A Framework for Automatic Algorithm Animation

Yi-Jing Lin

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-37
May 1991

A Framework for Automatic Algorithm Animation

Yi-Jing Lin

Department of Computer Science
Brown University
Providence, RI 02912

May, 1991

Abstract

Algorithm animation and data visualization have been shown to help people understand the behavior of algorithms and programs. Different approaches to algorithm animation and data visualization have been tried, but all the existing systems have drawbacks. On one hand, algorithm animation systems can generate nice animations, but users have to modify the original program and code the animation scenes by hand. On the other hand, data visualization systems are more automated, but they do not support smooth illustration of abstract operations and are not sufficiently general and flexible.

This research proposes another approach to data visualization and algorithm animation. Through automatic insertion of code, the manual modification of programs is avoided. By analyzing programs at preprocessing time instead of at run time, the program being monitored is made more efficient. By putting data objects and image objects into classes, the reusability of the animation methods and the flexibility of the whole system are increased. Finally, by automatically abstracting out high level operations from low level ones, high quality algorithm animation is generated without analyzing the algorithm manually.

1 Introduction

There have been many quite different approaches to algorithm animation and data visualization in the past. Systems like BALSA [4], TANGO [3], Incense, FIELD/DISP[5], and UWPI [10] all show great potential in this field. However, all these systems have their drawbacks. Algorithm animation systems like BALSA and TANGO generate nice animations, but users are required to modify the original program and code the animation scenes by hand. This is a laborious work, and the reusability of the additional code is not good. On the other hand, although data visualization systems like Incense, Amethyst, and FIELD/DISP are more automated, these systems are slow and be restricted to certain data types. The scenes generated by these systems are not smooth either.

In this paper, we propose another approach to data visualization and algorithm animation. Through automatic insertion of code, the manual modification of programs is avoided. By analyzing programs at preprocessing time instead of at run time, the program being monitored is made more efficient. By putting data objects and image objects into classes, the reusability of the animation methods and the flexibility of the whole system are increased. Finally, by automatically abstracting out high level operations from low level ones, high quality algorithm animation is generated without analyzing the algorithm manually.

This framework is designed for conventional languages like Pascal and C. In these languages, a program consists of a set of procedures. Procedures can be nested and recursive. For a procedure, the variables accessible to it include local variables, global variables, and its parameters. All the examples used in this paper are written in Pascal, although similar method can be applied to other languages like C.

1.1 Overall Structure

The overall structure of this framework is shown in Fig. 1. The framework is composed of four major parts: a source program preprocessor, an algorithm process, an animation process, and an user interface.

The program preprocessor takes a source program and a set of variable names as input. It analyzes the program and inserts message-sending calls to monitor the behavior of these variables. It also generates a symbol table for the user interface. The program generated by the preprocessor, called the *augmented source program,* can be compiled like a normal program.

The algorithm process is generated by linking the augmented program and a special module called the *shadow memory*. The primary function of the shadow memory is to filter out the false messages generated by the augmented program. When the augmented program calls an external function, the shadow memory is also used to check the effect of that function and to generate appropriate messages.

The animation process accepts messages from the algorithm process. These messages are guaranteed to be false-message free. They are first sent to the abstract operation generator, in which higher level operations like exchange, rotation, shift, and chunk operations are generated by sum-

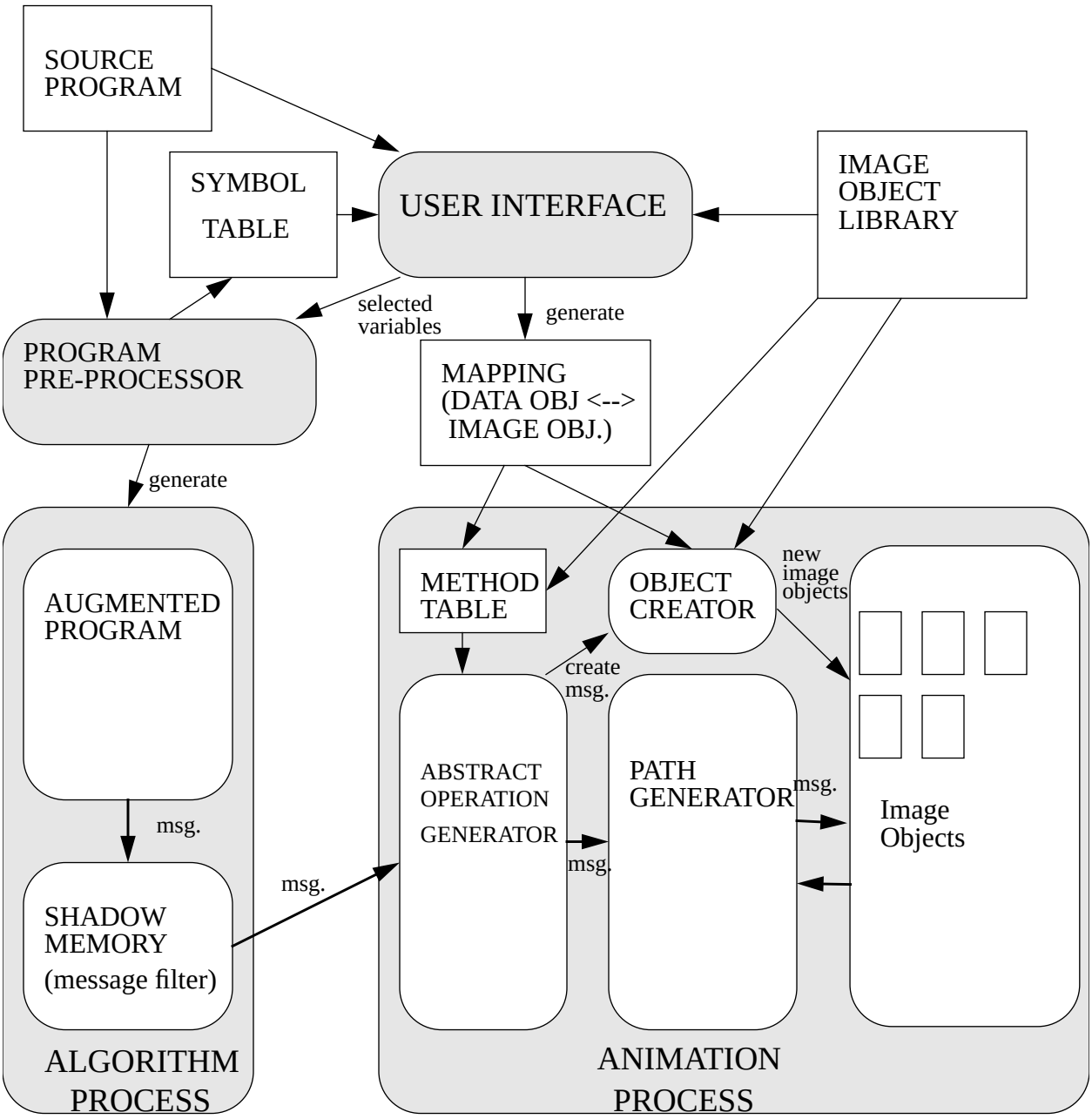

Fig 1. Overall Structure

ming up low level operations. Other data type dependent abstractions like push and pop of a stack can also be generated after consulting a method table. The path generator then takes these abstract operations, evaluates the resulting positions of images, and then generates paths for images by interpolation. Images in the animation process are object oriented objects. They are created when creation messages are sent to the object creator.

The user interface has two primary functions: selecting variables to be monitored, and establishing the mapping between classes of data objects and classes of image objects. To select monitored variables, the user interface needs the source program. To set up the mapping, it needs the symbol table of the program and the definitions of the classes of image objects. The user interface also checks the validity of the mapping.

1.2 The Flow of Messages

The communication between the major parts of this framework is based on message passing. Each message contains an message-type identifier and a list of parameters to describe the details of the message. As shown in Fig. 2, messages are generated by the augmented program, then processed by the shadow memory, the abstract operation generator, the path generator, and finally sent to image objects in the animation scene. As will be discussed in the next section, a message identifies its destination by the address of its associated data object.

In this section, we will only list the kinds of messages generated by each component of the system. More details about how these messages are generated and how they are used will be given in the following sections.

