcomings of our previous distributed evaluation algorithm described in

```
root program;
distributable module;

/****************** attribute declarations  ******************/

Program { synthesized EXPORTAGG allexports; }

Module  { synthesized EXPORTTBL exports;
          synthesized ID name;
          inherited STRING error; }

/********* abstract syntax and semantic equations ************/

Program ::= set of Module;
        { for each Module$i, where 1 <= i <= |set of Module|
              assign($$.allexports, Module$i.name,
                          Module$i.exports, Module$i.error,
                          "<-- duplicate module");
        }

Module ::= Name Export Import Decl Body;
        { $$.name = Name.id;
          $$.exports = ... ;
        }

Import ::= ModuleId VarId
        { local STRING error1, error2 = "";
          local EXPORTENTRY single_module_exports;

          single_module_exports =
                compute({program.allexports}, ModuleId.name);
          if single_module_exports = bottom then
                error1 = "<-- imported module unknown"
          else if single_module_exports = multiple then
                error1 = "<-- imported module duplicate"
          else if varid.name not in single_module_exports then
                error2 := "<-- variable not exported";

        }

/***************** attribute type definitions *****************/

EXPORTTBL : NULLEXPORTS()
        | EXPORTPAIR( EXPORTENTRY EXPORTTBL )
        ;

EXPORTENTRY : (ID TYPE ... )

EXPORTAGG: ID cross EXPORTTBL bottom NULLEXPORTS;
```

Figure 3-1: Specification of a simple modular language

section 2.2. Instead, we keep copies of only those components actually referenced by COMPUTE operations within a particular segment in an attribute associated with that segment. This is called the *uses* set of the segment.

Thus, the *uses* set of a segment is a subset of the aggregate. It is also organized as an AVL tree. However, for each element in *uses*, there is a list of references to attribute instances within the segment that use that particular component. There is also a pointer from each attribute instance back to the *uses* set. This is needed by the propagate algorithm described in

the next subsection.

The *uses* set of each segment residing on a workstation is communicated to the local APL. This information is used by the APL to determine which changes in component values to propagate to each local segment. The information from each local segment's *uses* set is used to build for each unique component in the aggregate the set of segments that should receive propagations if the value of that component changes. This is called the *used-by* set of the interface component. Thus, the APL indirectly links symbol definitions to their references and vice versa.

For the example of figure 3-1, the components of the interface aggregate attribute *allexports* are the exported symbol tables of each module in the system. The *used-by* set of the component for a module M is the set of modules that import facilities from M. Whenever one of the exported facilities of M changes, the change is propagated to all modules in the *used-by* set of the component of M. We can refine our notion of use-lists so that a *used-by* set is kept for each facility exported by M. This improves the efficiency of the attribute propagation algorithm even further since a change to an exported facility results in propagations only to those segments that reference the particular facility. This is accomplished in the general case by extending the finite binary relations to n-ary relations, and the key used by ASSIGN and COMPUTE to $n-1$ prespecified fields.

We give a simple calculation to compare the efficiency of the distributed attribute propagation algorithm with and without selective propagation.
Let

m = the number of modules in the system,

e = the average number of exported facilities per module,

i = the average number of imported facilities per module,

p = the average number of imported modules per module, and

c = the average number of changes to an exported facility throughout the lifetime of the system.

In our previously published algorithms, where the interface aggregate problem had not yet been solved, each module would receive $m \times e \times c$ propagations. Using finite relations for the type of interface aggregate attributes, which associate *used-by* sets with each module's exported symbol table, results in $p \times e \times c$ propagations per module. Note that p is usually much smaller then m. With *used-by* sets associated with each exported facility individually, this is improved even further to $i \times c$ propagations to each module.

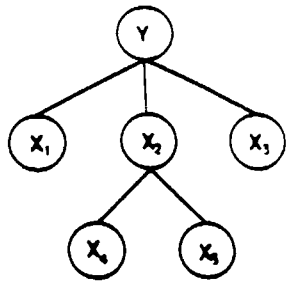
We now describe incremental algorithms to maintain the *uses* and *used-by* sets after each edit operation.

