

Encoding Program Executions

Steven P. Reiss and Manos Renieris

Brown University

Computer Science Department

Providence RI 02912

USA

+1 401 863 7600

{spr,er}@cs.brown.edu

ABSTRACT

Dynamic analysis is based on collecting data as the program runs. However, raw traces tend to be too voluminous and too unstructured to be used directly for visualization and understanding. We address this problem in two phases: the first phase selects subsets of the data and then compacts it, while the second phase encodes the data in an attempt to infer its structure. Our major compaction/selection techniques include gprof-style N-depth call sequences, selection based on class, compaction based on time intervals, and encoding the whole execution as a directed acyclic graph. Our structure inference techniques include run-length encoding, context-free grammar encoding, and the building of finite state automata.

KEYWORDS

Software understand, Program tracing, Dynamic program analysis

1 MOTIVATION

Software understanding requires inferring the behavior of software systems. Static techniques do this by analyzing the program's code, while dynamic techniques are based on running the program, collecting data as it runs, and then analyzing the resultant data. In the dynamic case, there are inherent tradeoffs concerning the amount of collected data, the types of analyses that can be performed on them, and the overhead of collection.

We regard frameworks suitable for dynamic analysis as having three parts. The bottom layer, closest to the program, is a suite of tools for gathering information as a program executes. The middle layer selects relevant portions of the data, which it later compacts and analyzes, effectively building a model of it. The top layer, closest to the user, displays these models. We are in the process of building such a framework. The first part is described in [17], but we also briefly review it in section 2. The bulk of this paper discusses our

middle layer tools. We envision the top layer as having at least the capabilities of our previous tools [15], but with enhanced abilities for specifying meaningful visualizations "on demand", as the task arises.

The purpose of the middle layer is to build models of the collected data. There are two reasons to build a model: first, it is impossible to visualize the raw data, because of the volume and complexity of it, and second, a model can be used to automatically check properties of the traces. We want our models to be useful in at least the following tasks:

- Summarizing parts of a program execution. Such summaries would allow us to show the execution at multiple levels of detail, helping the user to navigate through it.
- Modeling class behavior. We can track the outside calls to methods invoked on all objects of a specific class. If we have a suite of programs that use the class correctly, we can use the abstractions generated by the execution analysis as a model of how the class should be used.
- Modeling library usage. We can view a whole library as an object and track all external calls to it. This would allow us to build a model of correct library usage and test future users of the library against the model.
- Detecting unusual events. After we have a model for the program behavior, we should be able to find events that do not conform to that model. Such events are often significant in that they represent exceptional conditions. For example, if a program executes one sequence of synchronization steps almost all the time, but occasionally executes an alternate sequence, this denotes a potential problem.
- Checking against existing models. If we already have a model of aspects of the behavior of a system, say, from the specifications process, then we should be able to efficiently check whether the model produced by the traces is conforming to the original one. We also see here the opportunity of building models of design patterns and then verifying that a program implements them correctly. This is especially useful for behavioral patterns [8], which are hard to specify otherwise [16].

- Performance analysis. We want to be able to annotate our models with performance information. This is what existing performance tools are doing, except that they are using a fixed and limited model. Prof is using a simple function invocation model, and gprof [9] a digram model.
- Visualization. We want our models to be fit for further exploration by graphical means. In a visualization environment, we leave it up to the user to separate “data” from “noise”. Of course, what is noise for a particular application is useful for another; therefore it is essential that our models are rich enough to at least hint to what the user should focus on. On the other hand, we cannot include everything in a model, because of volume.

The work we describe in this paper provides the framework necessary for addressing these and other problems using trace data. The important aspects of this research are

1. A framework that separates the collection, analysis and encoding of data. The framework provides the flexibility needed to address the wide range of problems such as those described above.
2. Specific data selection techniques that address the above problems.
3. Specific encoding techniques that facilitate different types of analysis when combined with the various selection techniques.

The overall significance of the research thus lies in the specific techniques that are used, in the ability to combine them in various ways, and in the application of their combination to a broad range of problems.

The remainder of the paper is organized as follows: Section 2 describes the data that our trace tools currently generate. The different data compaction and selection approaches are described in Section 3. Section 4 describes the different sequence encoding techniques that we utilize in these approaches. Note that each of these techniques can be used with many of the approaches described in Section 3. Section 5 then discusses the results of these various encodings, commenting on their degree of compaction and their accuracy. We conclude with a discussion of the impact of this work and what extensions we are currently planning.

2 TRACE DATA

The first part of our framework collects data from a running program. We have a suite of tools that traces both binary executables and Java programs. The binary tracers are tuned for C and C++, but there is nothing that would prohibit them from working with executables produced from, say, a Pascal or Java native compiler. Table 1 summarizes the tracing data.

