A System for Efficient and Flexible One-Way Constraint Evaluation in C++

Scott E. Hudson

Graphics Visualization and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
E-mail: hudson@cc.gatech.edu

ABSTRACT

This paper describes the Eval/*vite* system for compiling one-way constraints into C++ objects, as well as the highly efficient lazy and incremental evaluation algorithm behind it. This system supports the creation of C++ object classes whose members are controlled by one-way constraint equations. These objects are declared with simple specifications much like C++ class declarations. The system is designed to work smoothly with other C++ code, producing type-safe code which supports normal inheritance and information hiding mechanisms and works with most types. In particular, the system efficiently supports constraints involving indirection either through pointers or C++ reference types. The system also works with other sophisticated constructs such as the use of function valued constraint equations.

Keywords: one-way constraint systems, object-oriented programming, incremental update, lazy evaluation.

1 INTRODUCTION

Constraints offer the ability to describe, in a high-level and declarative fashion, a set of relationships that are to hold among data items. Given a set of values and a set of relationships between them expressed as equations, a constraint maintenance or evaluation system can be employed to automatically maintain the proper relationships between these values in the face of changes. The automatic nature of this update allows very high level specifications to be employed for a number of tasks in the user interface software domain. These include, for example, controlling user interface presentations [Huds89, Huds90a, Myer93, Vand90], making connections between data objects and interactive views of those objects [Huds88, Hill92], support for construction of graphical documents [VanW82], and controlling animation [Born86].

Because of their inherent advantages, constraint systems of various sorts have been used in a number of user interface systems (see for example [Bart86, Born86, Myer90, Huds90b]). However, constraints are still considered an "exotic" technology in many circles. One reason for this is that, in the compiled languages that would typically be used for production user interface development (i.e., languages like Pascal, C, C++, Eiffel, etc.), there have been no systems that allow truly convenient and efficient integration of constraints with the rest of the user interface and application code. This paper introduces the Eval/*vite*[†] system which is designed to provide a practical solution to this problem.

The Eval/*vite* system accepts specifications for object classes whose fields can be controlled by one-way constraint equations. These specifications resemble C++ class declarations and are compiled into C++ classes and supporting code. The resulting classes can be used very easily with other C++ code. By employing a simple preprocessor and only a very slight notational change (i.e. adding an "@" symbol before the name), constrained values — what we will call *attributes* — can be accessed in essentially the same way as any other value. Further, the classes produced are type safe in the statically typed world of C++ and can enforce the same encapsulation and information hiding as other classes. Similarly, these classes can inherit from other classes (constrained or otherwise) using

This work was supported in part by the National Science Foundation under grants IRI-9015407, and DCA-9214947.

[†] "Vite" is French for fast and is pronounced like "beat".

either single or multiple inheritance and can act as base- or super-classes. Finally, and perhaps most importantly, a wide range of types can be used for attribute values. In particular, both pointer and reference typed attributes are fully supported without loss of efficiency or use of cumbersome notational constructs. Similarly, more exotic techniques such as function-pointer valued attributes are also fully supported.

In addition to ease of use with general C++ code, the system also has several other important advantages. First, it employs a simple but very efficient update algorithm. This algorithm is an enhancement of the one described in [Huds91] and is both incremental and lazy. Specifically, after a modification to the system, it only reevaluates attribute values which both could have changed value and have actually been directly or indirectly requested. Second, this algorithm can be implemented using no dynamically allocated storage and so does not incur significant overheads for storage management. Finally, the system allows constraint equations themselves to be compiled into C++ code. As a result, the system is practically efficient and can make good use of optimizing compiler technology.

All of these features combine to make the Eval/*vite* constraint compiler an attractive and practical system for implementing a range of user interface tasks.

The next section of this paper considers the tradeoffs involved in using one-way constraints and the importance of support for indirection. Section 3 will describe how constraints are specified in the Eval/*vite* system and give an example of their use in the interface to a parallel program visualization system. Section 4 will briefly consider how constraints can be compiled into C++ classes, while Section 5 will consider the underlying incremental evaluation algorithm. Section 6 will go on to show how this same algorithm can be used to support constraints with indirect references, and finally, Section 7 will provide a conclusion.

2. ONE-WAY CONSTRAINTS WITH INDIRECTION

The general problem of constraint satisfaction is unsolvable. Consequently, all constraint systems place some limitations on the classes of constraints that they can deal with. Among systems supporting user interface applications both so called "one-way" and "multi-way" constraint satisfaction systems have been used.

