Generation of Distributed Programming Environments

Gail E. Kaiser
Simon M. Kaplan*
Josephine Micallef

October 1986

CUCS-225-86

Abstract

This technical report consists of three related papers in the area of distributed programming environments. *Incremental Attribute Evaluation in Distributed Language-Based Environments* presents algorithms that extend existing technology for the generation of single-user language-based editors from attribute grammars to the cases of multiple-user concurrent and distributed environments. *Multi-User Distributed Language-Based Environment*, an extended abstract, provides additional information on how to apply the algorithms. *Reliability in Distributed Programming Environments* presents additional algorithms that extend our results to unreliable networks.

Part of this research was conducted while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA. *Dr. Kaplan is supported in part by a grant from the AT&T Corporation. Dr. Kaplan's address is Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana. IL 61801.

Incremental Attribute Evaluation in Distributed Language-Based Environments†

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

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

Summary

We present a model of distributed program editing and algorithms for the incremental checking of static semantic properties of modules that are at once semantically interdependent and physically distributed across a number of workstations connected by a high speed network. This makes possible the synthesis of modern program development hardware — workstations on high speed networks — and modern program development software — incremental, language-based program development systems — that until now have suffered from the problem of not being able to support incremental checking across distributed modules.

Introduction

This paper deals with the problem of performing incremental semantic[1] analysis across a program that is split into l modules that are distributed across a network of m machines, with n modules dormant (not being edited or inaccessible due to network or machine failure) and the other $l-n$ being concurrently edited.

A state–of-the-art program development environment provides a programming team with a number of workstations connected by a high speed network. Each module in the system under development is typically the responsibility of a programmer, and is resident on that programmer's workstation. The program under development is therefore distributed across the network.

State-of-the art software for program development support should also be provided. Such software will *incrementally* enforce a module's syntax and indicate any semantic errors as the program is constructed [Reps 84]. Syntax checking is an inherently *local*

† Appears in ACM Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada, August, 1986, pp. 121-130.

[1] In this paper, by "semantics" we mean static semantics.

process, but semantic checking may require information imported from another module [Kaiser 83]. If the semantic information in that module alters, all users of the module should have the change propagated to them so that their programs can be checked to see if they are still semantically consistent.

Currently no program development environments provide support for incrementally checking semantics across multiple distributed modules: they support only single monolithic programs (*i.e.* the entire program must be physically on the user's machine and not split into physically separate modules). There are several systems (most notably Cedar [Lampson 83]) that support the distributed program development effort of multiple programmers; however, these systems perform semantic checking at previously agreed checkpoints. This form of checking is not incremental.

Algorithms for incremental evaluation of semantic information are traditionally sequential in nature [Reps 82a, Johnson 85]. Having multiple programmers concurrently modify modules (and as a result propagate semantic information between the modules) requires an inherently concurrent view of semantic change propagation. We postulated that if we could derive a concurrent semantic evaluation algorithm for the single edit case, this would extend naturally to the multiple edit case and then to the distributed multiple edit case.

We begin by listing the original contributions of this paper. The next section briefly overviews attribute grammers, building language-based editors from grammars and sequential attribute updating strategies. Then we introduce and analyze our concurrent updating strategy for the case of a single edit. This is followed by a description and analysis of the multiple edit case, which in turn is followed by the distributed, multiple edit case. We relate this paper to other work in the field and end with some conclusions.

Original Contributions

- We introduce for the first time a model of incremental attribute evaluation amongst program modules distributed across multiple machines on a network.
- We provide algorithms for the concurrent incremental evaluation of attributes in a programming environment. We begin with a concurrent algorithm for incremental evaluation in a single user, single edit environment. We then expand this to handle multiple asynchronous edits, and then extend further to handle multiple asynchronous edits on modules that are distributed.

Overview of Attribute Grammars and Incremental Semantic Checking

An attribute grammar [Knuth 68] is a context-free grammar that is enhanced to be capable of expressing context-sensitive information by associating with every symbol in the context-free grammar a set of *attributes*, and associating with every production in the grammar a set of *attribute equations* that describe the relationships among the attributes. [Waite 84] and [Reps 82b] overview attribute grammars and their relation to construction of compilers and program development systems, respectively.

The context-free grammar can be used to parse a context-free sentence of the language specified by the grammar into *tree form*, where each node in the tree corresponds to an instantiation of a production of the grammar. Once this has been done, the attributes associated with each production can be added to the tree. Attribute flows will form a *directed graph* on the tree. A standard restriction employed by all incremental checking editors (and most compilers based on attribute grammars) is to consider only *noncircular* attribute grammars, *i.e.* those whose attribute flows form a *directed acyclic graph* (DAG) on the tree.

An attribute grammar specification can be used to generate a language-based editor that incrementally checks the consistency of a program. Edits in such a system are defined in terms of *subtree replacement* on the tree representation of a program. After each such replacement, the syntactic and semantic consistency of the program is reestablished.

After a single subtree replacement, the tree is syntactically consistent and all inconsistent attributes are located at the replaced node of the tree. A graph D of the dependencies (including transitive dependencies) amongst the attributes at that node, its parent, siblings and children is constructed from templates obtained by analyzing the grammar at the time that the editor was generated.[2] This graph has attributes at the vertices and dependencies shown by the edges. Because transitive dependencies are included, this graph can extend arbitrarily far on the tree. Any vertex in this graph with in-degree 0 is said to be *independent* because it cannot depend on any other inconsistent attribute. Because the attribute flows form a DAG, at least one such vertex must always exist.

An independent vertex is chosen and the associated attribute α reevaluated. If the value of the attribute has altered, D is examined to see if there are any attributes β dependent on α not already in D. If so, D is expanded to include *all dependencies* between each β and any attributes already in D, again including any transitive dependencies. The attribute α is removed from D, along with its associated edges, and the process is repeated until D is empty.

This algorithm is due to Reps [Reps 82a] and is asymptotically optimal in time; its time complexity is $O(|Affected|)$, where *Affected* is the set of attributes that change. The implied topological sort on D results in an attribute being reevaluated only when the reevaluation is guaranteed to yield its final value (because an attribute is only reevaluated after it becomes independent).

[2] In [Reps 82a] this graph is called the model.

Introducing Parallelism

We begin by proposing an algorithm that performs as many attribute evaluations in parallel as possible, by choosing the complete set of independent vertices at any time and evaluating all of them. This is a simple modification to the sequential optimal algorithm described above. Its major purpose is a *paradigm shift*; by introducing parallelism, we provide the ability to handle the interesting cases: multiple asynchronous subtree replacements (with their reevaluations) and distributed evaluations. The algorithm is:

```
startup($\mathcal{T}$, $\mathcal{R}$)
 let
  $\mathcal{T}$ = fully attributed tree
  $\mathcal{R}$ = node in tree with inconsistent attributes
  $\mathcal{S}$ = set of attribute instances
  $\beta$ = attribute instance
 in
   setup($\mathcal{T}$, $\mathcal{R}$, $\mathcal{S}$)
   $\forall$ $\beta$ in $\mathcal{S}$ do propagate($\beta$) od
   terminate

propagate($\beta$)
  let
    $\beta$ = attribute instance
    $\mathcal{S}$ = set of attributes
  in
    evaluate $\beta$
    if $\beta$ changed then expand($\beta$) fi
    remove($\beta$, $\mathcal{S}$)
    $\forall$ $\beta$ in $\mathcal{S}$ do propagate($\beta$) od
    terminate
```

Setup, *remove* and *expand* are all calls on an abstract data type $\mathcal{D}$, which maintains the attribute dependency graph (which may in practice be divided into a number of disjoint graphs).

Setup($\mathcal{T}$, $\mathcal{R}$, $\mathcal{S}$) takes as arguments a tree $\mathcal{T}$ and a particular subtree $\mathcal{R}$ of $\mathcal{T}$, which has inconsistent attributes at its root, constructs the dependency graph $\mathcal{D}$ and returns in $\mathcal{S}$ the attributes that are ready for evaluation because they are independent (they have in-degree 0 on the dependency graph).

Expand(β) takes as argument an attribute and expands the graph as described in the previous section.

Remove(β, $\mathcal{S}$) takes as argument an attribute name and removes it and its associated edges from the graph. It returns in $\mathcal{S}$ a list of attributes ready for evaluation.

These three operations must also maintain synchronized atomic access to the dependency graph, as well as a list of what attributes on the graph with in-degree 0 have been passed to a calling process as being ready for evaluation (to prevent an attribute being passed for evaluation more than once).

The *Startup* process calls *setup* to create the initial graph, and then *propagate*s a process for each element in the list returned by *setup*.

Each *propagate* process reevaluates its argument attribute. If this has changed, it *expands* the graph to include any new implied dependencies as discussed above. The argument vertex is then *remove*d from the graph along with its edges. A list of attributes ready for reevaluation is returned to the process, which then in turn spawns evaluation processes for each element in the list before terminating.

By the nature of $\mathcal{D}$, once an attribute is independent it can be evaluated entirely separately from any others. This implies that *remove* can return the fullest possible list of attributes waiting for reevaluation regardless of how they came to be included in $\mathcal{D}$, so that the maximum possible parallelism may be attained. Because the attribute flows form a DAG, we also do not have to worry about deadlock problems.

Analysis of the grammar when compiling it for use in an editor allows the preconstruction of templates for the dependency graphs. These can be instantiated in constant time when performing evaluations [Reps 82b]. Further, attribute reevaluations take constant time, so the only variable is the number of propagations needed to complete all attribute reevaluations.

