```
program ::= set of module;
     program.allexports =
          ∪_{i ∈ |modules|} (module_i.exports);


distributable module;
module ::= 'MODULE' id export import ... ;
     module.exports =
                    entry(id.name, export.exportid);


export ::= 'EXPORT' id;
     export.exportid = id.name;


import ::= 'FROM' id_1 'IMPORT' id_2;
     import.error =
                    if (id_1.name, id_2.name)
                        ∈ program.allexports
                    then ""
                    else "<-- cannot import this id";
```

Attribute grammar fragment for multiple segments

```
Decls:   synthesized attributes:  SymTabOut;
          inherited attributes:    SymTabIn;

Decl:    synthesized attributes:  SymTabOut, error;
          inherited attributes:    SymTabIn;

Id:      synthesized attributes:  Name;
          inherited attributes:    ∅;

Type:    synthesized attributes:  TpKind;
          inherited attributes:    ∅;
```

Context-free symbols of the attribute grammar and their attributes

```
p1  Program ::= ... Decls ...
                Decls.SymTabIn = NullTbl();

p2  Decls   ::=  /* empty rule */
                Decls.SymTabOut = Decls.SymTabIn;

p3  Decls1 ::= Decl Decls2
                Decl.SymTabIn = Decls1.SymTabIn;
                 Decls2.SymTabIn = Decl.SymTabOut;
                 Decls1.SymTabOut = Decls2.SymTabOut;

p4  Decl    ::= Id ':' Type ';'
                Decl.error = Member(Decl.SymTabIn, Id.Name)
                                ? "<-- Variable already declared"
                                : "";
                 Decl.SymTabOut =
                       Insert(Decl.SymTabIn, Id.Name, Type.TpKind);
```

Productions of the attribute grammar fragment and their semantic equations

A Attribute Grammar Examples

[48] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 41–48, Pittsburgh PA, April 1984. Special issue of *SIGPLAN Notices*, 19(5), May 1984.

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

[50] Lawrence A. Rowe and Sharon Wensel (editors). 1989 acm sigmod workshop on software cad databases. February 1989.

[51] R. W. Schwanke, E. S. Cohen, R. Gluecker, W. M. Hasling, D. A. Soni, and M. E. Wagner. Configuration management in biin sms. In *11th International Conference on Software Engineering*, pages 383–393, Pittsburgh PA, May 1989.

[52] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988. Technical Correspondence.

[53] Andrea H. Skarra and Stanley B. Zdonik. *Concurrency Control and Object-Oriented Databases*, pages 395–421. ACM Press, New York, 1989.

[54] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at kestrel institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, November 1985.

[55] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[56] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, New York, 1984.

[57] Janet A. Walz and Gregory F. Johnson. Incremental evaluation for a general class of circular attribute grammars. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 209–221, Atlanta GA, June 1988. Special issue of *SIGPLAN Notices*, 23(7), July 1988.

[58] Mark N. Wegman. Parsing for structural editors. In *21st Annual IEEE Symposium on Foundations of Computer Science*, pages 320–327, October 1980.

[59] Travis L. Winfrey and Gail E. Kaiser. Testing reliable distributed applications through simulated events. In *8th Symposium on Reliable Distributed Systems*, pages 160–166, Seattle WA, October 1989.

[60] Alan Zaring. Parallel attribute evaluation. Cornell University and Indiana University, 1986.

[36] Josephine Micallef. Incremental evaluation of ordered attribute grammars for asynchronous subtree replacements. Technical Report CUCS-380-88, Columbia University Department of Computer Science, July 1988.

[37] Josephine Micallef and Gail E. Kaiser. Version and configuration control in distributed language-based environments. In Jurgen F. H. Winkler, editor, *International Workshop on Software Version and Configuration Control*, volume 30 of *German Chapter of the ACM Berichte*, pages 119–143, Stuttgart, January 1988. B. G. Teubner.

[38] Josephine Micallef and Gail E. Kaiser. Support algorithms for incremental attribute evaluation of multiple asynchronous subtree replacements. Technical Report CUCS-450-89, Columbia University Department of Computer Science, November 1989.

[39] Josephine Micallef and Gail E. Kaiser. Extending the mercury system to support teams of ada programmers. In *1st International Symposium on Environments and Tools for Ada*, Redondo Beach CA, April 1990. In press.

[40] Erich Neuhold and Michael Stonebraker (editors). Future directions in dbms research. *SIGMOD Record*, 18(1), March 1989.

[41] Stephen B. Peckham. *Incremental Attribute Evaluation and Multiple Subtree Replacements*. PhD thesis, Cornell University, February 1990. TR 90-1093.

[42] Dewayne E. Perry and W. Michael Evangelist. An empirical study of software interface errors. In *International Symposium on New Directions in Computing*, pages 32–38, Trondheim, Norway, August 1985. IEEE Computer Society.

[43] Dewayne E. Perry and W. Michael Evangelist. An empirical study of software interface faults @y[m] an update. In *20th Annual Hawaii International Conference on Systems Sciences*, volume II, pages 113–126, Kona HI, January 1987.

[44] Calton Pu, Gail E. Kaiser, and Norman Hutchinson. Split-transactions for open-ended activities. In *14th International Conference on Very Large Data Bases*, pages 26–37, Los Angeles CA, August 1988.

[45] Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *9th Annual ACM Symposium on Principles of Programming Languages*, pages 169–176, Albuquerque NM, January 1982.

[46] Thomas Reps. *Generating Language-Based Environments*. The MIT Press, Cambridge MA, 1984.

[47] Thomas Reps, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language-based editors. In *13th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, St Petersburg FL, January 1986.

[24] Gail E. Kaiser and Dewayne E. Perry. Workspaces and experimental databases: Automated support for software maintenance and evolution. In *Conference on Software Maintenance*, pages 108–114, Austin TX, September 1987.

[25] Simon M. Kaplan. Incremental attribute evaluation on node-label controlled graphs. Technical Report UIUCDCS-R-87-1309, University of Illinois at Urbana-Champaign Department of Computer Science, May 1987.

[26] Simon M. Kaplan and Gail E. Kaiser. Incremental attribute evaluation in distributed language-based environments. In *5th Annual ACM Symposium on Principles of Distributed Computing*, pages 121–130, Calgary Alberta, Canada, August 1986.

[27] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[28] Randy H. Katz and Shlomo Weiss. Design transaction management. In *ACM IEEE 21st Design Automation Conference*, pages 692–693, Albuquerque NM, June 1984. IEEE Computer Society Press.

[29] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A transaction model supporting complex applications in integrated information systems. In *ACM-SIGMOD 1985 International Conference on Management of Data*, pages 388–401, Austin TX, May 1985. ACM.

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

[31] Donald E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, March 1971.

[32] Butler W. Lampson and Eric E. Schmidt. Organizing software in a distributed environment. In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1–13, San Francisco CA, June 1983. Special issue of *SIGPLAN Notices*, 18(6), June 1983.

[33] David B. Leblang and Jr Robert P. Chase. Computer-aided software engineering in a distributed workstation environment. In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104–112, Pittsburgh PA, April 1984. Special issue of *SIGPLAN Notices*, 19(5), May 1984.

[34] David B. Leblang and Jr Robert P. Chase. Parallel software configuration management in a network environment. *IEEE Software*, pages 28–35, November 1987.

[35] Raymond Lorie and Wilfred Plouffe. Complex objects and their use in design transactions. In *Engineering Design Applications, Annual Meeting Database Week*, pages 115–121, San Jose CA, May 1983. IEEE Computer Society Press.

[10] Bob Geitz. Asynchronous subtree replacement for language-based editors. Oberlin College and Cornell University, 1987.

[11] Carlo Ghezzi and Dino Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–579, July 1980.

[12] Irene Greif, editor. *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufman, San Mateo CA, 1988.

[13] William Harrison. Rpde^3: A framework for integrating tool fragments. *IEEE Software*, pages 46–56, November 1987.

[14] Maurice Herlihy. A methodology for implementing high concurrent data structures. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Seattle WA, March 1990.

[15] Roger Hoover. *Incremental Graph Evaluation*. PhD thesis, Cornell University, May 1987. Technical Report 87-836.

[16] Roger Hoover and Tim Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 39–50, Palo Alto CA, June 1986. Special issue of *SIGPLAN Notices*, 21(7), July 1986.

[17] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, pages 345–387, July 1989.

[18] Wenwey Hseush and Gail E. Kaiser. A network architecture for reliable distributed computing. *IEEE Network*, 2(4):28–44, July 1988.

[19] Scott E. Hudson and Roger King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems*, 14(3):291–321, September 1989.

[20] Larry G. Jones and Janos Simon. Hierarchical vlsi design systems based on attribute grammars. In *13th Annual ACM Symposium on Principles of Programming Languages*, pages 58–69, St Petersburg Beach FL, January 1986.

[21] Gail E. Kaiser and Elaine Kant. Incremental parsing without a parser. *The Journal of Systems and Software*, 5(2):121–144, May 1985.

[22] Gail E. Kaiser and Simon M. Kaplan. Reliability in distributed programming environments. In *6th Symposium on Reliability in Distributed Software and Database Systems*, pages 45–55, Kingsmill–Williamsburg VA, March 1987.

[23] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef. Multiuser, distributed language-based environments. *IEEE Software*, pages 58–67, November 1987.

bers of the Synthesizer Generator group at Cornell University provided much of the motivation for this work.

Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research. Kaplan is supported by the National Science Foundation through grant CCR-8809479 and by AT&T through the Illinois Software Engineering Project.

References

[1] Evan W. Adams, Masahiro Honda, and Terrence C. Miller. Object management in a case environment. In *11th International Conference on Software Engineering*, pages 154–163, Pittsburgh PA, May 1989.

[2] Hans-Juergen Boehm and Willy Zwaenepoel. Parallel attribute grammar evaluation. In *7th International Conference on Distributed Computing Systems*, pages 347–354, Berlin, West Germany, September 1987.

[3] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with applications to syntax-directed editors. In *8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, Williamsburg VA, January 1981.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, November 1976.

[5] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98, Palo Alto CA, June 1986. Special issue of *SIGPLAN Notices*, 21(7), July 1986.

[6] S. I. Feldman. Make – a program for maintaining computer programs. *Software – Practice & Experience*, 9(4):255–265, April 1979.

[7] 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.

[8] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[9] David Garlan. Views for tools in integrated environments. In *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 314–343. Springer-Verlag, Berlin, 1986.

unreliable distributed environment. Change propagation at a coarser granularity is obviously easier for file-based applications, such as mentioned above, but may not be easier for the database management systems (or the subclass known as object management systems or object-oriented database systems) now believed by many to be the basis for future environments (again see [50, 40]).

Database updates in this context may be viewed as subtree replacements of composite objects, where semantic tree nodes, their children and their attributes are "blown up" to database objects, their component objects and their attributes. The Cactis system [19] already applies single user, single edit, single cursor incremental evaluation algorithms to this problem. Because some databases support arbitrary links among objects as well as composite objects, it will be necessary to extend our multiple user, multiple edit, multiple cursor centralized and decentralized algorithms to subgraph replacements as well as subtree replacements. Preliminary results in this area have already been reported by Kaplan [25]. The object management system domain is an important area for future work.