Our tracer for binaries, WOLF, patches the executable, inserting hooks at the entry and exit of each routine. It also instruments basic blocks to keep count of how many instructions are executed per function invocation. The entry and exit hooks are used to generate a trace file for each thread of the program. For each call a record is emitted including the called address, the calling address, and the first argument of the call. The latter is used to determine the object used for method calls, the size for memory allocation, the thread for thread-related activities, and the synchronization object for a synchronization call. For each return, the record includes the address of the function that was called and the return value. In addition, the trace files contain periodic records recording the thread run time, the real time clock, and the number of instructions executed by the thread. The records of each thread are output on a separate file. We have a second tracer under development, which is doing minimal work, emitting only the called address and first argument for function calls, and piggybacking the count of intervening function returns on the next function call.

For Java programs, we have a different tool, TMON [17], that uses the standard Java profiling interface (JVMPPI) to again produce multiple trace files, one per thread. These files contain records for each method entry and exit, for each object allocation and deallocation, for thread creation and termination, and for monitor (synchronization) activity. The trace files also contain records about thread run time and real time.

The reason that we use one trace file per thread in both cases is that this avoids synchronizing calls that would otherwise not be synchronized. To get the intermingling effect back, we use another program, TMERGE, which outputs a unified trace file, and a dictionary file which maps identifiers (like function addresses for binaries, or object identifiers for Java) in the trace file to meaningful symbols.

Even though the data we are recording is fairly high-level, the amount of data for a large or long-running system can be immense. The resultant trace data files contain about one gigabyte of data for every two seconds of C/C++ execution or ten seconds of JITed Java execution (the minimal tracer reduces this by a factor of 2.5). This imposes certain requirements on the efficiency of the tools that have to deal with the files. Primarily, however, it creates the need for the compaction and selection techniques that we describe in the next section.

3 DATA COMPACTION AND SELECTION

Our data compaction and selection techniques range from lossless transformations to aggregation. Our trace data is basically the dynamic call tree. This tree includes a node for each routine invocation and an edge from each invocation to all the invocations that it directly generates. The tree can be augmented with time information (like run time, real time, and instructions spent in each call) and memory usage infor-

1. Common to Java and C/C++
 - Function Entry: Function Id; First argument (C) Object Id (Java)
 - Function Exit: Function Id
 - Object Allocate/Free: Object Id
 - Thread Start/End
 - Run Time: Execution time in thread
 - Real Time: Accumulated real time
2. C/C++ only
 - Lock (any pthread synchronization object) Create/Destroy/Wait/Test/Unlock: Lock Id
 - Memory Allocate/Free: Address; Size
 - Instruction Count: Instructions executed in thread
3. Java Only
 - Class Load/UnLoad: Class Id
 - GC Start/Stop: Number of non-garbage objects; Total non-garbage object space; Total object space
 - Monitor Enter/Exit: Object Id
 - Monitor Wait: Object Id; Timeout

Table 1: Trace Record Summary

mation (like the size and number of allocations done by each call).

In the following discussion of compaction/selection schemes, we will be talking about encoding sequences of calls and other events with strings and combining them using various encodings. The degree of compaction depends largely on which particular structures and operations we choose for that purpose. For example, we can represent a sequence of calls as a string or as a run length encoded string; but we can also represent the sequence as an automaton, and then combine nodes that would not be collapsed for strings, by combining their automata, by, for example, taking their union. Our different approaches to this are described in section 4. The effectiveness of the combination of compaction and choice of representation is described in section 5.

String Compaction

The simplest way of representing the dynamic call tree is with a string. For example, if function A calls function B and then function C, we could represent it as ABC. This is of course ambiguous, since the same string would result from A calling B and B calling C without returning. We need therefore to insert markers for return, and then the two cases become as follows: AB \vee C \vee and ABC $\vee\vee$. Note that each thread of control generates its own string. Thus the output of a string encoding of the execution tree is a sequence of encoded strings, one per thread. For Java where we have thread

names, we allow threads with similar names to be grouped for encoding purposes. Once we have those strings we can apply on them the techniques of section 4.

Class Selection

For some applications, like modeling the behavior of a class, parts of the trace are irrelevant. We are interested in what constitutes a typical use of a class; for this, we need a number of applications that use the class correctly. We can trace these applications, and isolate the calls of methods of a particular class (or class hierarchy) and group those by object. If we are interested into the inner workings of the class we can monitor the methods that an object invokes on itself; if we are only interested in the external interface we can forego them. For the grouping of calls per object, we need to track the first argument of all calls, which for methods is the ‘this’ pointer (this is largely system dependent behavior, but seems to be the norm). The pointer becomes interesting when it’s the first argument of a constructor, and becomes uninteresting again when it’s the first argument of a destructor.