If we conceive of the attribute propagation processes forming a tree $\mathcal{P}$, with the startup process forming the root, and each propagation forming the root of all the processes it propagates, and if $h(\mathcal{P})$ is the height of this process tree, then the time complexity is $O(h(\mathcal{P}))$. The maximum parallelism obtained is the max-cut of $\mathcal{P}$. In the worst case, with no parallelism (*propagate* runs as a procedure of *startup*), the tree (which now records calls to *propagate*) has height equal to the number of propagations, so the time complexity is related to the number of propagations. The number of propagations is the same size as the set *Affected* used by Reps, so in the sequential case we have the classic result for sequential incremental evaluation. In practice we are unlikely to achieve the maximum potential parallelism because of a shortage of processors; this point is discussed in the "Pragmatics" section below.

Handling Multiple Asynchronous Subtree Replacements

We have shown thus far that introducing parallelism scheduled by a topological sort on the attribute dependency graph can improve the running time of an incremental attribute evaluation algorithm. We look now at the case where multiple asynchronous subtree replacements are performed. This sets us up to consider the case of multiple asynchronous replacements across multiple distributed modules in the following section.

This section introduces a major result of this paper: given the parallel change propagation algorithm introduced above, we can support multiple asynchronous updates on the tree and still end up with an efficient algorithm.

We modify *setup* to be atomic and to *merge* the graph it creates with any other graph in $\mathcal{D}$. This merging operation is a *union* operation, so identical edges and vertices in the two graphs become one in the resultant graph. When a subtree replacement is performed, attribute evaluation proceeds as follows:

- Execute the *startup* process. This will *add* to whatever is already in $\mathcal{D}$, the graph of the initial dependencies amongst the attributes of the changed subtree.
- Continue exactly as before. From this point it makes no difference if the new dependency graph for the newly changed subtree overlaps with others or not; *remove* and *expand* will return the correct results regardless.

This is a significant result. We can now build environments for multiple users and know that attribute evaluations will succeed. Further, we can look at the truly interesting case, namely distributed editors that propagate semantic modification to one another.

The time complexity in the multiple subtree replacement case is slightly more complex than before. The dependency graphs grow on the fly, so when looking at an attribute with in-degree 0 in the dependency graph, we can be sure that it is truly independent of any evaluations propagated from the subtree replacement that ultimately was the cause of its evaluation. However, a graph from one replacement may grow to cover a part of the graph from another replacement that has already been evaluated, thus repeating the work of that evaluation. We need to show that there is an upper bound on this growth of attribute reevaluations.

Theorem. Given a tree on which k subtrees are replaced asynchronously, in the worst case any attribute is evaluated at most k times.

Proof. Suppose that each replacement is made immediately after the previous replacement's evaluations quiesce. This is the same as k separate replacements. We have seen that each replacement evaluates an attribute at most once; this implies that in the worst case an attribute is evaluated k times. Now suppose that the replacements overlap in time with evaluations. The potential exists for an attribute to be independent in $\mathcal{D}$ (and thus ready for evaluation), but that it will be reevaluated later by some other subtree replacement making $\mathcal{D}$ grow to include it again. In this case the propagations will chase each other. They will either catch up, thus merging the graphs and reducing the number of replacements, or they will not. All k subtree replacements can exhibit this chasing behavior, so in the worst case each attribute will be reevaluated k times. □

In the single replacement case, a tree $\mathcal{P}$ of process propagations is formed; in the multiple replacement case, a graph $\mathcal{G}$ with k starting points is formed. Note that *remove* returns *all* possible attributes that can be evaluated, regardless of where on the graph they appear, so there is no relation between the various paths through $\mathcal{G}$ and the replacement graph $\mathcal{D}$. Nonetheless, the time complexity of the multiple replacement case is $O(h(\mathcal{G}))$.

In general we would expect some *expands* on $\mathcal{D}$ will overlay parts of the dependency graph that are already there, thus reducing the number of evaluations required.

This model of users making arbitrary changes to the tree at any time in general poses some serious problems from the viewpoint of editing on the tree; a programmer wants to be sure that the part of the tree she is editing is not suddenly changed by some other user. We have found a solution to this problem, called *firewalls* which we discuss in the next section.

Maintaining attributes consistently across a distributed tree

We turn now to the problem of supporting semantic analysis for *programming in the many*. Whereas *programming in the small* refers to the problem of developing the contents of one module, and *programming in the large* refers to the problems associated with the combination of many modules to form large systems, *programming in the many* refers to the problem of coordinating the activities of many programmers as they attempt to create large software systems. In this paper we deal specifically with the problems associated with maintaining consistent semantic information between a number of modules distributed across a network. However, the algorithms developed are equally suitable to a time-shared mainframe environment.

We envisage a model of program development where several programmers each use a workstation to develop a module. with the workstations connected by a high speed network. Each programmer runs a copy of a programming environment such as Gandalf [Notkin 85], POE [Fischer 84], the Program Synthesizer [Teitelbaum 81], or SAGA [Kirslis 84]. To the programmer, the module she is currently editing is the *local* module. All other modules are *remote* modules, regardless of their physical location (some remote modules may in fact by resident on the local workstation but not being edited). All modules are internally represented by the programming environment as attributed trees. When complete the trees may be combined (across machines) to form large programs. Some examples of this situation are:

- Each programmer develops one or more modules, and all the modules are combined together to form a large program.
- One programmer owns the declarations for a procedure, the body of which is edited by another programmer on a remote machine.

Although a program can be broken into modules and split across a distributed system, there must still exist definitions in the context-free grammar that describe how the modules connect syntactically, and provide a channel for attribute flow. Every editor has a copy of the attribute grammar describing the language and its semantics. which it can use to determine when attribute flow crosses to a remote machine, and what the attribute dependencies there are. (Because of the precompilation of the attribute grammar. determining this information requires minimal overhead).

When a programmer replaces a local subtree, attribute reevaluation begins as usual. At each point that $\mathcal{D}$ is expanded a check is performed to see if the expansion would cause a flow of attribute propagation onto a remote machine, or become dependent on an attribute from a remote module. For the former case, we build $\mathcal{D}$ as normal, and then cut the graph so as to separate all remote attributes for a given remote module from the rest of $\mathcal{D}$. We now insert a special vertex, called *remote*, into $\mathcal{D}$ so that all attributes flowing over the cut *out of the local module* now flow via *remote*. We handle attribute values flowing *into* the local module by making them immediately independent and giving them the previous values they had when propagating into the local module (how this is actually done is described below). We then alter $\mathcal{D}$ so that anything flowing out of *remote* is discarded, all attributes flowing into the local module are independent, and continue evaluation as normal[3].

When the special vertex *remote* is independent, a packet of information is built containing local attribute information, which is then *propagate*d to the remote machine. We assume that *propagate* is modified to handle this. We further assume the existence of *support layers* of software that handle the message passing across the network and recovery from errors.

On the remote machine a semantic update will be triggered as if there had been a subtree replacement at the point where the syntactic link between the local and remote modules reaches the remote module. The dependency graph $\mathcal{D}$ for that remote module is now built in dual to the approach used for the local module. Attributes flowing into the (new) local module are treated as independent, and attributes that flow out are assigned a *remote* as described above.

This means that the subtree replacement model extends easily to handle remote attribute propagation. However, there are some problems with this. A module on a remote machine may be dormant (the module is not being edited, or it is inaccessible due to network or machine failure), or the user could be performing an editing operation that temporarily makes the tree unsuitable for receiving attribute information.

We therefore propose the introduction into our model of a general concept called a *firewall.* A firewall acts as a barrier behind which a module can shelter if it is not ready to accept semantic change propagations from other modules, a circumstance that may arise either because a local user is actually changing the module, or because the module is dormant. The former case arises relatively infrequently in relation to the total amount of time that a module is not dormant. Much of a programmer's time is spent browsing the code and deciding what changes to make. It is only while the actual subtree replacement is taking place that the firewall need be in place.

When an attribute propagation reaches a firewall that is in place, it queues until the firewall goes down, at which point the change is propagated to the module as if a subtree replacement had taken place at the firewall. If a module is dormant a propagation will queue until a programmer begins an editing session, at which time the propagation will enter the module.

There are a number of implications of this strategy. Suppose a change propagates to a remote module which, in response, will propagate back some semantic information. To the programmer this may appear as two entirely separate operations. The local

[3] Note that we are guaranteed to get all remote dependencies in place the first time the graph expands to imply remote relations, as the remote link is, from the viewpoint of the attribute grammar, just another link in the syntax tree. So expanding the graph to describe remote relations is just like expanding it to accommodate another node in the tree; the first attribute to get there triggers a complete expansion of $\mathcal{D}$.

attribute evaluations will cause a message to be sent to the remote module causing further propagation there. This may be indefinitely delayed because of a firewall being in place or because of a failure of the network[4]. The remote evaluations are not included in $\mathcal{C}$, so the internal attribute evaluation will run to quiescence in the normal way. When the remote propagation eventually starts, it will propagate back some semantic information that will be evaluated as if it were a new subtree replacement on the local module at the firewall.

Another important role of the firewall is to act as a place where previous values of remote attributes can be stored. Suppose an attribute depends on another attribute from a remote module. We do not want to have to go across to the remote machine to get the attribute value, which may in general not be possible because of remote firewalls being in place or network failure, both of which could cause undue increases in response time. Instead, we store on the firewall the value of every attribute that passes through it. When an attribute depends on a remote attribute, we need look no further than the firewall to discover its most recent value (which has to be correct as the most recent value will always by definition be propagated to the local module). Conversely, when a remote propagation triggers a local evaluation by passing a package of attribute information, the information on the firewall is updated.