3.3. Algorithms to Maintain Use Lists

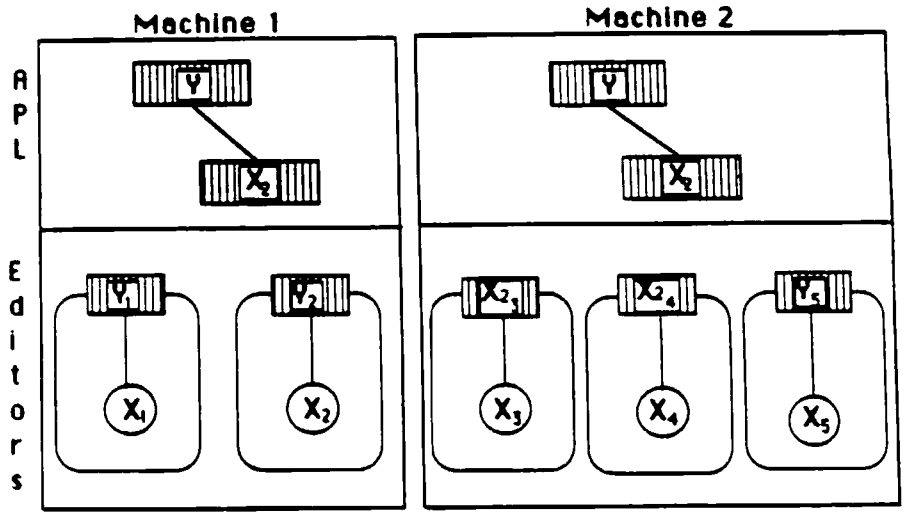
Figure 3-2 shows the physical representation of a software system, Y, whose modules are distributed among several workstations. The distributable segments are X_1, X_2, X_3, X_4 and X_5, where X_1 and X_2 are edited on Machine 1 and X_3, X_4 and X_5 on Machine 2. The interface attributes of system Y and subsystem X_2 are replicated in the APL at each workstation to support the multiple-level uses sets of the interface attributes that are finite relations. The APL associates each distributable segment with its own *uses* set, which is the appropriate subset of the *uses* set for its parent segment, and so on.

```
Y ::= set of X;
distributable   X;
X ::= <module spec>
    | . . . set of X . . .;
```

(a) Language specification fragment



(b) Software system structure



(c) Physical representation

Figure 3-2: Distribution of interface attributes

An attribute that is defined by a COMPUTE operation depends on two other attributes: (1) the aggregate (this is the first argument) and (2) the key of the desired component of the aggregate (the second argument). Figure 3-3 illustrates the dependency graph of an attribute defined by a COMPUTE operation and associated with the parse tree node T in segment X_1.

3.3.1. Change to a Segment's Uses Set

A segment's *uses* set changes if (1) a new use site is added, (2) a use site is removed, or (3) the key of a use site is changed. Removing a use site occurs if either (a) the parse tree node containing the key attribute identifying the component of that reference is deleted, or (b) the subtree decorated with the attribute instance that created the use is deleted. In our example language, these correspond to the module name and the enclosing import statement.

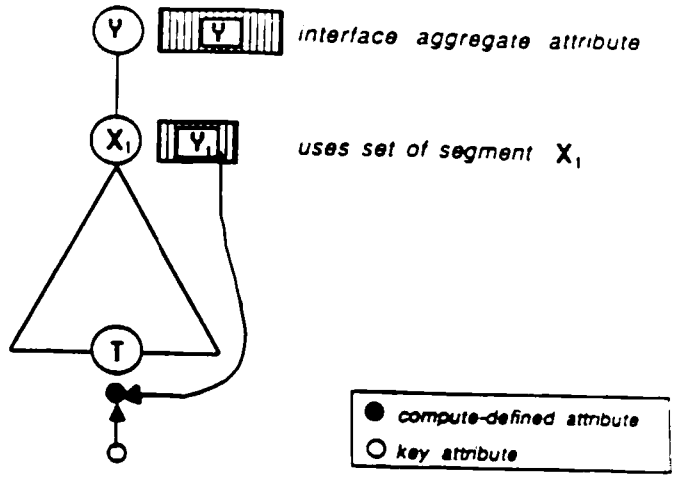


Figure 3-3: Dependency graph for attribute defined by COMPUTE operation

respectively. In the first case, deletion of the node containing the key leaves a *null* value for the key, so this becomes the same as changing the key of a use site (case 3). We present two algorithms for maintaining a segment's *uses* set, illustrated in figures 3-4 and 3-5 below.

The algorithm shown in figure 3-4 is a modified attribute evaluation algorithm that recognizes a new use of an interface aggregate, either by addition (case 1) or by change in value of the key of an already existing use (case 3).

