Algorithm Animation using Shape Analysis: Visualising Abstract Executions

Dierk Johannes, Raimund Seidel, Reinhard Wilhelm
Department of Computer Science
Universität des Saarlandes
66123 Saarbrücken, Germany
{johannes, rseidel, wilhelm}@cs.uni-sb.de

Abstract

This paper describes progress in a non-traditional approach to algorithm animation. We visualise the abstract execution of an algorithm instead of animating the algorithm for executions on concrete input data. Algorithms under consideration are imperative pointer-based algorithms. For our purpose, we need to compute invariants which describe the state of the heap. The most powerful method to accomplish this task automatically is shape analysis. However, the output of a shape analysis can get confusingly large and complex, and it may contain redundancy as well. We need to identify those configurations which are important according to our visualisation aim, such as teaching the algorithm. We present suitable methods for structuring the shape analysis output and for finding expressive configurations. Our approach is exemplified for binary tree algorithms.

CR Categories: D.2.11 [Software Engineering]: Software Architectures—Data abstraction; E.1 [Data]: Data Structures—Graphs and networks, trees; E.2 [Data]: Data Storage Representations—Linked representation; K.3.1 [Computer and Education]: Computer Uses in Education

Keywords: Abstract execution, Algorithm animation, Invariant, Shape analysis, Tree, Visualisation

1 Introduction

Algorithm animation has been a research topic for many years. The educational benefit for teaching algorithms is undisputed. Besides, this field shows a stimulating relation between theoretical concepts and challenges on the one hand, and practical applicability on the other hand. We refer to [Kerren and Stasko 2001] for an overview of algorithm animation, further references can be found there too.

Traditional algorithm visualisation systems have evolved from the idea of visualising the execution of concrete programs with concrete input. Usually they have more than just a good visual representation of a concrete program and data states, but they also offer quite an amount of helpful extra information like different views of the program and data states. The best algorithm visualisation systems are pretty elaborate and efficient. We also observe a tendency for finding more and more abstract representations of the program (or algorithm) and its data state and for offering the user more abstract views. This development is also reflected in algorithm animation taxonomies, where a distinction between program animation and algorithm animation is suggested, compare [Price et al. 1993] for example. In this paper, we always think of and argue about algorithms. But sometimes we use the word program synonymously, we find it for instance more customary to speak about program points.

In spite of the trend towards abstraction, traditional algorithm animation systems are still bound to concreteness: They visualise (animate) an algorithm with concrete data. They execute the algorithm with different input data sets and show how the program states change according to the data chosen. They are focussed on these changes. To make clear what stays the same, what is invariant, is beyond their capabilities. This limitation has practical implications. It is worth thinking about whether visualisation of concrete executions is the right approach for teaching algorithms. The student may be focussed too much on concrete data. Furthermore, this approach does not necessarily help students to see and understand similarities and differences of instances and how this affects the execution.

Our approach starts from the opposite direction. Usually, there are different data sets resulting in exactly the same (or very similar) execution paths. There is no need to visualise two such paths separately. The idea of representing and visualising a whole class of similar input instances simultaneously emerges quite naturally. For this reason we need an abstract description of sets of execution states, we need to compute invariants. The execution of an algorithm in terms of these abstract descriptions is the basis of our visualisation.

We want to visualise algorithms that are imperative and manipulate pointer structures. This is no real restriction. Quite to the contrary, pointers are semantically the most difficult programming language constructs. For reasons of efficiency, algorithms are usually implemented in an imperative programming language, where dynamic data structures are realised in a pointer-based way and stored in the heap.

The task of computing invariants is performed by shape analysis as implemented in the TVLA system (three-valued-logic analyser), see [Sagiv et al. 2002]. We will briefly describe shape analysis in the next section. This analysis results in an abstract description of the heap states by means of logical structures. These can be interpreted as graphs, so-called *shape graphs*. The set of shape graphs for some program point describes all heap configurations that can appear there. They are an invariant for this program point. These shape graphs are our objects for visualising the abstract execution

of the algorithm. In contrast to traditional approaches, we are more focussed on what stays the same, what is invariant.

In order to obtain such invariants, we first have to specify an "analysis". This seems to put our approach to a disadvantage compared to traditional algorithm visualisation systems. But this is not necessarily the case. This analysis involves a logical specification of data structures – which has only to be done once and for all per data structure – and an initial specification of heap configurations. The latter is no more expensive than implementing the algorithm and finding some expressive input data sets which is required for traditional systems in any case. We are as close to the aim of automatic visualisation as traditional systems are.

The output size of a shape analysis is generally large, at least too large to consider visualising it directly. The first data structures that were examined by the TVLA developers were lists. They produce only small-sized output. Now, research interest has proceeded to more complex data structures like trees. They reveal this problem of large output sizes. TVLA needs to generate and distinguish many shape graphs for a correct analysis. Often, redundancies are introduced as well. The visualisation, however, does not need this level of detail. For the purpose of teaching, for explaining the idea of an algorithm, we are for example more interested in general cases and not in special ones. Furthermore, there is no real need for visualising similar execution paths (the shape graphs for every program point are similar) separately, they offer no further insights. So, which shape graphs are important for visualisation? Concepts to answer this question will lead to an increase of explanatory quality of our approach to visualisation.

1.1 Subject of the Paper

This paper deals with mechanisms to handle shape analysis outputs of large size. We focus on *structural properties*. We introduce the notion of similarity between shape graphs to partition the set of shape graphs belonging to the same program point into classes. This *class partition approach* supports coarsening and refining of partitions to adjust the level of detail for visualisation. Moreover, we explain how to choose executions paths appropriate for visualisation. Thereafter, we show how to visualise a class of shape graphs. But we do not address how single shape graphs could be drawn. For instance, the choice of node position, edge routing, or use of colour are not discussed. Because of this focus on structural properties and not on the actual drawing process, we prefer to use the neutral term visualisation throughout this paper. Our approach is exemplified for binary trees; we use the search for an element in a binary search tree as a running example.

The above mentioned concepts make our approach to algorithm visualisation very versatile. The main intention is to explain an algorithm for the purpose of teaching. As pointed out, the level of detail can be changed from one visualisation to another, even during a single visualisation. So, we can adjust the explanation to the experience and knowledge of the student, and he is free to see parts of the algorithm with different precision. The class-partition approach is flexible enough that even experienced users can gain new insights into the behaviour of an algorithm.