Another advantage of a firewall is that it can be used to place a border on areas that a programmer may edit. A programmer may be allowed to cross a firewall while browsing through code, but might not be allowed to edit anywhere except behind her own firewall(s). This simple strategy guarantees that two programmers cannot simultaneously change the same part of a module, a standard feature of multiple user programming environments [Notkin 85, Leblang 85].

This model of distributed incremental attribute evaluation is chosen to strike a balance between the need for the programmer always to have absolutely correct attribute information, and the physical constraints on response time and dangers of ending up waiting for a network that is broken to yield a response. We adopted two complementary strategies to alleviate these problems. When a remote attribute needs to be reevaluated because of a local attribute change, the relevant information is packaged and transmitted to the remote site, which models the changed attributes as if they came from a subtree replacement. When a local attribute depends on a remote attribute, the most recent value (stored on the firewall) is just taken on the grounds that if a more recent value was available it would have been propagated to the local module. Note that more recent values may exist but be inaccessible due to network failure; we assume that in this case the network will eventually recover itself and perform the propagations.

This means that once an attribute propagation crosses over into a remote module, from the viewpoint of the local module, that attribute has quiesced. If the remote module

[4] We assume a suitable recovery mechanism for such network failures.

generates a propagation back into the local module, then that is considered a separate subtree replacement. Thus the complexity analysis for the multiple replacement case applies here also, from the viewpoint of one module. (We do not believe that considering the propagations across the entire network is meaningful or interesting; but the order across the entire network is as if all the affected modules were considered to be connected into one large tree).

A final issue worth a brief mention is that queueing information on a firewall also allows a way of handling multiple propagations of the same attribute from a remote module whilst the firewall is in place. The most recent propagation can be used, and the others discarded, thus reducing the potential problem of repeating the same work k times for k propagations of the same attribute.

Pragmatics

Even with the processing power of a high speed workstation, if too many attributes become independent concurrently, the number of *propagate* processes will swamp the processor(s) and remove any advantages accruing from the parallelism. Simple modifications to the algorithm can limit the effect of this problem. For example, we can put an upper bound on the number of *propagates* that can run concurrently. The operations on $\mathcal{D}$ can be modified to restrict the list of attributes ready for reevaluation returned in $\mathcal{S}$ so that swamping of the processor(s) cannot occur.

We would at least need two processes for each editor incarnation (on a workstation we would usually only have one editor running, but there is no reason why these algorithms could not be used on a time-sharing mainframe on which there are as many editor incarnations as there are users); the first would be the conventional editor process. The second would manage the firewall, merge any attribute propagations that arrive at the firewall into $\mathcal{D}$ as described above and handle transmission of packets of attribute values to remote editors.

The model of a module being dormant until awakened by a programmer may in practice be unrealistic. Consider, for example, the situation where module M propagates an attribute to module N, and the effect of that propagation in turn has an impact on some other modules in the system (possibly including M). If N is dormant for a significant period of time, many of the advantages of this editing model will be lost as other programmers will have to wait until N's programmer reactivates an editor for N and causes the effect of the changes from M to propagate through the system. Therefore, associated with each module there should be a watchdog which can invoke an editor for a module if it has been dormant with queued attribute propagations at its firewall for more than a given period. The editor invocation can then handle attribute propagations, log any errors for future use by a programmer and transmit any resultant remote propagations to other modules before returning to the dormant state.

We have said very little about the network that would be used to support the remote propagations; this network should contain an *attribute propagation layer* that interfaces the network to the editors and handles packing and unpacking of remote attribute propagations and the firewalls. The second editing process of the two described in the first part of this section would then be subsumed into this network layer.

Related Work

There is some previous research that relates in various ways to this project, which we discuss below. However, we can find no reference in the literature to any previous attempts to solve the problem of *incremental* attribute updating across a *distributed* program representation.

An obvious way to introduce parallelism into attribute evaluation is to treat attribute dependency graphs as petri nets and perform the obvious concurrent evaluation. Under circumstances where the attributes can be evaluated non-incrementally, this strategy works well and has been the basis for some non-incremental approaches to attribute evaluation [Fang 72, Kennedy 76]. In the incremental case, with graphs constructed on the fly, and with an attribute's propagation terminating if its value does not alter, there is no way that a vertex of the net can wait for all its inputs to yield a value before firing, as some might never do so. Some might also never be included in the *dependency* graph at all, as they are not affected in any way by a particular change in the tree. If we allow the vertices of the net to wait for all inputs they might well wait forever; and if we allow a vertex to fire each time it receives an input, in the worst case we get exponential explosion of processes. Such approaches are therefore unsuitable for the distributed, incremental situation.

To prevent a similar exponential explosion in a sequential incremental evaluation algorithm for single replacements in a single environment editing monolithic programs, Reps proposed a *topological sort* on the dependency graph to delay evaluation of attributes until that evaluation is guaranteed to yield the final result of the evaluation [Reps 82a]. Our use of the topological sort is adopted from this work. In [Reps 86], the single subtree replacement is replaced by *multiple simultaneous* replacements. This is done to model more powerful editing operations than single subtree replacements, such as program transformations. The algorithm obtained is unsuitable for asynchronous replacement, and is not extensible to distributed replacement. This result is subsumed into our asynchronous replacement result. It should be stressed that these, and all other incremental evaluation algorithms that we know of are sequential and restricted to single user systems.

An incremental evaluation strategy based on attribute propagation by message passing is proposed in [Demers 85]. The purpose of this work is to analyze more complex forms of static information concerning a program, such as dataflow information. This scheme does not support parallelism, multiple subtree replacements or distributed module editing.

A distributed word processing system based on Argus [Liskov 85] has been constructed to test problems in multiple paper authorship [Grief 86]. No semantic checking between distributed documents is performed.

Several incremental program development environments have been constructed [Teitelbaum 81, Feiler 81, Fischer 84, Garlan 84, Kirslis 84], but none of these support distributed program development. A number of distributed program development environments have been constructed [Donahue 85, Lampson 83, Swinehart 85, Leblang 85], but none of these support incremental semantic checking.

Discussion

We have introduced an algorithm for concurrent incremental attribute evaluation in program development environments. We have shown that this algorithm extends, naturally, first to supporting multiple asynchronous modifications to a program, and then to supporting incremental semantic checking across a set of program modules that are distributed on a network.

These are important results, because they pave the way for integration of modern programming hardware (workstations and high speed networks) with modern program development software (program development environments that guide and incrementally check the programmer).

We have not restricted ourselves to any particular editing system, but given general algorithms that we believe may be introduced into any existing language-based environment in order to extend it to the multi-user and distributed cases.

Further, the algorithms are host environment independent. By this we mean that they will function equally well on workstations used by a single programmer or mainframes time-shared between many programmers. The more processors a particular machine has, the more the parallelism in our algorithm will be capable of being exploited. Many more evaluation processes (*propagates*) than processors can be generated, but there exists a practical upper bound on any system where the overloading of the processors outweighs the advantages of the parallelism in the algorithm. In such restrictive situations, the algorithms are amenable to tuning to limit parallelism to prevent processor overloading.

References

[Demers 85] Demers, A., A. Rogers and F. K. Zadek, "Attribute Propagation by Message Passing", Conference Record of the ACM SIGPLAN Symposium on Language Issues in Software Development, *SIGPLAN Notices*, 20, 7 (July 1985).

[Donahue 85] Donahue, J., "Integration Mechanisms in Cedar", Conference Record of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments", *SIGPLAN Notices*, 20, 7 (July 1985).

[Fang 72] Fang, I, "FOLDS: A Declarative Formal Language Definition System", Ph.D Thesis, Stanford University (1972).

[Feiler 81] Feiler, P. H. and R. Medina-Mora, "An Incremental Programming Environment", *IEEE Transactions on Software Engineering*, SE-7, 5 (September 1981).

[Fischer 84] Fischer, C. N., G. F. Johnson, J. Mauney, A. Pal, D. L. Stock, "The POE Language-Based Editor Project", Conference Record of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *SIGPLAN Notices* 19, 5 (May 1984).

[Garlan 84] Garlan, D. B. and P. L. Miller, "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors", Conference Record of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *SIGPLAN Notices*, 19, 5 (May 1984).

[Grief 86] Grief, I., R. Seliger and W. Weihl, "Atomic Data Abstractions in a Distributed Collaborative Editing Environment", Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages (January 1986).

[Johnson 85] Johnson, G. and Fischer, C. "A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors", Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages, (January 1985).

[Kaiser 83] Kaiser, G. E. and A. N. Habermann, "An Environment for System Version Control", Conference Record, Spring CompCon '83 (February 1983).

[Kennedy 76] Kennedy K. and S. K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars", Conference Record of the Third ACM Symposium on Principles of Programming Languages (January 1976).

[Kirslis 84] Kirslis, P. and R. Campbell, "The SAGA Project: A System for Software Development", Conference Record of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Software Development Environments", *SIGPLAN Notices*, 19, 5 (May 1984).

[Knuth 68] Knuth, D., "Semantics of Context-Free Languages", *Mathematical Systems Theory*, 2 (1968). Correction, *ibid*, 5, 1 (1971).

[Lampson 83] Lampson, B. W. and E. E. Schmidt, "Organizing Software in a Distributed Environment", Conference Record of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, (June 1983).

[Leblang 85] Leblang, D. B. and G. D. McLean, "Configuration Management for Large-Scale Software Development Efforts", Conference Record of the GTW Workshop on Software Engineering Environments for Programming in the Large (June 1985).

[Liskov 85] Liskov, B., "The Argus Language and System", *Lecture Notes in Computer Science 190*, Springer-Verlag (1985).