7 Conclusions

We have presented algorithms that support (1) parallel evaluation on a centralized tree in response to a single edit; (2) parallel evaluation on a centralized tree operating concurrently with a sequence of multiple edits by the same cursor; (3) parallel evaluation on a centralized tree in response to multiple asynchronous edits by multiple cursors; (4) parallel evaluation on a decentralized tree in response to single or multiple asynchronous edits by different users; and (5) reliability of the evaluation algorithms as machines hosting segments of a decentralized tree become unavailable and return to availability. Algorithms (2) and (3) have not appeared previously, while the others have been substantially reformulated to take advantage of the insights gained in developing the new algorithms. Our algorithms are immediately applicable to multi-user language-based editors, but we believe that they also indicate directions towards a solution to the general change propagation problem for object management systems, particularly in the context of their application to multi-user software development environments.

Acknowledgments

Past and present members of the Mercury group are too numerous to mention, but we would like to single out Josephine Micallef, for many fruitful discussions that led to the clarification of our algorithms and for her contributions to the theory of incremental attribute grammar evaluation for asynchronous subtree replacements; Travis Winfrey, who pointed out some problems with our original approach to merging multiple propagations; and Mark Gisi, who found several ambiguities in an earlier version of this paper. We would also like to thank Steve Goering and Peter Sweeney for their comments. Tim Teitelbaum, Tom Reps and other former and present mem-

addresses the problems of allowing dependency graph updates while evaluation is in progress and of parallel propagation on a dependency graph. These cases correspond to the centralized parallel evaluation algorithms we presented in Section 3, and his work is the closest to ours. But since Hoover employs an approximate topological order, it is possible that his algorithm will unnecessarily evaluate the same attribute multiple times for even a single dependency graph modification.

Many software development environment projects have investigated *ad hoc* mechanisms for module interface checking. The Unix `make` tool automatically invokes specified manufacturing steps on all files dependent on other files that have changed; the interface checking, as a side-effect of recompilation and linking, is incremental at the granularity of files – if `make` is invoked every time a file is edited. During a system build, the Cedar System Modeller [32] tracks down the modules distributed across the network, copies them to one place, and checks all interfaces. Apollo's Domain Software Engineering Environment (DSEE) supports parallel builds [34]. It also provides a change notification mechanism [33], where a programmer can put a "marker" on any file to trigger an arbitrary action when the file is modified. NSE provides a similar facility, and also supports an extended transaction model for detecting conflicts and enforcing merging of changes [1]. BiiN's Software Management System (SMS) [51] extends the DSEE ideas in a different direction, supplying tools for extracting dependencies among modules from their files and a short-circuit mechanism for avoiding undesired recompilations.

Tichy's "Smart Recompilation" [55] is an optimization to all these file-based tools, where the decision of whether or not to recompile a file is based on whether any component of the file is actually dependent on the specific change made to another file; Schwanke and Kaiser [52] optimize this further, to distinguish safe from unsafe decisions to avoid recompilations that would by required by Tichy's algorithm. When used for compilation in addition to interface checking, incremental attribute grammar evaluation does even better than smart recompilation since the minimum subtrees within segments are recompiled (segments correspond roughly to files).

Most of the work described above does not address reliability of the distributed environment. In a previous paper [22], we compared our approach to reliability against those followed in distributed database systems, and explain why our algorithm is more suitable for change propagation in distributed LBEs but inadequate for distributed databases, due to the latter's requirement of serializable transactions [4]. Serializability is not an issue for incremental attribute evaluation, since all reorderings of the same set of subtree replacements that result in the same tree also result in the same set of attribute values. It is generally believed among software engineering and database researchers that serializability must be relaxed in database support for software development environments in general, to allow other forms of cooperation between multiple developers in addition to the change propagation described here (*e.g.*, see [50, 40]).

We take the viewpoint that the hard components of the change propagation problem are to support multiple asynchronous replacements at the subtree replacement granularity, both intra-segment and inter-segment, as well as in the context of an

segment propagation on and off. The implementation also makes it possible to selectively determine propagation with respect to particular remote hosts, allowing immediate change propagation within groups of users (using those hosts) working closely while avoiding interfering with other users who prefer to continue to use the previous versions of the rapidly changing modules.

6 Related Work

The subject of incremental attribute evaluation for multiple asynchronous edits and/or across a decentralized tree had not, to our knowledge, been previously investigated before our initial effort [26]. Boehm and Zwaenepoel [2] study parallel attribute grammar evaluation, but their goal is parallel compilation and they do not treat editing or incremental evaluation. Zaring [60] has developed an algorithm for parallel incremental evaluation for single edits on a centralized tree.

Reps, Marceau and Teitelbaum [47] present an algorithm that supports multiple *synchronous* subtree replacements. It keeps track of the interdependencies among the attributes affected by the distinct edits using an extension to Kastens's algorithm [27], in which all possible attribute dependencies among all pairs of nonterminal symbols are determined. These extended graphs are then used to construct and expand the model. This work was in fact the original inspiration for our attack on the asynchronous edits problem. Geitz [10] has more recently applied this approach to attribute evaluation for asynchronous edits, for the special case where the number of asynchronous edits proceeding in parallel with attribute propagations is bounded. This result would require some extensions analogous to our firewalls to support full decoupling of editing and evaluation. In any case, both of these algorithms work only for a restricted subclass of the noncircular attribute grammars (those that pass extensions of the first two steps of Kastens's algorithm).

Micallef [36] has developed an efficient algorithm for incremental evaluation for asynchronous edits on either centralized or decentralized trees, but its application is also limited to a subclass of attribute grammars (ordered attribute grammars, which pass all four steps of Kastens's algorithm). Peckham [41] has developed a similar algorithm for a subclass of the ordered attribute grammars, which he calls globally partitionable attribute grammars. Both of these algorithms use static evaluators, while most other work in this area employs dynamic evaluators. Although static evaluators typically have higher performance than dynamic evaluators, it does not seem possible to apply static techniques to the general noncircular attribute grammars, but only to restricted subclasses.

In his dissertation, Hoover [15] is primarily concerned with incremental graph evaluation, to support modifications of dependency graphs and evaluation of influenced vertices of the graph. He discusses incremental attribute evaluation for general noncircular attribute grammars in this framework, which also applies to a range of other incremental computation problems. Most of the thesis is based on the paradigm where the dependency graph modification and propagation alternate, and both are done sequentially. In chapter 8, Hoover relaxes this assumption and briefly

do not consider the possibility of gateways and other intermediate nodes separating portions of the network linking the machines hosting the segments of a single decentralized tree. However, there is no reason that this message cannot be generated by such an intermediary in the case that a previously physically partitioned network is now rejoined, to reestablish global consistency across the full network.

One concern with our reliability protocol is that the network traffic may be large, particularly since entire caches are broadcast (or multicast) rather than individual attributes. However, the number of interface attributes is a constant determined when the attribute grammar is written, and is rarely more than one pair in the module interface checking application, representing the symbol table of each segment and the conglomerate symbol table, respectively. Unfortunately, the size of a segment's symbol table or other synthesized interface attributes may be large; inherited interface attributes are fortunately never transmitted, but are always computed locally. Symbol tables are *aggregate attributes*, meaning they are composed of many independent entries. If reliability were not an issue, we could use the binary relations described by Micallef and Kaiser [37], which extend the finite functions developed by Hoover and Teitelbaum [16] to reduce evaluation costs in the classical algorithm. Binary relations provide a mechanism whereby only the actually changed entries of the aggregate attributes are transmitted, and only to those APLs where there is a local segment that actually uses that entry. But our reliability protocol depends on full replication of the interface attributes cache in the APL of every host where the distributed LBE resides.

There is only one APL process per machine, even though there may be segments in multiple *distinct* systems, in the same or different programming languages, residing on the same machine. Therefore, it must be possible for the APL to know which system a segment belongs to as well as which segment an interface attribute belongs to. The APL needs only minimal knowledge about the attribute grammar that defines the language for a particular segment, in particular the mapping from the names of synthesized interface attributes to the corresponding inherited interface attributes.

Several other practical considerations must be addressed in order to employ our algorithms in real software development environments. Creating new systems, adding and removing modules from systems, changing module names, unintentionally creating multiple modules with the same name, using a library module in multiple systems, multiple versions of modules, multiple system configurations, and other inevitabilities of real software development are relatively messy, but do not affect the incremental evaluation algorithms. Most of these problems are addressed in a previous paper [23] or by Micallef and Kaiser [37, 39].

A common "criticism" of the software development model behind this work is that each user should have control over the granularity of commitment of his interface changes (*i.e.*, outgoing inter-segment change propagation), as well as the granularity of notifications of interface changes by other users affecting his modules (incoming inter-segment change propagation). But these capabilities are in fact provided in the Mercury implementation, by simply providing a user command to turn inter-

5.3 Pragmatics

The protocol as described above seems to use the network software's broadcast primitive to send every message to every machine throughout the network, even though some machines may not support the distributed LBE. Unless broadcast is extremely cheap, this is probably a bad idea. One improvement is to multicast rather than broadcast, according to an APL host table maintained as part of each APL process. This does not require that the number of APL processes be "fixed", with a centralized process to update the database of known APLs on each host every time a new APL is added. Instead, the **update** message can include its network address and so forth, which would be added to its local table by each responding APL.

We can add a degree of fault-tolerance for the case where an APL is running but other APLs do not know this by employing any of three supporting protocols described by Hseush and Kaiser [18]. Their goal is to keep the local view of the machines available on the network (in this case the APL host table) very close to the global (actual) view (the set of APL hosts actually up and accessible at this point in time). The three protocols range from basing local views on the usual acknowledgments and timeouts, with pings at certain time intervals to check whether the local view is correct; piggybacking representations of the local view on top of other messages, to detect disagreement among APL hosts regarding their local views; and informing all the other accessible hosts whenever an APL process notices that another host has failed (or that a host previously believed to have failed is now accessible).

In the previously published version of our cache consistency protocol [22], the APL's *Receive* procedure (figure 17) behaved differently when it received a cache older than its local copy. In that case, rather than do nothing, it broadcast its more recent cache, so that the sending APL with the older version along with all the others currently accessible on the network would get the more recent information. In the case that the local APL itself had out-of-date information, this broadcast might reach another APL with even more recent information, which would then broadcast, and so on. This can quickly get very expensive in terms of network traffic. Our rationale for this extra load was that network partition is "silent". Two APLs may not be able to communicate for a period, even though both are operating correctly and are connected to the network; it is possible that there is no real failure, where later restoration would lead to an **update** request, but instead just a transient partition due to network congestion or some other problem that will soon correct itself – but in the meantime the two APLs may have become inconsistent. However, using only the original protocol they would have had no way of knowing that they have become inconsistent until the next time some other APL issues an **update** request. But the additional protocol described by Hseush and Kaiser allows a nice solution to this problem: an **update** request is generated whenever an APL process previously believed to be unavailable is discovered to be available, and so the much more efficient version of *Receive* described in this article is sufficient.

We have described the **update** request as coming from an APL process, and

```
Send(b)
 let
  b = an attribute instance
  i = a segment identifier
  t = a timestamp
 in
  t ← generate timestamp
  update timestamp of local cache to t
  save synthesized interface attribute b in local cache for i
  pack message with entire local cache for i
  broadcast message across network
 end
```

Figure 18: Send for Unreliable Network

If, instead, the newly arrived cache is the same as the local information, the APL does nothing. This will actually be the case most of the time, since each APL responds to the **update** message by broadcasting all its caches to every other APL process, not just to the newly accessible APL process that originated the message. Further, the APL does nothing when it receives a cache older than its local copy; it is not necessary to inform the APL that sent the old information, since this APL will eventually receive the more recent local cache already broadcast. This version of the *Receive* procedure also addresses the new scenario where a cache associated with a previously unknown segment arrives, because that segment was created while the network was unavailable. The new cache is installed, and all the interface attributes in the cache are propagated as a group into each local segment.