Using a similar approach we can provide models of the usage of a library, package, or other entity. While this is easy in principle, the difficulty in practice comes at attempting to define what is meant by a single use of a given library or entity. The trick we use for objects does not work in the general case. As an example, consider the C library file interface. A file is accessible to the program through a pointer to a FILE data structure. A specific address only becomes interesting when returned by a function (fopen). Also sometimes it is the first argument to a function (fprintf, fscanf) and sometimes the fourth (fread, fwrite). What we need is to somehow specify that the return value of fopen is linked to the first argument of fprintf and the fourth argument of fread. On top of the specification difficulty, it is harder to trace the program, since we might need to emit records with all the arguments. One approach is to interpose a library that does exactly this extra step of selecting arguments.

N-level Call Compaction

The standard UNIX prof tool provides use useful performance data by grouping all the calls to a single routine into a single node and collecting statistics for that node. Gprof accumulates statistics on calling pairs, but even that has been found inadequate [20]. Using the trace data, we can provide a simple n-level generalization of this.

To create an N-level call encoding we look at the call stack at the start of each call. We then create for $1 \leq i \leq n$, the i-tuple that includes this call and the (i-1) top items that are on the call stack at that point. For each such tuple, we accumulate the statistics that are collected from the trace data, computing both the sums for averaging and the sum of squares for computing the standard deviation.

Interval Compaction

Another way of compacting the data is to consider the program trace in chunks. We break the execution into a small

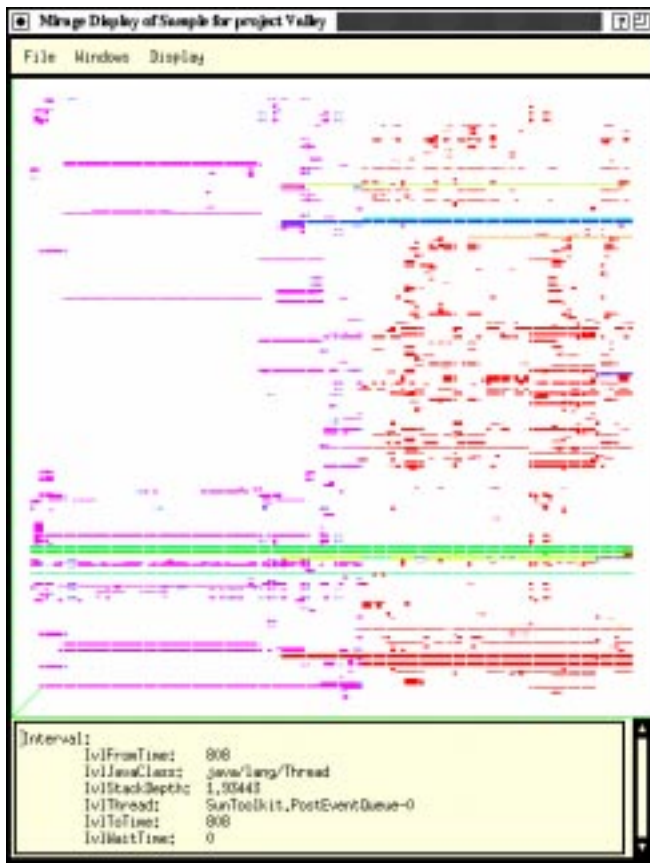


Figure 1: Interval Visualization

set of intervals (for example 1024), and then do a simple analysis within each of the intervals to highlight what the system is doing at that point. Currently, we define “what the system is doing” in two distinct ways, providing two different analyses for the intervals. The first one concentrates on the calls during the interval, combining them per class. For multithreaded applications, it also keeps track of how much time is spent waiting within a class. Using this model we can produce an overview visualization such as that shown in Figure 1.

The second interval analysis looks at allocations on a class basis. For each interval it records the number and size of allocations of objects of each class by each thread. This again can be used to provide a high level visualization of the allocation behavior of the system or it can be combined with the call interval analysis to provide additional details of the behavior of the system.

Dag Compaction

One effective way to compact the dynamic call tree is to transform it into a directed acyclic graph (DAG). The dag is built by traversing the tree in a postorder fashion and collapsing nodes with identical strings of calls. Every such string appears exactly once in the dag.

The algorithm to compact the tree is as follows:

1. In a post-order fashion, for all nodes of the tree: Construct a string consisting of
 - the function of the node,
 - an opening parenthesis,
 - the encoding of the strings of all its children, maintaining their order,
 - a closing parenthesis.
2. Compute a signature of that string, by hashing or otherwise.
3. If there is a node in the dag node containing the resulting string do nothing.
4. Otherwise, create such a node, and create an arc from it to each of the nodes containing the tree node’s children’s strings, retaining the order.

Once again we don’t have to keep strings as strings; we can use some other representation of a sequence, if we have an operation to replace concatenation, and some way of defining unambiguously the “head” of a sequence (what we do here with parentheses).