One-way systems (such as the one described here) place limitations on the form of equations that may be used such that information always flows only from the right hand side of a constraint rule to the left hand side — hence the designation as "one-way". In general, these systems support rules of the form:

$$A = F(B, C, D)$$

This rule states that the value of attribute A will always be equal to the value obtained by applying function F to (the updated values of) attributes B, C, and D. This form of rule allows only a single attribute on the left hand side and does not allow any given attribute to be defined by more than one equation. Typically these systems also require that there are no circular dependencies between attributes, however, this restriction is lifted in various ways in some systems. The discussions in this paper assume acyclic constraints, however the system behaves gracefully in the presence of cycles.

Multi-way constraints normally remove the asymmetry from rules and allow information flow in different directions at different times. In general they may support rules such as:

$$F(A,B) = G(C,D)$$

However, except in special cases (such as multi-linear equations) the system designer is normally responsible for providing a series of factorizations of the rule equation, each of which allows one attribute to be computed as a simple function of the remaining attributes.

In general, multi-way constraints are more powerful and can support some constructions that one-way constraints cannot. However, with this generality comes some cost.

First, multi-way constraint satisfaction systems can fail to find a solution to a system of constraints that arise at runtime — either because no such solution exists, or because the particular solution mechanism being employed cannot find one. Currently few analysis techniques exist for detecting or characterizing this possibility in advance. Consequently the system designer may need to explicitly provide for this possibility in all aspects of the system that use constraints (see

[Born87, Free90] for one approach to this). In contrast, one-way constraints are always guaranteed to find a solution (so long as the code for each individual constraint equation completes in finite time).

Second, multi-way constraint systems are much less predictable and understandable than one-way systems. For example, unlike one-way systems, multi-way systems can find themselves in under-constrained situations where more than one solution is valid. Robust approaches for specifying how the system is to respond in these cases have been developed (see [Born87, Free90]). However, it is can still be difficult to understand at design or specification time, just how the system will react in all cases at run-time. Again, this is in contrast to one-way systems which, because of their more restricted form are very easy to understand and predict.

Finally, because of the greater degrees of freedom in multi-way constraints, they are typically harder to debug[†] .

In addition to these traditional reasons for considering the use one-way constraints over the use of more general multi-way systems, there is also an additional new reason — full support for pointers and indirection. Dynamically changing pointers offer a significant challenge for constraint systems because they can cause the dependencies between values to change in the middle of the evaluation itself. Fortunately, in the case of one-way constraints efficient incremental update algorithms that work in the presence of general indirection do exist. (Two such algorithms are described in [Vand91] and a more efficient one is described here). At present no corresponding incremental algorithms have been published for multi-way constraints.

As described at length in [Vand91], the use of indirection opens up a range of new opportunities for constraint systems. These include increased opportunities for describing dynamic behaviors and improved facilities for specifying easily composable and reusable components. One of the central reasons for this significant increase in capability has to do with a hidden limitation in conventional constraint systems.

[†] In the past, multi-way constraints have also been considered much more inefficient than one-way constraint systems. However, with modern incremental algorithms the difference in efficiency between the approaches is small, and should probably not be a major factor in deciding which type of solver to use.

The equations that form constraints require that all objects they operate over be explicitly named in the equation. However, without support for indirection, this implicitly limits constraints to operating on objects whose existence is known at the time the constraint is constructed — typically at design time. This means that dynamic and unpredictable objects (for example, objects representing files in a desktop interface) are very difficult to deal with. Typically either these types of objects can't be supported, or less efficient interpreted approaches which allow dynamically construction of new constraint equations, must be employed.

In the Eval/*vite* system, the need for indirection is even more immediate. Normal programming in C++ makes heavy use of pointers. As a result, full support for pointers (and C++ reference types) is a must for any practical system designed to work smoothly with conventional C++ code.

All of these reasons combine to make support for one-way constraints with indirection a good choice for the Eval/*vite* system. The more general question of the relative merits of one-way versus multi-way constraints is still an issue for debate.

3. SPECIFYING CONSTRAINTS — AN EXAMPLE

A major goal behind the development of the Eval/*vite* system has been its convenient integration with conventional C++ code. To accomplish this, the system employs a translator which supports constraint specifications very similar to normal class specifications in C++ (an example appears later in Figure 3). In addition, the same translator also acts as a preprocessor for any additional code that needs to access attribute values. This preprocessor works using a very simple lexical extension. Specifically, any code which wishes to make use of an attribute value simply refers to the value as it normally would, but precedes the name of the attribute with an "@" symbol. The translator / preprocessor then converts this name into the proper code to transparently evaluate and return the updated value of the attribute. Since the same program performs both translation and preprocessing, constraint and normal code can be freely intermixed. All syntactic and semantic uses of normal values are