[Notkin 85] Notkin D., "The GANDALF Project", *Journal of Systems and Software*, 5, 2, (May 1985).

[Reps 82a] Reps, T., "Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors", Conference Record of the Ninth ACM Symposium on Principles of Programming Languages (January 1982).

[Reps 82b] Reps, T., "Generating Language-Based Environments", Ph.D Thesis, Cornell University (1982). Also published by M.I.T. Press, Cambridge (1984).

[Reps 84] Reps, T. and T. Teitelbaum, "The Synthesizer Generator", Conference Record of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *SIGPLAN Notices*, 19, 5 (May 1984).

[Reps 86] Reps, T., C. Marceau and T. Teitelbaum, "Remote Attribute Updating for Language-Based Editors", Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages (January 1986).

[Swinehart 85] Swinehart, D. C., P. T. Zellweger and R. B. Hagmann, "The Structure of Cedar", Conference Record of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments, *SIGPLAN Notices*, 20, 7 (July 1985).

[Teitelbaum 81] Teitelbaum, T. and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, 24, 9 (September 1981).

[Waite 84] Waite, W. and G. Goos, "Compiler Construction", Springer-Verlag (1984).

Extended Abstract: Multi-User Distributed Language-Based Environments

Gail E. Kaiser
Columbia University

Simon M. Kaplan
University of Illinois

Josephine Micallef
Columbia University

April 18, 1986

Abstract

We describe a system for *programming in the many* that adapts language-based environments for programming in the small so as to support the automatic checking of semantic interdependencies among modules as they are developed in parallel on a collection of workstations connected by a local area network. We believe that accurate and productive *programming in the large* is made possible by supporting cooperation among environments for programming in the small.

1 Introduction

The management of large software systems involves large teams of programmers whose members must cooperate together in the development of a software system. In general each programmer is responsible for the development of a piece of the system, usually a module. The module exports certain facilities to other modules, and in turn depends on facilities imported from other modules. Invariably a *communications* problem arises when module interfaces change or do not meet what programmers imagine to be their specifications.

One way around this problem is to use a message passing system *among the programmers*. In this scenario, when a programmer changes a module interface, he sends a message (usually electronic mail) to all other programmers that use the module to tell them of the change. However, in the real world the list of programmers that use the module is constantly changing, because of changes the programmer is making to the module and concurrent changes by other programmers to their modules. So he sends the message to the entire team, just to be sure. The result is a deluge of mail. Some programmers will spend vast portions of their time reading mail (and accomplish little work), while others will ignore their mail and gets lots of work done. Much of this work may need to be redone later because of outdated assumptions made about other modules.

The rise in popularity of networks of distributed workstations aggravates this problem as personnel become distributed along with the hardware.

The easiest way to solve a problem is to make it go away. If the programmers all use state-of-the-art language-based program development systems, then the problem of inter-programmer communication about module dependencies can be made to vanish by automating the process of identifying semantic interdependencies among modules. Then, when a module changes the set of facilities exported to the other modules in the system, the other modules can be automatically notified of the changes.

In this paper, we describe a solution to this problem: a *multi-user, distributed language-based programming environment*, where the environment is responsible for propagating changes. The environment propagates each change, in a timely manner, to the set of modules affected by the change, regardless of their physical location. Whenever an imported module changes in a way that affects an importing module, the programming environment automatically updates its view of the software database to include the new version and informs the programmer if any errors in his own module were introduced by the change in the imported module. The programmer can go about his business knowing that he will be informed of all changes that affect him. This information is provided in a manner that makes it easy for him to make any corresponding changes required in his own module. The programmer no longer has to spend hours sending and reading mail.

Our solution meets three important goals. Changes are propagated *automatically*. Changes are propagated *in a timely manner*. And each change is propagated to *exactly* the modules affected by the change. We achieve these goals by generating our programming environment from an attribute grammar [8]. Attributes are attached to each module to describe the interface of the module. Each interface has two parts: (1) the facilities exported by the module and (2) the modules imported by the module and the facilities actually required from these modules. In addition, we automatically maintain a *use list* [15] for each exported facility that records which modules import the facility. These attributes provide enough information to pinpoint the modules that are affected by a particular change to an exported facility: if a change doesn't involve an exported facility, then no inter-module propagation is required and none takes place.

The advantage of using an attribute grammar to describe these interfaces is that there are already incremental attribute evaluation algorithms [16,4,1,5], which support automatic propagation to exactly those attributes that are actually affected by the particular change. The propagation occurs immediately, as soon as the change occurs. We have extended these algorithms to a parallel implementation [7] that makes it possible to perform propagation in a distributed programming environment.

Section 2 discusses the contributions of our work and section 3 places it in the context of related work. The remainder of the paper describes our model of incremental semantic checking across distributed modules by means of a running example. Section 4 discusses how language-based environments work internally by means of a simple example. Section 5 expands traditional single-user language-based environments to multi-user environments to allow multiple asynchronous edits on a program and briefly discusses some synchronization and safety issues. Section 6 expands this further to allow automatic attribute propagation (and semantic checking) across distributed machines. We end with a brief discussion of our prototype implementation.

2 Contributions of this Paper

The primary contribution of this paper is the development of a system for programming in the many that supports incremental checking of semantic interdependencies among modules distributed across multiple machines on a network. Each module is being edited using a programming environment that is language-based and normally suited to programming in the small. By having many environments for programming in the small cooperate together in the manner described in this paper, we achieve a synthesis of programming in the small called programming in the many. This approach represents a way to achieve programming in the large.

3 Related Work

Cedar [14] is a distributed programming environment for the Mesa programming language. The Cedar System Modeller [9] makes it easy for a programmer to recompile his module(s) in the context of particular versions of other modules in the software system. If any errors are detected, the programmer can either modify his own copies of the modules that caused the problem or send mail to other programmers asking them to make appropriate changes in their modules. Cedar does not support incremental consistency checking and does not perform automatic change propagation.

The Apollo Domain Software Engineering Environment (DSEE) [10] is a language-independent distributed programming environment. Like Cedar, DSEE makes it easy for a programmer to recompile his module(s) in the context of selected versions of the other modules in the system. If any errors are detected, the programmer can either modify his own copies of the conflicting modules or submit a *task* requesting other programmers to make appropriate changes to their modules. DSEE provides support for monitoring the other modules and informing the original programmer when the other programmers have all checked off the activities listed in the task [11]. DSEE does not actually check whether or not all errors have been removed. It does not perform incremental consistency checking.

The Gandalf System Version Control Environment (SVCE) [6] supports incremental consistency checking across module interfaces. Any errors introduced by a change in an imported module are ..utomatically reported to the programmer. SVCE is not a distributed environment, but a multi-user environment for a mainframe. However, the problems of multi-user synchronization with respect to the software database (an attributed syntax tree) had not been solved at the time of the SVCE implementation [2], so SVCE is effectively a one-user-at-a-time environment. (The synchronization problem for attributed syntax trees has now been solved by our introduction of firewalls, discussed in this paper.)

The Unix[1] Source Code Control System (SCCS) [13] and the Revision Control System (RCS) [17] support synchronization among multiple programmers using file locking mechanisms. These mainframe systems use variants of the Make tool [3] to automate the recompilation and relinking of a program using the latest versions of modules after changes have occured. They do not automate change propagation or perform incremental consistency checking.