A new algorithm for deleting a subtree, shown in figure 3-5, updates the set of interface components used in a segment, the segment's *uses* set. If the subtree being deleted contains a reference to an interface component, that reference is removed from the component. If the component has no more references, it is removed from the segment's *uses* set; the APL is notified so that the segment's name is removed from the *used-by* set of the component in the APL.

3.3.2. Change to Component's Used-by Set

The *used-by* set for each aggregate component in the APL, indicating which local segments use a particular component of the aggregate, is affected by the two functions `get_value_from_apl(aggregate,key)` and `remove_use_from_apl(aggregate,key)` invoked in the algorithms of figures 3-4 and 3-5 above. The former adds the segment name of the segment that issued the call to the *used-by* set of the component whose key is specified. The latter removes the segment name from the *used-by* set. (Note that the name does not have to be an argument to the call since each segment communicates with its local APL over a unique channel, which uniquely identifies the segment.)

There are two problematic situations: (1) the key specifies a multiply defined component, or (2) the key specifies an undefined component. If the call to `get_value_from_apl` specifies a multiply defined component, then the segment name is added to the *used-by* set of any component with the specified key. Multiply defined components, as well as the program

```
/* Attribute instances defined by COMPUTE(aggregate,key) have an */
/* additional field, backptr, pointing back to the uses set of   */
/* the aggregate component specified by key argument to COMPUTE. */

function eval (ai: attribute instance): attribute value;
begin
(1) if ai is defined by compute(aggregate,key) then begin
        /* Case (1): a new use site not yet */
        /* added to multiple-level uses set */
(2)     if backptr of ai = nil then
            /* first reference to key within segment */
            /* add entry for key to local aggregate */
(3)         if key not in local aggregate at root of segment then begin
(4)             entry = get_value_from_apl(aggregate,key);
(5)             add entry to local aggregate;
(6)             add ai to uses set of entry;
(7)             set backptr of ai to entry;
            /* already references to same key within segment */
            /* reuse entry for key in local aggregate */
(8)         end else begin
(9)             entry = get_value_from_segment(aggregate,key)
(10)            add ai to uses set of entry;
(11)            set backptr of ai to entry;
(12)        end
        /* Cases (2a) and (3): an old use site */
        /* whose key may have changed */
(13)    else begin
            /* get previous entry from local aggregate */
            entry = follow backptr of ai;
            /* same key */
(14)        if key = key of entry then
(15)            do nothing;
            /* different key */
            /* remove from uses set of previous entry */
            /* add to uses set of new entry */
(16)        else begin
(17)            remove ai from uses set of entry;
(18)            set backptr of ai = nil;
(19)            Do lines (3) - (12);
(20)        end;
(21)    end;
    /* evaluate attributes not defined by COMPUTE */
(22) else begin
        ...
    end;
end; /* of eval */
```

Figure 3-4: Attribute evaluation algorithm

segments that define them, are treated as erroneous; the semantics of the ASSIGN operation require the APL to effectively remove erroneous segments from the program until the conflict is resolved. We describe below how to handle the deletion of a component such that the correct action is taken when a key that was multiply defined becomes unique. If `remove_use_from_apl` specified a multiply defined key, then the *used-by* set of each component with that key must be searched to delete the segment that invoked the function.

If `get_value_from_apl` specifies a key that is not defined in the aggregate in the APL, bottom is returned. A component is added to the APL aggregate with the specified key and the value bottom (if such a component does not already exist), and a *used-by* set for it is

```
procedure delete_subtree(r: treenode);
    begin
        for each attribute instance, ai, associated with every
           treenode in subtree rooted at r, excluding r, do begin
                if ai is defined by compute(aggregate,key) then begin
                    /* get entry for key and */
                    /* remove attribute from uses set */
                    entry = follow backptr of ai;
                    remove ai from uses set of entry;
                    /* last reference to key within segment */
                    if uses set of entry = nil then begin
                        remove entry from local aggregate;
                        remove_use_from_apl(aggregate, key);
                    end; /* of IF */
                end; /* of IF */
        end; /* of FOR */

        /* free storage taken up by r */
        ...

end; /* of delete-subtree */
```

Figure 3-5: Subtree deletion algorithm

created (if the component was already in the aggregate, the segment name is added to *used-by*). This is necessary to handle the correct propagations if a component with that key is defined later on. We mark such components as "demanded-but-undefined", and distinguish them from regularly defined components. Typically, defined components correspond to defined symbols, and the demanded-but-undefined ones are references to as yet undefined symbols.