While building the dag, we need to merge the performance statistics that are associated with the individual nodes. We do this by keeping for each dag node the count of the number of original nodes, the sums of each statistic from each of the original nodes, and the sum of the squares of each statistic for each of the original nodes. This lets us provide averages as well as standard deviations for each dag node.

The compaction is generally quite effective since much of an execution is repetitive in nature and the dag allows such repetitions to be collapsed into a single node.

When dealing with multiple threads, we start with the forest of their call trees, and we apply the same algorithm to all the trees simultaneously, but without starting with an empty dag every time. That is, we construct a single dag that represents all threads of control. The individual threads are then represented as root nodes of the resultant graph.

4 SEQUENCE ENCODINGS

The effectiveness of many of the compaction techniques relies on the representation of sequences of items. There are two approaches to such representations. The first is to provide an exact representation of a sequence, possibly compressed. The second is to provide a “lossy” representation, one that represents not only the original sequence but other sequences, too. One could, for example, detect repetitions in the sequence and then ignore the counts of them, so that, say, AAAABABABC would become “some A’s, some BA’s, and a BC”, or, in regular expression notation $A^*(BA)^*BC$.

This kind of approximation of sequences is particularly useful when one is attempting to identify similarities or, as in our case, when one is attempting to encode a group of related sequences using a single encoding. Several of the compaction techniques of the previous section can use this approach. For example, when compacting the tree into a dag, we can make nodes in the dag describe whole families of call sequences. As an example take the case that function A calls function B in a loop. Under the classical string representation, it would be mapped to a different dag node depending on the number of iterations through the loop. Under, say, a run length encoding scheme that ignores counts, it would be mapped to the same node. In such cases an approximation is better for understanding and visualization.

Our framework allows for a variety of sequence encoding techniques. These include both approximations and exact encodings. They vary from very simple to more complex. The correct one to use will depend on the particular compaction/selection scheme that is being used and the specific understanding task.

Run-Length Encoding

The simplest approach that we provide (other than no encoding) is to find immediate repetitions and replace them with a count followed by the item being repeated. Thus, the string ABBBBBBBBBCDDDBCDC is encoded as A 9:B C 3:D BCDC. This is very fast and often quite effective in terms of compression. The run-length encoding algorithm also takes a parameter k indicating the longest repetition to be expressed exactly. Any repetition of size longer than k will look the same. Thus, if $k = 3$, the above sequence would be encoded as A *:B C 3:D BCDC.

Grammar-Based Encoding

An alternative to simple run-length encoding that finds immediate repetitions, it to find all common subsequences and to encode the string using a grammar where each such subsequence is represented by a rule. This is a natural extension to the RLE encoding. One such approach is the Sequitur algorithm [14]. This algorithm builds a context-free grammar representing the original string. It ensures that:

- a) No pair of adjacent symbols appear more than once in the grammar; and
- b) Every rule is used more than once.

The algorithm works by looking at successive digrams and, whenever a new digram is formed that duplicates an existing one, adding a new rule. The process needs to be applied recursively when nonterminals replace digrams and rules need to be eliminated when their number of uses falls to one.

The standard Sequitur algorithm provides an exact encoding of a single sequence. Our implementation of the algorithm provides for encoding groups of sequences. This is done by building a global table of variables. Each sequence is then

encoded separately. When the sequence encoding is complete, its nonterminals are merged with those in the global variable table so that each unique right hand side of a variable only appears once. The final encodings are given in terms of these global variables.

As an example, consider the string ABBBBBBBBBCD-DDBCDC from above. It would be encoded as:

```
R1 ::= B B
R2 ::= R1 R1
R3 ::= B C D
S ::= A R2 R2 R3 D D R3 C
```

Here the subsequence BCD has been isolated and the sequence of B's has been encoded as the two rules R1 and R2.

The basic sequitur algorithm does a good job of finding all common subsequences and of encoding them efficiently. It produces a grammar that is compact and where each rule is meaningful. On the other hand, it does not do the best job of handling repetitions (it needs $\log n$ rules to encode a sequence of n identical symbols as in the encoding of the 9 B's), it does not take into account "balanced" strings (strings with the same number of calls and returns), and it does not handle alternation.

We have modified the basic algorithm in two basic ways. The first is to find immediate repetitions and represent them as we do in run-length encoding. This is done as a post processing step. Each rule is processed only once, after all the nonterminals used in it are processed. Each instance of a nonterminal in the right hand side of each rule is replaced by its newly computed expansion, and then the algorithm does a run-length encoding the new right hand side. The rule above become

```
R1 ::= 2:B
R2 ::= 2:(2:B) = 4:B
R3 ::= BCD
S ::= A 2:(4:B) R3 D R3 C = A 8:B R3 2:D R3 C
```

Since R1 and R2 are no longer needed, this yields the grammar

```
R3 ::= BCD
S ::= A 8:B R3 2:D R3 C
```

which is a more logical representation of the original sequence in that the repetition of B is shown explicitly.