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

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

MODULE N ;
   EXPORT Y ;
   FROM M IMPORT x ;
   ...
END N ;
```

Figure 1: Skeleton of Program with Two Modules

4 Incremental Semantic Checking

This section looks at how incremental semantic checking among modules is achieved in traditional single-user language-based programming environments (exemplified by the Cornell Synthesizer Generator [12]) by considering an edit on the program in figure 1. A logical representation of the tree structure of the program is given in figure 2. In this diagram the arrows represent attribute flow, not syntactic relationships.

The full paper will discuss a simple edit (removing x from the export list of M) and its effect, for the classical single-user case [16].

5 Multi-User Semantic Checking

We expand the traditional "programming in the small" language-based editor paradigm – a single user editing monolithic programs – to the situation where many programmers can edit the same program asynchronously. We assume that programmers will not be able to edit the same part of the tree, ie that there will exist some division of the program among programmers. The obvious place to make this division is at the module level. We therefore propose a model of editing where many programmers have access to a common program tree, but are each given an area on the tree that only they can modify.

Consider the case where programmers *Dick* and *Jane* are editing our simple program. *Dick* can edit module M only and *Jane* can edit module

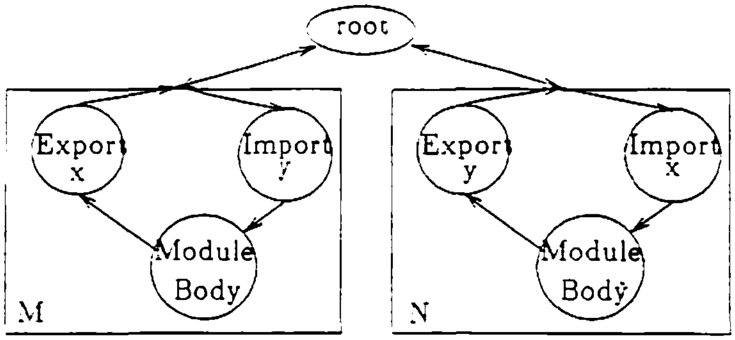

Figure 2: Logical Representation of program

N only. Suppose that *Dick* deletes x from the export list of M. While the attributes affected by the change are being propogated, *Jane* deletes y from the export list of N. A new set of attribute propogations for this change is started. We now have a situation where the tree representation of the program in figure 2 is being asynchronously modified by two processes.

In [7], we presented an algorithm that performs attribute reevaluation in the face of multiple asynchronous edits on a program. In this algorithm, the attribute propagations from the various replacements synchronize with one another to perform an optimal reevaluation of attributes. This is done by sharing the dependency graph structure (which is used by the algorithm to record dependencies among attributes and to select which attributes are *independent*, i.e., ready for (re)evaluation) among the attribute reevaluation processes. One of the novel features of this algorithm is that the number of attribute evaluation processes running at any time is dependent *not* on the original number of edit sites on the tree but on the number of attributes that are independent. This allows us to achieve the maximum degree of parallelism.

But what happens if module M is modified and the change is propagated to module N at just that moment when N is itself being edited (at the exact moment of subtree replacement). We do not want an attribute propogation to arrive in N when the tree representation of the module is in an inconsistent state[2]. To resolve this problem we introduce the concept of a *firewall*. A firewall can be *up*, in which case any attribute propagation attempting to cross the firewall is delayed, or *down*, in which case it is entirely invisible. The firewall provides a barrier behind which a program segment can shelter while it is being modified. It is generally a good idea to have the firewall at the same level as the split of programs among programmers, in this case

[2] Note that this is different to the *attributes* being in an inconsistent state.

at the module level. In figure 2 the boxes around the modules represent the firewalls. Firewalls need only be up *when a tree is actually being changed*; this is a minimal amount of time in relation to the time spent by the environment performing attribute evaluations and the time the programmers spend browsing the tree.

6 Distributed Semantic Checking

In section 5 we expanded the traditional single-user model of language-based editing to a multi-user model. This section further expands the model to distributed, multi-user language-based editing.

Having allowed multiple edits on a program, with firewall protection, the next step is to split programs across multiple machines. We believe that the advent of the inexpensive workstation is rapidly making distributed program development with cooperation among the programmers the preferred mode of software development. We split a program in such a way that a firewall-protected section (in most cases a module) is assigned to a workstation. (Naturally one workstation may be the home of many modules). In terms of figure 2 this means that modules M and N are each assigned to a workstation. The part of the figure representing the root of the tree becomes subsumed into the network. (For fault tolerance reasons the root information is duplicated on each workstation).

On each machine, attribute propogation proceeds as if it were the only machine in the network as long as the attribute propagations remain within the bounds of the firewall. Once the firewall is accessed by an attribute propagation that will propagate *outside* the firewall, it becomes necessary to deal with remote machines. It is a feature of the attribute propagation algorithm that once an attribute propagation reaches the firewall, all other attributes that will cross the firewall in company with this attribute become known immediately[3]. We can therefore wait until all the attributes that will cross the firewall together are ready for propagation (more formally, when they are all independent of any other attribute values), then build a packet containing their values and propagage that packet across the network to the set of modules that depend on the changed module (this information is determinable from the use lists mentioned in the introduction). We assume that the network has an *attribute propagation layer* that can perform the packing, unpacking and dissemination of attribute packets to the actual target

[3] In fact, this is a feature of Rep's original algorithm and *all* optimal incremental algorithms that we know of; it is this very fact that makes the optimality of the algorithms possible.

modules.

Conversely, when we need an attribute that originates in another module (such as the type of an imported variable) we do not want to have to go across the network to get this information (in the case of a network failure it may not be available). We therefore use the firewall as a *cache*. All attributes that pass through it are cached on the firewall and can be obtained from it when needed without any need to access the network. (An underlying assumption here is that the cache will be up to date on the grounds that if more recent information were available it would already have arrived).

We also have to deal with the situation that arises when a new attribute value propagates through a firewall and reaches another module. The attribute propagation layer unpacks this information and compares it to the most recent value for the attribute on the firewall. If these are different, then the new value is propagated into the module. This is achieved by simulating an edit on the module at the firewall. It does not matter what is happening to the the module internally as the module is ready to recieve new attribute values (because the firewall is down). We use the attribute propagation algorithm described in section 5 and [7], which supports multiple asynchronous edits and associated attribute propagations.

Finally, the firewall acts as a protection in the event that the module is *dormant* (not being edited). Attribute propagations are stored on the firewall until an editing session for that module is resumed. This strategy also allows a simple optimization: when a module reawakens, only the most recent changed attribute values are passed to it.

Thus, in our example, if the same editing sequence is followed as for the edits in section 5, the export list attribute from M will be bundled and passed across the network to N. If the firewall is up this will wait until the firewall comes down, at which point the attribute bundle will be unpacked and inspected. The attributes that differ in value from their values as cached on the firewall will have their cache values updated and then be propagated into N.

7 Implementation

We have implemented a prototype of the distributed language-based environment for a local network of two VAX 11/750's running UNIX 4.2 BSD. A 10 Mb Ethernet is used for communication between the machines. Attributes flow between local and remote machines by means of the interprocess communication (IPC) mechanism provided by Unix using the INTERNET TCP protocol. The user interface consists of a rudimentary language-based structure editor, which allows each user to create and modify sections of a program by a sequence of subtree replacements of the program's abstract syntax tree. The system can correctly handle asynchronous subtree replacements resulting from either a local editing operation or a remote attribute propagation. We are in the process of adding firewalls and plan to complete a larger-scale implementation on a heterogenous network consisting of a number of Unix workstations. In this version, the distributed incremental semantic analysis algorithm will be integrated with an improved user interface supporting multiple windows and mice pointing devices.

References

[1] Anne Rogers Alan Demers and Frank Kenneth Zadeck.
Attribute propagation by message passing.
In *Proceedings of the SIGPLAN '85 Symposium on Language Issues in Programming Environments*, pages 48–59, Seattle, WA, June 1985.

[2] Robert J. Ellison and Barbara J. Staudt.
The evolution of the gandalf system.
The Journal of Systems and Software, 5(2):107–119, May 1985.

[3] S.I. Feldman.
Make - a program for maintaining computer programs.
Software - Practice and Experience, 9:255–265, April 1979.

[4] Gregory F. Johnson and C.N. Fischer.
A meta-language and system for nonlocal incremental attribute evaluation in language-based editors.
In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 141–151, January 1985.

[5] Gail E. Kaiser.
Semantics of Structure Editing Environments.
PhD thesis, Carnegie-Mellon University, May 1985.
Technical Report CMU-CS-85-131.

[6] Gail E. Kaiser and A. Nico Habermann.
An environment for system version control.
In *Digest Of Papers of the Twenty-Sixth IEEE Computer Society International Conference (Spring CompCon '83)*, pages 415–420, February 1983.

[7] 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 (PODC)*, Calgary, Alberta, Canada, August 1986.
To appear.

[8] Donald E. Knuth.
Semantics of context-free languages.
Mathematical Systems Theory, 2(2):127–145, June 1968.

[9] Bulter W. Lampson and Eric E. Schmidt.
Organizing software in a distributed environment.
In *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1–13, San Francisco, CA, June 1983.

[10] 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.

[11] David B. Leblang and Robert P. Chase Jr.
Computer-aided software engineering in a distributed workstation environment.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104–112, Pittsburgh, PA, April 1984.

[12] Thomas Reps and Tim Teitelbaum.
The synthesizer generator.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[13] M. J. Rochkind.
The source code control system.
IEEE Transactions on Software Engineering, SE-1:364–370, 1975.

[14] Warren Teitelman.
A tour through cedar.
IEEE Software, 1(2):44–73, April 1984.

Also appears in Proceedings of the Seventh International Conference on Software Engineering, 1984.

[15] Carla Marceau Thomas Reps and Tim Teitelbaum.
Remote attribute updating for language-based editors.
In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages.*

[16] Tim Teitelbaum Thomas Reps and Alan Demers.
Incremental context-dependent analysis for language-based editors.
ACM Transactions on Programming Languages and Systems (TOPLAS), 5(3):449–477, July 1983.

[17] Walter F. Tichy.
Rcs – a system for version control.
Software - Practice and Experience, 15(7):637–654, July 1985.

Reliability in Distributed Programming Environments

Gail E. Kaiser[*]
Columbia University
Department of Computer Science
New York, NY 10027

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

September 2, 1986

Keywords: attribute grammar, change propagation, consistency, distributed system, language-based editor, programming environment, reliability, replicated data.

Abstract

We describe a system for *programming in the many* that adapts language-based editors for individual programmers to support the automatic checking of semantic interdependencies among modules as they are developed in parallel by multiple programmers on a collection of workstations distributed across a local area network. We focus on the reliability of these distributed programming environments as some modules become inaccessible and later return to availability. Our primary contributions are the decentralized control of the programming environment, *firewalls*, a mechanism that encapsulates individual modules to protect them from external failures, and a special network layer that enables the system to be highly available and reliable in the face of an

[*] This paper was written while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

[†] Supported in part by a grant from the AT&T Corporation

unreliable network. The firewalls and network layer together support re-establishment of consistency among fully replicated data in the context of distributed programming environments.

1 Introduction

The development and maintenance of large software systems involves teams of cooperating programmers. In general each programmer is responsible for the development of a piece of the system – a module.[1] Each module exports certain facilities to other modules, and in turn depends on facilities imported from other modules. Invariably communication problems arise as programmers change the interfaces of their modules, so they no longer meet the specifications expected by other programmers. We have solved this problem using a *distributed, language-based, programming environment* [11]. Using this system, programmers can develop modules in isolation and the system takes care of communicating changes among the relevant set of their colleagues, regardless of their physical location.

This paper focuses on reliability issues associated with the use of these environments. Specifically, we are interested in building a system that is both *reliable* and *highly available* [18]. Each module is changed using an *editor* that operates entirely on a single machine, where the distributed collection of editors makes up the *environment*. In this context, reliability requires that every editor should always have correct information. This information should also be as up to date as possible. By highly available we mean that an editor should be affected as little as possible by the failure of the network or of other machines. We have developed a system where control is fully decentralized: each editor operates independently of the others, and propagates information to other editors whenever the interface information of its module is altered. The most recent possible information from other modules is always available for use, regardless of the state of the network or the state

[1] By *module* we mean the unit assigned to an individual programmer for source code changes.

of the machines on which the other modules are located. We use a provably correct algorithm for the re-establishment of information consistency when recovering from network or machine failures.

We begin by listing, in section 2, the contributions of this paper. We then overview briefly in section 3 our distributed language-based programming environment. Section 3.1 discusses how the system has fully decentralized control. Section 4 describes our special network layer. and section 3.2 our *firewall* system for encapsulating modules. Section 5 considers the case that the network and machines are completely reliable. This somewhat unrealistic scenario leads into a discussion of the case where the network and/or machines are unreliable (section 6). We present the algorithm for re-establishing consistency, prove it is correct and address its complexity. We then look at some cases where we can do better than is implied by the worst-case of the algorithm. Finally. we compare our research with related work. specifically that in the field of distributed databases. in section 7. briefly describe the implementation in section 8 and conclude by summarizing our results.