1.2 Structure of the Paper

We briefly describe shape analysis in Section 2. Section 3 contains our major ideas. We present a similarity concept for shape graphs in Subsection 3.1. We continue in Subsection 3.2 with an approach to explore the structure of execution paths. Section 4 describes several methods on how to visualise a class of shape graphs. In Section 5 we report about a prototype implementation. Section 6 presents related work. In Section 7 we conclude the paper and mention directions for future work.

2 Shape Analysis

We want to visualise the abstract execution of an algorithm. Therefore, we need to compute invariants. The currently most powerful means to accomplish this task is shape analysis. This term does not refer to a special algorithm, but to static program analysis methods based on three-valued logic. The TVLA system implements the shape analysis method. Compared to other approaches available, shape analysis has some advantages that make its use as a basis for visualisation very attractive: it always terminates; it generates the analysis automatically without the need for user interaction; it has a semantic foundation, that is, the resulting analysis has a precise logical meaning.

But maybe the main advantage of shape analysis over other approaches available is the fact that shape analysis is parametric: The analysis can automatically be instantiated with different "vocabularies". Thus, by choosing an appropriate vocabulary, we can define the properties we are most interested in and neglect others. This allows to adjust the analysis to different levels of precision. Thus we have the freedom to choose a vocabulary that best fits our visualisation aim.

Shape analysis was developed to analyse imperative pointer-based programs. The work was motivated by the fact, that pointers are still difficult objects. For example, there are only a few theoretical treatments of pointer semantics. From a practical point of view, pointers are very error-prone. Two common mistakes are dereferencing NULL pointers and accessing previously deallocated heap storage.

There exists a vast literature about shape analysis. The ideas are explained in [Wilhelm et al. 2000], the theory in [Sagiv et al. 1999]. A more technical paper, which is mainly focussed on TVLA and its use, is [Lev-Ami and Sagiv 2000]. Furthermore, there are two treatises [Sagiv et al. 2002; Wilhelm et al. 2002b] that cover most aspects about and around this approach in a detailed manner. The method is precise enough to automatically prove partial correctness of algorithms, this is demonstrated in [Lev-Ami et al. 2000] for sorting programs that use singly linked lists.

```
// data structure for tree elements
type tree =
   record
      data:  integer
      left:  pointer to tree
      right: pointer to tree
   endrecord
```

In the following, we explain how shape analysis works and what shape descriptions (the descriptions of the heap states) look like. We remain at an informal level. Throughout this

Predicate	Meaning
$root(v)$	Does *root* points to v?
$x(v)$	Does x points to v?
$r[root](v)$	Is v reachable from *root*?
$r[x](v)$	Is v reachable from x?
$ancest[root](v)$	Is v an ancestor of *root*?
$ancest[x](v)$	Is v an ancestor of x?
$left(v_1, v_2)$	Is $v_1.left = v_2$?
$right(v_1, v_2)$	Is $v_1.right = v_2$?

Table 1: Predicates for heap description

paper, we will exemplify all ideas considering the search for an element in a binary search tree. This is our running example. The data structure for a tree element is shown above. Each heap cell consists of a data value and two pointers. The search algorithm is shown below in pseudo-code. Each program line is preceded by a "line number", we will refer to this as program point.

```
n0   // tree search
n1   x := root
n2   while (x ≠ NULL and x.data ≠ el.data) do
n3      if (el.data < x.data)
n4         x := x.left
n5      else
n6         x := x.right
n7   od
```

We search for an element with value *el.data* in our tree. We have a variable *root* pointing to the root node of the tree, and we have one additional variable x, which is our current element for comparison with *el*. In the while loop, we traverse the tree with x from the root along a path in the direction towards a leaf. Let us assume in the following that we have performed some iterations, say at most three.

We start with composing the "vocabulary" for the analysis. We describe heap properties logically using predicates. Their arguments are heap cells. TVLA supports predicates of arity at most two. Unary predicates express properties of a heap cell, binary predicates express a relation between two heap cells, and 0-adic predicates express properties for the whole structure. In our example, we will not use 0-adic predicates.

At first, we need to express pointer values with predicates. We introduce two binary predicates *left* and *right*. We have $left(v_1, v_2) = true$ iff pointer component *left* from heap cell v_1 points to v_2. The predicate *right* is defined analogously. We need one predicate for every pointer variable in order to know to which cells the variables points to. We have $root(v) = true$ iff pointer variable *root* points to heap cell v. The predicate $x(v)$ is defined analogously. These four predicates describe the most basic properties of the tree structure. They express the pointer relation of (pointer) variables and heap cells and are therefore contained in any specification.

Besides, we introduce some auxiliary predicates. It is convenient to have predicates expressing reachability. We define for every pointer variable a reachability predicate $r[root](v)$ and $r[x](v)$ which is true for all cells that can be reached from the variables *root* and x respectively through left and right pointers. In our example, the former one is always true (for tree nodes), because every node in a tree is reachable from the root node. We further introduce for every variable ancestor predicates $ancest[root](v)$ and $ancest[x](v)$. The latter is true for all strict ancestors of x, that is, for all cells

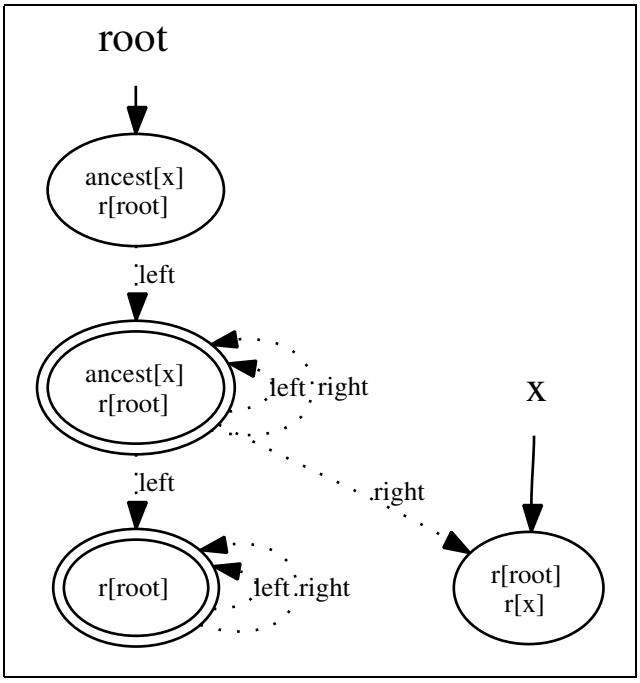

Figure 1: A sample shape graph

different from x lying on the path from the root to x. The former predicate is in our example always false because the root node has no ancestor cells in the beginning and the heap structure is not modified during the search.