This approach can then be combined with the run-length encoding where more than k repetitions are represented as $*:X$. This lets more rules be merged. The representation of the sequence ABBBBBBBBBCDDDBCDC using Sequitur with run-length encoding and a cutoff of 3 is:

R3 ::= BCD

S ::= A *:B R3 2:D R3 C

Our second modification to Sequitur is designed to produce balanced rules. This is useful for the simple string compaction of section 3. In his dissertation [14] Nevill-Manning notes that Sequitur can be restricted to only create rules that contain more closing symbols than opening ones. This is done by not considering digrams that are “unbalanced”. The resultant grammar is a start, but again it does not produce only balanced rules. We use a modification of this algorithm that ensures that all generated rules are balanced.

This is again done through post-processing. We start by restricting the grammar so that all generated rules are either completely balanced or the rule starts with an open bracket (either directly or through a nested rule) and contains more open brackets than close brackets. This is what Neville-Manning’s balancing algorithm does. Then we do the run-length encoding of the rules, as we did in the first modification. But any time we encounter an unbalanced non-terminal, we expand it in place. This makes the rules in the final grammar longer, and some subsequences appear more than once. However, the resultant grammar has only balanced rules.

Finite State Automata Encoding

A way to provide both alternation and repetition in the encoding is to use finite state automata that accept the sequence. One can vary the accuracy and precision of the encoding by using different means of constructing the automaton. At the one extreme, one can build a “chain” automaton that accepts only the given sequence. At the other extreme, one can build a single state automaton that accepts everything via self arcs for every possible input symbol.

Naturally, neither of these approaches is useful. What we want from the automaton is a good intuition for what sort of sequence is possible. In other words, the automaton should reveal something about the structure of the sequence. When the automaton represents multiple sequences their collective structure must be revealed.

There has been significant previous work on inferring finite state automata from input sequences. Most of this work has concentrated on the use of positive and negative examples, i.e. providing a set of inputs that are valid sequences and a set of inputs that are not. Other previous work has looked at interactive models where the inference procedure is able to ask the user whether a particular input is valid or not. Neither of these approaches is practical for the types of sequences that we are looking at.

In some ways, our sequences are special. In the path example, the automata should reflect the internal flow graph of a single procedure. It should encode loops, conditional branches, and pure sequential flow. This provides us with some direction. Other applications, such as using automata to describe how the methods of a class should be used, are not as clear cut. However, even here, the way that most classes are designed to be used should result into a fairly structured flow diagram.

In order to take advantage of the presumed structure behind the sequence and to still provide a reasonably high degree of abstraction while building automata, we have developed a new algorithm for inferring an automaton from one or more sequences.

The Basic FSA Construction Algorithm

Our algorithm constructs deterministic FSAs with the property that any given sequence of three symbols can start at only one state.

To formalize a little, we will use the definitions from Hopcroft and Ullman [10]. A deterministic finite state automaton is defined by:

- a set of states Q ,
- an input alphabet Σ ,
- an initial state $q_0 \in Q$,
- a transition function $\delta : Q \times \Sigma \rightarrow Q$,
- and a set of final states $F \subset Q$.

In addition, we define a string of length k as an element of Σ^k , and we use the extended definition of δ which allows its second argument to be a word (instead of a single symbol). Formally, $\delta : Q \times \cup_i \Sigma^i \rightarrow Q$. For our specific purposes, we define F to have only one element. We achieve that by extending all our strings with a sequence of length k of the special symbol $\$$.

We say that a state q has a k -tail $t \in \Sigma^k$ iff there exists a state q' such that: $\delta(q, t) = q'$, that is, iff starting from q , it’s possible to see the sequence t . We call this relation $Tail(q, t)$.

Our algorithm maintains two invariants: determinism ($\forall q, q_1, q_2 \in Q, x \in \Sigma : \delta(q, x) = q_1 \wedge \delta(q, x) = q_2 \Rightarrow q_1 = q_2$), and unique mapping of k -tails to states ($Tail(q_1, t) \wedge Tail(q_2, t) \Rightarrow q_1 = q_2$).

The basic algorithm maps every k -tail to one state. When two states have a k -tail in common, they are merged. Merging two states is actually a recursive process since when two states are merged, if they have outgoing arcs with the same symbol, the target states of those arcs must also be merged. The process can also affect the possible k -tails of some other state. However, the number of potential additions that need

to be considered as one adds an arc grows as n^k where n is the size of the input alphabet. This is prohibitive in building a large automaton. We therefore ignore such associations while building the automaton and then we merge states based on such associations only once, as a post processing step. The final postprocessing step is classical minimization of the FSA.

An example of the algorithm's output, when given the sequence `ABBBBBBBBBBCDDDBCDC` and $k = 3$ is shown in Figure 2.