The *Send* procedure of the previous section broadcasts the individual synthesized interface attribute that had changed; now we modify *Send* to broadcast the entire local cache for the segment rather than just the individual attribute (figure 18). This has the advantage of subsuming normal inter-segment attribute propagation into the consistency reestablishment protocol, rather than requiring one version of *Receive* for caches and another for individual attributes, and avoids the problem of maintaining separate timestamps for each synthesized interface attribute.

Although an APL broadcasts an **update** request each time it comes up, and therefore it is theoretically the only out-of-date APL, the protocol is fault-tolerant regarding the situation where all APLs have different information but think that they are up-to-date. In this case the protocol will reestablish the most up-to-date information at each APL, since they all receive each other's broadcast caches. (Termination requires sufficient network stability that there will not be another **update** request before the APLs are done inspecting the caches received for the previous one.) This extra degree of safety is an advantage over an alternative approach where the original **update** message triggers a leader election [8] and then only the leader actually transmits the caches.

```
Receive(message)
 let
  message = a segment identifier i, a cache C and a sender D
  D = identification of a machine
  C, L = caches of interface attributes
  i, j, k = segment identifiers
  b, c = attribute instances
  caches = complete set of attribute caches (local copies)
           for all segments in tree
  segments = set of local segments
  r = root of a fully attributed semantic tree representing segment
 in
  unpack message
  C ← newly arrived cache for segment i
   if ∃ a local cache for segment i  then
    L ← old cache for i
     if timestamp of C > timestamp of L  then
      replace L with C
      ∀ b in C different from corresponding b in L  do
        c ← ∪j ∈ caches b attribute for cachej
        ∀ roots r of local segments  do
              suspend until FireWallUp(r,  root)
              update inherited interface attribute c
              FireWallDown(r,  root)
         od
       od
     fi
   else
      install new cache C
      ∀ b in C  do
        c ← ∪j ∈ caches b attribute for cachej
        ∀ roots r of local segments  do
              suspend until FireWallUp(r,  root)
              update inherited interface attribute c
              FireWallDown(r,  root)
         od
       od
  fi
 end
```

Figure 17: Receive for Unreliable Network

```
Update()
 let
 in
  pack message with  update notice
  broadcast message across network
 end

Respond()
 let
  L = a local cache for one segment
 in
  ∀ local cache L  do
    pack message with entire local cache L
    broadcast message across network
   od
 end
```

Figure 16: Update and Respond for Unreliable Network

a more recent value of the same attribute has already arrived. Since a segment can only be edited on one machine at a time, there is no need for a global clock. Timestamps can simply be integers that are incremented by the originating APL each time an attribute is broadcast [7].

The reliability protocol is based on new versions of the *Update* and *Respond* procedures (figure 16, *Send* (figure 18) and *Receive* (figure 17). The protocol works as follows: When a machine comes up, or when its broken connection to the network is restored, its APL process broadcasts the **update** request over the network using the revised *Update* procedure. This message will eventually be received by all APLs that are accessible from this APL. On receipt of an **update** request, each APL process uses the *Respond* procedure to broadcast every local cache including its timestamp. Note that the **update** message is broadcast on the network, not just to a distinguished leader process, and *every* APL that receives the message responds with its full set of caches; this is because the leader itself might be down or inaccessible.

Further, the APL that originates the **update** also responds, broadcasting its local caches. This is necessary in the case where the network connection was broken and the local APL has now been reconnected, as opposed to the machine itself being down and now restored. Local segments may have been edited while the network was unavailable, resulting in changed interface attributes that must be propagated to the rest of the network.

On receipt of a cache, each APL executes the *Receive* procedure, given in figure 17, which has been modified to treat timestamps and entire caches. Propagation into the local segments is carried out only if the newly arrived cache has a more recent timestamp than the previous local copy of the cache for the same segment.

```
Update(D)
 let
  D = identification of new machine
 in
  pack message with  update notice and D
  send message to leader APL
 end

Respond(D)
 let
  D = identification of new machine
  L = a local cache for one segment
 in
  ∀ local cache L  do
    pack message with entire local cache L
    send message to D
   od
 end
```

Figure 15: Update and Respond for Reliable Network

5.2 Attribute Propagation on an Unreliable Network

In this section, we consider the practical problems of unreliable networks and unreliable machines. We assume that messages are never corrupted or lost, but otherwise make no assumptions about the reliability of the network; we do not assume that messages arrive at a machine in any particular order. We assume that the machines themselves are failstop, that is, they are either running correctly or down. Whenever a machine is up, its APL process 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 segment 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.[7]

In this context, the APL has the task of reestablishing globally consistent attribute information after failures. In other words, the APL solves the problem of consistency among replicated data (the caches) in the context of decentralized attribute propagation. Each APL process maintains a cache for each local and remote segment, now containing a timestamp as well as attribute information. The timestamp indicates the time of the most recent reevaluation that resulted in a changed value of some synthesized interface attribute from that segment. This is necessary because an old value of an interface attribute might arrive from a remote APL after

[7] Winfrey and Kaiser [59] have developed a tool for testing message passing applications, and its first use was testing how the Mercury implementation behaved when these assumptions are violated. Yes, they found bugs.

```
Send(b)
 let
  b = an attribute instance
  i = a segment identifier
 in
  save synthesized interface attribute b in local cache for i
  pack message with i and b
  broadcast message across network
 end

Receive(message)
 let
  message = a segment identifier i and an attribute b
  b, c = attribute instances
  i, j = segment identifiers
  caches = complete set of attribute caches (local copies)
           for all segments in tree
  segments = set of local segments
  r = root of a fully attributed semantic tree representing segment
 in
  unpack message
  save b in local cache for i
  c ← ∪_{j ∈ caches} b attribute for cache_j
  ∀ roots r of local segments do
        suspend until FireWallUp(r, root)
        update inherited interface attribute c
        FireWallDown(r, root)
   od
 end
```

Figure 14: Send and Receive for Reliable Network

up. If this would conflict with the editing cursor's firewall, the APL waits until the cursor's firewall goes down. Then it simulates a subtree replacement at the root of the segment, in order to update the interface attribute. When the root firewall goes down, the new value of the inherited interface attribute is propagated by the $FireWallDown$ procedure.

Each APL process maintains a local cache of the most recent value of every synthesized interface attribute for every local or remote segment (including those that are dormant). These caches are needed so that when local attributes that depend on the values of dormant and/or remote attributes need to be evaluated due to local changes, it is not necessary to load the dormant segment from disk or communicate with the remote site, respectively, to get the arguments of the semantic equation. Also, this information is needed by our algorithms for the unreliable network case, discussed in Section 5.2; notice that the attribute cache "database" is fully replicated.

The *Send* and *Receive* procedures introduced in Section 4 are modified as shown in figure 14 (1) to handle transmission of attributes across a network in addition to between processes on the same machine, and (2) to distinguish between segment caches, which are fully replicated in the APL process of every machine, and the actual segments, which reside only on one machine. The *Send*ing APL also participates in the *Receive*, to propagate to the local segments, but this special case is not distinguished; in the following section concerned with unreliable networks, it is important for correctness that this case not be distinguished.

If the network is completely reliable, there are only two situations in which consistency among caches must be reestablished, namely a new machine is added to the network or a machine has previously been removed in an orderly fashion and is now being returned. We assume that no machine containing relevant segments is ever removed from the network, but instead its dormant segments are first moved to other machines. (We do not discuss segment movement, except to say that the corresponding cache identifiers would have to be modified.)

In both cases, the local APL process transmits an **update** packet to a distinguished "leader" APL process using the *Update* procedure. The leader receives the **update** packet and uses the *Respond* procedure to return its local cache information to the originator of the **update**. For the new case, this gives the originating APL a set of caches to make available to its local segments, and in the restarting case the originating APL can now decide what attributes have changed with respect to the previous contents of each cache and propagate them to local segments. No segments local to the new/restarted APL may be edited during this initialization process. The *Update* and *Respond* procedures are shown in figure 15. The *Respond* procedure could be rewritten to combine all the caches maintained by the leader APL as a single large message, which would be broken down into packets by the operating system's network software, but the separation among caches for different segments becomes relevant in the unreliable network case discussed in the next section.

interface problem regarding change simulation is exactly what to tell the user: should he see the actual source code inside each affected module, with the "error" attributes displayed – thus breaking at least encapsulation and possibly access rights, or should he be told simply that new static semantic errors have been introduced into each of a list of modules?

Kaiser and Perry [24] have considered both change propagation and change simulation in the context of large scale software systems involving large teams of programmers. They advocate isolation of change propagation to subgroups of the programmers, with groupings determined according to an interdependency metric with respect to the modules being edited by the programmers. This is motivated by two hypotheses: (1) interface errors are most likely to occur among closely coupled modules, by a simple probabilistic argument, and (2) it is counter-productive to overload the programmers with information about modules only weakly coupled to their own. Kaiser and Perry also discuss a facility for temporarily breaching this isolation for change simulation purposes when special circumstances warrant, in order to detect interface mismatches early rather than late, when they are usually much more expensive to repair.

5 Attribute Propagation Layer

Our decentralized attribute evaluation algorithm assumes an attribute propagation layer (APL) that resides on every machine hosting the distributed software development environment. The layer implements a protocol that supports propagation among segments on the same or different machines and provides facilities to support recovery from failures. For ease in exposition, we first describe the APL assuming a reliable network and reliable machines, and then explain the recovery mechanisms for machine and network failures.

5.1 Attribute Propagation on a Reliable Network

The purpose of the APL is to support propagation of interface attributes among segments. The APL consists of a process continuously running on every machine. For a particular APL process, the segments being edited on the same machine are *local* segments, as are *dormant* segments not currently being edited by any user but stored on the local disk (if there is one). All other segments are *remote* segments. From the viewpoint of a segment, there is one *local* APL process and all the others are *remote*.

When a synthesized interface attribute of a segment changes in value, its local APL broadcasts it to all the remote APLs, where the corresponding inherited interface attribute is computed as described in Section 4 (*i.e.*, by collection into an unordered set by the **union** operator). Then each of these APLs updates the corresponding inherited interface attribute in all its local segments. To do this, the local APL interacts with the root firewalls of each local segment. When an attribute propagation is transmitted to a segment, the APL attempts to bring its root firewall

which, and setting up interprocess communication are omitted. All *Send* does is save the outgoing attribute in a cache and call *Receive*; in the distributed case, discussed in Section 5, *Send* instead broadcasts the changed attribute across the network, and it is received by an APL process on every machine. Thus it makes sense for *Receive* rather than *Send* to compute the incoming attribute value, so less information must be transmitted across the network. *Receive* first computes the new value of the inherited attribute as the **union** of all the corresponding synthesized attributes from all the caches (each representing a segment). It then waits until it can bring the root firewall up (which happens immediately unless the regular editing cursor is currently pointing to the root of the segment), updates the interface attribute, and brings the root firewall down to initiate propagation into the segment.