1.2.1 Messages Generated by the augmented program.

The following messages are generated by the augmented program:

- *Create* <address, type, initial_value>
  Creates a data object.
- *Destroy* < address>
  Destroys a data object.
- *Move* <to_address, from_address>
  Moves a value from a monitored data object to another monitored data object.
- *Arithmetic/Logic Operation* <target_address, op, new_value, operand1_addr, operand2_addr>
  Updates a value through the application of arithmetic or logic operations on monitored data objects or constants. This kind of operation is seldom seen in algorithm animation, but it is included to make our model complete.
- *Update* <address, value>
  Updates a value other than through movement and arithmetic/logic operations. This kind of operations occurs when the return value of a function is assigned to a variable, or when the source of a movement operation or an arithmetic operation is not monitored.
- *Compare* <address1, address2>
  Compares two monitored data objects.
- *Flush* <>
  Flushes the buffers in abstract operation generator. This message is generated at the end of

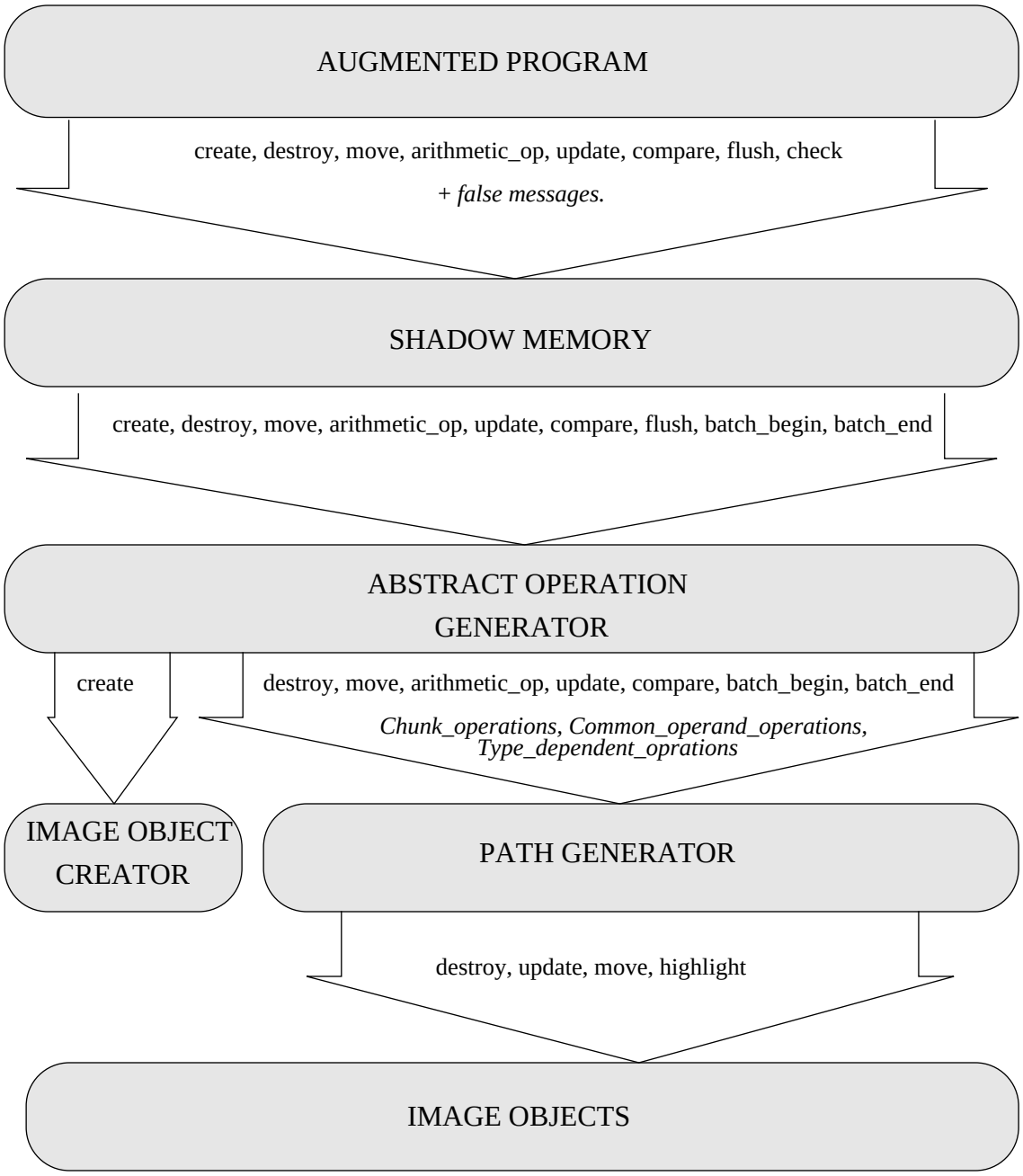

Fig.2 Message flow.

each procedure and before each I/O operation.

- *Check* <list_of_variable_addresses>
  Checks if the values of certain variables are modified by an external function. It is generated after an external function call.

The messages described above are similar to the instructions of an assembly language. This is not a coincidence. Actually, these messages describe the operations that can be applied to a data object.

1.2.2 Messages sent by the shadow memory.

The shadow memory will remove false messages and relay the correct ones. In addition, upon receiving a check message, it will compare variables with their old values and send appropriate messages to report the changes. These changes include creation and destruction of nodes in linked data structures and value modifications of variables.

If an external function causes more than one changes, the order of these changes is unknown. Since one might want to represent this fact in the animation, e.g., to update all the data at once instead of one by one, the shadow memory should also send the following two messages to mark the beginning and the end of these messages.

- *Batch_begin*
  This message is sent when the shadow memory receives a check message.

- *Batch_end*
  This message is sent when the checking is finished.

Check messages are not used beyond the shadow memory. Therefore, they will not be relayed to the abstract operation generator.

1.2.3 Messages Sent by the Abstract Operation Generator

The abstract operation generator tries to generate high level operations from low level ones. The primary target is the movement operation, which is the most common operation in algorithm animation applications. By buffering and analyzing message sequences, operations like exchange, rotation, shift, and chunk movement can be easily detected. Type dependent operations like node insertions of a linked list are more complicated but can be recognized after consulting the method table, which contains description of the underlying data structure of each monitored data object.

The abstract operation generator generates new messages listed below. All messages that fail to form a higher level abstraction are relayed to the path generator, except creation messages, which are sent to the object creator.

- *Chunk Operations*

Manipulates several fields of a record or several elements of a array simultaneously. This kind of operations can be further divided into*:*

- *Chunk Movement*

  Moves several fields of a record or array to another record or array.

- *Chunk Update*

  Updates several fields of a record or array.

- *Chunk Arithmetic Operation*

  Applies the same arithmetic operation to fields of same records or elements of same arrays and assigns the result to a record or array.

• *Common Operand Abstract Operations*

  Moves values between several data objects. This kind of operations can be further divided into:

- *Rotation*:

  Rotates values among several monitored data objects.

- *Exchange*

  Exchanges the value of two monitored data objects. This is a special case of rotation, but it is more commonly seen and hence much more important.

- *Shift*

  Shifts values among data objects and overwrite the value of the first target.

- *Initialization*

  Copies a value to several data objects.

• *Type dependent abstract operations*

  Operations like push and pop of a stack. These kinds of abstract operations are defined only on certain types of data objects.

1.2.4 Messages Generated by the Path Generator

The path generator generates paths for the transition of image objects. It receives messages from the abstract operation generator, determines the resulting position or status of the image objects, and generates the paths by interpolation. Abstract operations like exchange and shift are again resolved into simple operations like movement and updating, but with the transitions of the involved image objects being interleaved. Comparison operations are resolved into a sequence of high-light messages. More details about the path generator will be discussed later.

2 Program Preprocessor

Given a set of variables to be monitored, the program preprocessor analyzes a program, decides where the status of these variables may be changed, and then automatically inserts some message-sending calls to report the behavior of these variables. The program generated by this preprocessor is called the augmented program, which acts just like the original program except that it sends out messages to report the changes of status of monitored variables.

The data objects monitored include simple variables (e.g., integer and real numbers), structured variables (e.g., records and arrays), and linked data structures (e.g., linked lists and binary trees). To monitor a simple variable or a structure variable, the user can simply select its name. To monitor a linked data structure, the user should select the *handle pointer* of it. A handle pointer is a pointer through which all nodes of a linked data structure are reachable.

The messages generated by an augmented program may include some *false messages*. False messages are messages about the update of some data object that is not being monitored, or messages about the creation and destruction of nodes in some linked data structures that are not of our interest. As will be discussed, false messages are not always avoidable but can be filtered out by the shadow memory.

Although the method discussed here is originally designed for an animation system, similar methods can be applied to other systems where the status of certain variables is of interest. Although this paper only discusses the design of a preprocessor, the same method can be used to design a special-purpose compiler to avoid the redundancy of parsing.

2.1 Variable Status and Code Insertion

The status of a variable can be described by its activation and its values. During the execution of a program, a variable is *activated* when it is mapped to some memory location. A variable is *deactivated* when it does not map to any memory location. Although data objects are usually designated by variable names, the mapping between data objects and variable names is not one-to-one. A data object may be associated with several names when aliases exist. A heap-allocated data object may be anonymous, that is, no variable is mapped to it. A local variable of a recursive function may also have several instances in the memory simultaneously.

The attribute that identifies a data object uniquely is its address. Therefore, addresses should always be included in each message to identify which data object the message is associated with. Besides, a user request to monitor a variable *X* is interpreted as to monitor all the instances of *X*. When the data object being monitored is a pointer, all the memory locations reachable from it are monitored.

The status of a variable is changed when a new instance of it is created, when an instance of it is destroyed, and when the values of its instances are modified. Message-sending calls should be inserted at places where the status of monitored data objects are subject to change.

2.1.1 Creation Message and Destruction Message

A creation message-sending call is inserted right after a data object is created. It cannot be inserted before the creation, because the address of the new data object is unknown. Similarly, a destruction message-sending call should be inserted right before a data object is destroyed. It cannot be inserted after the destruction, because the address of the old data object will not be available.

A global variable is created when the program starts, and is destroyed when the program ends. A local variable is created when the control enters the procedure which defines it, and is destroyed when the control leaves. Therefore, for each global variable being monitored, a creation message-sending call and a destruction message-sending call should be inserted at the beginning and the end of the main routine respectively. And for each local variable being monitored, a creation message-sending call and a destruction message-sending call should be inserted at the beginning and the end of the procedure which defines it.

For heap allocated data objects, the creation occurs when the function *new()* is called, and the destruction occurs when the function *free()* is called.Therefore, if *p* is a pointer being monitored and a statement

```
new(p);
```

is found, then a creation message-sending call with the new address contained in p should be inserted after it. If *p* is a pointer being monitored and a statement

```
free(p);
```

is found, then a destruction message-sending call with the address contained in p should be inserted before it.