For reasons of simplicity and lucidity, we do not introduce more predicates. Even though we are talking about search trees, we omit all predicates expressing relative order information of data elements. Table 1 summarises the predicates for reference. Now we have characterised the heap cells with the predicates mentioned above. But we are left with finding a representation of the heap of finite size. The idea is to partition the heap cells in equivalence classes. For this reason, we declare some of the predicates to be *abstraction predicates*. We call two heap cells equivalent if they agree in the values of the abstraction predicates. Sets of equivalent cells are represented by one abstract cell. The default setting is that all unary predicates are abstraction predicates.

The resulting structures can be interpreted as labelled directed graphs, so called *shape graphs*, where nodes and edges are labelled with predicates (predicate values). Figure 1 shows an example of a shape graph that arises during the search procedure. Its nodes are the classes of heap cells. Two types of nodes are distinguished. Classes with just one single heap cell are individual nodes, classes with more than one cell are summary nodes. Both are shown as ellipses, the latter one with a double-line boundary. The values of the unary predicates are shown inside the ellipses. If just the name of a predicate is given, then its value is true. If a predicate name is absent, then its value is false.

Note: The figures in this paper are generated directly from the TVLA output. In particular, its graph drawing algorithm is not specialised for trees. This is not the way we intend to actually draw shape graphs in our visualisation, see the comments in Section 5.

The path from the root to x consists of three nodes in the

shape graph, the individual nodes v_r and v_x pointed to by *root* and x respectively, and the summary node s between them. The left child of the root is contained in the heap cells represented by s, but not all cells from s are left children of the root. Because we never want to report anything wrong, the value $left(v_r, s)$ can be neither true nor false. We need to introduce a third truth value 1/2 with the meaning "don't know". For binary predicates, this truth value is shown with a dashed line.

Besides the predicate set, we need to specify for every statement in the algorithm an action for TVLA. It describes how the statement affects the predicate values, how the heap descriptions (shape graphs) are transformed by the statement. This task has only to be done once per data structure. Furthermore, we need to specify shape graphs representing initial heap structures for the algorithm. TVLA starts with this initial shape graphs and successively applies actions according to the control flow until a fixed point is reached. At the end of the computation, we get for every program point a set of shape graphs. The shape graphs in one set conservatively describe all possible heap configurations for this program point.

There are TVLA actions that need to substitute a single shape graph with a set of (more precise) shape graphs representing the same heap structure. In our example, let us assume that we continue the search from node x to its right child. A pointer variable can only point to an individual node, but not to a summary node. TVLA needs to "materialise" an individual node from a summary node. Several cases can occur. The node x may have no right child. If it has, then this subtree can either be a single node or a bigger subtree. In both cases, x may or may not have a left child. These cases are distinguished using different shape graphs.

3 Visualisation with Shape Analysis

A first step in using this approach to visualisation is to specify a shape analysis. We have to find and formulate meaningful predicates, and we have to formulate predicate updates as TVLA actions. Additionally, we have to specify initial shape graphs.

The time and space requirements for the analysis depend on the number of structures produced and their sizes, which is mainly the number of nodes. The number of nodes of a shape graph is controlled by the set of abstraction predicates, in the worst case it is exponential in the number of unary abstraction predicates. In general, if we allow a larger number of nodes in a graph, the more structures will be possible and likely to appear during an analysis. In terms of time and space, shape analysis is more efficient if shape graphs have only few nodes. The extreme would be to have just one or two nodes. In this case we have to express all relevant properties in relations between heap cells using binary predicates (or using 0-adic ones). Depending on the intended use of the shape graphs, this may be viable. On the other hand, shape graphs with a larger number of nodes are normally more precise, they represent a more specific heap state.

For visualisation, our first priority should lie in the resulting explanatory quality. For this purpose, it will be of great benefit if a shape graph looks somehow similar to the heap structure it represents. If we are dealing with trees, then the shape graphs should look like trees. Thus in most cases, the resulting explanatory quality will increase if we do not restrict the number of nodes to a minimum.

Let us have a closer look at our running example, the search for an element in a binary search tree. We start a search at the root and traverse a path p in the direction towards a leaf. Let us assume, we have performed some iterations and our current element x is an interior node. An intuitive view suggests that certain tree parts (not necessarily subgraphs) should also appear as separate elements (nodes) in the shape graphs. First, we have two individual nodes for the root and the current tree node x. They are distinguishable from all other tree elements because variables are pointing to them. The path p between them contains all nodes visited so far, it encodes the history of the execution. Therefore it should be represented differently from tree parts not or not yet visited. Along p, subtrees may hang off. While traversing the path, we passed by, but have not visited them. Below x are the not yet explored subtrees of the children of x. They contain the future of the algorithms execution and should also be represented separately. Altogether, we discovered five tree parts that should be distinguishable in each shape graph for the sake of explanatory quality. Compare the shape graph of Figure 1. We have chosen the predicates introduced in Section 2 according to these requirements.

We have argued that for the sake of visualisation quality, we want shape graphs with a meaningful number of nodes (distinguished heap regions). But this means that the shape analysis will produce (a great) many shape graphs. They are given to us as sets for each program point. In the search example, there are more than 200 shape graphs per program point of the while loop. There are several cases of how the path from the root to node x can look like: x can point to the root, x can point to a child of the root, there can be exactly one node between the root and x, and there can be a summary node between them. In every case, there may be a summary node below x, and there may be a summary node for the subtrees hanging off the path. In all cases, pointers may be either left or right. Note that we chose an analysis specification with only few predicates. A more detailed specification or a more complicated algorithm will for sure lead to a much higher number of resulting shape graphs. This is in contrast to (singly linked) lists, where the number of shape graphs is small enough to be visualised one by one.