3.3.3. Change to Component's Value

A component is changed by an ASSIGN operation, according to the dependency graph illustrated in figure 3-6.

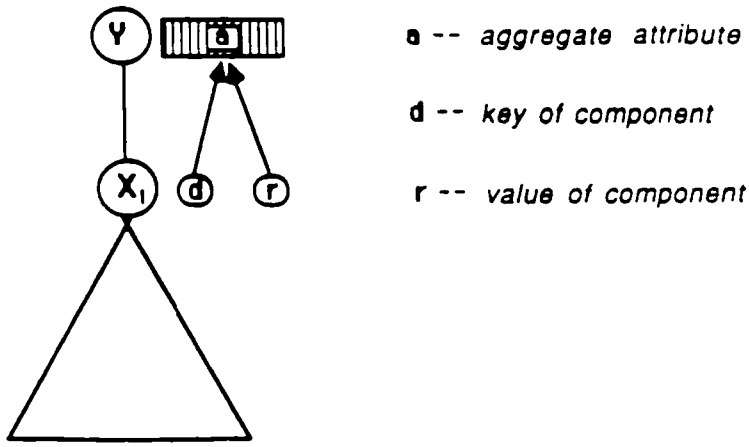


Figure 3-6: Dependency graph for attribute defined by ASSIGN operation

Since in the physical representation, *d* and *r* are attributes in the segment (the key and value of the component, respectively) and the aggregate *a* is stored in the local APL, a change to either

d or *r* results in a change to a component of the attribute instance *a* in the APL, which in turn causes propagations to affected segments. The other way the component can change is if the segment containing the attribute instances *d* and *r* is deleted. The following algorithms handle changes in the definitions of aggregate components.

1. Change from *r* to *r'* in segment — The component (*d*, *r'*) is transmitted from the segment where the change occurred to its local APL, which then broadcasts it to all other APLs. Each APL propagates the component with the changed value to segments that use that key, indicated by the component's *used-by* set. This arises, for example, when the exports list of a module is modified.

2. Definition of new component — This happens when a new key is defined, i.e., a new segment is created. The new component is broadcast to each APL.
   - If the key is already in a defined component of that aggregate, then the error attribute is set, and the component (*key*, **bottom**) is propagated to all uses of that key.
   - If there is a demanded-but-undefined component with the same key as the newly defined component, then mark the component as defined. Propagate the value of the newly defined component to all reference sites as indicated by the components *used-by* set.
   - If no component with the specified key exists, then add the component to the aggregate, initializing its *used-by* set to empty.

3. Deletion of component from aggregate —
   - If the component is removed because the key *d* became undefined, then
     - If this was a duplicate component, concatenate *used-by* set for this component with the *used-by* set of another component with the same key. Then remove the component. If only one component is left with the key of the deleted component, then propagate the remaining component to the segments on the *used-by* set. This is appropriate, for example, when one instance of a multiply defined module is removed.
     - If the component was not a duplicate, then mark the component as demanded-but-undefined, changing the *r* value to **bottom**, and propagate to the *used-by* set.
   - If the program part containing the attribute instances *d* and *r* is removed, then the delete subtree algorithm operates similarly to how it handled deleting a subtree containing a reference site. However, if *d* and *r* are associated with the root of the distributable segment, they cannot be deleted unless the entire segment is removed. In practice, this would mean deleting the file containing the module from the file system, so it is more complicated than the other case.

4. Dealing with Multiple Versions and Configurations

This section describes how the algorithms given in the previous section are augmented to cope with more than one version of each program segment, and consequently, more than one system configuration. In this context, program segments are almost invariably modules, so we refer to them as such in this section. Controls are needed to reduce the chaos that can result if

programmers were to work on versions and build systems without any communication and coordination between them. Managerial controls, such as controls imposed by a chief programmer on what the other team members can change, and when and how the system is built, are insufficient [3]. The environments described here provide an automated approach that can support and enforce managerial directives.

The attribute grammar specifications determine the exact functionality that a generated environment supports, but the following is the kind of support we have in mind:

1. Static semantic analysis of the modules that comprise the software system, and
2. Code generation for error-free modules or fragments thereof.

The environment is capable of performing these functions after every edit operation. This is the default mode of operation, and the hardest to support. However, the programmer can select other modes of operation where the analysis and code generation are performed less frequently. For example, the programmer can set environment options to request notification of changes at the end of each editing session, or only when he issues a special "get changes" command.