As with the centralized tree case of Section 3.4, attribute evaluation considering only a single segment is optimal, that is, no more than $\bigcirc(|\mathit{Influenced}|)$ attributes are evaluated in response to propagations initiated from within the segment. The restrictions placed on inherited interface attributes make it possible to subsume the dependency between the paired synthesized and inherited interface attributes into the original model, that is, for the segment in which the synthesized interface attribute changed in value. The propagation in this segment can wait for *Receive*'s simulated subtree replacement at the root of the segment before evaluating any attributes that depend on an inherited interface attribute. Code to handle this special case is omitted from the figures.

4.3 Pragmatics

Our simple segmentation scheme has been extended by Micallef and Kaiser [39] to support hierarchical segments and multiple kinds of segments with possibly different sets of interface attributes, which is required for complex languages like Ada. Their extended segmentation scheme does not require any modifications to the decentralized incremental evaluation algorithm presented here, except that it is no longer practical to "wait" for the receipt of the new value of an arbitrary interface attribute at an interior segment leaf or root because the computation of these attributes is more complex. In particular, Micallef and Kaiser describe different extensions to the attribute grammar formalism and new operations defined on what they call *conglomerate* attributes, which are the same as our inherited interface attributes constructed with the **union** operation, in order to detect naming errors, determine compilation context and check that there is a noncyclic compilation order.

A special case of our decentralized algorithm, to simulate global propagation of changes to the interface attributes of a single segment, is useful in its own right to apply the power of parallel processing to recompilation after changes and to support *change simulation*. Change simulation enables a user to ask a "what if?" question: "how would this proposed change affect the rest of the software system?" In this case, the inter-segment propagation algorithm would be applied to copies of the other modules in order for the results to be invisible to the other users and returned only to the curious user, who has not yet committed his interface change. One user

```
Send(b)
 let
  b = an attribute instance
  i = a segment identifier
 in
  save synthesized interface attribute b in cache for i
  Receive(b)
 end

Receive(b)
 let
  b, c = attribute instances
  caches = complete set of caches for segments in tree
  segments = complete set of segments in tree
  i = a segment identifier
  r = root of fully attributed semantic tree representing segment
 in
  c ← ∪i ∈ caches b attribute for cachei
  ∀ roots r of segments do
    suspend until FireWallUp(r, root)
    update inherited interface attribute c
    FireWallDown(r, root)
   od
```

Figure 13: Send and Receive for a Single Machine

```
FireWallDown(r, u)
 let
  T = a fully attributed semantic tree, shared
  NeedToBeEvaluated = a set of attribute instances, shared
  M = a directed graph, shared
  r = a node of T containing any inconsistent attributes
  N = a directed graph
  S = a set of attribute instances
  u = either  root or null
  b = an attribute instance
 in
   if u =  root  then
    ∀ attributes b of r and its children  do
          color b green in D(T)
     od
   else
    ∀ attributes b of r and its parents and siblings  do
          color b green in D(T)
     od
   fi
  N ← r.C ∪ r.C̄
  M ← M ∪ N
  insert all attributes in N into NeedToBeEvaluated
  S ← the set of vertices in M with indegree 0
  ∀ vertex b ∈ S  do
     if an evaluation thread for b is suspended  then
          resume evaluation thread for b
     else
          send Propagate message to b
     fi
   od
 end
```

Figure 12: FireWallDown for Multiple Segments

```
atomic FireWallUp(r, u)
let
 T = a fully attributed semantic tree, shared
 M = a directed graph, shared
 r = a node of T to be replaced by an edit command
 u = either  root or null
 b = an attribute instance
in
  if u =  root  then
    if the editing cursor points to r or a child of r  then
     return false
    else if r or a child of r is in M  then
     return false
    else
     ∀ attributes b of r and its children  do
           color b red in D(T)
      od
     return true
    fi
  else
    if any node in T_r, the parent of r,
         or a sibling of r is in M
    then
     return false
    else
     ∀ attributes b of r and its parents and siblings  do
           color b red in D(T)
      od
     return true
    fi
  fi
end
```

Figure 11: FireWallUp for Multiple Segments

```
Propagate(b)
 let
  T = a fully attributed semantic tree, shared
  NeedToBeEvaluated = a set of attribute instances, shared
  M = a directed graph, shared
  b, c = attribute instances
  changed = Boolean
  OldValue, NewValue = attribute values
 in
   if b in D(T) is colored red  then
        suspend evaluation thread for b
   else
    changed ← false
    if b ∈ NeedToBeEvaluated  then
     remove b from NeedToBeEvaluated
     OldValue ← value of b
     evaluate b
     NewValue ← value of b
     if OldValue ≠ NewValue  then
      changed ← true
      if b is a synthesized interface attribute  then
           Send(b)
      fi
      if M does not contain all the successors of b in D(T)
      then
           Expand(b)
      fi
     fi
    fi
   ∀ successor c of b in M  do
     remove edge (b, c) from M
     if changed  then
      insert c into NeedToBeEvaluated
     fi
     if indegree of c in M is 0  then
      if an evaluation thread for c is suspended  then
           resume evaluation thread for c
      else
           send Propagate message to c
      fi
     fi
    od
  fi
 end
```

Figure 10: Decentralized Attribute Propagation

so they are always available within the process to compute the **union** when needed. The root of the decentralized semantic tree can be thought of as residing in the APL, although it need not be explicitly represented as such, and segments are added to and removed from a tree by informing the APL; the details of how this is done are not discussed here. The *Receive* procedure simulates a subtree replacement at the root of a segment, using its root firewall, in order to initiate propagation after a change to an inherited interface attribute. The APL is explained further in Section 5.

The *Propagate*, *Expand*, *FireWallUp* and *FireWallDown* procedures are defined to operate only within a single segment: the semantic tree T represents only a single segment, and M, $NeedToBeEvaluated$, T and $D(T)$ are shared only among the invocations of these procedures applied to the *same* segment. To support inter-segment propagation, we modify the *Propagate* procedure for asynchronous edits slightly in order to handle interface attributes, as shown in figure 10. When a synthesized interface attribute changes in value, the *Send* procedure is called to propagate the change to all the other segments. There is a corresponding *Receive* procedure, explained later, that raises the root firewall, updates the inherited interface attribute, and brings the firewall down to propagate the change into each segment. The *Expand* procedure is the same as for asynchronous edits with multiple cursors (figure 6). There is no global scheduling among segments, and thus the propagation algorithm is fully decentralized.

Recall that in Section 3.4, it was not allowed for a firewall to come up if there was another editing cursor inside the region covered by the firewall; we also ignored the possibility of a subtree replacement at the root since this would have replaced the entire centralized tree. In our new context of multiple segments, however, there is *always* an editing cursor inside the subtree protected by the root firewall, that is, the subtree corresponding to the entire segment; further, if there is any propagation in process at all in the segment, some node inside the subtree protected by the root firewall will be in the model. The root firewall must be allowed to come up as long as the editing cursor is not pointing to the root of the segment or one of its immediate children and neither the root nor one of its children is in the model. We consider the segment root and its children, rather than the subtree replacement root and its parent and siblings, because now the propagation comes from above rather than from below. Thus only the attributes of the segment root and its immediate children are colored red in this case, and only these attributes should be colored green when the firewall comes down.

The *FireWallUp* and *FireWallDown* procedures are modified to handle the root firewall specially, and each is supplied an additional argument to indicate whether it is being invoked with respect to the root firewall or the editing cursor firewall; this is depicted in figures 11 and 12, respectively. Note that *FireWallUp* and *FireWallDown* both assume only a single editing cursor, but the changes to support multiple editing cursors are straightforward combinations of the procedures shown here and those employed in Section 3.4.

The basic forms of *Send* and *Receive* are given in figure 13. Details such as initializing a cache for a new segment, keeping track of which interface attribute is

2. This **set** construct is restricted to appear only on the right hand side of the production that defines the start symbol (or axiom) of the grammar. The effect is to represent the program as a two-level hierarchy, with a system root and an unordered collection of segments. This simplifies the representation of segment interconnection information to only the one set.

3. The goal symbol of a selected production must be declared to be a **distributable unit** (*i.e.*, the root of a segment), and this distributable unit must also be the element type of the **set** described above. The effect is to restrict the system to one kind of segment. This guarantees that all segment root nodes will have the same attributes.

4. Certain attributes of the distributable unit symbol are declared to be *interface attributes*, meaning they represent dependencies across segments (*e.g.*, the imports and exports lists of modules). This makes it clear when attribute evaluation must propagate across segment boundaries.

5. Interface attributes are defined in pairs, a synthesized interface attribute and an inherited interface attribute. When a synthesized interface attribute in one segment changes in value, the corresponding inherited interface attribute in all segments must be evaluated. This simplifies the dependencies among the interface attributes of segments.

6. The semantic equation for an inherited interface attribute of a segment is limited to the **union** of the corresponding synthesized interface attributes of all segments. In the standard attribute grammar formalism, such attributes would instead be defined as concatenations of attributes from nodes in the skewed trees, so there is no significant difference except for lack of ordering. But by explicitly limiting the definitions of inherited interface attributes in this way, we simplify the implementation of the root of the decentralized semantic tree, as explained in Section 4.2. Other than the addition of union and this restriction, we assume that semantic equations can use the same primitive operations supported by the Synthesizer Generator.

A fragment of an example extended attribute grammar, specifying attribute propagation across multiple segments, is given in the appendix.

4.2 Multiple Segment Evaluation

To support attribute propagation across segment boundaries, *i.e.*, distributed evaluation, we define a new process called the *attribute propagation layer* (APL). The interface between a user process and the APL is provided through two new procedures, *Send* and *Receive*. *Send* is used to propagate a change to a synthesized interface attribute out of the segment. The APL computes the corresponding inherited interface attribute and transmits it to all the segments. To do this, it keeps a cache of the current values of all the synthesized interface attributes of all segments,

editing (*i.e.*, a module) and the unit of distribution (and also typically the unit of separate compilation in languages whose compilers support separate compilation). This aspect of our model is identical to the mode of software development common when using traditional tools, where each user edits only one module at a time in a text editing buffer.

The primary difference between our software development model and the traditional approach arises when a programmer edits the interface to his or her module. If one user changes the facilities exported in such a way as to introduce static semantic errors into other modules, these other users are *immediately* informed of the errors as soon as the editing command is issued. In contrast, in the traditional approach, these errors are detected only after the full editing sessions have been completed and someone runs `make` [6] or a similar tool to rebuild the system. By this point, several of the programmers may have already spent many hours editing their modules based on no longer valid assumptions about the interfaces of other modules.[6] It would of course be possible to back up to previous versions in these cases, but this wastes the efforts of the users who generated the new versions as well as not allowing much forward progress. Our approach is to inform all users of each interface change that affects them as soon as possible.

Thus the (conceptual) semantic tree representing the entire software system is divided into segments. There is one user, one editing cursor and two firewalls per segment; the second firewall, called the *root firewall*, is used to protect incoming changes, as explained in Section 4.2. Each user moves his cursor and makes subtree replacements within his own segment. Propagation among segments can arise only if a subtree replacement within a segment causes an attribute at the root of the segment to change in value. In this case, there may be other attributes at the roots of other segments that depend on the changed attribute, and its new value must be propagated to these other segments. In this section we present our primary result, an algorithm for decentralized attribute evaluation across multiple segments.