Handling Self Loops

The basic algorithm does reasonably well at finding an appropriate automaton. It fails, however, to detect sequences of k or less of a single token. Consider the input sequence `ABBBC`. The automaton generated for this using the basic algorithm would consist of seven states each with a single transition to the next one. (There are seven rather than six states because the algorithm adds a transition on the end symbol `$`). What we actually want to generate is an automaton with a self-loop that indicates the repetition of `B`.

This requires two simple modifications to the above algorithm. The first occurs while we are building the automaton. If the current state already has an input on the current token, instead of creating a new state we create a self-loop. This is sufficient to create self-loops for repetitive input sequences.

Consider the result of applying the modified algorithm to the example shown in Figure 2. The self-loop for input `B` in state S_1 is created because of determinism. This is not enough for the 3 `D`s in state S_2 . The first `D` takes us to a new state S_3 . When we get to the second of the three `D`'s in the sequence, the new check fires and a self loop is created. This is shown in Figure 3 (a). After the last `D` the next sequence is `BCD` and its associated state S_1 . However, the self-loop in state S_3 on `D` would then require that S_1 and S_3 be merged. The result is shown in Figure 3(b).

The second modification is a bit more complex. Self-loops tend to cause spurious state merging. Consider the automaton in part Figure 3(b). Both states S_1 and S_2 have a potential successor string of `DDD`. In general, if a state q has a self-loop on input X ($\delta(q, X) = q$), then any state q' that has a transition into that state with input X ($\delta(q', X) = q$) would be merged with it. This is not the behavior we want. We therefore changed the postprocessing step, so that, in situations like this the initial instance of X causes a transition to the state representing the loop and is not part of the loop itself.

5 Experience

We have used the various encoding approaches with a variety of program traces from both Java and C++. The programs that generated the traces include the following:

KnightsTour solves the problem of finding a Hamiltonian

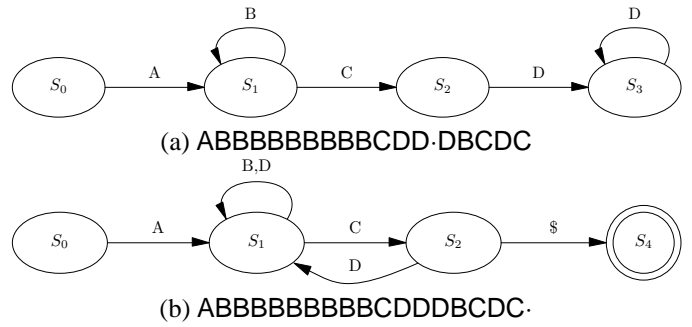


Figure 3: Examples of self loops

path on the graph induced by the knight's moves on a chessboard. It is written in C++.

Romp 1 and 2 simulate the motion of pendulum in a magnetic field. They are both written in Java.

OnSets is an engine for the on-sets board game. Its core is a program that builds valid logic formulas out of a set of characters. Its written in C++.

Decaf is an optimizing compiler for a small subset of Java. It is written in C++.

ShowMeanings is a webserver whose main function is finding alternate meanings of words to elaborate web searches.

The raw trace data ranges in size from one megabyte for a simple C++ program implementing Knight's tour to twenty gigabytes for a test of a commercial Java system that handles web-based requests. We have tried various encodings on each of these traces. To evaluate the quality of the results, we have done spot checks on each trace and a more detailed analysis on the Knight's tour example.

The encodings can be evaluated in two ways. One of the goals of the various encodings, especially the dag and call encodings, is to provide a more concise version of the trace data. In these cases, we can measure the amount of compaction that the encoding provides. Note that this is not a good absolute measure. A program that is very repetitive will have a more condensed encoding than one that is not, independent of the encoding technique. Note also that we are giving up information in doing the encodings. The encodings typically only look at the dynamic call graph. They ignore memory management and synchronization information in the trace. Moreover, they also discard information about individual objects. Based on our experience, however, they still encode about a quarter of the original trace data. Finally, note that the raw trace files are already quite dense since they contain packed binary data while the encoded output files are text files containing XML data.

The compression that results from the various encodings can be seen in Table 2.

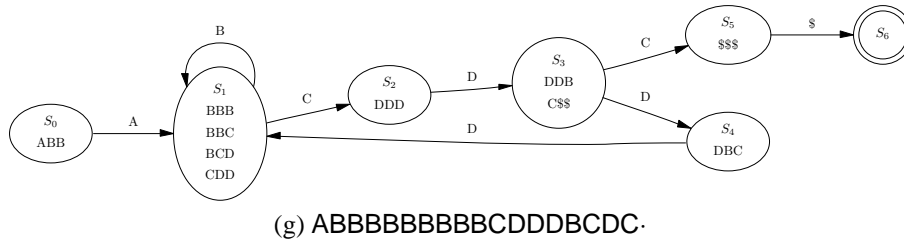


Figure 2: Sample run of the FSA inducing algorithm

The grammar encodings seem to grow with the size of the trace, while the FSA encodings grow with the number of functions in the program. Grammars also tend to grow larger than the FSAs. However, it would be dangerous to make any definite assertions about the sizes of the models.