Code generation by a compiler generated from an attribute grammar is usually accomplished by having a code attribute associated with the root of the parse tree, where this attribute contains the generated code for the entire program. This is grossly inefficient in an incremental environment, since an edit to the program necessarily requires recomputation of code attributes all along the path from the point of modification to the root. Incremental generation of code can be performed efficiently if the code attributes containing fragments of the generated code are dispersed throughout the tree and coalesced only when they are needed for system build. This makes it feasible to update the code attributes after each (or a number of) edit operation(s). The environment can evaluate the code attributes opportunistically, that is, when it is not running the normal attribute evaluation algorithm. This delay in attribute evaluation is acceptable because the results of these computations are not visible to the programmer in the way that error messages resulting from semantic analysis are.

Since the modules of the system are distributed among different machines in the network, our environments do not automatically link the code objects into one executable image. This would require the remote copying of the code from other nodes in the network, a very expensive operation, and therefore not suitable to be automatically invoked by the environment. Pfreundschuh also makes use of attribute grammars for specifying system builds [22]; her work differs from ours because the system build is not applied incrementally, but only at user command after the modules of the system have been analyzed. Pfreundshuh's work relies on our previous algorithms for distribution capabilities.

4.1. Version and Configuration Control

Our environments utilize an external mechanism for storing the different versions of modules. We use RCS, but there are other candidates such as SCCS, the History Manager of Apollo's Domain™ Software Engineering Environment (DSEE™) [19], etc. RCS keeps track of *revisions* of the modules by storing the differences (called deltas) between successive revisions. Parallel lines of development require the revision relationship to form a tree, where parallel versions are represented as paths in a subtree rooted at the common ancestor revision.

RCS and other source version control facilities provide additional services, notably a reserve/replace mechanism that allows a programmer to work on a revision without interference from other members of the team. Revisions are immutable; to make a change, the programmer reserves the module. If this branch is not already reserved by another programmer, he retrieves a copy of the latest revision of the module, makes any changes on that copy, and when finished puts the module back under control of RCS. Once checked in, the revision can no longer be modified. While being edited using our environment, the copy is called a *working copy*.

Another service is the naming of revisions. RCS provides a default naming of revisions: 1.2.3.1 means the first revision on the third branch of the second revision of this module. Alternatively, the user can give symbolic names to the revisions, and then use the symbolic names to identify what he wants to reserve/replace.

Our distributed environments provide a configuration manager to control the activities needed to "build" a software system from its modules. The facilities provided by the configuration manager, similar to those provided by DSEE, require the following information to be maintained by the environment:

- A *system model* [18] describing the structure of the software system, that is, (1) which modules make up the system, and (2) the interdependencies among these modules.
- A *configuration thread* for selecting particular versions of each module in the system model. Options for version selection allow the selection of the latest revision, a named revision, or a revision that satisfies certain properties (e.g., the latest revision targeted for a VAX).

The configuration manager guarantees that a consistent system is (incrementally) built: (1) the modules are internally consistent, and (2) the interfaces between the modules are consistent. We now describe how these version and configuration control facilities are integrated with our distributed attribute grammar evaluation algorithm.

4.2. System Model

For our environments, a system model specifies the modules that comprise the software system. The dependencies among the modules, which determine which modules must be reanalyzed after a change to one of them, are captured by the *used-by* sets. Recall that each component in an interface aggregate attribute has an associated *used-by* set that contains all

the modules that use that component, and therefore should receive propagations of changes to that component. Note that the system model does not contain information about the manufacturing process, that is, the commands that must be executed to go from a *primitive* component (a source module written by a programmer) to a *derived* component (its object code) [2]. In our environments, translation rules for going from primitive modules to derived objects are given by the AG specification. These rules are effectively the same as manufacturing steps, but incremental.

We provide the programmers with a *system structure editor* (SSE) for describing a system model, separate from the editor for constructing modules (that is, program segments). The SSE is a language-based editor derived from the same language description, but all the pieces in the grammar not dealing with distributable modules have been filtered out. What remains is an editor for describing the modular structure of the system. For example, figure 4-1 illustrates the system structure for a typical compiler as it is displayed by the SSE.

```
program Compiler is composed of {
    module Lexical is composed of {
        module GetToken is composed of
            <module set>;
        module <name> is composed of
            <module set>;
        <module set>
    }
    module Analyzer is composed of {
        <module set>
    }
    module CodeGenerator is composed of {
        <module set>
    }
```

Figure 4-1: Example of system model

Editing a system model results in the creation of a new working copy of the system model. This change is propagated to other programmers in exactly the same way as changes in a module — the programmer might want to be notified whenever the current system model changes, or he might want to continue using the original one. If he chooses the former, and the change was an addition of a new module to the system, then he must select which version of the new module he wants. We describe the mechanism for accomplishing this in the following subsection.