For more complex data structures, like trees, we therefore need methods for handling big sets of shape graphs. We will tackle this problem by imposing an additional structure on the set of shape graphs. The methods presented below are applicable and useful in general. But especially for binary trees they lead to pretty satisfactory explanation results.

We introduce our approach with some intuitive observations. Shape graphs at the same program point are often related. They may be "similar", representing heap states that do "not differ too much". This suggests that the execution paths leading to them are "similar" too. In this case we may not wish to distinguish them and, instead, choose to visualise them as a unity. To show the idea of an algorithm, the "general" cases are usually more meaningful than the "special" ones. If we consider a loop, for example, we will often have shape graphs only occurring during initial or final iterations, while the intermediate ones represent the more general loop cases. In the subsections below, we will exploit this kind of information to structure the shape graph

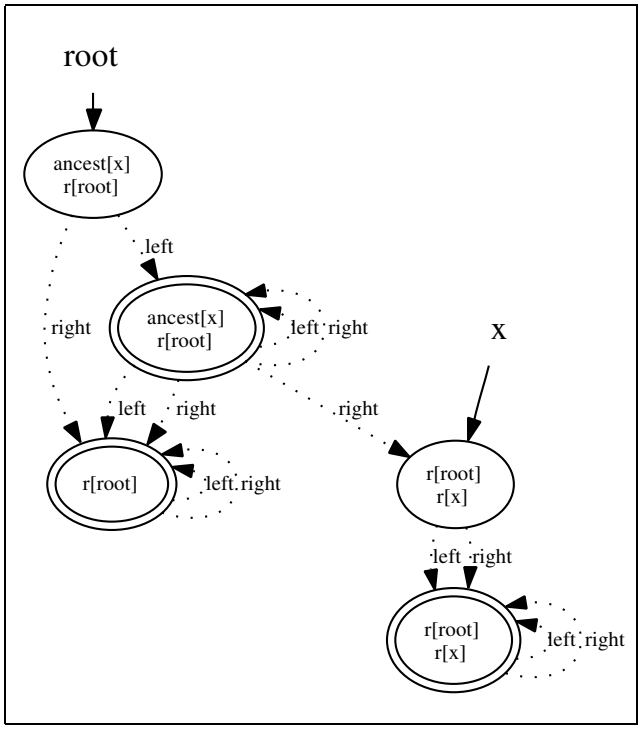

Figure 2: A maximal shape graph

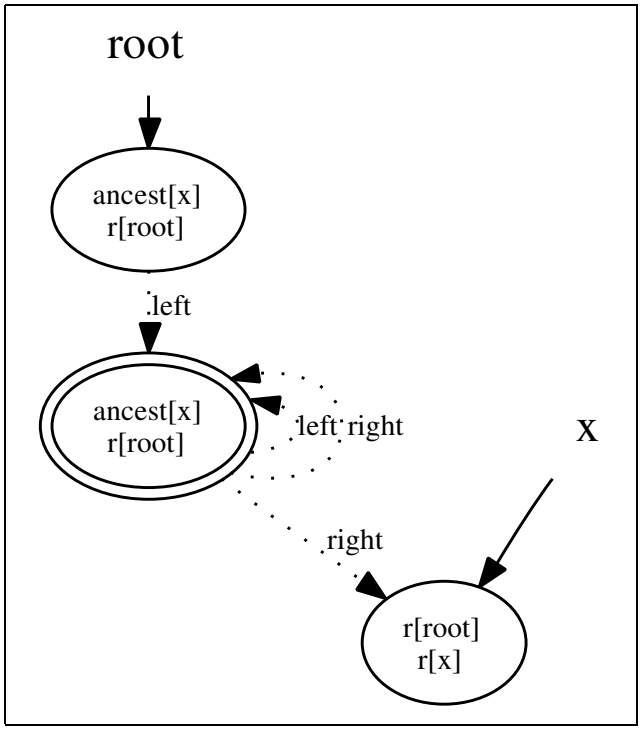

Figure 3: A path configuration

sets and to derive additional information advantageous for visualisation.

3.1 Partitioning Shape Graph Sets

In this subsection we address the problem of giving the set of shape graphs at some program point an additional structure, exploitable for visualisation.

The situation is as follows: We have carried out a shape analysis using TVLA and obtained for every program point a set of shape graphs. When the visualisation reaches a program point n, we have to show the information given by the shape graphs at this program point. In reaching this goal, it will be of great help if we can impose a structure on the set of shape graphs associated with n.

Let us consider the search example. We start the search with the root node and traverse the tree along a path leading to a leaf. Figures 1 and 2 show two shape graphs for this situation. They have the same structure of the path leading from the root to the current element x. They differ only in the subtrees which hang off (or not) along this path. Because the path encodes the history of the execution so far, one may wish to put emphasis on path structures and collect all shape graphs with the same path structure in one class.

In the following, we introduce a method for partitioning sets of shape graphs into classes according to a common substructure (subgraph). This substructure is defined in terms of predicates used for the shape analysis specification. Let D be a subset of this predicates. TVLA supports predicates of arity at most 2. Therefore, it is often convenient to interpret D as a (disjoint) union of its 0-adic, unary, and binary predicates, $D = D_0 \cup D_1 \cup D_2$.

Definition 1 *Let D be a subset of the predicates used for specifying the analysis. Let S be a shape graph. The* substructure of S induced by D *is defined as follows: It consists of firstly, all 0-adic predicates from D, secondly, all nodes v of S such that there exists an unary predicate $p \in D$ with $p(v) \geq 1/2$, and thirdly, all binary predicates from D restricted to the new node set. We denote this substructure by S_D.*

As usual, we can interpret S_D as a graph. It consists of all nodes of S for which at least one unary predicate from D_1 has non-zero value, its edges are the edges from S with labels from D_2. Besides, it has a set D_0 of unary predicates attached. Thus, S_D is simply a subgraph of S.

Definition 2 *Let D be a subset of the predicates used for specifying the analysis. Let S and T be two shape graphs belonging to the same program point. We call S and T* equivalent according to D *if the substructures induced by D are equal.*

This relation is an equivalence relation. The shape graphs at each program point are partitioned into (disjoint) equivalence classes according to D. Each class consists of pairwise equivalent shape graphs, thus all shape graphs in one class share the same substructure. Note that every subset D of the predicates defines a partition.