4.1 Attribute Grammar Extensions and Restrictions

To support fully decentralized attribute evaluation, we make a few restrictions and modify the standard attribute grammar formalism as follows:

1. A new construct, **set**, is added to the context-free grammar notation as a means for defining an unordered sequence; standard context-free grammars can define only ordered sequences, by a skewed tree using a left- or right-recursive pair of context-free productions. This extension makes it possible to specify a collection of segments without requiring an explicit order among the segments. The goal is to avoid the overhead of synchronization among multiple users that would be needed to specify a particular ordering as new segments are added to a system during development.

[6] An empirical study of a large software system has shown that about 52% of reported errors involve interface problems among multiple files [42, 43].

3.5 Pragmatics

We have presented support for multiple editing cursors, without discussing the issues of multiple *human* users controlling those cursors. There are a few problems with multiple users editing the same centralized tree. First, it will be necessary for multiple user processes to share memory, or at least appear to share memory. Essentially any technique for simulating shared memory would reduce response time to some extent. There is what might be called a sociological problem with "flickering" error attributes, as one user makes some edit that happens to correct a static semantic error introduced only a moment before by another user, and all users see the error message appear and disappear on their displays. These kinds of issues are being addressed by researchers concerned with computer-supported cooperative work [12], but they are far from being solved.

Fortunately, while it may sometimes be necessary for multiple users to simultaneously edit the same module, it is usually acceptable in these cases that they edit separate *copies* and merge together these divergent versions later on. Sun's Network Software Environment (NSE) [1] detects when multiple users have edited the same file and provides a text-based tool to help with merging two files; Horwitz, Prins and Reps [17] have developed an algorithm that detects semantic interference and automatically integrates some classes of changes.

However, supporting multiple *cursors* operating asynchronously within a semantic tree might solve part of the problem of parallelizing a program transformation system (such as Refine [54]). Indeed, we see our algorithm as a potential enabling technology for the development of new software development environment paradigms where several tool fragments (for example, [9, 13]) may simultaneously modify the same module.

Multiple users as well as multiple editing cursors must be considered in order to apply our results to database management systems, assuming editing within some kind of shared database rather than a "checkout" model [28, 35]. Some kind of cooperative transaction scheme would be needed, as proposed by many researchers, for example, [29, 53, 44]. This seems to be a promising area for future work.

4 Decentralized Attribute Propagation

We now turn our attention to the problem of parallel evaluation on a decentralized tree distributed among multiple user processes. By a user process, we mean a heavyweight operating system process, with protected virtual memory, its own thread(s) of control, *etc.* In Section 3 we considered multiple lightweight threads of control executing within a single shared virtual memory on a shared representation of the centralized tree; thus the entire system was within what we now consider a single user process.

Our algorithm assumes a model of software development where several programmers work together to develop and maintain a large software system. Each system is divided *a priori* into a number of *segments*, where a segment is both the unit of

```
atomic FireWallUp(r, u)
let
 T = a fully attributed semantic tree, shared
 M = a directed graph, shared
 r = a node of T to be replaced by an edit command
 u,v = editing cursors
 b = an attribute instance
in
  if ∃ a cursor v, v ≠ u,
       that points to any node in T_r,
       the parent of r, or a sibling of r
  then
   return false
  else if any node in T_r, the parent of r,
        or a sibling of r is in M
  then
   return false
  else
   ∀ attributes b of r and its parents and siblings
    do
         color b red in D(T)
    od
   return true
  fi
end
```

Figure 9: FireWallUp for Multiple Cursors

if all the edits themselves are combined prior to any propagation to make the minimal change to the semantic tree. Reps, Marceau and Teitelbaum [47] have investigated incremental evaluation algorithms for such *synchronous* subtree replacements.

For asynchronous subtree replacements, the lower bound is $\bigcirc(|\ \mathit{Influenced}\ |)$, where *Influenced* is the minimal set of attributes that must be evaluated given the timings among edits and propagations. Timing is an issue since an attribute may become inconsistent due to one edit, and its evaluation completed *prior* to the real-time occurrence of a second edit. This attribute may become inconsistent again due to the second edit, and have to be evaluated a second time, but it is not possible to avoid the extra evaluation. Thus *Influenced* is in general larger than *SimultaneousAffected*. It is an open question whether $\bigcirc(|\ \mathit{Influenced}\ |)$ can be achieved for asynchronous edits when both bookkeeping costs and attribute evaluations are considered, but our algorithm is optimal in the number of attribute evaluations.

3.4 Asynchronous Parallel Evaluation with Multiple Cursors

The extension from single to multiple editing cursors making multiple asynchronous edits on a shared semantic tree is relatively simple: multiple firewalls. Each editing cursor has a corresponding firewall, which covers the subtree rooted at the current node, its parent and its siblings. The editing model is as follows. Each cursor operates independently, bringing up its firewall to make a subtree replacement, and then bringing it down and merging its new model into the shared existing model. The cursor can make another subtree replacement.

For the same reasons discussed above, incorrectness can result if a subtree replacement is made in a region of the tree covered by the model, and/or if the model expands into a region of the tree where a subtree replacement is in progress. This continues to be handled by the $FireWallUp$ procedure. In addition, there is now a potential problem of overlapping firewalls. Thus cursor movement operations (not shown) are modified to disallow movement to a node under a firewall, and $FireWallUp$ is extended as shown in figure 9 to disallow bringing up a firewall if it would cover another cursor.[5]

The $Propagate$ and $Expand$ procedures are the same as for the single cursor case (figures 5 and 6, respectively). $FireWallDown$ is also the same as for the single cursor case (figure 8), if we assume immediate merging of models rather than the possibility of multiple models as mentioned in Section 3.3. If there may be multiple models, then it would be necessary to keep track of any collisions with the firewall while it is up, and merge the new model together with the colliding models when it comes down.

[5] Although straightforward extensions to the algorithms described here would permit certain cases of overlapping firewalls, we require for the purposes of this article that firewalls be disjoint – in general, it does not make sense to be editing inside a subtree that is about to be replaced or is about to serve as a replacement.

covered by M, it is necessary to either fill in the region between M and N, or to insert summary transitive edges between the two subgraphs (and subsequently allow expansions inside the model as well as at the apex and frontier). This bookkeeping effort may be significant, although doing so guarantees that the number of attribute evaluations is minimized. It may be the case that overall performance can be improved by accepting non-optimality in the number of attribute evaluations in order to reduce bookkeeping costs.

There is a spectrum of alternative approaches to this problem, which have been delineated by Micallef and Kaiser [38]. They characterize this spectrum by recognizing the models M and N as disjoint, and merge their propagations only if one model "collides" with the other. They give three basic definitions of collision: (1) one model expands to contain a node currently in the other model; (2) one model expands to contain a node that at some point had been in the other model; and (3) a new model is initialized. Note that in cases (1) and (2), there may be more than one existing model M_i in addition to the new model N, reflecting multiple previous edits whose propagations have not yet collided. In each of the three cases, merging between two models must consider the current dependency graph and restore any dependencies no longer or never previously reflected in the now combined model; the dependencies can change because the characteristic graphs used in the original construction of a model may have been invalidated by subsequent subtree replacements.

The tradeoff between bookkeeping and evaluation costs is different in each case, ranging from case (1), minimal bookkeeping but the possibility of k evaluations of the same attribute for k edits to case (3), elaborate bookkeeping but minimization of evaluations. Our first published algorithm implicitly assumed (1), while here we assume (3). Micallef and Kaiser focus on case (2), and present algorithms for updating characteristic graphs, collision detection and merging that balance bookkeeping and evaluation costs. They handle both a single editing cursor and multiple cursors, which we address in the next section.

We do not address this question further here, except to note that probably the most dramatic example where cases (1) and (2) allow extraneous re-evaluations is when there is repeated delete and undo (and redo and undo, *etc.*) of a large subtree, particularly a subtree containing many identifier declarations when incremental evaluation is being applied to static semantic analysis of a program. Of course, even in this scenario it is possible that each propagation will complete before the subsequent edit command, so k evaluations for k edits is the best that can be achieved even if case (3) is employed.

In other words, if the cursor makes k edits to the program, the best we can say is that the number of attributes evaluated is no worse than $\bigcirc(|\ \mathit{SerialAffected}\ |)$, *i.e.*, where *SerialAffected* is determined by serializing the k asynchronous edits, and summing the total attributes affected by each separate propagation process. Using k repetitions of the sequential algorithm in Section 2, it is possible to achieve $\bigcirc(|\ \mathit{SerialAffected}\ |)$ time complexity, since bookkeeping costs are proportional to the number of attribute evaluations. Of course, what we would like to achieve is $\bigcirc(|\ \mathit{SimultaneousAffected}\ |)$, where *SimultaneousAffected* is the set of attributes affected

```
FireWallDown(r)
 let
  T = a fully attributed semantic tree, shared
  NeedToBeEvaluated = a set of attribute instances, shared
  M = a directed graph, shared
  r = a node of T containing any inconsistent attributes
  N = a directed graph
  S = a set of attribute instances
  b = an attribute instance
 in
  ∀ attributes b of r and its parents and siblings  do
        color b green in D(T)
   od
  N ← r.C ∪ r.C̄
  M ← M ∪ N
  insert all attributes in N into NeedToBeEvaluated
  S ← the set of vertices in M with indegree 0
  ∀ vertex b ∈ S  do
     if an evaluation thread for b is suspended  then
          resume evaluation thread for b
     else
          send Propagate message to b
     fi
   od
 end
```

Figure 8: FireWallDown for Single Cursor

```
atomic FireWallUp(r)
let
 T = a fully attributed semantic tree, shared
 M = a directed graph, shared
 r = a node of T to be replaced by an edit command
 b = an attribute instance
in
  if any node in T_r, the parent of r,
       or a sibling of r is in M
  then
   return false
  else
   ∀ attributes b of r and its parents and siblings  do
         color b red in D(T)
    od
   return true
  fi
end
```

Figure 7: FireWallUp for a Single Cursor

the new model should be propagated; some attributes may have been independent in the original model but are no longer independent in the united model after the subtree replacement and consequent updates to the dependency graph, and these should not be propagated until they become independent again. Thus it is not necessarily the case that all threads suspended when the firewall is up will be resumed as soon as the firewall is taken down. Again, we assume that the set of independent attributes are collected during the union operation rather than requiring a search of the united model.

3.3 Merging Models

There is some subtlety in one line of the $FireWallDown$ procedure,

$\mathtt{M} \leftarrow \mathtt{M} \cup \mathtt{N}$

Depending on the definition of $\cup$, there may be significant differences in the complexity of the overall propagation algorithm.