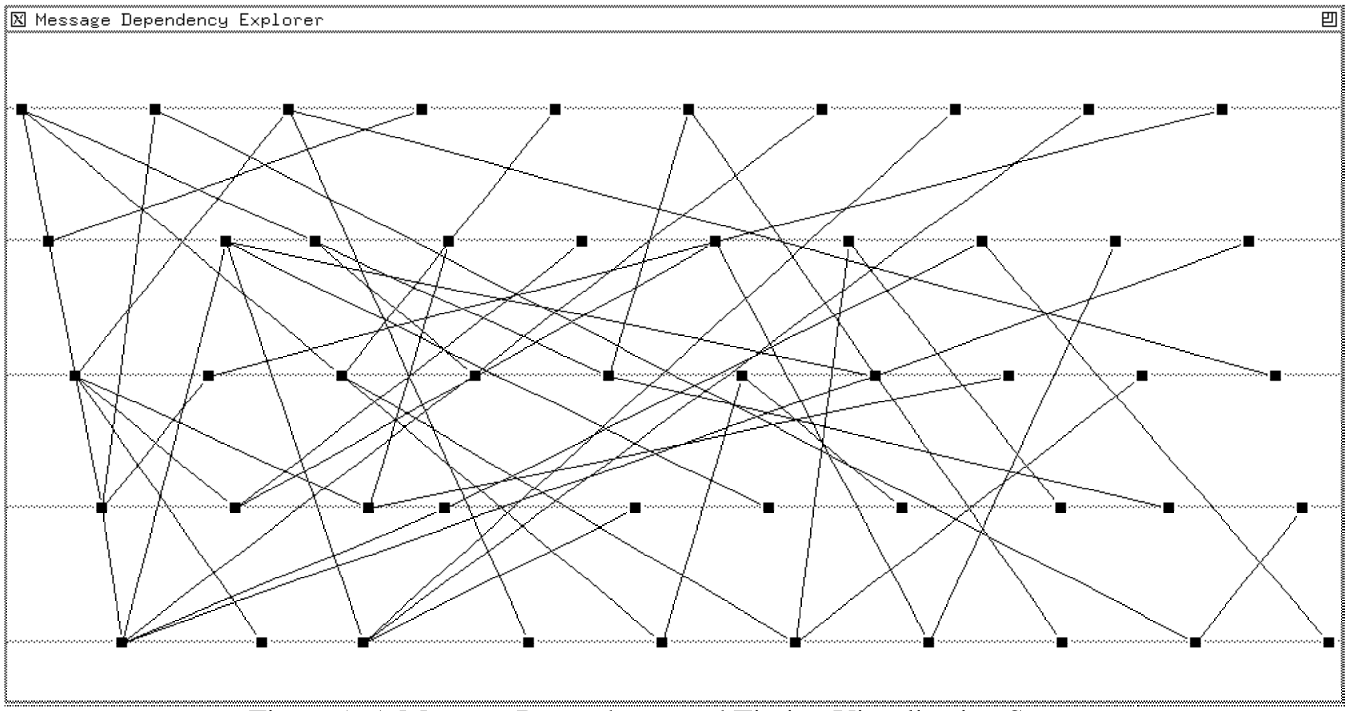

Figure 1. A Message Dependency and Timing Visualization System

also supported for attribute values. In addition, attributes support assignment in precisely the same form as any other program variable.

To illustrate how these specifications work, a small example will be used — a system for visualizing the dependencies and timing relationships in message trace data captured from the processes of a parallel program.

Figure 1 shows a display from this system. Each horizontal gray line in this visualization corresponds to a single process executing over time from left to right. Each small icon placed on the process line corresponds to an event indicating that a message has been sent or received. Lines