Let S and T be the shape graphs of Figure 1 and 2 respectively. We set $D = \{root, x, ancest[x]\} \cup \{left, right\}$. Both, S_D and T_D are the graph shown in Figure 3. Thus, S and T are equivalent according to D. If $D = \emptyset$, then all shape graphs are collected in one class. If on the other hand D consists of all predicates, then $S = S_D$ for every shape graph S. In this case every class contains just a single shape graph. This is equivalent to not partitioning the set of shape graphs.

It is natural to ask how a partition changes if we add predicates to D or remove some. To this end we introduce the following notion.

Definition 3 *Let $\mathcal{P} = \{C_1, \ldots, C_m\}$, $\mathcal{P}' = \{C'_1, \ldots, C'_n\}$ be two partitions of the set of shape graphs at some program point. We call $\mathcal{P}'$ a* refinement *of $\mathcal{P}$ if for each $C'_j \in \mathcal{P}'$ there exists $C_i \in \mathcal{P}$ with $C'_j \subseteq C_i$. In this case $\mathcal{P}$ is alternatively called a* coarsening *of $\mathcal{P}'$.*

We are not interested in all possible partitions. We only consider partitions that arise by a defining subset of predicates. So, what does refinement for partitions mean for the defining predicate sets?

Theorem 1 *Let D and D' be subsets of the predicates used for specifying the analysis, and let $\mathcal{P}(D)$ and $\mathcal{P}(D')$ be the partitions of the set of shape graphs at some program point according to D and D' respectively. If $D \subseteq D'$, then $\mathcal{P}(D')$ is a refinement of $\mathcal{P}(D)$.*

Proof: We prove the proposition for the case $D' = D \cup \{p\}$, where $p \notin D$ is a single predicate. The general case $D \subseteq D'$ follows from iteratively applying this argument. We present the proof for the case that p is an unary predicate. The cases that p is a 0-adic or a binary predicate are similar.

Let C' be a class in the partition $\mathcal{P}(D')$. All graphs in C' have some substructure S' in common. It consists of those nodes where for all shape graphs from C' at least one unary predicate has a value greater or equal to 1/2. Let V_p denote the set of those nodes from S' with $p(v) \geq 1/2$ and $q(v) = 0$ for all unary predicates $q \in D$. We remove all nodes from V_p and all its incident edges from S'. We call the resulting substructure S. This is a substructure according to D. Every shape graph from C' also has S as a substructure. Hence all shape graphs in C' are equivalent according to D, they belong to the same class of the partition $\mathcal{P}(D)$. □

Note that the converse is not necessarily true. Only a slightly weaker proposition holds. (We omit the proof.)

Theorem 2 *Let D, D', $\mathcal{P}(D)$, and $\mathcal{P}(D')$ be as above. If $\mathcal{P}(D')$ is a refinement of $\mathcal{P}(D)$, then there exists a predicate subset D'' with $\mathcal{P}(D'') = \mathcal{P}(D')$ and $D \subseteq D''$.*

We have introduced the concept of partitioning the set of shape graphs at some program point. We define a partition using a subset D of the predicates. The size of the partition, that is the number of classes, is controlled by D. Every class consists of shape graphs with a common substructure derived from D. We will interpret shape graphs in the same class as equivalent, as not significantly different. Therefore, we are also going to visualise a class as a whole. The number of classes expresses the visualisation complexity.

Changing the subset D means to control the amount of detail used for visualisation. This feature allows us to adjust the level of detail to the knowledge of the user. If we want to visualise an algorithm to someone unfamiliar with it, he may in the beginning prefer a presentation with not too many details, that is a coarse partition, while an experienced user may prefer a more detailed view, that is a fine partition. We can even change the partition during a single visualisation, showing parts of the algorithm with different levels of precision.

3.2 Execution Paths

In the last subsection, we were concerned with shape graphs itself. We introduced the notion of similarity between shape graphs in order to define a class partition. In this subsection, we are concerned with the relation in which shape graphs stand to each other according to their occurrence and order in execution paths. This will allow us to distinguish shape graphs representing special cases from those representing (more) general cases.

We assume that the preceding shape analysis not only produces for every program point a set of shape graphs, but a *transition graph*: Its nodes are the shape graphs, more precisely the *abstract states* (n, S), where n is a program point and S a shape graph occurring at n. There is a directed edge from (n_i, S) to (n_j, T) if shape graph S can be transformed into T by a TVLA action which leads from program point n_i to n_j. The transition graph was originally not generated by TVLA. It was introduced for the purpose of visualisation in [Bieber 2001], where it was called trace graph. Note that not all versions of TVLA support transition graph output.

At first, we show how this combines with the partitioning approach introduced in the previous subsection. Let D be a subset of the set of predicates used for specifying the shape analysis. The set D defines a partition $\mathcal{P}(D)$, the shape graphs for every program point are partitioned into classes. Let p be a path in the transition graph, traversing shape graphs in classes $C_{n_1}, C_{n_2}, \ldots, C_{n_k}$ in this order where n_i is the corresponding program point. Let q be a different path traversing the same sequence $n_1, n_2, \ldots, n_k$ of program points, and for every program point n_i the corresponding shape graph also belongs to C_{n_i}. All shape graphs in one class are treated as equal. We cannot distinguish the paths p and q, they are equal according to D. Therefore, there is no need to visualise them separately.

Definition 4 *Let D be a subset of the predicates. The* reduced transition graph TG_D according to D *is the graph that emerges from the transition graph as follows: For every class, we identify its abstract states by shrinking them to a single (super)node. There is a directed edge from node (class) C to C' if there exist abstract states $(n_i, S) \in C$ and $(n_j, T) \in C'$ such that there is an edge from (n_i, S) to (n_j, T) in the transition graph.*

The size of the graph TG_D is strongly related to the *visualisation complexity* with regard to D. The graph TG_D consists of all nodes and paths which are not equivalent, which are distinguishable, according to D. By changing the partition defining subset D, we can vary the visualisation complexity and adjust to the users needs and wishes.

Now we discuss how to find expressive execution paths. For the time being, TVLA supports only intra-procedural shape analysis. Hence we do not need to consider procedure calls. The main building blocks of algorithms are statement sequences, conditions, and loops. The first two lead to more or less straightforward execution paths. The latter is the most interesting because it comprises iterations. In the following, we will examine a loop separately. The ideas and results can then be applied to whole execution paths. For reasons of simplicity, we will assume that the loop shows "good programming style," that is, the loop has exit conditions only at the beginning and end of the loop, there are no jumps inside the loop body and so on.