We do not synchronize changes to the system model between the programmers. In practice, each programmer may have his own distinct system model. However, conflicts among separate system models are flagged with error messages using the same incremental attribute evaluation algorithm as for consistency checking among modules (i.e., program segments). Here, a conflict is defined to mean any differences, rather than the direct contradictions required for modules. A programmer can choose to continue using his own, or browse through the competing models to select a new one. Access controls could be implemented on

top of this system restricting the changing of the system structure to a few "trusted" persons in the team.

4.3. Configuration Thread

When there are multiple revisions of each module, each programmer must specify which particular ones should be used in his configuration of the system. A programmer specifies a configuration thread by a fill-in-the-blanks form provided by the environment, with the collection of modules determined by the programmer's current system model. The following possibilities for version selection are available:

- Latest working copy of a module; if there are multiple branches of development of the module, the branch must be specified.
- Latest checked-in revision of a module; same as above for multiple branches.
- Specification of a particular revision of the module. The details about how a particular revision is specified, for instance, by name or by revision number, depends on the source version control system. For RCS, where revision names are assigned only at check-in, this implies a checked-in revision.

The notification of changes in a module range from full notification after every change for the first possibility, changes done between last check-in and previous check-in all at once for the second possibility, and no change notification for the third possibility, since checked-in revisions are immutable. For the first two, notification of changes can also be set by the programmer to happen on command or at the end of each editing session, as described earlier.

The configuration thread is dynamic if it contains modules selected according to the first two possibilities; the system configuration changes as the modules evolve. The environment can support various levels of interaction between the programmers. During the initial development effort, when too much interaction would be detrimental because everyone is changing everything, each programmer can select null revisions for the other modules in the system, effectively preventing notification of changes. Later on, when baseline revisions are defined, he can select these to check consistency of his module only against "correct" modules. During system integration, on the other hand, close interaction is required, and at this time programmers might select latest working copies of the modules, or latest checked-in revisions for the more cautious approach.

4.4. Attribute Propagation Algorithm

The attribute propagation algorithms described in section 3 are modified to take the programmer's configuration thread into account. The change is in how the *used-by* sets are computed. Now we have multiple revisions, and possible branches among the revisions, of each module. This requires that the APL's attribute cache contain corresponding revisions of the interface aggregate attributes. Figure 4-2 shows the APL cache organization for a system with multiple versions of two modules, *A* and *B*. Module *B* has branched into two parallel lines of development.

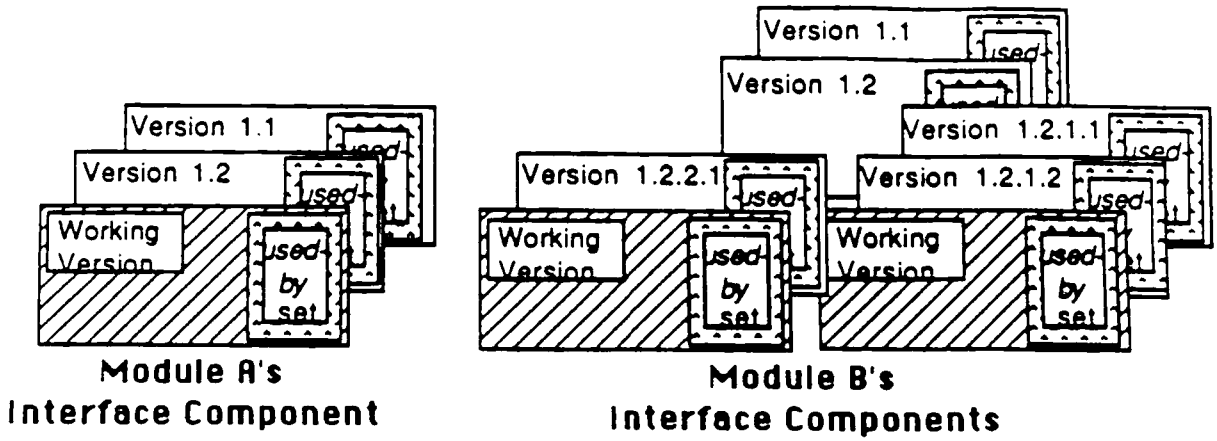


Figure 4-2: APL cache organization

A *used-by* set is associated with each revision of an interface component. The revisions of the interface attributes in the APL are also immutable. Working versions exist for each component in the aggregate; changes to the interface attribute resulting from an edit operation are made to these working versions. The operation of checking in a module to RCS also saves the version of the interface component defined by that module. When there is no working copy derived from a particular revision, the working version of its interface attributes is a copy of the last checked-in revision.