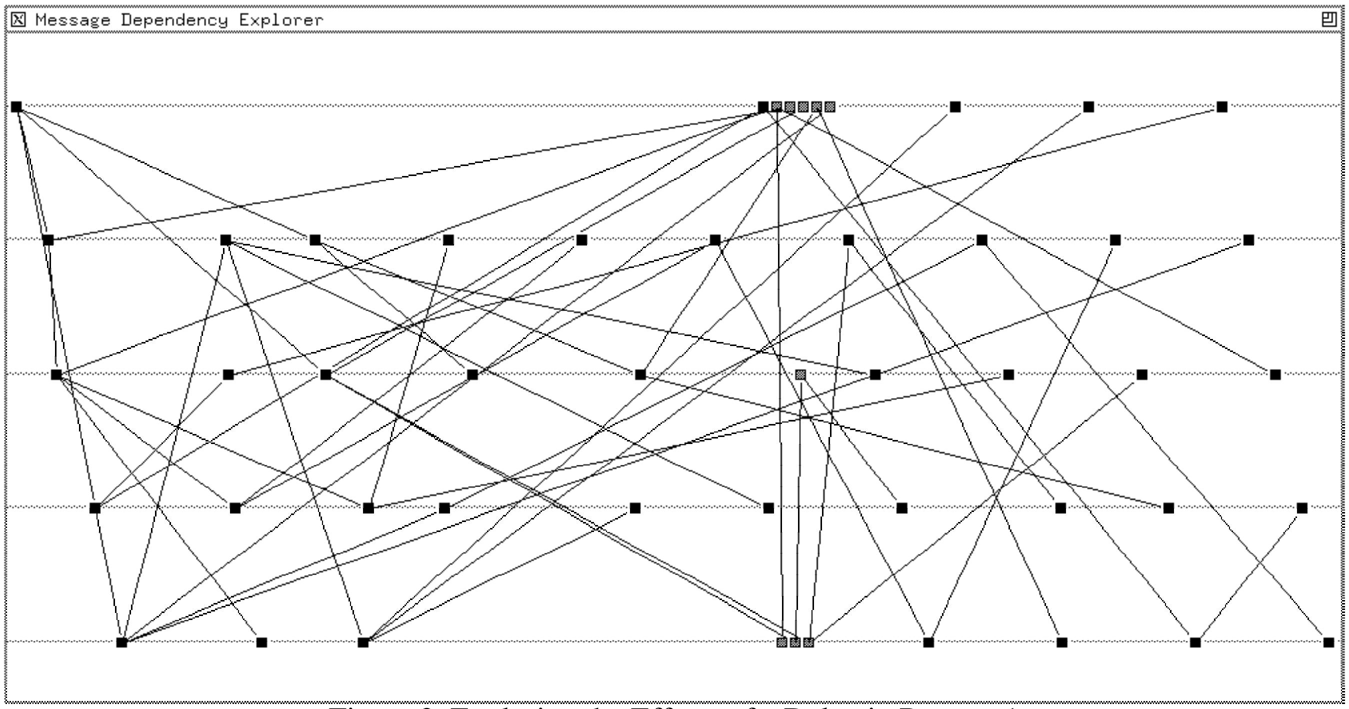

Figure 2. Exploring the Effects of a Delay in Process 1

between processes represent messages (in these figures, a random set of messages has been used for illustration purposes).

One of the major difficulties in working with parallel programs is that coincidences and small perturbations of timing can cause bugs to appear and disappear. In addition to serving as a visualization of a specific event trace, the system shown in Figures 1 and 2 also allows the specific timing in the trace to be modified in order to explore what might have happened in different executions of the same program. In particular, it allows the user to ask "what-if" questions for debugging and performance tuning purposes by modifying the timing directly in the display. However, it only allows the timing of events to be changed in ways which are consistent with the partial order inherent in the original execution (i.e., messages cannot be received before they are sent and events within a single process cannot be reordered).

Figure 2 shows an example of such a user manipulation. Here the user has grabbed the second event in the first process and forced it later in time by dragging it to the right (introducing a new delay that was not in the original execution). The system responds by repositioning event icons in the display so that they remain consistent with the partial order requirements implied by the original execution trace. In each case where the event has been forced to change its timing, the icon is changed from black to gray. In this example, we can see that this delay in the first process has relatively little effect on the other processes, only affecting the timing of four events outside the process.

A natural method for modeling trace data of this form in C++ is to use a graph structure constructed from objects linked by pointers. Each event object is linked to the process that it occurs in and the event immediately preceding it in the process. In addition, receive events are linked to the event which sent the message.

```
@class evt_display : public drag_icon {
 protected:
   proc_display *proc        = (proc_display *)0;
   evt_display    *prev_evt = (evt_display *) 0;
   evt_display   *recv_from = (evt_display *) 0;

   coord             natural_x  = default_natural_x;

   bitmap  *icon_image ::=
     if @natural_x == @x_pos
       then small_black
       else small_gray;

 public:
   constructor : drag_icon(0,0,small_black);

   coord x_pos ::= max3(
     @natural_x,
     if @prev_evt
       then @prev_evt->@x_pos + min_pspace
       else 0,
     if @recv_from
       then @recv_from->@x_pos + min_ospace
       else 0
   );

   coord y_pos ::=
     if @proc
       then @proc->@y_pos
       else floating_y_pos;

 members {
   ...
 };
};
```

Figure 3. An Example Class

Figure 3 shows such a class declared using the Eval/*vite* system. Those familiar with C++ will recognize that this declaration is very similar to a normal C++ class declaration. The fact that this is an Eval/*vite* class definition — what we will call an *attribute class* — rather than a normal class is signaled by the use of the @class keyword instead of the normal class keyword. Note that this class inherits from the drag_icon class in a conventional fashion. (The drag_icon class comes from the Artkit user interface toolkit [Henr90] used to implement this interface).

Within an attribute class the visibility operators private:, protected:, and public: apply as usual. Any data members declared in the class are treated as attributes. Member functions (methods) and conventional (*non-attribute*) data members may be declared in the separate members section shown at the bottom of the class.