2.1.2 Value-Update Messages

Value-update messages include movement message, update message, and arithmetic-operation messages. These messages are sent when the value of a monitored memory location is modified.

The value of a variable will be modified when it appears at the left hand side of an assignment statement. For example, in a statement

```
a := b + c;
```

the value of *a* will be modified. A message-sending call should be inserted right after this kind of statements if *a* is being monitored.

Beside assignment statements, some other program constructs can also modify the value of a

data object. In Pascal, the value of a *control variable* for a for-loop is modified every time the loop is iterated. Therefore, an update message-sending call should be inserted at the beginning of the loop body if the control variable is monitored. In C language, the *increment operator* "++" and *decrement operator* "--" can also change the value of a variable. These operators can appear either in the postfix form or in the prefix form. When they appear in prefix forms like *(++i)*, such expressions can be substituted with expressions like *(MSGsenda("update", &i, ++i), i)*. When they appear in postfix forms like *(i++)*, such expressions can be substituted with expressions like *(MSGsenda("update",&i,++i), i-1)*. Here we take advantage of the *comma operator* "," of C language to ensure that the semantics of the program are not changed.

The value-update messages can be further divided into movement messages, arithmetic operation messages, and update messages. The movement message is used when the new value of a monitored data object is copied from a constant or another monitored data object. The arithmetic operation message is used when the new value comes from the result of a arithmetic operation on two or more monitored data objects and constants. The update message is used in all other cases.

2.1.3 Check Messages and Comparison Messages

When an external function is called, global variables and call-by-reference parameters of the external function are subject to change. Therefore, a check message-sending call should be inserted after the external function call when some of the call-by-reference parameters are being monitored, or when some monitored global variables are visible to that external function. The list of these variables can be decided by analyzing the program statically, and is included as parameters in the check message.

A *comparison* message-sending call is inserted when two monitored data objects are compared by comparison operators like "=", ''>", and "<>." The insertion point can be either before or after the comparison expression, because the value of these data objects will not be modified by a comparison operation.

2.2 Alias and Generalized Alias

When two or more variable names denote the same memory address, the names are *aliases.* Obviously, if a variable is monitored, then all its aliases should also be monitored. An update through any of these aliases will have the same effect on all the aliases.

Aliases can be formed by pointers. If p and q are pointers point to a same address, then **p* and **q* are aliases. For example, if the following statement appears,

```
p := q;          /* p and q are pointers.*/
```

then **p* and **q* form a pair of aliases. Or, if *p* is a pointer that points to another variable *x*, then **p*[1] and *x* are aliases. For example, if the following statement appears,

1. **p* denotes the memory address pointed by *p*, as in C language.

```
p := addr(x);      /* addr() return address of a variable.*/
```

then **p* and x forms a pair of aliases.

Call-by-reference arguments of procedures can also form aliases of variables. If the arguments of *B()* are call-by-reference, then a procedure call like *B(x, x)* makes the two arguments of *B()* a pair of aliases. Using a global variable as a call-by-reference parameter will also form a pair of alias. For example, in the following program:

```
program foo;
    var
        y: integer;
    procedure A(var x : integer);
        begin
            x := 5;
        end;
begin
    A(y);
end.
```

x and *y* will form a pair of aliases after *A()* is called from the main procedure.

2.2.1 Generalized Aliases

To monitor linked data structures, we have to generalize our notion of alias. When two or more names refer to the same linked data structure, these names are *generalized aliases*. If a linked data structure is monitored, then all its generalized aliases should also be monitored, because operations through any of these names will affect the monitored linked data structure.

The behavior of generalized aliases is very similar to that of “standard aliases.” They are formed by assignment to a pointer variable or to a pointer field of a node in a linked data structure. For example, each of the following statements will make *node1* and *node2* a generalized alias pair.

node1 ^.*rson := node2;*
node1 := node2 ^.field;
node1^.*field1* := *node2*^.*field2*;

Linked data structures are usually represented by their *handle pointers*. A handle pointer is a pointer though which all nodes of a linked data structure are reachable. For example, a binary tree is usually represented by a pointer points to its root, a linked list is usually represented by a pointer points to the first node in the list. With the notion of generalized alias, the user can monitor a whole linked data structure simply by choosing its handle pointer. While nodes of a linked data structure are usually allocated from the heap, handle pointers are usually non-heap-allocated.

2.2.2 Trivial Generalized Aliases

Often it is useful to monitor an array or a record as a whole. Under this circumstance, it is necessary to recognize all the updates of its elements. Therefore, names refer to elements of an array or a record are considered as *trivial generalized aliases* of their parent structure. If a structured data object is monitored, then all its trivial generalized aliases should also be monitored.

For example, if an array *A* is monitored, then all names with the form *A[i]* are considered as generalized aliases of *A* and should also be monitored. If a record *R* is monitored, then all names with the form *R.field* are considered as generalized aliases of R and should also be monitored.

Similarly, names formed by the "^." ("->" in C) operator are trivial generalized aliases of their parent structures, because they always point to the same linked data structure. For example, *node^.next^.next* is considered as a trivial generalized alias of *node*.

Complex names can be formed by using "^", "[ ]", and "." together. For example, *node1^.field1^.field2[4]* is legal in Pascal. But all these names can be recognized easily, and they are generalized aliases of their parent names. Therefore, they will not be treated explicitly in the following discussion.

In C, the address arithmetic is allowed. This will not be too much of a problem, because the address arithmetic is defined only when applying to arrays, and its effect is similar to the use of "[]." Address expressions not applied to arrays are undefined and rejected.

2.3 False Message

False messages are messages about something the user is not interested in. They are not always avoidable, because the alias pairs and generalized alias pairs obtained by analyzing a program represent the *possible* relations. Sometimes a name X might become the alias of another name Y during certain executions, but during some other executions, X and Y might refer to different addresses.

Take as an example, consider the following program:

```
program foo;
    var
        x, y: integer;
    procedure A(var z : integer);
        begin
            z := z + 5;
        end;
begin
    A(x);
    A(y);
end.
```

If *x*, but not *y*, is the variable we want to monitor, then after our processing the program should look like:

```
program foo;
    var
        x, y: integer;
    procedure A(var z : integer);
        begin
            z := z + 5;
            MSGsenda("update <address and new value of z>");
        end;
begin
    MSGsenda("create <address of x>");
    A(x);
    A(y);
    MSGsenda("destroy <address of x>");
end.
```

In this example, the main routine calls *A()* twice. In the first call x is used as the parameter, and the update of x in *A()* is reported correctly. But in the second call, y is used as the argument and the update of y is also reported, although the value of y is not of our interest.

The problem of false message is not caused by the automation of code insertion. The same problem will still be there if we analyze the program and insert code manually. However, we can avoid some of them by using better analysis algorithms, and get rid of others by using the shadow memory.

When trying to remove false messages, non-heap-allocated data objects and heap-allocated data objects are considered separately. For non-heap-allocated data objects, a creation message for a data object exists if and only if the data object is monitored. Therefore, a false value-update message can be recognized since there is not any corresponding creation message. In the example above, there is a creation message for x, but there is not any creation message for y, so the update message with the address of y is a false one.

For heap-allocated data objects, the situation is more complicated. The creation message itself can be a false message, because a new node might be created with a possible generalized alias. Therefore, false messages are detected by checking whether a data object is reachable from some non-heap-allocated data object. This is done with the shadow memory, as will be discussed in the next section.

2.4 Find All Names That Should Be Monitored

Given a set of variables to be monitored, the program preprocessor should find all their aliases and generalized aliases. This section discusses three approaches to do this. The first one is very conservative but also very simple. The second can give us more precise information with moderate cost in computation and storage. The third can give us the most precise information, but at the expense of complex algorithms and expensive computation.

2.4.1 Approach 1: A Conservative Algorithm

If a variable name *X* becomes an alias of another variable *Y*, it does not mean that *X* will remain as an alias of *Y* through the program execution. *X* may be assigned some new address and then become an alias of another variable. Similarly, if a pointer *p* points into a linked data structure *D*, it does not mean that p will always point to some node of *D* through the program execution. Some other addresses may be assigned to it.

However, to simplify the problem, we can assume that once a variable name *X* becomes an alias of a monitored variable *Y*, it will always be an alias of *Y*. Therefore, *X* is monitored through the whole program. Similarly, if a pointer *p* ever points to a monitored data structure, *p* is monitored throughout the program.

This conservative assumption treats a possible alias pair formed by a statement as being an alias pair through the whole program. Thus, the relation of possible alias pairs is treated as an *equivalence relation*, that is, a symmetric, reflexive, and transitive relation. These properties can be used to find all the names that should be monitored:

Algorithm 1: A Conservative approach to find all names to be monitored.

Input: A program *P* and a set *N* of names of *P* which are requested to be monitored by the user.

Output: A set *M* that contains all the names that should be monitored in *P*.

Method:

1. For each statement *S* in *P*:
   For each alias pairs p implied by *S*:
   put p in a set *R*;
2. Find the reflexive, transitive closure of *R*, call it *R**; /* Alias pairs are unordered. */
3. For each alias pair (*X*,*Y*) in *R** :
   if (*X* in *N*)
   put *Y* in *M*;

This algorithm requires very little computation and is very simple. However, it might monitor some unrelated names and thus generate more false messages, because the possible alias relation is not actually transitive. Alias pairs *(p, q)* and *(p, r)* do not necessarily imply the alias pair *(q, r)*. For example, consider the following fragment of code:

```
statement          alias-pair formed
-----------------------------------------
p := q;            /* (q, p) */
new(p);
r := p;            /* (p, r) */
```

If we use Algorithm 1 and apply the transitive property to these relations, we will get the alias pair *(q, r)*. But actually *r* will not become an alias of *q*, because the definition p := q is *killed* by the statement *new(p)*.