The second way of evaluating the encodings, especially for the automata-based ones, is to see if the resultant automata reflect the intuitive behavior of the system. For this purpose, we spot-checked routines in the Knight’s tour example, classes in our Java program, and the I/O classes of the C++ standard library, to see what the corresponding automata looked like. An example is given in Figure 4. In all our spot checks, the automata that were generated seem to correspond quite well what one would expect given the code.

One thing that we evaluated in generating the encodings was determine good values for the k parameter for automata. For most of the cases we looked at, a value of k below 2 collapsed too many states while a value of k above 4 did not make that much difference in the resultant automata. Moreover, in most cases, the automata did not change much between $k = 2$ and $k = 4$. While one can come up with examples that require an arbitrary k to find the right intuition, a value of $k = 3$ seems to be a good compromise.

6 RELATED WORK

Tracing programs can be done in a number of different ways. Essentially, the program has to emit records about its state. One can modify the program code to generate the trace data. This is the approach taken by some data visualization systems, like Balsa [2] and Polka[18]. This approach works very well if the goal is something like algorithm animation, where tracing is actually part of the finished program. A second approach is to modify the executable. This can be done either a compile time, like gprof does, or on the finished executable. Apart from our system, EEL [13] takes the same approach. Lastly, one can run the same executable under a tracing environment, like the JVM or a modified JVM, like Jinsight [5].

There has been significant work in encoding the call tree. Ours is essentially the algorithm of Jerding et al. [11], except that they incorporate some of the steps we keep for the encoding phase. Other approaches ([1], [21]) lose some information in the process.

Jerding et al. [11] also focus on class selection. Prof and Gprof [9] kept statistics on function invocations and caller-callee pairs respectively. When we try to derive an automaton based on the usage of classes, we are, effectively, discovering path expressions [3]. This kind of result is akin to the work by Ernst and Notkin [6] who are trying to discover data invariants of a program.

Sequitur has been used to compress basic block trace data by Larus [12].

Neville-Manning reports on the work of Gaines [7] that focused on discovering control flow, albeit with the use of the absolute value of the program counter. Discovering FSAs only by positive examples has a long history. In general, it’s impossible to discover the correct minimal FSA. It is impossible even under the Probably Approximately Correct model. Our algorithm is closest to that of Cook and Wolf [4], with the difference that they collapse two nodes iff one includes all the k -tails of the other one. Since their constraint is stricter than ours, they end up with bigger FSAs.

7 FUTURE WORK

The research described in this paper represents a first step toward a system that will afford a broad basis for understand and visualizing the dynamic behavior of large complex systems. Our current efforts involve extending this basis in a variety of ways.

In the area of trace data collection, we are working on extensions to reduce the size of the trace files and the impact of tracing on program execution. We are working on an interface for specifying which trace records are interesting, and outputting only those. We are also incorporating the minimal tracing of section 2 into the larger framework.

In addition we are implementing a variety of minor extensions to let our trace collection system be used effectively with multiple process, distributed systems.

Next, we are working on developing and incorporating additional selection/compaction techniques. The one we are looking at first involve generating sequences of memory events (e.g. allocations, memory compaction involving moving of objects by the garbage collector, frees, garbage collections), and at generating sequences to reflect the use of a library or arbitrary program abstraction.

Program Name	#Functions	Raw Trace Size	Dag—Grammar	Dag—FSA	String—Grammar	String—FSA
KnightTour	268	700K	120K	208K	12K	40K
Romp 2	3108	87M	1352K	2.3M	408K	576K
Romp 1	1862	500M	937K	1.5M	230K	350K
OnSets	542	833M	6M	464K	1.5M	80K
Decaf	5443	2.6G	29M	5M	6.1M	700K
ShowMeanings	1488	21G	82M	623K	34M	110K

Table 2: Compression achieved by the encodings.

Finally, we are working on developing additional encoding techniques. For the sequence encodings, we are investigating the use of probabilistic models, like hidden Markov models to approximate the FSAs. Probabilistic models are very flexible, and it's well known how to train them. They are capable of various things that an FSA cannot do, like segregation of a program trace in phases. On the other hand, one has to tune a lot of parameters for them to work properly, and since the learning process is an optimization procedure, sometimes the resulting model gets stuck in local optima which are not very meaningful.