Since code is generated for modules that have been (incrementally) analyzed and found error-free, the environment must guarantee that the revisions of the modules that they have been checked against exist. This is not as strange as it sounds; remember that one of the possibilities for version selection is to receive notifications of changes after every change. Consider the case where a module *M* is checked against the latest working copy of module *N*, analyzed to be error-free, and checked in. Now *N* is changed in a way that causes an error in module *M*, and then *N* is itself checked in. Thus, there is no checked-in version of module *N* corresponding to the version that *M* was checked against. So the code generated for *M* is useless, since it cannot be used to build the system without also having the code for the transient version of *N*.

To solve this problem, the environment checks in the module after each edit that causes a change in the module's interface component. This is done only for working versions that have non-empty *used-by* sets. We call these revisions *system-induced*, and distinguish them from the revisions checked in by the programmers.

System-induced revisions can be garbage collected as follows. For each module, say module *X*, the environment keeps track of the check-in times of all modules that use *X*'s exported facilities and that use the latest version of *X*. These modules are the ones in the *used-by* set of

some interface component for the latest version of X. We only need to keep a system-induced revision if it corresponds to the state of X when one of these modules that use X was checked in, or if one of these modules is a working copy (since this working copy may later be checked in while it still uses this version of X). All such revisions are marked, and all unmarked system-induced revisions of X discarded. This can be optimized by considering at each garbage collection only the system-induced revisions of X created since the last garbage collection, plus the system-induced revisions of X at that time with working copies of other modules in their *used-by* sets.

The garbage collection algorithm assumes a reliable network; if a new module is created, starts using the latest version of X, and becomes partitioned from the machine where X resides before it is added to X's used-by set, then a system-induced revision corresponding to the state of X that this new module is using may be incorrectly discarded. The *fossil collection* algorithm used by the Time Warp System [11] (a distributed simulation mechanism) for recovering storage associated with simulation times so far in the past that they are no longer relevant may be applicable to the unreliable case.

4.5. Space Optimization

The APL must keep a different revision of each interface attribute corresponding to every checked-in version of each module, as well as the latest attribute value for each working module. For a practical environment, it is crucial to store these revisions efficiently. We represent interface attributes as applicative AVL trees [21]. An AVL tree is applicative if the operations for manipulating the tree do not change the tree, but produce a new "copy" of the tree that is changed. The algorithm does not copy the entire tree, only the part of the tree that changed (delta from previous revision). Therefore, list operations on applicative AVL trees have $O(\log n)$ space complexity, making them an attractive representation for storing multiple revisions of interface attributes [7].

5. Conclusion

We have presented algorithms that extend our previous work in distributed language-based environments by making them appropriate for the real world, where efficiency matters and there are multiple versions of modules and system configurations. The primary contributions of this paper are:

- Significant improvement in the efficiency of the distributed attribute propagation algorithm.
- Supporting multiple versions of the modules of the software system being developed or maintained.
- Handling multiple versions of the system model, where each programmer can select which system model should be in effect with respect to his efforts.

The environments described in this paper support software systems composed of modules in a hierarchical structure, where groups of modules form subsystems, which in turn form other

subsystems, and eventually the system itself. However, they do not solve the general case of nesting. We are currently working on algorithms to fully support arbitrarily nested program segments within block-structured constructs, such as internal packages in Ada.

Acknowledgements

Simon Kaplan worked with us on the development of the previously published algorithms referred to here. Wenwey Hseush, Joe Adipietro, Katherine Feldman, Rowan Maclaren, Luke McCormick, Michal Melamede, Linda Mischel, Michael Tavis, Su-Chuan Tsai, Chen Yen, and Bulent Yener participated in their implementation as part of the Mercury system. This implementation is a modification of the Synthesizer Generator, which was developed at Cornell University by Tom Reps and Tim Teitelbaum; we acknowledge their effort in its development, and also wish to thank Tom Reps for his assistance in converting the Generator.