2.4.2 Approach 2: Expand Algorithms that Find Aliases

To overcome the problem stated at the end of previous subsection, data-flow analysis is used to calculate the possible alias relations on a statement-by-statement basis. This section discusses the off-the-shelf methods to find "standard aliases", and discusses how to tailor them to our need.

Aliases detection is an well-known problem in the field of compiler design, and the techniques to deal with it are quite mature. There are individual methods to find aliases in a program [13], and frameworks to solve the general *data-flow problem*[1][2][11]. These methods build a *control flow graph (CFG)* from a program, in which a node represents a statement or a segment of code, and an edge represents a possible transfer of control between segments of code. Associated with each node is some data that describe the facts at that point in the program. Usually, these facts can be represented as elements of a semi-lattice. Associated with each edge is a function which takes the facts at the source node of the edge as input, and feeds its output to the destination node of the edge. These functions describe the data-flow effects of edges. A *meet* operation is defined on elements of the semilattice and is used when several edges lead into a single node.

Since the behavior of generalized aliases is very similar to that of the "standard aliases", these compiler algorithms can be slightly modified to solve our problem. Actually, the CFG builder, the data structures at each node, and the meet operation are all the same. The functions that describe the effect of statements which generate generalized aliases is the only part that must be re-defined.

Let *GALIAS(p)* be the set of all generalized aliases of a variable *p*. Then for an assignment statement *S* which assigns a new value to a pointer variable *p*, like *p := q* or *p := q^.field*, the function associated with S is:

```
(1) FOR each variable r in GALIAS(p) DO
        GALIAS(r) := GALIAS(r) - {p}
(2) FOR each variable s in GALIAS(q) DO
        GALIAS(s) := GALIAS(s) U {p}
(3) GALIAS(p) := GALIAS(q) U {p}
```

For an assignment statement which assigns a new value to a pointer field "accessed through" *p*, like *p^.field1 := q^.filed2* or *p^.field := q*, the function associated with the statement is

(1) FOR each variable *r* in *GALIAS(q)* DO
GALIAS(r) := *GALIAS(r)* U *GALIAS(p)*
(2) FOR each variable *s* in *GALIAS(p)* DO
GALIAS(s) := *GALIAS(s)* U *GALIAS(q)*

The great advantage of this approach is that all these techniques are quite mature. There are many different algorithms available, some of which are more general, some more precise, and some more efficient. To improve the efficiency, interval analysis and basic blocks can be used [12]. To deal with separate compilation, summary information describing the effect of functions in a file can be used. To build an interactive environment, there are also incremental algorithms available [11]

2.4.3 Approach 3: Use Storage Shape Graph

The third approach is to use algorithms which are originally designed to detect conflicts between structure accesses [7] [9] [14]. These methods also use CFG to represent programs. But

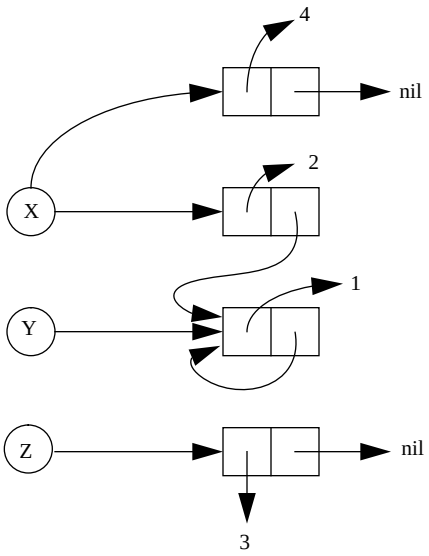

Fig3. A Storage Shape Graph. (SSG)

at each node of the CFG, the possible configuration of the heap allocated data structures is summarized by a graph, called the *Storage Shape Graph* (SSG) or *Alias Graph.* In the SSG, one node corresponds to possibly many nodes in the heap. For example, the SSG in Fig. 3 tells us that *X* and *Y* might point to a same data structure, while *Z* cannot be a generalized alias of *X* or *Y*.

This approach provides the most precise information about the relation between heap-allocated data objects. It distinguishes different fields of a record. It can discover data structures which are “true lists” or “true trees.” And it can also detect conflicts between references to a storage location. But this approach has its drawbacks. These techniques are expensive because the manipulation of SSG will take extra time and extra space, while still requiring the algorithms described in approach 2 to discover the “standard aliases”. In fact, these techniques give us more information than is necessary. In our problem, neither the information about the conflicts between data accesses nor the distinction between different fields of a record is necessary.

2.4.4 Tracing the Aliases Backward

The second and the third approaches provide information on a statement-by-statement basis. This information is more accurate, but accompanying it is a special problem that does not occur in the first approach.Consider the situation in which a binary tree is built from bottom up, as shown in Fig. 4. If the user requests to monitor the pointer *tree* only, then since p is not yet a generalized alias of *tree* at step (1) and step (2), the creation and update of node *p^* will not be monitored. When this node is finally connected to the tree at line (3), the animation process which receives messages from this program will find that *tree^.rson* points to an undefined location.

To solve this problem, we can simply request the user to put *p* under monitoring. But this is

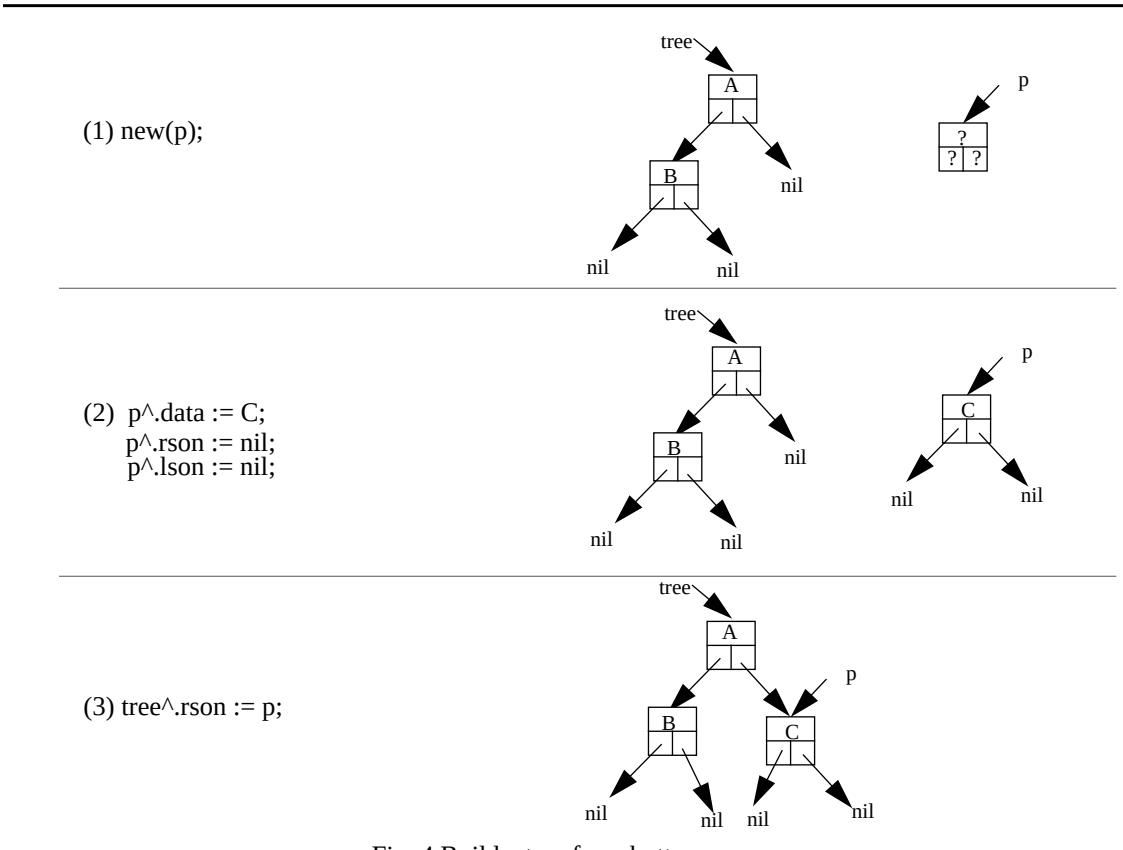

Fig. 4 Build a tree from bottom up.

inconvenient in this situation. Alternatively, this problem can be solved by tracing the aliases backward. Using the previous example, we can trace the generalized alias pair *(tree, p)* backward from step (3) and put *p* under monitoring at step (1) and step (2).

By recognizing that this is a *backward data-flow analysis problem*, the methods described in approach 2 to can be reused solve this problem. Actually, the same CFG, with the direction of all edges reversed, is used. In addition, similar, if not the same, functions describe the effect of each statement.

2.5 Comparison with Related Works

Every data visualization system and algorithm animation system needs some mechanism to monitor the behavior of variables, but different systems use different methods to meet this need. One commonly used method is to load a specially compiled program into a debugger, and then use the debugger to monitor the behavior of variables, as in FIELD/DISP [5] and *Incense* [15]. Another commonly used method is to ask the programmer to insert code manually, as in BALSA [4] and TANGO [3].