This class begins by declaring three attributes for the pointers described above. Each declared attribute must either be assigned an initial value (using "=") or be given a constraint or *evaluation* rule (using "::="). In this case, they are each initialized to null. Both initializations and evaluation rules are expressed using C++

expressions. The only unsupported features are the ++, --, assignment, and new operators. In addition, because of their importance in constraints, a more convenient if-then-else syntax for conditional expressions is also supported (although the more obtuse ?: form can still be used by die hard C programmers).

Our example class next declares an attribute for the natural position of the event. This attribute indicates where the event would be if it were not constrained by other events. This will be initially set to the position corresponding to the event's timestamp in the original trace, and will be modified whenever the user drags the event to change its timing.

Finally, the actual interactive behavior of the event display object is implemented with three attributes and their associated rules. The icon_image attribute selects one of two appearances for the object (black if the natural position is in use, gray otherwise), while x_pos and y_pos determine the position of the object on the screen. The y position of the object is constrained to match the y position of the process it is associated with, and the x position is constrained to be the maximum of either the natural position or position offset from event(s) that it cannot be placed before.

Several additional features of the specification can be seen in this class declaration as well. First, a constructor for the class can be declared (using the "constructor :" syntax). This allows initializations for base classes and non-attribute members to be declared. In addition, the members section at the bottom allows member functions and non attribute data members to be declared using full C++ syntax.

Attributes are inherited just like any other member of a class. A feature not shown here is the ability to override an attribute's initialization or constraint rule in a subclass. To do this, the attribute's declaration is simply preceded by the keyword override. This allows the inheritance system to be fully utilized for attribute as well as non-attribute members.

Finally, although not apparent from our example, the Eval/*vite* system does place several limitations on the C++ declaration syntax that can be employed. The most serious of these is that types used in attribute declarations are limited to simple names, pointers to simple named types, or references to simple named types. More complex type constructions are supported by the system, but must be declared with a separate typedef statement and introduced to the system using a special @type declaration. This syntactic limitation was placed on declarations so that the system was not forced to parse, represent, and understand the full complexity of the C++ type system (a task more difficult than the rest of the translator implementation combined).

4. COMPILATION OF CONSTRAINTS

The system compiles each attribute class declaration into a corresponding C++ class declaration plus additional code to implement evaluation for the declared constraints. A special attribute object is declared for each attribute in the class. This object contains the storage for the attribute's value as well as all the bookkeeping needed by the evaluation algorithm. For each constraint rule, a special rule object is created and attached to the corresponding attribute. Finally, for each dependency edge resulting from a constraint rule, a dependency edge object is declared. These objects maintain the bookkeeping associated with dependency edges as discussed in the next section. One dependency edge occurs for each attribute value reference in the constraint equation (e.g., for each name prefaced by an "@" sign).

Almost all code for evaluation is simply inherited from a base class found in the run-time support system or instantiated from a class template. Beyond declarations, the only specific code generated is an evaluation function for each constraint. This function is responsible for evaluating enough of the equation and its parameters to determining if the value could change, then if necessary, evaluating the full rule.

For code which is not inside an attribute class declaration, the translator acts as a preprocessor, translating all attribute value references (marked with an "@") into proper code to evaluate and return the attribute's up-to-date value.

5. AN EFFICIENT UPDATE ALGORITHM

The code embedded in run-time support base classes (and class templates), as well as the code generated by the system, works together to implement a very efficient incremental and lazy update algorithm. This algorithm is incremental in that it only reevaluates attributes that actually could have changed value since the last time they have been accessed. The algorithm is lazy in that it only evaluates attributes whose values are actually requested. This algorithm is optimal in the set of attributes reevaluated after a change but is sub-optimal in the total work performed (see [Huds91] for discussion and proofs regarding the base algorithm). Note that the optimality claim made here is stronger than the one for Reps' optimal update algorithm [Reps83] because it allows lazy evaluation. In particular, this algorithm will never evaluate more attributes than Reps' algorithm and in some cases may take advantage of laziness to evaluate fewer (although it may do more total work in some cases).

```
    @class {
      int A ::= f(@B);
      int B ::= @E + @D;
      int C ::= f(@D);
      int D ::= @E / 2;
      int E = 99;
      int F = 42;
  };
```