We have presented the algorithm as *merging* the new model N into the existing model M, such that at any time there is only one model being used as a scheduling graph. The classical algorithm assumes that M models a connected region of the tree, and includes edges representing direct dependencies within this region and transitive dependencies at the apex (root) and frontier (leaves) of the region. But since N may be initiated arbitrarily far away from the region of the semantic tree

```
atomic Expand(b)
let
 T = a fully attributed semantic tree, shared
 M = a directed graph, shared
 b, c, d = attribute instances
in
  if ∃ c, a successor of b in D(T)
       ∧ edge (b, c) is not in M
       ∧ TreeNode(c) is a child of TreeNode(b)
  then
   M ← (M - TreeNode(b).C) ∪ ExpandedSubordinate(TreeNode(b))
   ∀ vertex d of ExpandedSubordinate(TreeNode(b))
          whose indegree in M is 0
    do
      if an evaluation thread for d is suspended  then
           resume evaluation thread for d
      else
           send Propagate message to d
      fi
    od
  fi
  if ∃ c, a successor of b in D(T)
       ∧ edge (b, c) is not in M
       ∧ TreeNode(c) is the parent or a sibling of TreeNode(b)
  then
   M ← (M - TreeNode(b).C̄)
               ∪ ExpandedSuperior(TreeNode(b))
   ∀ vertex d of ExpandedSuperior(TreeNode(b))
          whose indegree in M is 0
    do
      if an evaluation thread for d is suspended  then
           resume evaluation thread for d
      else
           send Propagate message to d
      fi
    od
  fi
end
```

Figure 6: Expand for Asynchronous Parallel Evaluation

```
Propagate(b)
 let
  T = a fully attributed semantic tree, shared
  NeedToBeEvaluated = a set of attribute instances, shared
  M = a directed graph, shared
  b, c = attribute instances
  changed = Boolean
  OldValue, NewValue = attribute values
 in
   if b in D(T) is colored red  then
        suspend evaluation thread for b
   else
    changed ← false
    if b ∈ NeedToBeEvaluated  then
     remove b from NeedToBeEvaluated
     OldValue ← value of b
     evaluate b
     NewValue ← value of b
     if OldValue ≠ NewValue  then
      changed ← true
       if M does not contain all the successors of b in D(T)
       then
            Expand(b)
       fi
     fi
    fi
    ∀ successor c of b in M  do
     remove edge (b, c) from M
     if changed  then
      insert c into NeedToBeEvaluated
     fi
     if indegree of c in M is 0  then
       if an evaluation thread for c is suspended  then
            resume evaluation thread for c
       else
            send Propagate message to c
       fi
     fi
    od
   fi
 end
```

Figure 5: Asynchronous Parallel Attribute Propagation

than actually doing the attribute evaluation and possible subsequent expansion, the evaluation thread is temporarily suspended. Alternatively, we could restrict the red coloring to the attributes at the root of the replacement subtree and then prevent expansion into a red region of the tree as opposed to propagation; the distinction is whether the check to see if an attribute is red is done before or after it has been evaluated.

It is necessary to prevent an evaluation thread from accessing the attributes at the root of a subtree while the subtree is being replaced, because there may be an arbitrary mix of pointers to the old subtree root, pointers to the new subtree root, and pointers into deallocated storage. But it is not necessary for correctness to prevent another thread from accessing the attributes at the parent and siblings. However, although the root of the replacement subtree must have the same set of synthesized attributes as the replaced subtree (assuming these attributes are actually used in the parent production), the dependencies among them and between these attributes and those of the parent and siblings may be different. So we prefer to delay propagation rather than wait until expansion in order to avoid the situation that an attribute of the parent or a sibling is evaluated while the firewall is up and then must be re-evaluated after it comes down. The firewall is up only for the time that it takes to do the subtree replacement, so such delays are minimal.

The *Propagate* and *Expand* procedures are modified as illustrated in figures 5 and 6. The semantic tree T and its dependency graph $D(T)$ are now shared among the threads initiated by the same or different edits, while for the synchronous algorithm they were shared only among those threads initiated by a single edit. The notion of coloring the dependency graph $D(T)$ is introduced, and the propagate procedure must check whether an attribute is at a firewall boundary before evaluating it. If at the boundary, the attribute's evaluation thread is suspended until some later point, as elaborated below; whenever an attribute becomes independent, it is necessary to check whether it already has a suspended thread that has not previously been resumed. A separate *Startup* procedure is no longer necessary, since its work is now done by the $FireWallDown$ procedure.

We introduce two new procedures, $FireWallUp$ and $FireWallDown$, given in figures 7 and 8, respectively. These take care of the interaction between the firewall and the evaluation threads. All vertices in the dependency graph $D(T)$ are colored, red for vertices under an up firewall, and green for those that are not covered. Thus one purpose of $FireWallUp$ and $FireWallDown$ is to maintain the coloring of the vertices correctly. The red coloring must be atomic, so $FireWallUp$ is a critical section. As with *Expand*, the critical section could be replaced by making the dependency graph a concurrent data structure with appropriate operations. The reason we color vertices of the dependency graph rather than the actual attributes of the tree is to avoid introducing another level of synchronization on the tree.

Two further tasks for $FireWallDown$ are to restart any attribute propagations that may have been suspended by *Propagate* due to collision with the firewall when it was up, and to take over the job of *Startup* to initiate the propagation for the new edit. These two roles are combined, since only attributes that remain independent in

the particulars of a given grammar and the actual trees constructed by the user.

3.2 Asynchronous Parallel Evaluation with a Single Cursor

In the case of a relatively small tree, representing for example a single module or a small program, the above synchronous algorithm is adequate. But in the case where there is a large tree consisting of many modules (*i.e.*, all the modules are combined in a single centralized tree), the wait time for full propagation may be extreme even when independent attributes are evaluated in parallel. To avoid performance degradation, we decouple editing and attribute evaluation to obtain an asynchronous edit/evaluation model, as follows: The editing cursor points to a particular node, a subtree replacement is made, and attribute propagation is initiated. The editing cursor is permitted to go ahead and make another subtree replacement at the same or a different node while the first propagation is still in progress. After a second subtree replacement, another attribute propagation is initiated and merged together with the first, if it is still in progress. The cursor is free to make a third subtree replacement, and so on. This new model of editing, and its supporting algorithm, is the main result presented in this section.

One challenge is possible interference between editing and propagation, when the cursor attempts to make a subtree replacement in a region of the tree covered by the model, and/or the model expands into a region of the tree where a subtree replacement is in progress. It becomes necessary to synchronize access to regions of the tree, as well as to the shared data structures discussed in Section 3.1. This is done by providing a *firewall* that covers the entire subtree rooted at the node currently under the cursor, as well as the parent and siblings of that node. When an edit command is entered, the firewall is placed *up*. Then the modification to the internal representation of this portion of the semantic tree and its dependency graph can safely take place behind the firewall.

After the structure is modified, the firewall is taken *down* and evaluation of attributes at the root of the new subtree can begin. Any other attribute evaluation threads that attempt to enter the region of the tree protected by the firewall are suspended at its boundary, and subsumed into the newly initiated propagation when the firewall is lowered. Conversely, a firewall may not come up, and thus an edit is not permitted, if an evaluation thread is already operating on the region of the tree under the firewall. Note that the firewall stays up only long enough to update the records and pointers of the internal parse tree representation of the program and comes down before the new attribute propagation begins on the now syntactically consistent semantic tree.

The firewall works as follows: When a firewall is placed up, attributes at the root of the replacement subtree and its new parent and siblings become colored *red*; other attributes remain colored *green*. The $FireWallUp$ and $FireWallDown$ procedures, presented later, maintain these colorings on the corresponding vertices in the dependency graph $D(T)$, which reflects the direct dependencies among all attributes in the tree. If propagation into a red region of the tree is possible, rather

```
atomic Expand(b)
let
 T = a fully attributed semantic tree, shared
 M = a directed graph, shared
 b, c, d = attribute instances
in
  if ∃ c, a successor of b in D(T)
     ∧ edge (b, c) is not in M
     ∧ TreeNode(c) is a child of TreeNode(b)
  then
   M ← (M - TreeNode(b).C)
       ∪ ExpandedSubordinate(TreeNode(b))
   ∀ vertex d of ExpandedSubordinate(TreeNode(b))
                 whose indegree in M is 0
    do
         send Propagate message to d
    od
  fi
  if ∃ c, a successor of b in D(T)
     ∧ edge (b, c) is not in M
     ∧ TreeNode(c) is the parent or a sibling of TreeNode(b)
  then
    M ← (M - TreeNode(b).C̄)
        ∪ ExpandedSuperior(TreeNode(b))
   ∀ vertex d of ExpandedSuperior(TreeNode(b))
                 whose indegree in M is 0
    do
         send Propagate message to d
    od
  fi
end
```

Figure 4: Expand for Synchronous Parallel Evaluation

```
Startup(r)
 let
  T = a fully attributed semantic tree, shared
  NeedToBeEvaluated = a set of attribute instances, shared
  M = a directed graph, shared
  r = a node of T containing any inconsistent attributes
  b = an attribute instance
  S = a set of attribute instances
 in
  M ← r.C ∪ r.C̄
  NeedToBeEvaluated ← the set of vertices in M
  S ← the set of vertices in M with indegree 0
  ∀ vertex b ∈ S  do
        send Propagate message to b
   od
 end

Propagate(b)
 let
  T = a fully attributed semantic tree, shared
  NeedToBeEvaluated = a set of attribute instances, shared
  M = a directed graph, shared
  b, c = attribute instances
  changed = Boolean
  OldValue, NewValue = attribute values
 in
  changed ← false
   if b ∈ NeedToBeEvaluated  then
    remove b from NeedToBeEvaluated
    OldValue ← value of b
    evaluate b
    NewValue ← value of b
     if OldValue ≠ NewValue  then
      changed ← true
       if M does not contain all the successors of b in D(T)
       then
        Expand(b)
       fi
     fi
   fi
  ∀ successor c of b in M  do
    remove edge (b, c) from M
     if changed  then
      insert c into NeedToBeEvaluated
     fi
     if indegree of c in M is 0  then
           send Propagate message to c
     fi
   od
 end
```

is synchronous, that is, the editing and attribute evaluation activities are synchronized (and exclusive); there is no implication that the multiple attribute evaluation threads are synchronized other than as shown in the algorithms.

The propagation algorithm consists of three procedures, *Startup* and *Propagate* given in figure 3, and *Expand* shown in figure 4. Following a subtree replacement, *Startup* constructs the initial model M and initializes the set of attributes that need to be evaluated ($NeedToBeEvaluated$) to the set of attributes at the root of the replacement subtree r. By sending the *Propagate* message to each of the initially independent attributes, it spawns as many threads as necessary to evaluate these attributes. These evaluation threads share M and $NeedToBeEvaluated$, as well as the semantic tree T and its dependency graph $D(T)$. If an attribute changes in value, its thread expands the model using the *Expand* procedure. The thread then removes the attribute and all its edges from the model, and spawns threads to evaluate all its newly independent successor attributes. We assume that the set of independent attributes is collected during the construction or expansion of the model, so it is not necessary to search the model in a separate pass, but this is not shown in the figures. We also assume that $\forall$ statements run in parallel rather than iteratively, to increase parallelism, although this is not necessary for correctness; this is why *Startup* spawns threads for all attributes in the separate worklist S, rather than the current contents of M, because otherwise race conditions might result in multiple evaluation threads for the same attribute.

The entire *Expand* procedure is a critical section with respect to all other concurrent threads, including new threads spawned during *Expand*. This is not strictly necessary, since expansions on disjoint regions of the shared model could correctly execute in parallel; further, restricting execution of *Expand* limits the possible concurrency, which is an acceptable bottleneck only if attribute evaluation is significantly more time consuming than bookkeeping operations on the model. An improvement might be to implement the model as a concurrent data structure [14], but we do not address this. Instead, we treat both M and $NeedToBeEvaluated$ as atomic data structures, where there may be multiple reading operations or one writing operation. It is not necessary to synchronize access to attributes, however, because this is done implicitly by the evaluation algorithm: either there is one writer and no readers (during evaluation of that particular attribute) or no writers and perhaps multiple readers (after evaluation of that attribute, when other attributes that depend on it are being evaluated). It is also unnecessary to lock T or $D(T)$, since all access by the evaluation threads is read-only and the editing cursor is effectively locked out until propagation terminates.