The disadvantage of the first method is that the debugger is a low level primitive tool. When it is not running in the interpretive mode, the user has to manually specify all the places where the value of certain variables should be reported. This is inconvenient, and users may fail to recognize the aliases of a variable. On the other hand, when it is running under interpretive mode, the whole process will be slow. And although the user can request to monitor certain addresses, he still cannot monitor a linked data structure automatically.

In comparison, the approach in this paper analyzes programs at preprocessing time and inserts code only when necessary. This strategy is more efficient. Furthermore, this method can monitor arrays, records, and linked data structure. Finally, this method also gives the user more information. For example, it can generate comparison messages and distinguish between movement operations and arithmetic operations.

In the manual code insertion approach, the major disadvantage is that it requires a lot of work from the programmer, and it will modify the original program. Although it gives the programmer freedom to define his own *abstract operation* and choose the interesting points, the insertion of code is a difficult and laborious task, especially when developing a new algorithm or program. Besides, modifying the original program makes it importable.

In comparison, the approach in this paper requires no manual modification of the original program. All the analysis and code insertion is done by the preprocessor. The user does not even have to understand the algorithm of the monitored program. This will be especially helpful when a user wants to know the behavior of an unknown program. In addition, since data visualization or animation can be acquired without any change to the original program, the system can serve as a high level debugger.

3 Shadow Memory

The shadow memory has two functions: false message filtering and message generating for external function calls. Functions to manipulate the shadow memory are put in a library and are linked with the augmented program generated by the program preprocessor.

3.1 Structure of Shadow Memory

The shadow memory maintains a *type table* and a *data object table*. The type table contains the definitions of types which have instances being monitored. It is generated by the program preprocessor and is read-only in the shadow memory package. The data object table contains an entry for each data item being monitored. Its contents are maintained by the functions associated with the shadow memory.

The type table is similar to the symbol table used by a compiler. It describes the structure of user defined types. Each entry in this table contains the following information:

- Type identifier: An unique identifier for the type.
- Type size.
- Pointers to element entries if the type is a record or an array.

Each entry of the data object table contains the following fields:

- Address
  The address of the data item. This is the key field of the data object table.
- Type
  The type of the data item. This is a pointer to an entry in the type table.
- Value
  The value of the data item.
- Reference count
  The number of paths from non-heap-allocated data objects. A zero means this data object is not reachable.
- Time stamp
  The relative time of last modification. This information is used when several unreachable nodes become reachable at once (described below). The clock can be implemented as a simple counter.
- Handle pointer
  The handle pointer of the linked data structure to which this data item belongs. This field

is used for heap-allocated data objects only, and is undefined for unreachable nodes.

3.2 False-Message Filtering

As discussed in the previous section, to remove false messages, non-heap-allocated data objects and heap-allocated data objects must be considered separately. For non-heap-allocated data objects, false messages can be recognized due to the fact that there are no creation messages for the addresses not being monitored. Therefore, an entry in the data object table is created only when a creation message is received. Value update messages that have no corresponding entry in the data object table are considered false message.

For heap-allocated data objects, messages for nodes which are not reachable from any non-heap-allocated data object are considered as false messages. The reference count field in the data object table serves to detect these false messages. Once the reference count of an entry becomes zero, it is considered as an unreachable node. Value update messages for a unreachable node will not be sent to the animation process.

Based on these discussion, the following actions are associated with each kind of message from the augmented program:

• Create <X,...>

```
If (X is heap-allocated)
        Create an entry for X in the data-object table, set the reference count to 0;
else
        Create an entry for X in the data-object table, set the reference count to 1;
set the time stamp of X
```

• Update <X,...>, Move <X, from,...>, or Arithmetic_op<X,...>

```
If there is an entry for X in the data object table
then
        Update the value of X;
        Update the time stamp of X;
        if (X is a pointer)
                for each entry Y reachable from the old value of X
                        Decrease the reference count of Y by 1
                        if (the reference count of Y is 0)
                                delete the entry of Y;
                for each entry Y reachable from the new value of X
                        Increase the reference count of Y by 1;
                        Set the handle_pointer field of Y to X;
                        if (the reference count of Y is 1)
                                Generate a creation message for Y and put it in buffer;
                Send messages in the buffer by the order of their time stamps;
else
        Do nothing, this is a false message.
```

• Destroy <X,...>

```
if (X is a pointer) or (X contains pointer fields)
        for each entry Y reachable from X
                Decrease the reference count of Y by 1;
                if (the reference count of Y is 0)
                        Delete the entry of Y;
Delete the entry of X;
```

• Compare <X, Y...>, Flush<>

Do nothing.

3.3 Generate Messages after an External Function Call

Whenever the augmented program calls an external function, call-by-reference parameters and global variables are subject to modification. Therefore, the values of these variables should be compared with their old values in the shadow memory. The following algorithm CHECK_VARIABLES is called when receiving a check message from the augmented program:

```
Algorithm CHECK_VARIABLES(L: list of call-by-reference variables and global variables)
    for each variable X in L
        call CHECK(X, false);


Procedure CHECK(X: address; New_Reference: boolean)
    if (X is an array or a record)
         for each element of X.
            call CHECK_ELEMENT(X, New_Reference);
    else
        call CHECK_ELEMENT(X, New_Reference );


Procedure CHECK_ELEMENT(X: address of variable; New_Reference: boolean)
    if (there is not an entry of X in the shadow memory)
        create an entry for X in the shadow memory;
        set the reference count of X to 1;
        set the time stamp of X;
        generate the creation message for X;
        if (X is a pointer)
            call CHECK(value_of_X, true)
    else
        if (New_Reference)
            increase the reference count of X by 1;
        if ((value of X in shadow memory) <> (value of X in memory))
            update the value of X in shadow memory;
```

```
        set the time stamp of X;
        generate an update message for X;
        if (X is a pointer)
                for each entry Y reachable from the old value of X
                        decrease the reference count of Y by 1;
                        if (the reference count of Y is 0)
                                delete the entry of Y;
                call CHECK(new_value_of_X, true)
else if (X is a pointer)
        call CHECK( value_of_X, New_Reference);
```

4 Mapping between Data Objects and Image Objects

An image object in the animation process is the visual representation of a data object in the algorithm process. The mapping from classes of data objects to classes of image objects is established in the user interface. It is used when creating a new image object in the image object generator and when generating class-dependent abstract operations in the abstract operation generator. With the mapping mechanism, users can visualize the same data structure with different methods, or visualize different data objects with different methods simultaneously.

4.1 Classes of Data Objects

Before the mapping can be established, data objects should be classified, and these classes should be given unique identities. Fig.5 shows the hierarchy of classes of data objects. In this hier-

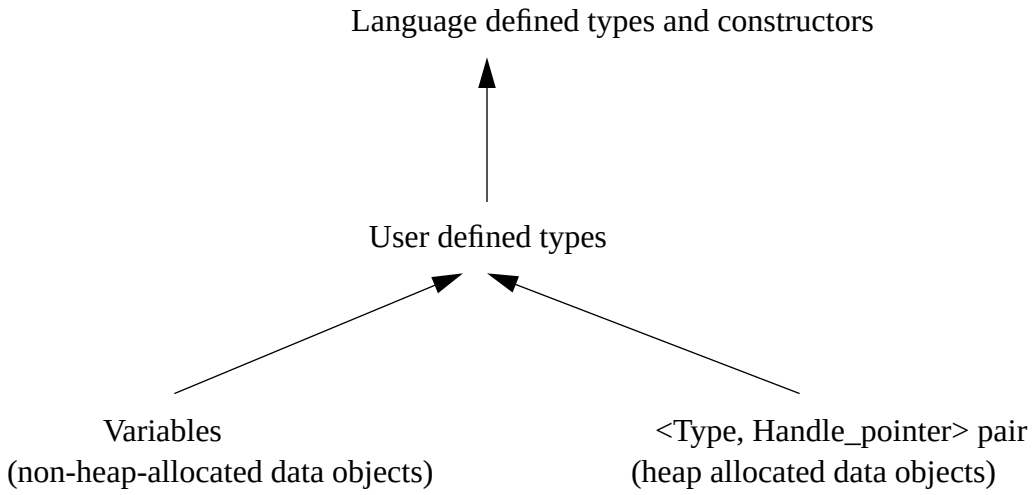

Fig.5 Hierarchy of classes.

archy, classes of data objects are put into three levels. At the top level are classes of language defined types and constructors, like integer, real, array, and record. At the second level are classes of user defined types. For non-heap-allocated data objects, at the bottom level are classes of vari-

ables. For heap-allocated data objects, at the bottom level are classes of <type, handle pointer> pairs.

If mapping is not defined for a certain class, then the mapping of its super-class is used instead for data objects in this class. If the user does not specify a mapping for a certain language defined type or constructor, which has no super-class, then the default mapping should be used.

For non-heap-allocated data objects, each variable forms its own class. For instance, given the declaration

```
type
        E = (red, green, blue);
        R = record
                f1, f2: E
        end;
var

        r, s: R;
```

the corresponding class diagram is shown in Fig. 6.