The search example consists of just one loop apart from the

first statement. We will derive a structure from the transition graph which allows us to analyse the behaviour of the loop separately. The structure explicitly represents the paths through the loop body. (In our example, the loop entrance program point mentioned below is n_2.)

Definition 5 *The* loop transition graph *LTG of a loop is a directed graph. Its nodes are the shape graphs of the loop entrance program point n_e plus two additional nodes A and B. Two original shape graphs S and T are connected by a directed edge if there is a path in the transition graph from (n_e, S) to (n_e, T) that only uses abstract states belonging to (program points of) the loop. There is an edge from A to an original shape graph S if there is a path entering the loop in (n_e, S). There is an edge from an original shape graph S to B if there is a path from (n_e, S) which leaves the loop in the next iteration.*

We can distinguish shape graphs which can only occur in the beginning or final iterations of a loop (these are the neighbours of A and B) from those that appear in intermediate iterations. The former ones will intuitively in general be considered as special cases. This simple distinction is not very helpful in case of the search example. We only have a few loop entrance graphs, but we can exit the loop with every shape graph because the current element can be the one searched for. All shape graphs are loop exit graphs and it is not clear which one should belong to "intermediate iterations." Another obstacle in finding an order in which shape graphs appear on execution paths is the existence of cycles in the transition graph. But on the other hand, if we have a shape graph which in one loop iteration can be transformed into itself – we have a heap configuration which repeats –, then it can be seen as a "general" case. We call such shape graphs *stable*. But this notion is very restrictive, we weaken it by allowing several loop iterations:

Definition 6 *We call a shape graph S from LTG* semi-stable*, if S lies on a cycle. If T also is a node of this cycle, then we call S and T transformable into each other.*

If we compare transformable shape graphs with non-transformable ones, we can understand transformable ones as similar. They represent heap configurations that within some loop iterations can be transformed into each other.

Let us consider the shape graph S of Figure 2. If we continue the search by proceeding the search with the right child of x, then one of the resulting shape graphs is S again. The shape graph S is stable. If we proceed with the left child of x, then one of the resulting shape graphs T looks the same as S besides that the ingoing edge to x is labelled with *left*. If we continue in a second iteration with the *right* child, then one of the resulting shape graphs is once again S. The shape graphs S and T are transformable into each other.

The notion of semi-stable nodes can be related to the graph theoretic concept of *strongly connected components*. These are maximal subgraphs in which every node is reachable from every other node. A formal definition and an (optimal time) algorithm to compute them can be found in [Cormen et al. 2001], for example.

Theorem 3 *A shape graph from LTG is semi-stable iff it has a loop or belongs to a strongly connected component of size at least 2.*

Proof: Let S be a semi-stable shape graph. Let C be the nodes of a (directed) cycle containing S. If $C = \{S\}$, then S has a loop. Assume that C consists of more than one node. For every $A, B \in C$ there exists a directed path from A to B and from B to A. (This argument also holds for $A = B$.) Thus, the node set C belongs to a strongly connected component.

Let shape graph S belong to a strongly connected component of size 1. Then S is (semi-)stable iff S has a loop. Let us now assume that S belongs to a strongly connected component of size at least 2. Then, there is another shape graph T, $T \neq S$ in this component. There exists a (directed) path from S to T and from T to S. Connecting these two paths yields a cycle which contains S. □

We compute all strongly connected components of LTG. For each component, we identify its nodes. The graph LTG' emerges from LTG by shrinking each strongly connected component to a single (super)node. This graph is acyclic. (If it had a cycle, then the nodes from this cycle would belong to a strongly connected component.) So, the graph LTG' encodes the order in which shape graphs occur on execution paths through the loop. For practical purpose, it is beneficial to sort the nodes topologically, see [Cormen et al. 2001]. Now we can choose among several heuristics to extract appropriate executions path for visualisation. The choice will, of course, depend on the algorithm, but primarily on the visualisation aim. For example, we could prefer longer paths over shorter ones. If we are interested in paths representing "general loop cases", we should favour nodes corresponding to strongly connected components.

4 Visualising Shape Graphs

In the previous section, we introduced methods for giving the set of shape graphs at a program point an additional structure. In particular, we partitioned shape graphs into classes. All shape graphs in one class share some common property, which was expressed in terms of subgraph equality. In this section we show how this structure can be exploited to reduce the visualisation complexity.

We are going to present three methods of how to visualise a class of shape graphs. Our primary concern is to reduce the visualisation complexity of a class, that is the number of graphs to be shown. But this reduction should avoid a loss of explanatory quality. We assume that at a specific program point the set of shape graphs has been partitioned into classes according to some class defining predicate subset.

Embedded Visualisation

This method uses the approach of selecting some representative shape graphs of a class for visualisation and neglecting the others. The members of this subset will be visualised simultaneously.

At first, we look at the search example. Assume that we choose to define the class partition according to the shape of the path from the root to the current element. We will refer to this as a path configuration. The shape graph S of Figure 3 is an example of a path configuration. We denote by C the set of all shape graphs belonging to the loop entrance program point n_2 with a path configuration equal to S. Of course, S belongs to C. The shape graphs in C have or do not have subtrees along the path and have or do not have summarised subtrees as children of x. Figures 1 and 2 show

further shape graphs belonging to C. Altogether, C consists of a total of 16 shape graphs.

The shape graphs in one class do generally not describe heap configurations which are disjoint. Quite often, we have the situation, that there are shape graphs S and T such that T describes a heap configuration that includes the heap configuration given by S. In this case, S is called *embeddable* in T. We remain at this intuitive view, a formal definition can, for instance, be found in [Sagiv et al. 1999; Sagiv et al. 2002].

In our example, let S and T be the shape graphs of Figure 1 and 2 respectively. Shape graph S cannot be embedded in T. Shape graph S describes a heap configuration where exactly one heap cells is reachable from x, whereas T describes a heap configuration where at least two cells are reachable from x. Let S' emerge from S by "adding" an individual or summary node below x. This shape graph S' can be embedded in T.