2 Contributions

This paper makes several contributions in the area of reliable. distributed, program development systems:

- It describes a system for communicating change information about modules among the programmers developing a software system in which control of the distribution of the information. and the effects of the distribution, are completely decentralized.

- The system described is highly available in that any machine can operate regardless of the state of the network or the state of any other machine. Information from other machines is always guaranteed to be correct, although it may be old [14].

- The system operates reliably in that it will restore information consistency after a network or machine failure.

3 Run-Time Support for Distributed Change Propagation

Rather than building a particular distributed programming environment, we have developed a system for generating the desired environments from formal descriptions. The formal description is given as an *attribute grammar* [12, 13], which describes the context-free and context-sensitive properties of the programming language. Our generator translates the description for a particular programming language into an internal representation understood by a language-independent kernel, which provides the common run-time support for these environments. This paradigm was originally developed by Reps [6], and has since been applied by many researchers to the generation of single-user, single-machine programming environments [16, 10, 7]. These environments support syntax-directed editing, type checking, code generation and other programming tools.

All of these environments process the program incrementally, after each subtree replacement command to the syntax-directed editor. For example, any static semantic errors are immediately flagged as soon as the programmer enters the erroneous part of the program and the object code is always kept up to date. Auxiliary data structures, called *attributes*, represent the symbol table, the object code, *etc.* and are modified as the program changes. For efficiency, only the attributes actually affected by the subtree replacement are recalculated. Reps developed an optimal algorithm for updating this internal information [17].

In [11], we present an extension to Reps' algorithm that permits optimal updating when the programming environment is implemented on a multiple processor machine and/or distributed across a network. For the first time, a programmer working on a workstation could immediately be informed of errors in his module caused by changes to other modules being modified simultaneously by other programmers on other workstations. The details of the internal workings of the algorithm are not relevant to this paper, and so are omitted; interested readers are referred to [11].

The main idea is as follows. Within any particular editor, attributes are re-evaluated as necessary in response to changes in the source code of the module. When an attribute changes in value, all attributes that depend on the first attribute must also be re-evaluated, and so on. This is called *attribute propagation.* As long as attribute propagation remains within a single module, the modules are effectively independent and distribution is of no concern.

However, certain attributes attached to each module depend on attributes in other modules. For example, the symbol table of a module depends on the symbol definitions imported from other modules. When any such attribute changes in value, it must be propagated across the net to all the external attributes that depend on it. When attribute propagation passes from one module to another, a new process is forked within the receiving editor. This process simulates a subtree replacement at the boundary of the second module, causing a chain of local attribute re-evaluations. This propagation may proceed concurrently with an attribute propagation initiated locally and with other propagations initiated externally.

Sometimes the same attribute is affected by multiple propagations with respect to the same module. Rather than repeat attribute evaluation for each process, we effectively combine the separate threads of control by synchronizing on a data structure called the *dependency graph.* The dependency graph is used by our algorithm, and by Reps', to order the evaluation of attributes so that inputs of attribute equations are always evaluated and set to their final values before outputs are calculated, and no attribute is evaluated more often than necessary. The details are rather messy, so we refer the interested reader to our previous paper [11].

3.1 Decentralized Control

In Reps' attribute evaluation algorithm. there is only one process and one thread of control. A single dependency graph is used to order the calculation of attribute values. The obvious distributed extension to Reps' algorithm would have been to simply distribute the program tree across the network. where certain links between nodes in the tree were implemented by inter-machine references rather than pointers. This would require a centralized dependency graph to order the evaluation of attributes throughout the distributed tree.

However. a centralized data structure would make our distributed programming environment inherently unreliable. The data structure would reside on some particular machine. acting as a server for the client processes on the other machines. If the server went down. attribute propagation would be impossible. even within the boundaries of a single machine. Program editing could continue. however. without local error checking. When the central dependency graph was restored. then an off-line attribute evaluation procedure would be applied to restore consistency across the distributed programming environment. Similarly. if the network was broken so that some machines could not access the central dependency graph. then attribute updating could not continue on these machines. Program modification could continue as before: the locally stored attributes could not be updated until the network was restored.

To solve these problems. we have *decentralized* the run-time control for our attribute evaluation algorithm. We maintain a separate, local dependency graph for every editor, which describes the dependencies among only the locally maintained attributes. Any dependencies that cross module boundaries are represented by special *pseudo-vertices* in the dependency graph. These pseudo-vertices represent both the points where attribute propagation passes into the local module and the points where propagation leaves the module. The only incoming and outgoing edges of each graph are with respect to other local vertices.

This representation allows attribute evaluation to proceed independently within each local editor, regardless of the state of the network. When a local edit causes local attributes to change in value, updating of dependent local attributes is carried out as described in the previous section. The only difficulty arises when a propagation reaches a pseudo-vertex.

Since a pseudo-vertex does not have any outgoing edges to the dependent attributes in other modules, it is impossible to determine locally exactly where to propagate the changed attribute. Instead, the changed attribute, including its identification and value, is broadcast across the net. When the attribute arrives at each destination, its identifier is compared to the attributes expected by the local pseudo-vertices. If there is a match, local attribute propagation continues from this point: if there is no match, the attribute is ignored.

3.2 Firewalls

This works fine as long as the network itself is reliable, so that every attribute sent is eventually received at all the other machines. Serializability cannot be guaranteed, but this is not a concern – attribute updating does not depend in any way on the order in which subtree replacements are made. After all the changes are made and attribute propagation is run to quiescence, the result is always the same.

Our previous paper assumed that the network was reliable: the purpose of this paper is to extend our previous work to the unreliable case. When the network is not reliable, additional support is needed. We need a way to guarantee that each change to an attribute on the external boundary of a module will eventually be received by all the other modules. On the flip side, we need a way to get the value of an external attribute on the boundary of another module that is an input to the calculation of a local attribute, in spite of the fact that the remote module may not be accessible.

We solve these two problems with *firewalls*. A firewall is conceptually a barrier that encapsulates a module. Firewalls can be up, in which case any remote attribute propagation that reaches a module with a firewall in the up position is queued until the firewall changes state, or down, in which state all attribute propagations can flow through it. A firewall is only up if the module is physically being modified (which takes very little time relative to the total time spent in an editor) or if the module is *dormant* - not loaded into an editor.

A second role of the firewall is as a *cache*. All attributes that pass through the firewall are (logically) cached on the firewall. Thus, all external attributes are *replicated* at every machine. Any reference to a nonlocal attribute can be satisfied simply by going to the firewall, on the grounds that if more recent information were available it would have arrived. In this way we make our programming environment highly available, because local editors can continue to operate even in the case of total disconnection from the network. The information thus obtained from the firewall is always correct, that is, self-consistent, but may not be up-to-date, because of network or machine failures.

Firewalls are actually a higher-level abstraction: we discuss their implementation as a layer of the networking software in the following section.

4 Attribute Propagation Layer

This section introduces a special network layer - the *attribute propagation layer* (APL), which interfaces the distributed programming environment to the network. Editors communicate entirely with this layer, and the facilities to support firewalls and recovery from failure are built into it. This section discusses the APL and our assumptions about networks and machines.

We make no assumptions about the size, topology or reliability of the network, or the order in which it delivers messages, other than that the net supports broadcast. We do not assume that the messages arrive at a node in any particular order. but do assume that they are never corrupted. We allow the network to have internal nodes whose role is to pass along packets of information (for example. gateways between ethernet rings).

We assume that machines are failstop, that is, they are either running correctly or down. Whenever a machine is up. its APL is running, even if the machine is disconnected from the network. The APL remains running until the machine is shut down (or fails). APLs are assumed to be robust, and communication between each editor and its local APL is assumed to be reliable. Further, we assume that the APL knows the status of its connection with the network at all times. The APL has the task of re-establishing consistent attribute information after failures. In other words. APLs solve the problem of consistency among replicated data in the context of distributed attribute propagation.

Each APL can interface several editors to the network; the number is not fixed, but depends on the machine. For example, a workstation is likely to have one editor, but a large mainframe could have several editors running at any time. The editors on the local machine are called *local* editors; all others are considered to be *remote*, and similarly we can have local and remote modules. Modules can be either *active* (being edited) or *dormant* (not loaded into an editor). In either case. when an attribute from another module that affects a particular module is broadcast. the module should receive the attribute eventually, even if it is currently dormant.

To support this, the APL maintains for each local module an attribute cache and a boolean flag which together implement a *firewall*. The flag is set whenever the firewall is in the up state. If the flag is set and an attribute propagation arrives for a module, the cache for the module is updated, and no further action is taken. When the firewall flag is reset. the new cache value is propagated to the editor for the module. When a module is made dormant, the APL cache information is stored with the dormant module's internal forms. When the module is made active again. the stored value of the cache is compared to the current APL cache values, and any necessary updates are propagated to the module.

For each remote module the APL maintains a similar cache, but no flag. The remote caches are needed so that when local attributes that depend on the values of remote attributes need re-evaluation, there is no need to go to the remote site to get the information. In this way the two functions of the firewall are implemented in the APL. Also, this information will be needed in the unreliable network case discussed in section 6.

The APL performs the following functions (disregarding for the moment issues of unreliability):

- When it receives a changed[2] attribute from a local editor, that attribute is then passed to every local module, and broadcast on the net.
- When it receives an attribute broadcast from a remote APL, the relevant cache is updated and that information is passed to all local modules.

We assume that each APL and each module associated with an APL are uniquely identifiable.

5 Reliable Attribute Propagation

If the network is completely reliable, there are only two situations in which consistency among attributes must be established, namely: a new APL is added to the net or an APL has been removed from the net in an orderly fashion and is now being returned. In both cases, the APL broadcasts an *update* packet on the network. Every other APL that receives the packet returns its *local* cache information to the originator of the *update*. For the former case, this gives the originating APL a set of caches to pass to local

[2] It is possible that an attribute can be propagated from an editor and have exactly the same value as it had on the previous propagation; in this case the APL, following the policy defined for our attribute evaluation algorithm, can discard the attribute. We therefore assume that unchanged attributes never pass through an APL.

modules, and in the latter the originating APL can now decide what information has changed and propagate this to local modules. We assume that no editors local to the new/restarted APL are active while this startup process is in progress.

6 Unreliable Attribute Propagation

In this section, we present an algorithm for maintaining consistent attribute information in the face of an unreliable network. We extend the information stored for each cache to include a timestamp, and then give an algorithm that restores consistency of attribute information among APLs after a network failure. This is a pessimistic algorithm in that it makes as few assumptions about good network behavior as possible: we require only that the network does not corrupt data. We prove that the algorithm is correct and terminating, and discuss its complexity. We then show how we can do better by making certain assumptions about network topology and reliability.

As well as containing attribute information, each element of an APL cache is labeled by a timestamp and the identifications of the originating APL and the module within that APL. Timestamps can simply be integers that are incremented by the originating APL each time an attribute is broadcast [8].

The consistency re-establishment algorithm works as follows: When an APL comes up, or when its broken connection to the network is restored, it broadcasts an *update* request over the net. This will be received by all APLs that are accessible from this APL. On receipt of an *update* request, an APL broadcasts each attribute that it has cached, along with its timestamp and originating label: from now on, we refer to all this information as a cache. On receipt of a cache, the APL executes the algorithm given in figure 1.

When an APL receives a changed attribute from a local module, it broadcasts the corresponding cache, and then all the APLs follow the same strategy on receipt of the cache. Normal attribute propagation among APLs is thus subsumed into the consistency re-establishment algorithm. We assume that