REFERENCES

- [1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 85–96, 1997.
- [2] M. H. Brown and R. Sedgewick. Interesting events. In Stasko et al. [19], chapter 12, pages 155–171.
- [3] R. H. Campbell and A. N. Habermann. The specification of process scheduling by path expressions. In *Lecture Notes in Computer Science, No. 16*. Springer Verlag, 1974.
- [4] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–49, July 1998.
- [5] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 1999.
- [6] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–225. ACM Press, May 1999.
- [7] B. R. Gaines. Behaviour/structure transformations under uncertainty. *International Journal of Man-Machine Studies*, 8:337–365, 1976.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [9] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126. ACM, ACM, 1982.
- [10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [11] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of the 1997 International Conference of Software Engineering*, Boston, MA, USA, May 1997.
- [12] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 259–269, 1999.
- [13] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, California, 18–21 June 1995.
- [14] C. G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, New Zealand, may 1996.
- [15] S. P. Reiss. Software visualization in the desert environment. *ACM SIGPLAN Notices*, 33(7):59–66, July 1998.
- [16] S. P. Reiss. Working with patterns and code. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Jan. 2000.
- [17] S. P. Reiss and M. Renieris. Generating java trace data. In *Proceedings of the ACM 2000 Java Grande*, San Francisco, CA, June 2000. ACM Press.
- [18] J. Stasko. Smooth, continuous animation for portraying algorithms and processes. In Stasko et al. [19], chapter 8, pages 103–118.

```

Boolean
KnightSquareInfo::findRestOfTour(KnightSolution sol)
{
    KnightSquare sq;
    KnightHeuristic heur;
    Integer i;

    sol->addMove(row_number, column_number);
    if (sol->isValid()) return TRUE;
    for (i = 0; i < num_moves; ++i)
        legal_moves[i]->markUsed(this);
    heur = new KnightHeuristicInfo(num_moves, legal_moves);
    while ((sq = heur->nextMove()) != NULL) {
        if (sq->findRestOfTour(sol)) return TRUE;
    }
    delete heur;
    for (i = 0; i < num_moves; ++i)
        legal_moves[i]->markUnused(this);
    sol->removeMove(row_number, column_number);
    return FALSE;
}

```

[19] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. M.I.T. Press, Feb. 1998.

[20] D. A. Varley. Practical experience of the limitations of Gprof. *Software — Practice and Experience*, 23(4):461–463, Apr. 1993.

[21] J. Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, CA, June 2000. ACM Press.

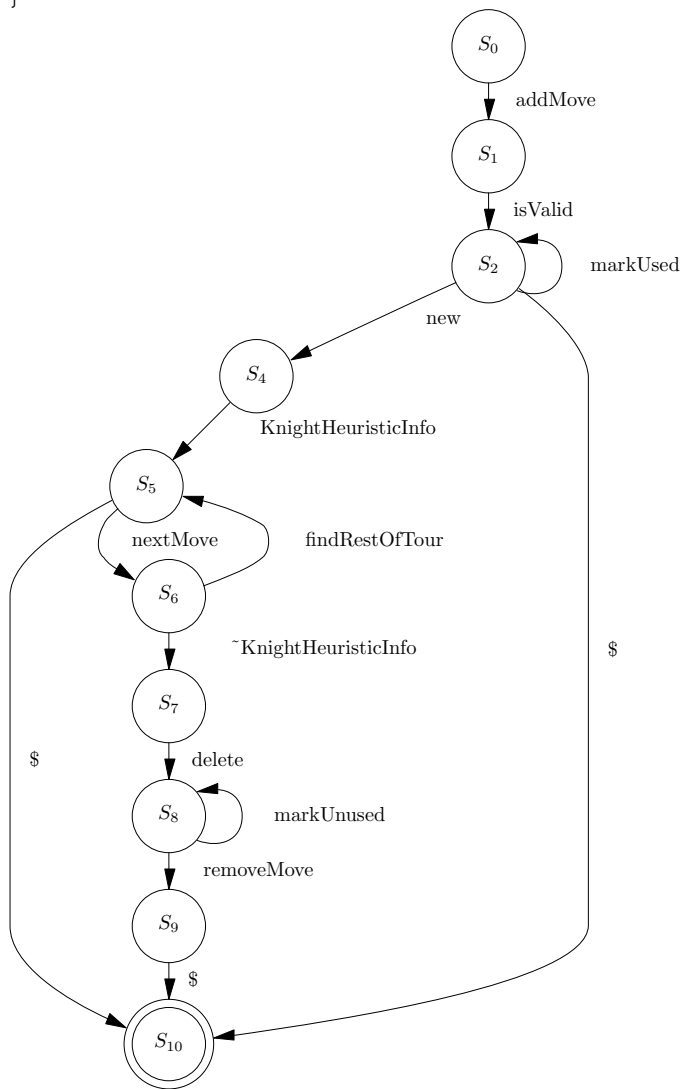


Figure 4: A case where the FSA algorithm discovers the control flow