We "remove" all shape graphs from C which can be embedded in another shape graph from C. In our example, the resulting set consists of only 4 shape graphs: the one of Figure 3, this graph with a summary node for the subtrees of the children of x, the graph of Figure 2, and this graph without the summary node below node x. Quite often, this method reduces the number of shape graphs remarkably.

We define this concept in a formal manner. The embedding relation is an ordering for shape graphs, so C is a partially ordered set with respect to the embedding relation $\sqsubseteq$. We call a shape graph $S \in C$ *maximal* if there is no shape graph $T \in C$, $T \neq S$, with $S \sqsubseteq T$, that is, S is not embeddable in any other shape graph of C. Note that elements of a poset may be incomparable like the shape graphs of Figure 1 and 2. Let M be the set of maximal elements of C. Every shape graph from C can be embedded in a maximal shape graph. Therefore, this set M is an appropriate choice for visualisation. Every non-maximal shape graph is visualised by a more approximative one. We still have a conservative description of all possible (concrete) heap states belonging to C, no error is introduced.

V-Node Visualisation

Let us first return to our example. All shape graphs in a class consist of the same path configuration. Along this path, subtrees may or may not hang off, and there may or may not be subtrees below node x. We cannot transfer the notion of "may" into terms of shape graphs. Shape analysis only knows two kinds of nodes, individuals, which stand for single heap cells, and summary nodes, which stand for one or more heap cells. It has no concept for zero or more heap cells (or for zero and one). We introduce this kind of concept for the purpose of visualisation. As in the subsection above, we also use the notion of an embedding $S \sqsubseteq T$, but in a generalised way. We mark a subset V_T of the nodes of T which we call *virtual nodes*. We allow that some of them (and all its incident edges) can be removed before embedding S. If we remove the node set $V_S \subseteq V_T$ and all its incident edges from T, then we denote the resulting graph by T_{V_S}.

Definition 7 *Let S be a shape graph. Let T be a shape graph with a marked subset V_T of virtual nodes. We call T a* visualisation graph for S *if there exists a subset $V_S \subseteq V_T$ of the virtual nodes such that S is embeddable in T_{V_S}. Let C be a class of shape graphs. We call T a* visualisation graph for C *if T is a visualisation graph for every $S \in C$. If additionally $\bigcup_{S \in C} V_S = V_T$ holds, then we call T* tight.

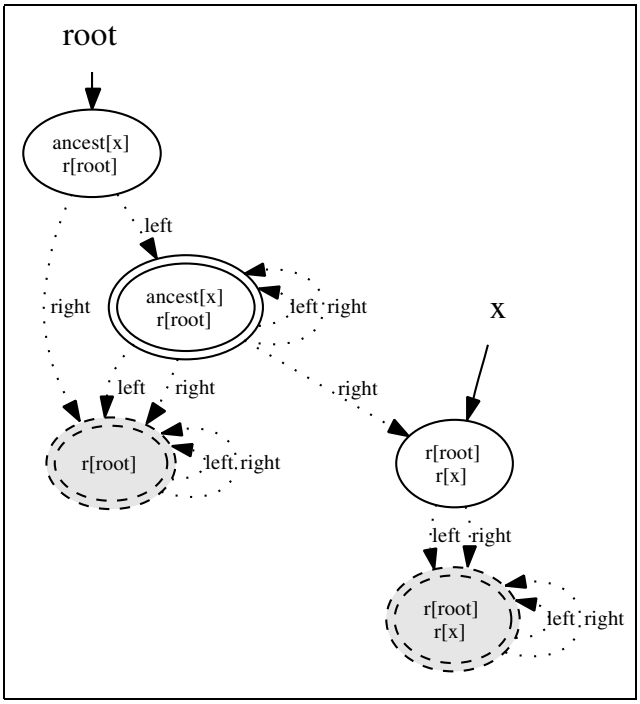

Figure 4: A visualisation graph

Note that for a given class of shape graphs, it is always possible to compute visualisation graphs. But it is non-trivial to find "good" ones. We omit the details. The graph T of Figure 4 is an example of a visualisation graph. Virtual summary nodes are depicted as light-grey ellipses with dashed double boundary lines. If we consider the class C of shape graphs with path configuration as shown in Figure 3, then T is a visualisation graph for this class. We have already seen that all shape graphs of C can be embedded in just 4 (maximal) shape graphs. To show that T is a visualisation graph for C, it is sufficient to show that T is a visualisation graph for each of these 4 graphs. Let us for example examine the graph S of Figure 1. It has no subtrees below x. So we choose V_S to be the virtual summary node below x. Removing it and all its incident edges results in a graph in which S can be embedded. If we consider the graph of Figure 3, we have to remove both virtual nodes. This shows that T is actually a tight visualisation graph.

Our example works very well with this method, the resulting visualisation graphs look quite natural. The main reason is, that we can find visualisation graphs where all virtual nodes are leaves. If we had more complex shape graphs or virtual nodes appearing as internal nodes, then the visualisation graph would also look more complex and the visualisation quality would not necessarily be that apparent.

Simultaneous Visualisation

This is the most intuitive approach. We simply take all shape graphs and visualise them in a traditional way one after another, or simultaneously. This method is best appli-

cable if the set is rather small. For example, the embedded visualisation method generally produces only a small set of shape graphs. Besides, this method has the advantage that we do not introduce new imprecision due to further approximation, which we did in the first two methods.

5 Prototype Implementation

The approach to shape analysis introduced in Section 2 was implemented by Tal Lev-Ami. The resulting program is called TVLA, see [Lev-Ami 2000] and [Lev-Ami and Sagiv 2000].

Soon after TVLA was available, work began to use shape graphs for algorithm visualisation. The program ALEXA by Ronald Bieber arose, see [Bieber 2001]. It was followed by the work of Gints Klavins, see [Klavins 2003]. The first examples, which where considered by the TVLA designers, were singly linked lists. This data structure is simple, the resulting shape graphs are small and the analysis produces only few of them. There was no need for sophisticated additional methods to handle them. The work on ALEXA was focussed mainly on how to draw shape graphs arising from singly linked lists in an appropriate manner. The problem of animating the change of shape graphs from one program point to the next was addressed too.

Now, interest has moved on to more complex data structures like trees. A prototype implementation of our methods is under development, but it is too early to report any experiences.