```
receive($p$) is
let
    $c$ = a local cache
    $r$ = a remote cache received from the net.
    $time$ = the time it was sent.
    $info$ = the information.
in
    if (there exists a local cache entry $c$ corresponding to $r$)
      then if ($c.time \leq r.time$)
              then if ($r.info = c.info$) then update local modules fi
              else broadcast($c$)
           fi
      else create a cache for $r$
    fi
end
```

Figure 1: Consistency Reestablishment Receive Part

intermediate nodes in the network can also broadcast *update* requests if they rejoin pieces of the network that have become separated, in order to make the newly rejoined pieces consistent.

Note that a broadcast will reach the APL that broadcasts it, too. So it will also participate in the restabilization of the net. This is necessary because its local modules may have been altered while the network was inaccessible, resulting in changes to propagate to the rest of the net.

Although we have assumed that each time an APL comes up it broadcasts and *update* request, and that therefore it is technically the only out-of-date APL, the algorithm is fault-tolerant of this assumption and will deal with the situation where all APLs have different information and think that they are up-to-date. In this case the algorithm will reestablish most up-to-date information at each APL by the time it terminates.

6.1 Correctness and Complexity

In this section, we argue for the correctness of the algorithm above, and show that it is terminating. We then investigate the complexity of the algorithm. We make several assumptions in order to simplify the arguments; these assumptions are not required in practice, so we end this section with an argument of correctness and complexity ignoring these assumptions. The assumptions are:

- Local modules do not emit any attribute changes while the restabilization is progressing.
- Only one APL returns to the network at a time.

Lemma 1 *The attribute consistency strategy is terminating.*

Each time a cache and its timestamp is broadcast, the network is (virtually) partitioned into these sets:

- $\mathcal{N}$ - The set of APLs whose cache holds more recent information than the packet just broadcast.
- $\mathcal{O}$ - The set of APLs whose cache holds less recent information than the packet just broadcast.
- $\mathcal{S}$ - The set of APLs whose cache information is the same as that of the APL originating the broadcast.
- $\mathcal{D}$ - The set of APLs that are inaccessible and will therefore not receive the broadcast.

Let $\mathcal{T}$ be the set of all APLs on the network, and let sidebars denote the size of a set, as in $\mid \mathcal{T} \mid = \mid \mathcal{N} \mid + \mid \mathcal{O} \mid + \mid \mathcal{S} \mid + \mid \mathcal{D} \mid$. Now, suppose that some APL α broadcasts an attribute cache. $\mid \mathcal{O} \mid + \mid \mathcal{S} \mid$ APLs will absorb the new broadcast and produce no new broadcasts as a result, $\mid \mathcal{D} \mid$ APLs will simply fail to receive it, and $\mid \mathcal{N} \mid$ APLs will decide that they have more recent information and rebroadcast their packets. In the worst case. $|\mathcal{N}| = |\mathcal{T}| - |\mathcal{D}| - 1$ (every other APL has more recent information than α); it can be much smaller. Each APL that receives the broadcast will in turn divide the net into the $\mathcal{N}$, $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{D}$ sets. Each element of a $\mathcal{N}$ set will rebroadcast, and each element of the $\mathcal{O}$ sets will absorb the information (there has to be at least one such APL – the originator of the first broadcast). So if $|\mathcal{N}|_\alpha$ is the size of the $\mathcal{N}$ set of APL α. and β is some other APL in $\mathcal{N}_\alpha$. $\mathcal{N}_\beta$ must be smaller than $\mathcal{N}_\alpha$ (in the worst case, $\mathcal{O} = \{\alpha\}$). Each time a broadcast is made. $\mathcal{N}$ must be smaller than previous $\mathcal{N}$ sets in this way. and so the algorithm eventually terminates. □

Lemma 2 *When the consistency establishment algorithm terminates. all APLs that participated in the broadcasting process have the same cache values.*

Proof Outline: To prove correctness. we show that if $\mathcal{N}$ is empty. then $\mathcal{O}$ must become empty also. Then $|\mathcal{T}| = |\mathcal{D}| - |\mathcal{S}|$. and all APLs have the same information or are not accessible. Note that an APL only broadcasts if it has more recent information than information it receives: and APLs only update themselves if they receive information more recent than that which they already have; so the algorithm tends to give each APL the most recent possible information. If $\mathcal{N}$ is empty for every APL in the network, there can be no APL that has more recent information. $\mathcal{O}$ may not be empty, but the APLs in this set will simply absorb the information of the current broadcast. and thus bring themselves up-to-date. Since we know that the algorithm is terminating, $|\mathcal{N}|$ steadily decreases, and the information updating in an APL tends to make bring the APL up-to-date. it follows that when $\mathcal{N}$ is empty for each APL, all APLS either have the most up-to-date information or it has been broadcast to them. Since all broadcasts are eventually absorbed by all APLs. all APLs eventually get the same, most recent possible, information. □

Theorem 1 *The consistency establishment algorithm is correct.*

Proof Outline: Follows directly from the preceding two lemmata. $\Box$

This algorithm has the desired updating properties necessary to restore APLs to a consistent state after a failure. Note that it will also (as a trivial case) handle normal attribute propagations and automatic extension of APL caches when new modules (and APLs) dynamically appear on the net. It is fully decentralized: There is no single machine or APL that acts as a control on the process.

The worst case-complexity of the algorithm (where we view complexity as a factor of the number of broadcasts) for the update of a single attribute cache is $\sum_{i=1}^{n} i$. where n is $|\mathcal{T} - \mathcal{D}|$, and every APL has a different value. The crippling factor here is that the APLs could agree on a most recent value early on. but still have many messages with older information to process. Each such message will yield a broadcast of the older information. which will get passively absorbed by the other APLs. However. this is a pessimistic algorithm: in section 6.3 we discuss ways of improving the algorithm in practice.

6.2 Removing the Assumptions

In the previous section. we assumed that the editors yield no attributes for remote propagation during the re-establishment of consistency, and also that only one APL comes up at a time. These assumptions were made purely to simplify the analysis. In practice there is no reason why editors cannot propagate at any time, or why APLs cannot come up at any time. The algorithm will work in these cases. but the effective complexity of the algorithm (by this we mean the number of broadcasts that would be seen by an observer watching the network) would increase in proportion to the number of additional propagations caused by the editor yielding an attribute for propagation or an additional *update* request on the part of an APL.

6.3 Doing Better

There are several ways that we can do better in practice than the worst case of the algorithm suggests. However, each of these implies some assumptions about the network topology or stability, or requires more space. We enumerate some of these improvements below, and indicate their assumptions.

- *Maintain history information.* This improvement requires that each APL maintain a table that shows for every other APL the most recent information it has received from the other APL. Then if information less recent than the table entry is received, it can be discarded. This improvement is only useful in the event that messages arrive out of order. The space to store the table is a factor of the number of attributes flowing through the APL and the number of other APLs. The table has to be capable of dynamic expansion as new APLs and modules are added.

- *Use point-to-point communication.* In this improvement, messages are passed point-to-point rather than broadcast. The disadvantages are that the number of APLs and modules is 'fixed' (a super-process is required to update the database of APLs and modules on each machine every time a new APL module is added, a difficult process in the event that the network is unstable), and that the network must be sufficiently stable for several message pairs to pass between two points to complete an update. Also, history information must also be maintained as described above. However, broadcasts are the natural communication strategy for an ethernet anyway (our implementation vehicle – see section 8). Further, the broadcast algorithm given in figure 1 is safer: Consider the case where APLs α and β communicate and become consistent, but an APL γ is excluded from the process although it has more recent information because neither α or β know this. The broadcast strategy would dynamically add γ to the updating process and the other two APLs would get more recent information as a result.

- *Use knowledge of the network topology.* For example, if the network is a star, point-to-point communication with the central server node is all that is needed. If a ring-of-stars configuration exists, the nodes on the ring can use the basic algorithm (plus the improvements enumerated above where relevant), and use point-to-point for the elements of the star. (Note that this is a generalization of the special case where the APL has multiple local modules and does point-to-point communication with them).

7 Comparison to Distributed Databases

In this section, we compare our approach against those taken in distributed database systems, and explain why some of the problems that beset the distributed database case are not an issue here, and why our solution would in turn not be adequate for the distributed database case. We look first at some differences between the programming environment and database cases, then at availability of the system, and finally at reliability issues. We briefly relate our work also to other approaches to providing fault-tolerant systems.