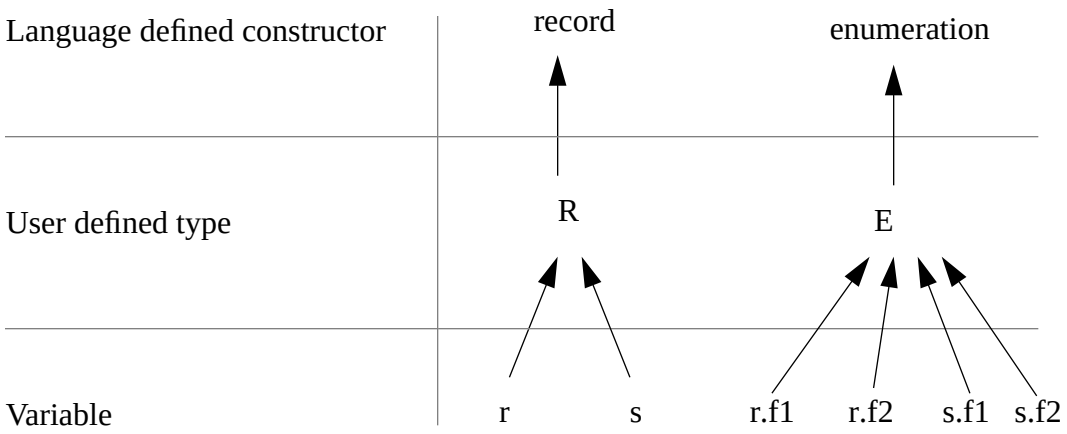

Fig. 6

That is, every instance of variable r in the program belongs to class r, which is a sub-class of R, which is a sub-class of record.

Non-heap-allocated variables are numbered (given identity) in the program preprocessor. This numbering is sent to the user interface through the symbol table. It is also used in the augmented program, as this unique number is included in each messages associated with the variable. Notice that the addresses of variables cannot be used as identifiers here, because the absolute addresses

are not known before the program is loaded into memory, and because a variable can have several instances with different addresses during the execution of a program.

For heap-allocated data objects, the situation is more complicated. First, there is not any variable name associated with a heap-allocated data object. Second, linked data structures may contain nodes of different types. Third, the user might want to visualize two linked data structures with different methods even when they are composed of the same kind of nodes. Therefore, both types and handle pointers are used to discern heap-allocated data objects. Given the declaration

```
type
        LTP = ^LT;                  /* pointers to left sons. */
        RTP = ^RT;                  /* pointers to right sons. */
        LT = record                 /* record for left sons. */
                lson: LTP;
                rson: RTP;
        end;
        RT = LT;                    /* record for right sons. */
var
        tree1, tree2: TP;           /* tree1 and tree2 are binary trees. */
```

the corresponding class diagram is shown in Fig. 7.

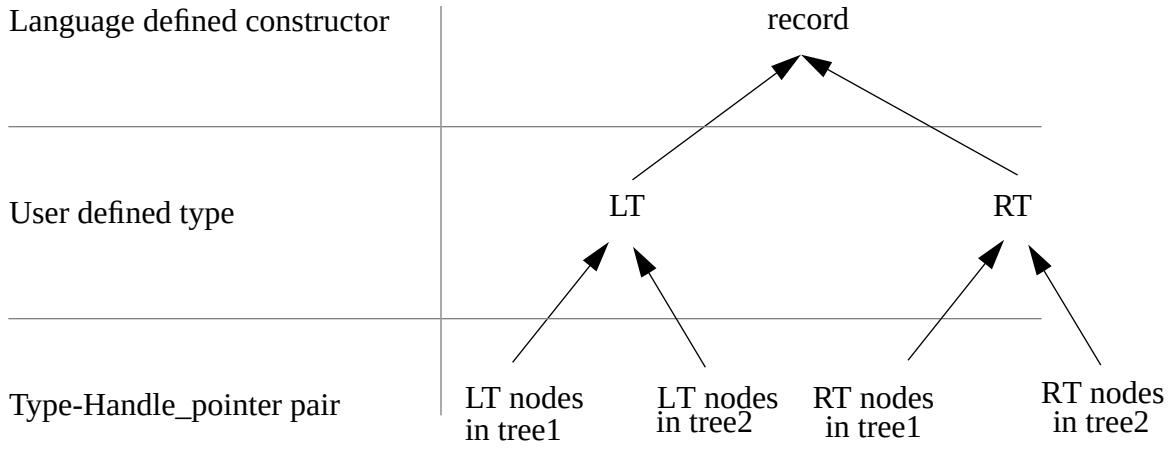

Fig. 7

Heap-allocated data objects are classified in the shadow memory, where the reachablility of nodes are decided. If a node of type T is reachable from a handle pointer P, then <T,P> defines the class of the node. Conflicts (multiple inheritance) might occur when a node is reachable from two different handle pointers. In this case, the user to decide the priority of classes, or a class is chosen randomly.

4.2 Image Objects

In data visualization and algorithm animation applications, it is more natural to design image objects by an object oriented approach for many reasons. First, images in an animation scene are inherently objects with i*dentify*. Each image object represent a certain data object in the program, so it is distinct from other images. Second, classifying objects into classes allows the creation of new images by instantiation without having to redefine every attribute of a new image. Third, some operations like move and highlight should be *polymorphic* so that they can be applied to all kinds of images. Finally, classes of image objects form an inheritance hierarchy very naturally. They should all share some basic attributes so they can be manipulated by the layout functions. All image classes that define the same data type or constructor should also share some attributes and operations.

The design of image objects should meet two criteria: it must be flexible enough to allow different layout methods be used, and it must be rigid enough so that different objects will be able to work together. The class diagram of the image objects is shown in Fig. 8. Here, the *Object Modeling Technique (OMT)* notations proposed by [18] are used.

The key point of this design is to distinguish between address images and value images. Each address image is associated with a data object in the algorithm process. Their positions may depend on each other. Each value image is associated with an address object. They are used to represent values of address images. The position of a value image depends on the position of the address image it is associates with.

4.2.1 Position of Address Objects

One of the great challenges of the design of image objects is to allow different layout methods. For value images, the layout problem does not exist because they are fixed in associated address objects and are moved if and only if the associated address images are moved. For window objects, a single geometric algorithm can be used to allocate space for windows, so the layout is not a problem either. But for address objects, different layout methods are possible. Based on previous work [3][4][5][10], the position of an address object can be defined by one of the following methods.

(1) Plot against window coordinates
examples: scalar pairs in 2-D, scalar triples in 3-D, Double Linked Edge Lists (DLEL)[8]. No value image is associated with this kind of address objects. The address object is moved when its value is updated, or when the window is moved.

(2) Fixed in its parent structure
examples: elements of an array, elements of a record.
This kind of address object is moved when its parent node is moved.

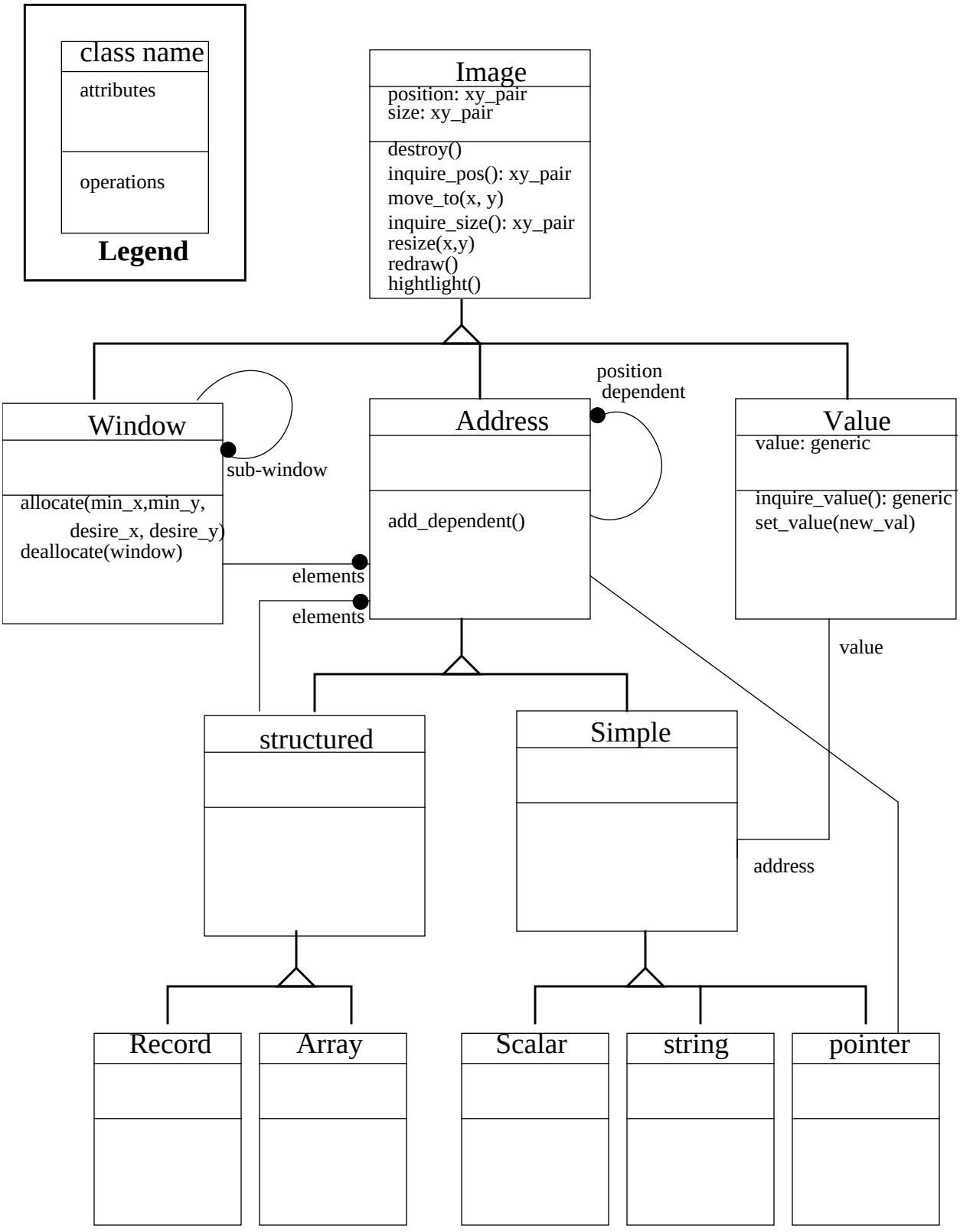

Fig. 8 Classes of Image Objects

(3) Decided by one of its field
examples: nodes in a inverted tree.
This kind of address object is used for record which contains a fly pointer or an index to an array. It is moved when position of that field is changed.