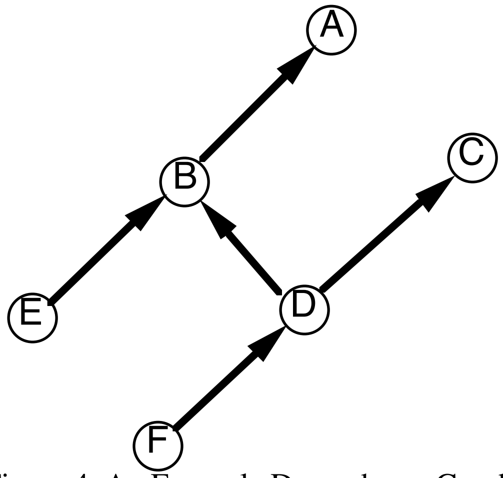

Figure 4. An Example Dependency Graph

The base algorithm (ignoring for the moment the issue of indirect references) is fairly simple. The algorithm works on the basis of a dependency graph. Each attribute forms a node in the graph and an edge is placed in the graph for each attribute dependency. An attribute A is said to *depend* on another attribute B if B's value is needed to compute A's value. In this situation we also call attribute B a *parameter attribute* of A (since it forms a parameter to the evaluation function for A). The set of potential dependencies can be established by examining the constraint rules. If no indirection is allowed, an attribute A may depend on an attribute B only if B is mentioned in its constraint rule. Figure 4 shows a system of constraints (within a single class) and the resulting dependency graph.

In addition to the dependency graph, the algorithm maintains a small amount of additional bookkeeping. With each attribute it keeps a Boolean mark indicating if the attribute's value is currently known to be up-to-date with respect to any constraint rule attached to it. With each dependency edge a Boolean mark is also kept which indicates whether a change is *pending* across that edge — that is, whether the "upstream" attribute has changed value since the "downstream" attribute has incorporated that change into its own value.

The update algorithm works in two parts or *phases*. Whenever an attribute without a constraint attached to it is assigned a new value, all attributes which directly or indirectly may depend on it are marked out-of-date with a simple graph traversal. The out-of-date mark indicates that the attribute's value may be incorrect with respect to its defining equation. Figure 5 illustrates the effects of modifying attribute F. Attributes which are marked out-of-date are shown filled with gray and dependency edges which are pending are marked with an X.

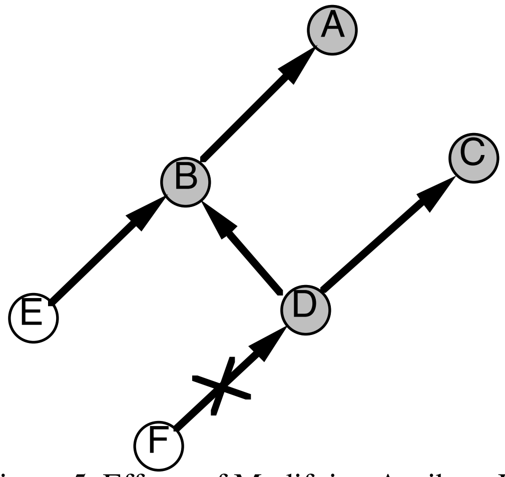

Figure 5. Effects of Modifying Attribute F

Whenever an attribute's value is requested the second (*evaluation)* phase of the algorithm is begun. This phase starts by examining the out-of-date mark of the attribute. If the mark indicates that the attribute might be out-of-date, an evaluation process is started. This process works recursively starting from the attribute whose value is requested always ensuring that an attribute's parameters are up-to-date before bringing the attribute itself up-to-date. When recursive processing reaches an already up-to-date attribute (for example, one with no constraint equation attached to it), its value is simply returned. For an out-of-date attribute, first all its parameter attributes are recursively brought up-to-date, then if necessary, the attribute's constraint equation is used to compute a new value.

Figures 5 and 6 illustrate what happens in the evaluation process when attribute D is requested. When D's value is found to be out-of-date, the system recursively evaluates all of its parameter attributes (in this case, just the attribute F). Since F is up-to-date, its value is simply returned. At this point attribute D is evaluated. Assuming its value actually changes, the system will be left as shown in Figure 6.

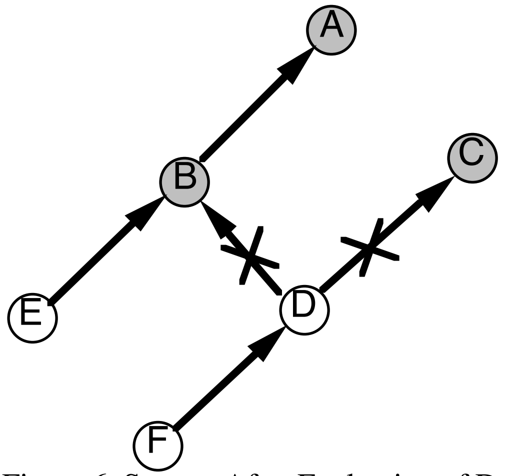

Figure 6. System After Evaluation of D

Each time an attribute computes a value that is different from its old value, it marks each of its outgoing dependency edges as pending to indicate that the change has yet to propagate across those edges. For example, in Figure 6, each of the outgoing edges from attribute D are marked pending because D's new value has not be used in computing B or C.