Since the only effective change thus far to the classic editing/evaluation model is to allow independent attributes to evaluate concurrently, it is not surprising that the time complexity of this algorithm is in the worst case (no parallelism obtainable) identical to that of the classic work, *i.e.*, $\bigcirc(|\mathit{Affected}|)$. The best case behavior is $\bigcirc(|\mathit{Affected}|\,|/m)$ where m is the max-cut of an image of the model containing all edges and nodes inserted into it during an evaluation and any cut of the graph yields the max-cut. Generally, solutions are between these two extremes and depend on

extra attributes, the algorithm maintains a special set called $NeedToBeEvaluated$, which includes only those attributes dependent on a changed attribute; since this data structure is a set, the insertion operation prevents duplicate elements. The worklist, S, is the set of attributes that have become independent but have not yet been evaluated.

Note that the procedures shown in the figures are not precise, in that no distinction is made between attribute instances in the semantic tree T, their corresponding vertices in the dependency graph $D(T)$ and their corresponding vertices in the model M. In particular, b and c are attributes in T, but are also treated as vertices in $D(T)$ and M. However, it should always be clear whether a reference is to an attribute instance or to its associated vertex in some graph. We continue this imprecision in our own algorithms, presented in the rest of this article, since the alternative would be cluttered procedures with little or no additional insight. It is also possible to implement the dependency graph and the model such that they are interlaced with the semantic tree, in which case the same record structures fulfill all three roles.

3 Centralized Parallel Attribute Evaluation

In this section, we focus on parallel evaluation on a centralized semantic tree in response to single or multiple edits. We first describe the case where editing and attribute propagation proceed synchronously: the single user enters a single edit command and then "waits" for attribute propagation to restore consistency among the attributes throughout the tree.[4] We then describe how we disconnect editing and attribute propagation, to allow asynchronous editing with a single cursor while evaluation is in progress, with special protection in cases where editing and attribute propagation attempt to affect the same subtree. Finally, we consider asynchronous edits by multiple editing cursors. This is a prerequisite to our algorithm for parallel evaluation on a decentralized tree, presented in Section 4, which requires the equivalent of a second cursor in each segment to represent attribute propagations arriving from other segments.

3.1 Synchronous Parallel Evaluation

For our synchronous parallel evaluation algorithm, the model of editing is the same as for sequential evaluation as described in Section 2. The editing cursor points to a particular node, a subtree replacement is made, attribute propagation is initiated and completed, and then it is possible to make another subtree replacement. The important innovation presented here is parallelization of attribute propagation. Multiple threads of control are spawned to evaluate independent attributes in parallel. The threads communicate via shared data structures, particularly the model and the set of attributes that need to be evaluated. Note that it is the editing model that

[4] Cursor movement and other non-editing commands could be permitted while the user "waits", but we do not discuss this.

```
Expand(M, b, S)
 let
  M = a directed graph
  b, c = attribute instances
  S = a set of attribute instances
 in
   if ∃ c, a successor of b in D(T)
      ∧ edge (b, c) is not in M
      ∧ TreeNode(c) is a child of TreeNode(b)
   then
    M ← (M - TreeNode(b).C)
        ∪ ExpandedSubordinate(TreeNode(b))
    insert into S all vertices of
        ExpandedSubordinate(TreeNode(b))
        whose indegree in M is 0
   fi
   if ∃ c, a successor of b in D(T)
      ∧ edge (b, c) is not in M
      ∧ TreeNode(c) is the parent or a sibling of TreeNode(b)
   then
    M ← (M - TreeNode(b).C̄)
        ∪ ExpandedSuperior(TreeNode(b))
    insert into S all vertices of
        ExpandedSuperior(TreeNode(b))
        whose indegree in M is 0
   fi
 end
```

Figure 2: Classical Algorithm for Expansion of the Model

```
Propagate(T, r)
 let
  T = a fully attributed semantic tree
  r = a node of T containing any inconsistent attributes
  S, NeedToBeEvaluated = sets of attribute instances
  M = a directed graph
  b, c = attribute instances
  changed = Boolean
  OldValue, NewValue = attribute values
 in
  M ← r.C ∪ r.C̄
  S ← the set of vertices in M with indegree 0
  NeedToBeEvaluated ← the set of vertices in M
   while S ≠ ∅ do
    select and remove a vertex b from S
    changed ← false
     if b ∈ NeedToBeEvaluated then
      remove b from NeedToBeEvaluated
      OldValue ← value of b
      evaluate b
      NewValue ← value of b
       if OldValue ≠ NewValue then
        changed ← true
         if M does not contain all the successors of b in D(T)
         then
          Expand(M, b, S)
         fi
       fi
     fi
     while ∃ c, a successor of b in M do
      remove edge (b, c) from M
       if indegree of c in M is 0 then
          S ← S + c
       fi
       if changed then
          insert c into NeedToBeEvaluated
       fi
     od
   od
 end
```

Figure 1: Classical Algorithm for Propagation

r everywhere in the subtree rooted at r. The latter is called the *superior characteristic graph* and denotes (transitive) dependencies among the attributes of r and the remainder of the tree not including r's subtree. The characteristic graphs are constructed on the fly during cursor motion, using dependency graphs $D(p)$ for each production p, which can be constructed in advance by a simple analysis of the attribute grammar.

M's vertices represent attributes and its edges represent the dependencies among attributes. Because transitive dependencies are included, the model can reflect information arbitrarily far away in the tree. Any vertex in this graph with indegree zero (0) is said to be *independent* because it cannot depend on any other inconsistent attribute. Because the attribute dependencies are restricted to form a DAG, at least one such vertex must always exist.

M is used as a scheduling graph for attribute evaluation. An independent vertex is chosen and the associated attribute b is evaluated. If the value of the attribute has changed, M and the dependency graph $D(T)$ are examined to see if there are any attributes c that directly depend on b but are not already represented by a vertex in M. If so, M is *expanded* to include vertices for each such c, edges for all dependencies between the new vertices, and edges for all dependencies between each c and any attributes already in M, again including any transitive dependencies. The attribute b is removed from M, along with its associated edges, and the process is repeated until M is empty. The algorithm is guaranteed to terminate if the attribute grammar is noncircular. This evaluation process, to restore consistency among the attributes throughout the semantic tree, is known as *attribute propagation*. After propagation has completed, another subtree replacement can be made, initiating a new propagation.

This algorithm was first described by Reps [45] and is asymptotically optimal in time: its time complexity is $\bigcirc(|\,Affected\,|)$, where *Affected* is the set of attributes that change in value. The implied topological sort on M results in an attribute being evaluated only when the evaluation is guaranteed to yield its final value, because an attribute is only evaluated after it becomes independent. The algorithm is implemented in the Synthesizer Generator.

This approach is shown in figures 1 and 2. The algorithm is broken into two procedures: *Propagate*, which controls the evaluation of attributes, and *Expand*, which expands the model M after an attribute changes in value.[3] They use an extended version of the characteristic graphs, produced by procedures called $ExpandedSubordinate$ and $ExpandedSuperior$, not shown. These take as argument a node in the semantic tree, and return a dependency graph. $ExpandedSuperior$ returns the union of the dependency graphs of the node, its parent and its siblings; $ExpandedSubordinate$ gives the union of the dependency graphs of the node and its children. Because these graphs are constructed from subgraphs generated in advance from the attribute grammar, they include all the attributes of these nodes, not just those actually dependent on the changed attribute. In order to avoid unnecessary evaluations of the

[3] These procedures are taken (with minor notational amendments) from Reps's dissertation [46].

A syntactically correct attributed parse tree is called a *semantic tree*.[1] A semantic tree for which all attributes have correct values is called *consistent*. An *inconsistent attribute* is an attribute whose value is not correct, that is, it is not the same as would be obtained by evaluating its semantic equation; a semantic tree with one or more inconsistent attributes is also inconsistent. Attribute dependencies, as implied by the semantic equations, form a directed graph on the semantic tree. A common restriction is to consider only *noncircular* attribute grammars, *i.e.*, those whose attribute dependencies will form a directed acyclic graph (DAG) on all trees that can be generated using the grammar [31].[2] We denote this graph $D(T)$ for a tree T; $D(p)$ is the dependency graph for a single production p.

An attribute grammar specification can be used to generate an LBE that incrementally checks the static semantic correctness of a module while guaranteeing syntactic consistency. Edits in such a system are defined in terms of *subtree replacements* on the semantic tree representation of a module, using templates corresponding to the programming language constructs. After each replacement, the attributes are incrementally re-evaluated to restore consistency of the semantic tree.

Before a subtree replacement, the editing cursor points to a particular node in the consistent semantic tree. After replacing the subtree rooted at that node with a new subtree, the tree is syntactically consistent and all inconsistent attributes are located at the root r of the replacement subtree (we assume that the replacement subtree is also fully attributed). To re-establish consistency among all the attributes in the semantic tree, these inconsistent attributes must be evaluated, and any other attributes that depend on attributes whose values have changed must also be evaluated.

To minimize evaluation costs, the inconsistent attributes must be evaluated in some order that avoids ever having to evaluate any attribute more than once. There are several difficulties with determining this order. Evaluating the attributes of r may cause other attributes to become inconsistent, in particular, the value of an attribute of r's parent, sibling or child may no longer be the same as the result of applying its semantic equation, since one or more of the updated attributes of r may be inputs to the equation. Further, there may not be any direct dependencies between a particular pair of attributes of r, but there may be transitive dependencies: a chain of attribute dependencies from inputs to outputs of semantic equations can go from some attribute of r to arbitrarily far away in the semantic tree, and then return to another attribute of r.

The classical solution to this problem is to construct a graph M that represents the direct and transitive dependencies among attributes, and use this graph to order the evaluation of inconsistent attributes. The graph M, called the *model*, is constructed from the *characteristic graphs* of the root node of the replacement subtree. For a node r we denote these as $r.C$ and $r.\overline{C}$. The former is called the *subordinate characteristic graph* and denotes (transitive) dependencies among the attributes of

[1] In general, we use the terminology defined in Reps's dissertation [46].

[2] However, some researchers have described incremental evaluators for circular attribute grammars using fixed points [20, 5], while others have addressed interpretation [57].

algorithms for parallel evaluation on a centralized tree. Section 4 gives our main result, by extending the algorithms of the previous section to support parallel evaluation on a decentralized tree, but segmented among user processes on the same machine. Section 5 explains how we support attribute propagation among segments on a network, first assuming a reliable network and then presenting our protocol for an unreliable network. We then discuss related work and summarize our contributions. The appendix includes simple examples of both a standard attribute grammar and an extended attribute grammar for a segmented program.