In the distributed database case, a major problem is that of *transaction commit* - the effort involved in processing a transaction is distributed among several database servers and then the servers have to agree on whether the transaction has been successfully processed, and commit it to the database or to abort the transaction. Reliability and availability are intertwined here; On the one hand, one wishes to commit only when all servers agree on an answer, so that every local database has the same information; this gives high reliability, but potentially low availability. On the other hand, one wants to keep servers running as much as possible; this can give high availability but low reliability. Another problem is the *byzantine* problem which arises in case that processors yield faulty answers. [8] proves that if even one processor is faulty, a commit decision cannot be made. Because only one node on the net in our system can *change* attribute values, we do not have a commit problem, and the impossibility result is not relevant in this system.

A second problem is that of *database consistency* where the database is replicated in different network partitions and then changed. The issue is to reestablish consistency among the replications in different partitions. [4] discusses strategies for partitioning the database network into smaller sections and then achieving reliability and availability within partitions, together with a strategy for re-establishing consistency among partitions when they rejoin the network. Algorithms for the reestablishment of consistent information in this case are different to what we have used because the data can be modified by more than one node on the network. In the case considered in this paper, only one node can change particular attribute information, and all others treat it in a read-only fashion. We can therefore reestablish consistency by referring only to timestamps. In the distributed database case, however, the data would need consideration also. [9] solves the problem by having a database administrator's tool that can be used to patch the database; [15] partition the network and have a "coordinator" within each partition to deal with reestablishment of consistency; in their work control is therefore not decentralized. [3] assumes that the network size is known, and that its topology is sufficiently stable that each site can establish the topology before recovering, an assumption that we do not make, and also assumes coordinating sites. Further, they require "master" and "slave" sites whereas all our APLs are on an equal footing. Also, both require some knowledge of network topology, assume that some sites know they are "correct" while others know they are "recovering", and assume sufficient network stability to allow a number of rounds of messages between recovering sites and correct sites to reestablish consistency. Our algorithm makes none of these assumptions.

While not directly related to environments, [1] is relevant in that it proposes a system in which *processes* are highly available. In this system processes are made available and fault-tolerant by replicating them on many machines. Networks are assumed to be stable and of known size/topology, processes are assumed to know if they are correct or recovering, and recovery is made either by copying the state of a correct process and continuing or by rollback [5] [2]. We need none of these assumptions in our system, and recovery is by message-passing information, not by rollback.

In the case of distributed programming environments, the problems are different to those in the distributed database or replicated process areas, and it is these differences that make our solutions work. First, the number of attributes that flow among modules is a very small constant determinable from the attribute grammar. This means that the amount of data that must be replicated at each APL is very small compared with, say, a complete database. Thus, full replication is an effective strategy for making information highly available. Second, only one module within a specific APL is responsible for establishing new values of a particular attribute for propagation. Therefore the commit problem does not exist in our system. Instead we have a consistency problem: If an attribute is transmitted from an APL, every other APL must eventually receive the transmission. We do this with our consistency re-establishment algorithm. This algorithm has poor worst-case complexity, but it is very forgiving of network and machine failures. For example, if an APL α transmits a new attribute value, and it is received by some APL β, a third APL γ that is not on the net will eventually get the attribute when it returns even if α is no longer up, because the information will be automatically propagated from β as part of the consistency re-establishment process. It is not clear that the approaches cited in related work can match this feature.

Because each APL replicates the attributes of all other APLs, the remote attributes are highly available to a module. The values thus obtained may not be the latest possible, but local operations will continue and the later values of attributes are guaranteed to arrive as soon as network topology and machine status permit them to do so. In this we do better than [14], who also allow old information to be used but have no way of propagating newer information to clients when it becomes available. Because of the consistency re-establishment results, the system achieves the highest reliability possible in face of the reliability of the underlying network.

8 Implementation

We have implemented a prototype system that generates distributed language-based programming environments for VAX 11/750s connected by a 10Mbit ethernet. This system employs the simple strategy outlined in section 6.1 above, which assumes a reliable network. As the ethernet employs message broadcast we saw no need to use point to point communication, and because the number of APLs in the test system is very small, the complexity of the algorithm does not pose a practical problem at this stage. We are currently implementing a new system as an extension of the Cornell Synthesizer Generator [16], primarily to take advantage of their user interface. Our new system is designed to test algorithms for distributed attribute propagation and strategies for re-establishment of consistency of attribute information under the conditions of an unreliable network. Future development work is focusing on trying on optimizations of the basic algorithm to obtain better complexity results.

9 Conclusions

We discussed a programming environment where programmers and modules are distributed on a variety of machines and the system propagates change information among the modules as they are altered. We presented our solutions to the problems of making the system highly available and reliable in the face of unreliable machines and an unreliable network linking the machines. We employed several complementary attacks:

- By having fully *decentralized control*, any editor can operate independently of any other, regardless of the state of the network or the state of other machines on the network. Also, no "coordinator" is needed to ensure reestablishment of consistency.
- By *replicating attribute information* we have made the system *highly available* in that any editor can always get the most recent possible

values of attributes from remote modules.

- By introducing a provably correct *attribute consistency re-establishment algorithm*. the system is guaranteed to restabilize itself after machine and network failures by providing each APL with the most recent possible attribute information.

Our algorithm is very general and widely applicable in that it makes no assumptions about network topology or size, but it does have rather high worst-case complexity. If a particular network topology and/or size is given - as will often be the case in practice - then we can do much better than is implied by the worst case.

Our research addresses the increasingly important issues of distributed programming environments. Programming environments. also known as software engineering environments. are reaching a state of maturity where they are produced and used commercially. It is crucial in this context that solutions be found to the problems of high availability and reliability of these environments.

References

[1] Kenneth P. Birman. Replication and fault tolerence in the isis system. In *10th ACM SIGOPS Symposium on Operating Systems Principles*. pages 79–86. Orca Island, Washington, December 1985.

[2] Roy H. Campbell and Brian Rendell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, August 1986.

[3] Danco Davcev and Walter A. Burkhard. Consistency and recovery control for replicated files. In *10th ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–96, Orca Island, Washington, December 1985.

[4] Susan B Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341-370, September 1985.

[5] C. T. Davis. Data processing spheres of control. *IBM Systems Journal*, 17(2):178–198, 1978.

[6] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with applications to syntax-directed editors. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages (POPL)*, January 1981.

[7] Alan Demers, Anne Rogers, and Frank Kenneth Zadeck. Attribute propagation by message passing. In *Proceedings of the SIGPLAN '85 Symposium on Language Issues in Programming Environments*, pages 48–59, Seattle, WA, June 1985. Proceedings published as Gi'SIGPLAN Notices', 20(7), July, 1985.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[9] H. Garcia-Molina, T. Allen, B. Blaustein, R. M. Chilenskas, and D. R. Reis. Data-patch: integrating inconsistent copies of a database after partition. In *Proceedings of the 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 150–162, 1983.

[10] Gregory F. Johnson and Charles N. Fischer. Non-syntactic attribute flow in language based editors. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages (POPL)*, January 1982.

[11] 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 (PODC), pages 121–130, Calgary, Alberta, Canada, August 1986.

[12] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[13] Donald E. Knuth. Semantics of context-free languages: correction. *Mathematical Systems Theory*, 5(1), March 1971.

[14] Barbara Liskov and Rivka Landin. Highly available distributed services and fault-tolerant distributed garbage collection. In *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 29–39, Calgary, Alberta, Canada, August 1986.

[15] A. V. Ma and G. G. Belford. A failure and recovery detection protocol for optimistic partitioned operation on distributed database systems. In *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems*, pages 532–539, Cambridge, Mass., May 1986.

[16] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[17] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):449–477, July 1983.

[18] Andrew S. Tannenbaum and Robbert van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.