After recursively processing all of its parameter attributes, an attribute can determine if it must reevaluate itself by examining the pending marks on its incoming edges. If no edges are pending, then no parameter attributes have changed. This means that the attribute's value will not change and that reevaluation (and further propagation of pending marks) can be skipped.

Figure 7 illustrates how this works. Here we assume that attribute A has been requested. This will result in a recursive evaluation of B. B will further request recursive evaluation of D and E. Since these attributes are already up-to-date, no new evaluations will be performed and their values will simply be returned. However, as indicated by the mark on the edge between D and B, attribute B must still be evaluated since it might change as a result of the earlier change to D. Let us assume that after evaluating B we find that its new value is the same as its old value. In this case, pending marks would not be placed on outgoing dependency edges. This would leave the system as shown in Figure 7.

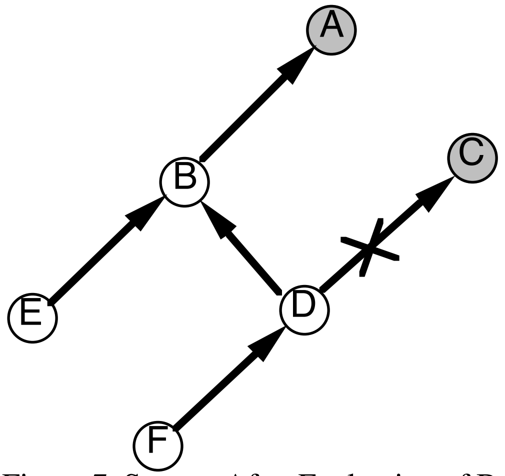

Figure 7. System After Evaluation of B

Now when attribute A completes its evaluation it can note that none of its parameters have changed value (since there are no incoming pending edges), consequently it cannot possibly change value. As a result, actual reevaluation of A can be skipped and the attribute can simply be marked up-to-date.

In addition to normal expressions, the algorithm can also make an important optimization for conditional expressions (and other forms of non-strict functions). In particular the system can act in a lazy fashion to avoid reevaluation of the branch of the condition that is not actually used during evaluation. For example, a rule such as:

```
test_result ::= if @user_wants_it
                   then @expensive_test
                   else 1;
```

can be used to dynamically turn on and off expensive features of a system. Without this ability, a system's response time will always be governed by the total set of features built into it, rather than the subset of features that are currently in use.

To support efficient evaluation, the system generates code for evaluation rules in two parts. The first part evaluates enough of the expression to determine if the value can change and the second actually computes the new value for the attribute. For normal expressions, the first part of the evaluation consists of the recursive evaluation of each parameter attribute and a test for pending edges. For a conditional expression, the first part of evaluation begins by evaluating the Boolean control expression. Based on this result, one of the two sub-expressions is chosen to complete the initial test.

6. SUPPORTING INDIRECT REFERENCES

The algorithm described so far is simple and efficient, but does not support the use of indirection in constraint equations. Indirect references present problems for evaluation systems since they can cause the dependency graph to change in the middle of the evaluation. Consider for example a rule such as:

```
A ::= F(@B)->@C
```

Here A is clearly always dependent on B. However, it may be dependent on many different C's over time, and the particular C that it is currently dependent upon cannot, in general, be determined until after F(@B) has been evaluated.

In order to support indirect references of this type we exploit a unique property of the base algorithm. In particular, under some circumstances the algorithm can produce correct and efficient results even when the dependency graph is partially incorrect. Specifically, the algorithm can tolerate incorrect incoming dependency edges for any attribute that is currently marked out-of-date.

To see why, consider the consequences of one or more incorrect edges into an out-of-date attribute. In all cases, the second (evaluate) phase of the algorithm does not present a problem, since the evaluation function need not use the dependency graph at all to compute a proper result — it can simply evaluate an expression which inherently resolves all names and addresses involved correctly.

The potential difficulty is in the first (mark out-of-date) phase of the algorithm. Here it might be possible to either mark an attribute out-of-date when it shouldn't be, or fail to mark an attribute out-of-date when it should be. These two possibilities correspond directly to the two cases of an incorrect edge that might occur: an extra edge that should not go to the attribute can be present, and/or an edge that should go to the attribute can be missing. However, if the attribute is already marked out-of-date, neither of these possible difficulties is of consequence. Marking the attribute out-of-date when it should not be will have no significance since that attribute is already marked and marking it again will have no effect. Similarly failing to mark the attribute a second time will also have no effect.

This analysis, plus the fact that any changes which could invalidate the incoming edges of the dependency graph (e.g. changing attribute B in our example) will also mark the attribute out-of-date are central to establishing the correctness and efficiency of the algorithm in the face of dynamically changing dependencies. In particular, the algorithm can be shown to be correct and efficient so long as the incoming dependency edges are always correct whenever the corresponding attribute is marked up-to-date. This can in turn be ensured by properly implementing evaluation rules such that they internally correct their local portion of the dependency graph before the attribute is fully evaluated.