2 Background

The *attribute grammar* formalism was invented by Knuth [30]. An attribute grammar consists of a context-free grammar that is enhanced to be capable of expressing context-sensitive information by (1) associating with every symbol in the context-free grammar a set of *attributes*, or properties, and (2) associating with every production in the grammar a set of *semantic equations*, each of which defines the value of an attribute of one symbol in the production as a function of other attributes of the same and other symbols in the production and/or the values of terminal symbols in the production. When a semantic equation defines the value of an attribute of the goal symbol of the production, the attribute is said to be *synthesized*; when an equation defines the value of an attribute of another symbol on the right hand side of the production, the attribute is *inherited*. Computing the value of an attribute by applying the function to the current values of its arguments is called *evaluation* of the attribute.

Attribute grammars can be applied to detection of static semantic errors and code generation, among other problems in software development environments; they can also support maintenance of user interfaces and other uses of "triggers" in database management systems. A simple example of an attribute grammar is given in the appendix. Waite and Goos [56] and Demers, Reps and Teitelbaum [3] discuss the use of attribute grammars in the construction of compilers and language-based editors, respectively. Hudson and King [19] describe some applications to database management systems.

In the software development environment domain, 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 parse tree is an instantiation of a symbol of the grammar; alternatively, a template-based syntax-directed editor can be used to force edits to always maintain a correct parse tree. The attributes associated with each production can be evaluated after parsing is completed, or can be maintained incrementally during syntax-directed editing, in which case the editor is called a *language-based editor* (LBE). In this article, we assume the latter model, but note that a text editing front end could be provided using incremental parsing techniques (see, *e.g.*, [11, 58, 21]).

This article extends our previous work in fundamental ways. Our previously published algorithm for parallel evaluation on a centralized tree assumed shared memory, one editing cursor, and multiple synchronous threads of control. The threads were synchronous in the sense that first the user edited, then the parallelized incremental algorithm was executed to completion, and then control returned to the user. We now assume only shared memory, and there may be single or multiple editing cursors, and multiple asynchronous threads of control, where new edits and thus new evaluation threads may start while previous evaluation threads are still in progress. Thus our algorithms for parallel evaluation on a centralized tree in response to multiple asynchronous edits with one editing cursor and parallel evaluation on a centralized tree in response to multiple asynchronous edits with multiple editing cursors are new.

Our previously published decentralized algorithm similarly combined the units of shared memory, editing, and multiple synchronous threads of control into the single notion of segment. The threads were synchronous in the sense that a user edit had to wait for the previous evaluation to complete and propagations arriving at a segment from another segment had to wait for a user edit to complete. We now equate a segment only with shared memory, and allow fully asynchronous threads of control and multiple editing cursors within a segment. Thus we can apply our new algorithms for parallel evaluation for a centralized tree to each individual segment of a decentralized tree, resulting in new decentralized algorithms.

The Mercury project at Columbia University has been in progress for several years and has involved many people, most of them contributing primarily to the implementation of Mercury by modifying the Synthesizer Generator. However, there have been a number of significant results by other members of the Mercury group, not presented here, including support algorithms for correctly maintaining certain internal data structures when there are multiple editing cursors, for detecting collisions between propagations within the same segment, and for merging these propagations in order to avoid unnecessary re-evaluations of the same attributes for each of the evaluations [38]; algorithms for efficient propagation of aggregate attributes, such as symbol tables, over a distributed tree and for multiple versions and configurations of modules represented as segments [37]; extensions to the segmentation scheme and extended attribute grammar formalism presented here to support the multiple kinds of compilation units and hierarchy of compilation units required for complex languages such as Ada [39]; a static rather than dynamically controlled algorithm for a restricted subclass of attribute grammars called the ordered attribute grammars [36]; a low level facility for reliable multicast of messages, originally developed for broadcast among segments [18]; and a tool for simulating interprocess communication and network events, used to "stress test" our implementation of the original decentralized evaluation algorithm and reliability protocol [59]. We reference these results as applicable in the body of the article.

The rest of the article is structured as follows. Section 2 provides background on attribute grammars and the classical sequential algorithm. A reader who is already familiar with this work may skip this section. Section 3 presents our three

tree for each module; interface checking would be done later, as a side-effect of recompilation of modified modules and their dependent modules.

2. Next we extend to parallel evaluation on a centralized tree, again representing a single module, in response to a single edit by a single user. Strictly speaking, this algorithm does not directly address any of the requirements outlined above, but is a prerequisite to the later algorithms. This algorithm might also be useful for a traditional single-user LBE, to potentially improve performance when the user's machine is a shared memory multiprocessor.

3. We go on to parallel evaluation on a centralized tree, again representing a single module, but in response to multiple asynchronous edits by a single user with a single editing cursor. Thus the user no longer needs to wait for the evaluation process initiated by one edit to complete, but can continue on immediately with new edits.

4. The next step is parallel evaluation on a centralized tree in response to multiple asynchronous edits by multiple editing cursors within a shared address space, and discuss some of the ramifications of this approach. Again, this algorithm is a necessary prerequisite to the full decentralized algorithm.

5. Now we present the full algorithm for parallel evaluation on a decentralized tree in response to multiple asynchronous edits, where different segments of the tree reside in different user processes on the same machine. Multiple edits may be made within the same or different segments, but in practice would most likely be confined to one editing cursor per segment – plus special facilities that mimic a second editing cursor to represent propagations entering the segment from other segments.

6. Finally, we provide the remaining piece, a supporting protocol to permit segments to reside in different user processes on the same or different machines connected by a network. The support layer guarantees reliability of the incremental evaluation algorithm as machines hosting segments of a decentralized tree become unavailable and return to availability. This result is described in a manner aimed to demonstrate that the protocol is applicable to improving performance, availability and reliability across a range of distributed applications, not just incremental attribute evaluation.

We have previously published algorithms for parallel evaluation on a centralized tree within the classical single-user model and for parallel evaluation on a decentralized tree, where each segment represents a simple extension of the classical single-user model allowing propagations from one segment to another by simulating subtree replacements at the root of each receiving segment [26]. We have also published an earlier form of our reliability protocol suitable for the latter [22]. The second and third algorithms are implemented in the Mercury system [23] at Columbia University, which generates distributed multi-user LBEs from attribute grammars.

The seminal result in incremental attribute evaluation is due to Reps, Teitelbaum and Demers [49], who developed an asymptotically optimal algorithm for the single-user case described above, and the first major application was to the Synthesizer Generator [48] developed at Cornell University. Using an LBE generated by the Synthesizer Generator, a user alternately edits and "waits" while attribute evaluation is performed and any errors displayed. The "wait" is typically insignificant in the case of individual modules and small programs, since there is only one possible source of attribute changes – an edit by the user to his module. In general, the user's "think time" is much larger than the time required for attribute propagation.

Scaling up to multiple programmers and large software systems, however, makes this wait unacceptable. If one user edits the now much larger parse tree representing the entire system and all users wait during the attribute evaluation, which propagates changes throughout the entire system, and then another user gets a chance to edit and all of them wait again, the ability of members of the group to work independently as far as possible will be severely impaired. no one will get much work done. Thus one requirement is to decouple editing from attribute evaluation, so every user can edit while propagations initiated by his own and/or other user's edits are in progress. Another way to phrase this is that there should be separate threads of control for each editing cursor and for each attribute propagation initiated by an edit.

If the multiple programmers share the same machine, they presumably employ a multi-tasking operating system that supplies each user with a separate process, with its own context, thread(s) of control, and protected virtual memory. Thus another requirement imposed by multiple users is a decentralized parse tree, because there is no memory shared among the users. The parse tree must be *segmented* according to some criteria, one segment to a user process. For pragmatic reasons, we choose to define segments according to the modular units of the programming language, and extend the attribute grammar formalism to designate such units. If the multiple programmers work on more than one machine, then this distribution adds the possibility that some segments will be available while others are not. A third requirement is support for reliability, so that users are not affected by hardware and software failures of other users' workstations, or by failures in the network linking the workstations.

We have adapted the incremental attribute evaluation paradigm to fulfill these three requirements, in order to solve the problem of change propagation among multiple users, whether working on the same machine or on their own workstations connected by a network. In this article, we present our algorithms to implement such change propagation. We define these algorithms in a step by step progression, to aid the intuition of the reader.

1. We start by sketching the classical sequential algorithm due to Reps, Teitelbaum and Demers [49] for incremental attribute evaluation on a centralized tree in response to a single edit by a single user, to provide the necessary background and terminology for presentation of our results. The centralized tree represents a single module or a small program. Multiple modules may be edited simultaneously by multiple users, but in isolation with a separate

1 Introduction

We present a family of efficient algorithms for incremental reestablishment of attribute consistency in response to subtree replacements on an attributed tree. The distinguishing features of these algorithms are the ability to perform parallel attribute evaluation on centralized or decentralized trees in the face of single or multiple asynchronous subtree replacements by single or multiple editing cursors. These algorithms thus extend previous work on incremental attribute evaluation for single users [49, 16, 47] in two complimentary directions: We present algorithms for parallel evaluation that will be useful when programmers have ready access to multiprocessor workstations, and we present algorithms that support the cooperative effort of groups of programmers working on related parts of a project using a distributed software development environment. All the algorithms apply to the class of general noncircular attribute grammars [30, 31]. In order to provide realistic solutions to the distributed incremental evaluation problem, we also present a protocol for guaranteeing reliability of the decentralized algorithms in the face of unreliable networks.

The incremental evaluation algorithms are applicable to a range of *change propagation* problems in software development environments and database management systems, where derived or otherwise dependent data must be efficiently recomputed in response to changes in the base data manipulated by programmers or database end-users, respectively. The context-free grammar underlying the attribute grammar together with the attributes of symbols in the grammar specify a hierarchical data model, while the semantic equations define the dependencies among data items and the recomputations required when data items are updated. The reliability protocol solves a general cache consistency problem, and thus is suitable for a range of applications in addition to attribute evaluation.

Our development of these algorithms was originally motivated by the need for change propagation in multi-user software development environments distributed across a network of workstations. A class of environments known as language-based editors (LBEs) have long supported change propagation within the limited context of a single user's own activities, to keep the user aware of how editing one part of his module or small program introduces inconsistencies with respect to other parts. One of the most widely accepted mechanisms for implementing this change propagation is to represent the module internally as an attributed parse tree and treat each edit as a subtree replacement on this tree. The static semantic properties of the programming language are expressed as an attribute grammar, and the LBE performs incremental evaluation of the attributes in response to each edit, propagating a change in one attribute to all the directly and transitively dependent attributes. This *attribute propagation* makes it possible for the LBE to immediately detect any inconsistencies among module parts introduced by the edit and notify the user of the inconsistencies by displaying the values of distinguished "error" attributes. As errors are removed through new edits, the attributes are reevaluated and the values of error attributes return to empty strings, and thus are no longer displayed.

Contents

1 Introduction **2**

2 Background **6**

3 Centralized Parallel Attribute Evaluation **11**
3.1 Synchronous Parallel Evaluation 11
3.2 Asynchronous Parallel Evaluation with a Single Cursor 15
3.3 Merging Models 19
3.4 Asynchronous Parallel Evaluation with Multiple Cursors 22
3.5 Pragmatics 24

4 Decentralized Attribute Propagation **24**
4.1 Attribute Grammar Extensions and Restrictions 25
4.2 Multiple Segment Evaluation 26
4.3 Pragmatics 32

5 Attribute Propagation Layer **33**
5.1 Attribute Propagation on a Reliable Network 33
5.2 Attribute Propagation on an Unreliable Network 36
5.3 Pragmatics 40

6 Related Work **42**

7 Conclusions **44**

A Attribute Grammar Examples **50**