6 Related Work

Algorithm visualisation has been a research topic for many years, and there exists a vast amount of work in this area that could be cited here. However, we have made it sufficiently clear in the introduction that our approach is quite different from existing ones. Therefore, we deliberately do not compare our work with the existing literature.

We are only aware of one other approach that uses ideas similar to the ones presented here. Amir Michail developed a visual programming system for teaching binary trees, which he called OPSIS, see [Michail 1996a] and more detailed [Michail 1996b]. He shares with us the approach of using abstract states to represent a set of concrete binary trees sharing some property. The abstract states are identified with visual diagrams. However, his approach has no semantic foundation, like shape analysis for example. We briefly describe his approach.

He uses two concepts for elements of a tree. The first is a node which simply represents a single element. This is the same as an individual heap cell in our terminology. The other is a tree fragment, which is a (possibly empty) connected subgraph of a tree. Note that a fragment is not necessarily a subtree. Apart from the fact that fragments may be empty, they correspond in their nature to summary nodes in our terminology.

For the purpose of programming, the OPSIS system offers a set of operations. An algorithm is represented using a state graph, which in our terminology is the transition graph. The nodes are the abstract states, and two nodes are joined by a directed edge if there is an operation transforming the visual diagram corresponding to the first node into the other one. For implementing a program, one starts with an initial state and combines operations which manipulate the visual diagrams. If a diagram results that matches an already existing one, their corresponding abstract states are identified. This is the way how loops are created. OPSIS further supports a mechanism for handling similar cases to some degree automatically, see [Michail 1998].

Michail also points out the connection to proofs. He views the resulting programs as being close to correctness proofs. Let us consider a loop, for example. An abstract state, a visual diagram, is through the operations of the loop body transformed into the same visual diagram, into the same abstract state. The sequence of visual diagrams shows that the visual diagram is maintained as a loop invariant, the sequence supports in a visual way a structural induction argument. But the absence of a semantic foundation makes this relation between algorithm and proof slightly vague. A practical drawback is, that loop invariants have to be found and formulated by the programmer. User experiments mentioned in [Michail 1996b] indicate that this task is difficult.

7 Conclusion and Future Work

We discuss a non-traditional approach to algorithm visualisation (animation). In contrast to traditional approaches, our point of origin is the visualisation of the abstract execution of an algorithm. Algorithms under consideration are imperative pointer-based algorithms. We need an abstract representation of the heap and invariants for every program point. They are computed using the shape analysis implementation TVLA as a preprocessing step. This results in a logical description of the heap states in terms of shape graphs. We use these shape graphs as a basis for visualising the abstract execution. We believe that this approach is very suitable for algorithm explanation, especially for the purpose of teaching.

Shape analysis for algorithms using more complex data structures than lists results in a large output size, that is, in many shape graphs. We propose several methods for finding those shape graphs that are most important with respect to the visualisation aim, namely teaching the idea of an algorithm. We introduce the notion of similarity for shape graphs, so that visualisation does not need to distinguish these cases. This concept also allows to adjust the level of detail. Moreover, we explain how to use transitional properties for further structuring the set of shape graphs and for finding expressive execution paths. Thereafter, we discuss methods for visualising classes of shape graphs.

Finally, we like to mention some problems which appear to be worth for future examination. We define class partitions according to a subset of the predicates used for specifying the analysis. This may be too restrictive for an experienced user who wants to use the partitioning approach for getting deeper insights in the behaviour of an algorithm. In order to gain more flexibility, we could specify partitions by (or in combination with) a set of formulae over the predicates.

We did not address the problem of how to actually draw single shape graphs. This involves the subproblems of finding good positions for nodes and routings for edges to start with.

This issue must not be neglected. The quality of the visualisation will depend to a great extent on the quality of the drawing. Furthermore, our approach to visualisation is not restricted to put visual emphasis on those binary predicates that represent pointers. For (singly linked) lists, it is discussed in [Wilhelm et al. 2002a] how to visualise uncertainty and non-structural properties like sortedness. Transferring these ideas to binary trees will lead to views different from those shown in our figures. In this setting it is not obvious at all how to tackle the drawing problem satisfactorily.

References

Bieber, R. 2001. *Alexa – Algorithm explanation by Shape Analysis. Extensions to the TVLA system.* Master's thesis, Universität des Saarlandes, Germany.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms*, 2nd ed. The MIT Press, Cambridge.

Kerren, A., and Stasko, J. T. 2001. Algorithm Animation – Introduction. In *Software Visualization*, 1–15.

Klavins, G. 2003. *Algorithm Visualization Using Shape Analysis.* Master's thesis, Universität des Saarlandes, Germany.

Lev-Ami, T., and Sagiv, S. 2000. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*, 280–301.

Lev-Ami, T., Reps, T. W., Sagiv, S., and Wilhelm, R. 2000. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 26–38.

Lev-Ami, T. 2000. *TVLA: A Framework for Kleene-Based Static Analysis.* Master's thesis, Tel-Aviv University, Israel.

Michail, A. 1996. Teaching binary tree algorithms through visual programming. In *Proceedings of the IEEE Symposium on Visual Languages*, IEEE Computer Society Press, Washington, 38–45.

Michail, A. 1996. Teaching binary tree algorithms through visual programming. Tech. rep., University of Washington.

Michail, A. 1998. Imitation: An alternative to generalization in programming by demonstration systems. Tech. Rep. UW-CSE-98-08-06, University of Washington.

Price, B. A., Baecker, R. M., and Small, I. S. 1993. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing 4*, 3, 211–266.

Sagiv, S., Reps, T. W., and Wilhelm, R. 1999. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages*, 105–118.

Sagiv, S., Reps, T. W., and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACP Transactions on Programming Languages and Systems 24*, 3, 217–298.

Wilhelm, R., Sagiv, S., and Reps, T. W. 2000. Shape analysis. In *Computational Complexity*, 1–17.

Wilhelm, R., Müldner, T., and Seidel, R. 2002. Algorithm explanation: Visualizing abstract states and invariants. In *Software Visualization*, Springer Verlag, vol. 2269 of *LNCS*, 381–394.

Wilhelm, R., Reps, T. W., and Sagiv, S. 2002. Shape analysis and applications. In *The Compiler Design Handbook.* CRC Press, 175–218.