(4) Decided by what it points to
examples: fly pointer, index of array, inverted tree.
No value image is associated with this kind of address object. It is moved when its value is updated, or when the image object it points to is moved.

(5) Decided by what points to it
examples: linked list, standard binary tree, 2-3 tree.
This kind of address object is moved when the node pointing to it is moved.

(6) Layout globally
examples: graph represented as adjacency matrix, as v-list and l-list, or as linked nodes.
All the address objects in the data structure should be redrawn when the relation between two of them is modified.

The position dependency among address objects is represented by the association *position dependent*. An address object X is position dependent on another address object Y if the position of X depends on the position of Y. For objects that fall into the second and the third categories, this relation is established when a new address object is created. For objects that fall into the fourth and the fifth categories, this relation is established when a node is linked onto a linked data structure. For objects that fall into the last category, their handle pointer or their parent structure is position dependent on each element.

When the value of an address object is modified, the address object should check if its own position should be changed. If the answer is yes, it should inform all its position dependents to make a similar check.

4.2.2 Coordinates within Windows and Address Objects

Data structures can be deeply nested. In order to deal with complex data objects, image objects should be able to draw themselves without knowledge of the type and the position of their parent structure. A simple scheme to meet this requirement is to have each window and each address object define a coordinate mapping. Externally, they have their own sizes and positions. Internally, each of them defines a coordinate system from (0,0) to (1,1). Sub-images (including windows, address objects, and value objects) are plotted against this internal coordinate system. By enforcing that the internal coordinate system is from (0,0) to (1,1), windows and address objects can be recursively plotted inside others without consideration of the real size and real position of the images.

Although three dimensional data visualization is not commonly used (except for plotting scalar triples), it is straightforward to define 3-D windows and 3-D address objects. They can define pro-

jections from 3-D to 2-D, or mappings from 3-D to 3-D. Again, all sub-images should be positioned within the (0,0,0)-(1,1,1) box.

4.3 The Mapping

A mapping from classes of data objects to classes of image objects is established in the user interface.The mapping does not have to be one-to-one, several classes of data objects can map to the same class of image objects.

Some image objects need information other than the data object they are mapped to. For example, if an integer is used to represent an index of an array, then the image object representing this integer requires the address of the array and the size of its elements. This kind of relation is also established in the user interface.

5 Abstract Operation Generator

As shown in Fig. 9,the abstract operation generator is further divided into three stages, the *common operand detector*, the *chunk operation detector*, and the *type dependent operation detector*. Each of these stages takes messages from the previous stage and generates operations of higher abstraction.

5.1 Common Operand Detector

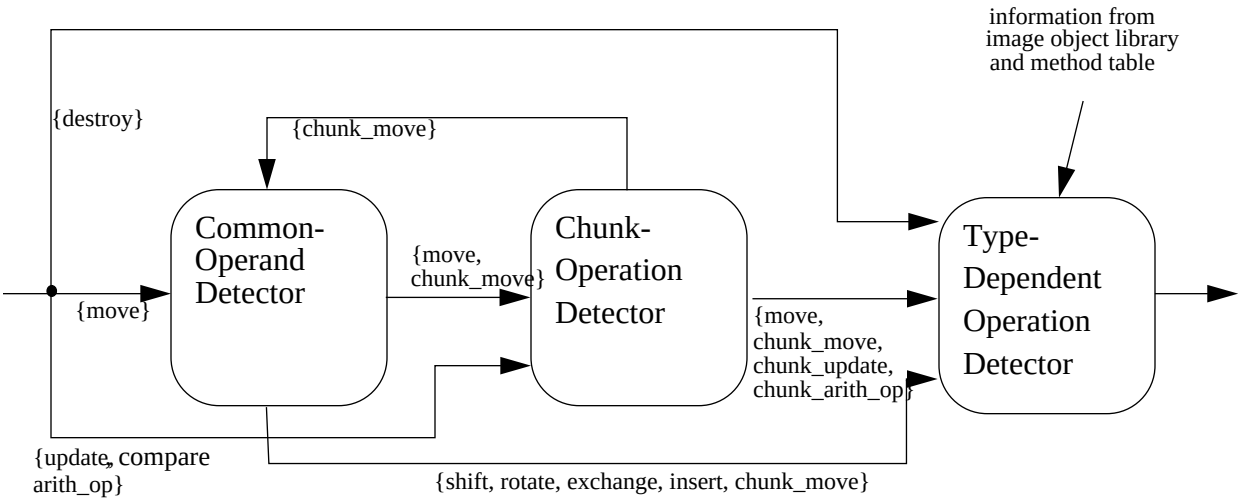

Fig. 9 Stages of Abstract Operation Generator

The first kind of abstract operation is generated by observing the sharing of operands between consecutive movement messages. For example, if the following statements are used to exchange the contents of two variables X and Y in a program:

```
tmp := X;
X := Y;
Y := tmp;
```

Then the following messages will be generated:

```
     op      to      from
1:   move    <tmp,   X>
2:   move    <X,     Y>
3:   move    <Y,     tmp>
```

Here the *from-field* of the first message is the same as the *to-field* of the second, the from-field of the second is the same as the to-field of the second, and the from-field of the third is the same as the to-field of the first.

Other abstract operations that fall into this category include shift, rotation, and initialization. Each of these abstract operations corresponds to a sequence of messages in which operands are shared between adjacent messages. In general, such an abstract operation can be described by two conditions, the *sequence condition*, which defines the relation between adjacent messages, and the *boundary condition*, which defines the relation between the first message and the last message. Table 1 lists these operations and the conditions defining them, where *message[i].from* and *message[i].to* represent the from-field and the to-field of the *i*-th message respectively.

abstract operation	Sequence condition (Relation between adjacent messages)	Boundary Condition (Relation between the first and the last messages)
Exchange	message[i-1].from = message[i].to	message[first].to = message[last].from last - first = 2
Rotation	message[i-1].from = message[i].to	message[first].to = message[last].from
Shift	message[i-1].from = message[i].to	
Initialization	message[i-1].from = message[i].from	

Table 1: Common-Operand Abstract Operations

These kinds of abstract operations can be detected by buffering messages and then comparing the current message with the buffered messages. Such an algorithm is listed below. Messages of common-operand abstract operations are sent to the Type-dependent Operation Detector. Other move messages that fail to form abstract operations are sent to the chunk movement detector.

```
If (the message buffer is empty)
        buffer the new message;
```

```
        possible_abstrations := {};
else if (the message buffer contains one message only)
        for (each abstract operation A whose sequence condition is satisfied by the
              new message and the message in the buffer)
                put A in the set  possible_abstrations;
        if (possible_abstrations = {})
                send the message in the buffer;
        buffer the new message;
else
        if (no abstract operation in possible_abstrations has a sequence condition
           which is satisfied by the new message and the last message in the buffer)
                Summarize the messages in the buffer;
                Send the summarized message;
                Clear the message buffer;
        else
                Delete all the abstract operations in possible_abstrations whose
                sequence condition is not satisfied;
        buffer the new message;
```

5.2 Chunk Operation Detector

The second kind of type independent abstract operation is based on the sharing of a parent data structure between operands of consecutive messages. If several messages are of the same kind of operation and all operate on elements of a same record or array, a higher level operation can be used to represent these operations.

For example, consider these two movement messages:

```
move <to1,        from1>
move <to2,        from2>
```

If *to1* and *to2* belong to the same record, and f*rom1* and *from2* also belong to the same record, then these two messages are combined into a single chunk-movement message. The address of the parent record or arrays should also be included in the chunk-movement message for future processing.

Messages that fail to form a chunk-operation are sent to the next stage, the *type dependent operation detector*. Chunk-operation messages are sent back to the previous stage, the shift/rotation detector, where chunk-operation messages are treated as normal messages. Since arrays and records can be nested, the chunk-operation detector may combine several chunk-operation messages into a single higher level chunk-operation message.The operands of a nested chunk-operation can be represented by a Lisp-like s-expression.

abstract operation	Type of Message	Sequence Condition
Chunk_Movement	Movement	parent(message[i-1].to = parent(message[i].to) parent(message[i-1].from = parent(message[i].from)
Chunk_Update	Update	parent(message[i-1].target = parent(message[i].target)
ChunkArith_Op(1)	Arith_op	parent(message[i-1].operand1) = parent(message[i].operand1) parent(message[i-1].operand2) = parent(message[i].operand2) message[i-1].op = message[i].op
ChunkArith_Op(1)	Arith_op	parent(message[i-1].operand1) = parent(message[i].operand1) parent(message[i-1].operand2) = parent(message[i].operand2) message[i-1].op = message[i].op
ChunkArith_Op(2)	Arith_op	message[i-1].operand1 = message[i].operand1 parent(message[i-1].operand2) = parent(message[i].operand2) message[i-1].op = message[i].op
ChunkArith_Op(3)	Arith_op	parent(message[i-1].operand1) = parent(message[i].operand1) message[i-1].operand2 = message[i].operand2 message[i-1].op = message[i].op

Table 2: Chunk Operations

Each chunk operation is described by a sequence-condition and the type of message it is summarized from. Table 2 lists these operations. The algorithm to detect chunk-operations is basically the same as the one used to detect common operand abstractions, but since there are no boundary conditions for chunk operations, the summarization of low level messages will be easier.