With this small addition, the base incremental update algorithm performs correctly and efficiently in the presence of indirect references.

7. IMPLEMENTATION AND FUTURE WORK

The Eval/*vite* system has been implemented in C++ with the help of the YACC and Lex compiler generation tools. The translator itself consists of 3364 lines of C++ code, 948 lines of YACC specification and 311 lines of Lex specification. The classes comprising the run-time system consist of another 1004 lines of C++ code. The system is currently being beta tested at a few selected sites. In addition to the system proper, a User's Manual is also available [Huds93].

Future plans for the system include small changes such as support for parameters to attribute class constructors, as well as more substantial enhancements such as an extension to allow constraint equations to be treated as completely separate objects which can be dynamically attached to attributes at run-time. In addition, we are currently exploring the possibility of using the system in the next release of the Artkit user interface toolkit [Henry90].

8. CONCLUSION

This paper has described the Eval/*vite* one-way constraint to C++ compiler and its underlying incremental update algorithm. This system operates in a highly efficient manner, supports nearly all C++ constructs (including indirection through pointers and references) and makes it easy to integrate systems of one-way constraints into conventional C++ code.

REFERENCES

[Bart86] Barth, P., "An Object-Oriented Approach to Graphical Interfaces", *ACM Transactions on Graphics*, v5, n2, April 1986, pp. 142-172.

[Born86] Borning, A., Duisberg, R., "Constraint-Based Tools for Building User Interfaces", *ACM Transactions on Graphics*, v5, n3, Oct. 1986, pp. 345-374.

[Born87] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M., "Constraint Hierarchies", *Proceedings of OOPSLA '87*, October 1987), pp. 48-60.

[Free90] Freeman-Benson, B., N., Maloney, J., and Borning A., "An Incremental Constraint Solver", *Communications of the ACM*, v33, n1, Jan. 1990, pp. 54-63.

[Henr90] Henry, T.R., Hudson, S.E., Newell G.L., "Integrating Gesture and Snapping into a User Interface Toolkit", Proceedings of the *ACM Symposium on User Interface Software and Technology*, Oct. 1990, pp. 112-121.

[Hill92] Hill, R.D., "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications", *Proceedings of SIGCHI '92*, April 1992, pp. 335-342.

[Huds88] Hudson, S. E., and King, R., "Semantic Feedback in the Higgens UIMS", *IEEE Transactions on Software Engineering*, v14, n8, August 1988, pp. 1188-1206.

[Huds89] Hudson, S. E., "Graphical Specification of Flexible User Interface Displays", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 105-114, November 1989.

[Huds90a] Hudson, S. E. and Mohamed, S. P., "Interactive Specification of Flexible User Interface Displays", *ACM Transactions on Information Systems*, v8, n3, pp. 269-288, July 1990.

[Huds90b] Hudson, S. E., "An Enhanced Spreadsheet model for User Interface Specification", University of Arizona Technical Report TR90-33, 1990.

[Huds91] Hudson, S., "Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update", *ACM Transactions on Programming Languages and Systems*, v13, n3, July 1991, pp. 315-341.

[Huds93] Hudson, S.E., "Eval/vite User's Guide (v1.0)", *Georgia Institute of Technology technical report GIT-GVU-93-xx*, 1993.

[Myer89] Myers, B.A., Vander Zanden, B., and Dannenberg, R. B., "Creating Graphical Interactive Applications Objects by Demonstration", *Proceedings of the ACM Symposium on User Interface Software and Technology*, Williamsburg, VA, Nov. 1989, pp. 95-104.

[Myer90] Myers, B.A., et al., "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces", *IEEE Computer*, v23, n11, Nov. 1990, pp. 71-85.

[Nels85] Nelson, G., "Juno a Constraint-Based Graphics System", Proceedings of SIGGRAPH '85, San Francisco, CA July 1985, pp. 235-243.

[Reps83] Reps, T., Teitelbaum, T., and Demers, A., "Incremental Context-Dependent Analysis for Language-Based Editors", *ACM Transactions on Programming Languages and Systems*, v5, July 1983, pp. 449-477.

[Vand90] Vander Zanden, B. and Myers, B.A., "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces", *Proceedings of SIGCHI '90*, Austin TX, April 1990, pp. 325-330.

[Vand91] Vander Zanden, B., Myers, B. A., Giuse, D., and Szekely, P., "The Importance of Pointer Variables in Constraint Models", *Proceeding of the ACM Symposium on User Interface Software and Technology*, November 1991, pp. 155-164.

[VanW82] Van Wyke, C.J., "A High-Level Language for Specifying Pictures", *ACM Transactions on Graphics*, v1, n2, April 1982, pp. 163-182.