This research is supported in part by grants from AT&T Foundation, IBM, Siemens Research and Technology Laboratories, and New York State Center of Advanced Technology — Computer and Information Systems, and in part by a Digital Equipment Corporation Faculty Award.

References

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

[2] Ellen Borison.
A Model of Software Manufacture.
In Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (editors), *Advanced Programming Environments*, pages 197-220. Springer-Verlag, Berlin, 1986.

[3] Frederick P. Brooks, Jr.
The Mythical Man Month.
Addison Wesley, Reading, MA, 1982.

[4] David R. Cheriton.
Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design.
In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 190-197. IEEE Computer Society, May, 1986.

[5] Alan Demers, Anne Rogers and Frank Kenneth Zadeck.
Attribute Propagation by Message Passing.
In *SIGPLAN 1985 Symposium on Language Issues in Programming Environments*, pages 48-59. Seattle, WA, June, 1985.
Proceedings published as *SIGPLAN Notices*, 20(7), July, 1985.

[6] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software, 1(4)October, 1984.

[7] Christopher W. Fraser and Eugene W. Myers.
An editor for Revision Control.
ACM Transactions on Programming Languages and Systems, 9(2):257-276, April, 1987.

[8] Harald Ganzinger, Knut Ripken and Reinhard Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In *Information Processing 77*, pages 535-540. North-Holland Pub. Co., New York, 1977.

[9] Roger Hoover and Tim Teitelbaum.
Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars.
In *SIGPLAN '86 Symposium on Compiler Construction*, pages 39-50. Palo Alto, CA, June, 1986.
Proceedings published as *SIGPLAN Notices*, 21(7), July, 1986.

[10] Susan Horwitz and Tim Teitelbaum.
Generating Editing Environments Based on Relations and Attributes.
ACM Transactions on Programming Languages and Systems, 8(4):577-608, October, 1986.

[11] David Jefferson.
Virtual Time.
ACM Transactions on Programming Languages and Systems, (3)July, 1985.

[12] Gregory F. Johnson.
An Approach To Incremental Semantics.
PhD thesis, University of Wisconsin at Madison, 1983.

[13] Gregory F. Johnson and C.N. Fischer.
A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors.
In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 141-151. January, 1985.

[14] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiple-User Distributed Language-Based Environments.
IEEE Software, :58-67, November, 1987.

[15] Gail E. Kaiser and Simon M. Kaplan.
Reliability in Distributed Programming Environments.
In *Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 45-55. Kingsmill—Williamsburg, VA, March, 1987.

[16] Simon M. Kaplan and Gail E. Kaiser.
Incremental Attribute Evaluation in Distributed Language-Based Environments.
In *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 121-130. Calgary, Alberta, Canada, August, 1986.

[17] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory, 2(2):127-145, June, 1968.

[18] Bulter W. Lampson and Eric E. Schmidt.
Organizing Software in a Distributed Environment.
In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1-13. San Francisco, CA, June, 1983.
Proceedings published as *SIGPLAN Notices*, 18(6), June, 1983.

[19] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[20] James G. Mitchell, William Maybury, and Richard Sweet.
Mesa Language Manual, Version 5.0.
Technical Report, Xerox Palo Alto Research Center, April, 1979.

[21] Eugene W. Myers.
Efficient Applicative Data Types.
In *Eleventh ACM Symposium on Principles of Programming Languages*. January, 1984.

[22] Mary Patricia Pfreundschuh.
A Model for Building Modular Systems Based on Attribute Grammars.
PhD thesis, University of Iowa, December, 1986.

[23] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems, 5(3):449-477, July, 1983.

[24] Thomas Reps.
Generating Language-Based Environments.
M.I.T. Press, Cambridge, MA, 1984.

[25] Thomas Reps.
Private communication.
June, 1986

[26] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator Reference Manual
Cornell University, Ithaca, New York, 1987.

[27] M. J. Rochkind.
The Source Code Control System.
IEEE Transactions on Software Engineering, SE-1:364-370, 1975.

[28] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM, 24(9)September, 1981.

[29] Walter F. Tichy.
Software Development Control Based on Module Interconnection.
In *4th International Conference on Software Engineering*. September, 1979.

[30] Walter F. Tichy.
RCS — A System for Version Control.
Software — Practice and Experience, 15(7):637-654, July, 1985.

[31] Niklaus Wirth.
Programming in Modula-2.
Springer-Verlag, New York, 1982.

The authors can be contacted at Department of Computer Science, 450 Computer Science Building, Columbia University, New York, NY 10027.