5.3 Type Dependent Operation Detectors

At this stage, type dependent abstract operations are summarized from all kinds of messages, including messages for primitive operations and messages for type independent abstract operations. Some of the type dependent abstract operations are:

- Linked list: node insertion, node deletion.
- Stack: push, pop.
- Array: insertion.
- Vector: inner product, cross product.

The detectors for these operations are defined in the image object library. Each of them is associated with a class of image object. A type dependent operation detector is activated only when some instances of its class are being used. Each of these detectors also defines an *applicable condition.* A message will be processed by a certain detector if and only if the applicable condition is

satisfied. The applicable conditions, along with the definition of each detector, are stored in the method table.

Type dependent abstract operations are more complicated than the type independent ones. Besides the sharing of operands and parent structure, these detectors might need other information. For example, to detect abstract operations of a stack, the detector must know which integer is used as the stack pointer. In general, more information can be acquired from the image objects, where the relation between objects are represented as links. Each class of image object should support appropriate inquiring operations needed by its associated abstraction detector.

5.4 Delay Caused by the Buffering of Messages

Because of the buffering of messages in the abstract operation detectors, there is some delay between the status of the algorithm process and the scene shown by the animation process. To avoid unnecessary delay and to keep the user from noticing the delay, the following actions are taken.

First, the messages of an abstract operation are assumed to be consecutive, and the buffers are flushed whenever the sequence of messages is discontinued. This assumption is true for most of the time, and this rule is used in algorithms discussed in this section.

Second, it is assumed that messages of an abstract operation will not span across an I/O operation like *read()* and *write()*. According to this assumption, message buffers are flushed right before an I/O function is called. This is done by inserting a flush message-sending call in front of each I/O function call in the program preprocessor. Because users communicate and synchronize with programs by these functions, this rule prevents users from noticing the delay. Every time a user communicates with the algorithm process through an I/O operation, the animation process correctly shows the status of the program.

Third, it is assumed that messages of an abstract operation do not span across the boundaries of functions, and the buffers are flushed at the end of each function. This is done by inserting a flush message-sending call at the end of each function in the program preprocessor. Again, this assumption is usually true.

6 Path Generator

The path generator is actually the manager of image objects. Its operation is divided into three stages. First, the path generator resolves the type independent abstract operations back into low level messages and sends these low level messages to the image objects as a batch. Second, the image objects calculate their new status (position e.t.c.) and register the required changes with the path generator. Finally, the path generator generates paths for these transactions by interpolation, and interleave the operations of each image object by sending appropriate messages to the image object again.

In general, interpolation generates smoother transitions between animation scenes, but the interpolation of some kinds of data is meaningless. For example, interpolation of a changing pointer

value would only confuse the user. Therefore, when registering the required change with the path generator, an image object also indicates whether the interpolation is necessary.

Basically, a single interpolation algorithm is used for all interpolations. The interpolation of a structured data object is done by interpolating each of its elements. As in some graphic animation systems [20], we can support multiple interpolation methods and let users choose one among them.

7 Conclusion

The framework proposed here is different from other data visualization systems and algorithm animation systems in three aspects. First, it uses a preprocessor to find interesting points of a program and insert code automatically. This provides more information and is more efficient than a debugger. It is also more convenient than inserting code manually. Second, this framework classifies data objects and image objects into hierarchies of classes and allows different mappings between these two kinds of objects. With this mechanism, users can choose to visualize data with different methods, while default methods are still available for classes of data objects. Third, it uses an abstract operation generator and a path generator to detect and animate abstractions automatically. This helps users understand a program without analyzing it manually.

The computational bottleneck of this system is in the layout of data structures. The scan conversion of graphic primitives usually takes a lot of time. Experience shows that this is also the bottleneck of previous systems like BALSA, TANGO, and FIELD/DISP. The trend of computer graphics is to use special designed hardware to accelerate the drawing process. This might improve the system performance of our system greatly.

In terms of space, again the bottleneck is in the layout of data structures. Interestingly enough, the limitation appears in the available screen space, not in the available memory. Since the memory space and the screen space required by this system are both proportional to the number of data items being monitored (if zooming is not considered), a limitation in one of them will prevent the monitoring of more data. While workstations nowadays usually support giga bytes of virtual memory, screens are typically thousands of pixels by thousands of pixels.Therefore, the information displayable on a screen is rather limited. A solution to this problem is to use scrollable screens. But scrollable screens need larger frame buffers or more complex drawing modules, and thus cause some overhead in computation and in memory space.

This framework can be considered as an advanced data visualization system, or an automatic algorithm animation system. It generates scenes automatically like a data visualization system, and demonstrates abstractions smoothly like an algorithm animation system. But this system is not suitable for the simulation of real-world activities. Algorithm animation systems like TANGO are sometimes used to demonstrate scenes in the real world, like to show the movement of disks in a *Hanoi Tower* problem. This system cannot do this because the internal data structure in this kind of program does not have to correspond to the status in the real world. However, when performing this kind of animation, a system like TANGO degenerates to a drawing package. Actually, this kind of simulation is more suitable for simple but efficient drawing packages like PHIGS[17] and SRGP[19].

References

[1] A.V. Aho, R. Sethi, and J. D. Ullman. “Code Optimization” in Compilers, Principles, Techniques, and Tools, pages 585-722, *Addison-Wesley* (1988).

[2] M. S. Hecht. “Flow Analysis of Computer Programs,” *Elsevier North-Holland*. (1977)

[3] John T. Stasko.”TANGO: A Framework and System for Algorithm Animation.” Brown University Computer Science Department Technical Report CS-89-30. (May 1989)

[4] Marc. H. Brown. “Algorithm Animation.” Brown University Computer Science Department Technical Report CS-87-05. (April 1987)

[5] Steven P. Reiss, “Interacting with the FIELD environment,” Software Practice and Experience 20(S1) pages 89-115 (June 1990)

[6] S. Horwitz, P. Pfeiffer, and T. Reps. “Dependence analysis for pointer variables.” *Proc. SIGPLAN’89 Symp. on Compiler Construction.* (June 1989) Published as *SIGPLAN Notices Vol. 24*, No. 7.

[7] N. D. Jones and S. S. Muchnick. “Flow analysis and optimization of LISP-like structures.” In S. S. Muchnick and N.D. Jones, editors, “Program Flow Analysis, chapter 4, pages 102-131. *Prentice Hall.* (1981)

[8] Franco P. Preparata and Michael lan Shamos. “Computational Geometry, an Introduction,” *Springer-Verlag*. pages 15-17 (1985)

[9] J. R. Larus and P. N. Hilfinger. “Detecting conflicts between structure accesses.” Proc. SIGPLAN’88 Symp. on Compiler Construction, pages 21-34 (July, 1988) Published as *SIGPLAN Notices Vol.* 23, No. 7.

[10] Robert. R. Henry, Kenneth. M. Whaley, and Bruce Forstall. “The University of Washington Illustrating Compiler.” *Pro. of the ACM SIGPLAN’90 Conference on Programming Language, Design and Implementation*, pages 223-233. (June, 1990)

[11] M. Burke. “An interval-Based Approach to Exhaustive and Incremental Interprocedual Data-Flow Analysis.” *ACM Trans. on Programming Languages and Systems*, Vol. 12, No. 3, pages 341-395. (July, 1990)

[12] R. Tarjan. “Testing flow graph reducibility.” Compute. Syst. Sci. 9, 3, pages 355-365. (Dec. 1974)

[13] J. P. Banning. “A Method for determining the Side Effects of Procedure Calls.” Ph.D. Dissertation, STAN-CS-78-676, Stanford University. (August, 1978)

[14] D. R. Chase, M. Wegman, and F. K. Zadeck. “Analysis of Pointers and Structures.” *Proc.*

SIGPLAN'90 Conf. on Programming Language Design and Implementation, pages 296-310. (June, 1990)

[15] Brad A. Myers. "INCENSE: A system for displaying data structures." *Computer Graphics, Vol* 17, No. 3, pages 115-125. (July, 1983)

[16] Brad A. Myers, Ravinder Chandhok, and Atul Sareen. "Automatic Data Visualization for Novice Pascal Programmers." In IEEE Computer Society Workshop on Visual Languages, pages 192-198, Pittsburgh, PA. (Oct., 1988)

[17] ANSI (American National Standards Institute), AMerican National Standard for Information Processing Systems - Programmer's Hierarchical Interactive Graphics System (PHIGS) Functional Description, Archive File Format, Clear-Text Encoding of Archive File, ANSI, X3.144-1988, ANSI, New York, (1988)

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. "Object-Oriented Modeling and Design" *Prentice Hall, Inc*. (1991)

[19] James D. Foley, Andries vanDam, Steven K. Feiner, and John F. Hughes. "Computer Graphics: Principles and Practice" Second Edition. *Addison-Wesley*. pages 25-66. (1990)

[20] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John Hughes, and Andries van Dam. "An Object-Oriented Framework for the Integration of Interactive Animation Techniques." Proceedings of the ACM SIGGRAPH, Computer Graphics. Vol. 25, No. 4 (Aug., 1991)