RC25023 (W1007-058) July 16, 2010
Computer Science

IBM Research Report

Discovering Algorithms in Angelic Programs

Shaon Barman, Rastislav Bodík
University of California
Berkeley, CA

Satish Chandra, Emina Torlak
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Discovering Algorithms in Angelic Programs

Shaon Barman[†] Rastislav Bodík[†] Satish Chandra[⋆] Emina Torlak[⋆]

[†]University of California, Berkeley [⋆]IBM T.J. Watson Research Center

Abstract

In *angelic* programming, the programmer asks an oracle for a demonstration of an algorithm under development. He writes a program with nondeterministic choose statements, with the goal of divining values that he does not yet know how to compute. Given an input, angelic choose statements produce values that, if possible, pass all the assertions in the program. A trace of these values typically exposes the insight behind the algorithm, allowing the programmer to refine the angelic program into a desired deterministic program.

A challenge with angelic programming is that not all safe traces correspond to a plausible algorithm; some achieve a safe execution by abusing the clairvoyance of the oracle. For example, a trace may first break a data structure, only to miraculously correct it later. Clearly, we want a method for identifying (sets of) traces that correspond to a plausible algorithm.

We approach this problem by computing sets of angelic traces that can each be abstracted with a single trace. The abstraction is based on entanglement of choose statements. Intuitively, two choose statements are entangled when their values are correlated, *i.e.,* if you change one, you may have to change the other in order to meet the specification. The abstract trace exposes entangled angels and thus often also the logic behind the algorithm in these traces.

In general, correlation of angels can be a complex relation. We approximate entanglement as an equivalence relation, which allows us to represent entanglement as simple partitioning of angels. We also introduce a two-lattice structure that relates partitions of angels to sets of traces. We develop useful queries on this structure and design efficient algorithms for computing this structure on demand. Finally, we show on several case studies that entanglement helps identify algorithms in angelic programs.

1. Introduction

Background *Angelic* nondeterminism can be a valuable tool in program development [5]. In this style of development, a programmer can use nondeterministic choose statements in his program with the goal of divining values that he does not yet know how to compute. The use of nondeterminism lets the programmer defer part of the reasoning involved in program development to an oracle and allows execution of incomplete programs.

Given a representative input, for each evaluation of a choose statement, the runtime system supplies a value that continues the execution to a successful termination, *i.e.,* the execution violates no assertion. A value is supplied if at all possible, hence the term *angelic*. When a safe execution exists, the programmer obtains a trace of values, one for each choose statement encountered dynamically. In fact, a programmer obtains all such traces. Successful traces can suggest possible ways in which the program can be determinized. When no safe traces can be found, the programmer learns that the development of the program is already on the wrong track; if the oracle cannot complete the program, neither can the programmer.

After examining the safe traces, the programmer *refines* the nondeterministic program to make it more deterministic. Technically, a programmer achieves this by a program transformation that curtails the number of safe traces that the program can produce, until only one safe trace exists for each input. Informally, refinement can be achieved by adding assertions in the program, or by replacing a choose statement by deterministic code. The details of these transformations are not pertinent to this paper. (See [5].)

We illustrate angelic development on the problem of checking whether a graph has a bipartite decomposition. The purpose of the algorithm is to assign each vertex of the graph a 0 or 1 polarity so that edges connect only vertices of opposite polarity. The kernel of the program is adapted from Immerman [9]. Here beta enforces the condition that the subset of edges that have been handled form a bipartite graph: e.handled $\Rightarrow$ e.src.pol != e.dest.pol.

Program $Bipartite_0$

```
for (j <− 0 to numedges − 1) {
  val e : Edge = choose(Edge) // angel selects an edge
  assert (! e.handled)
  if (j == 0) { // first edge processed
    assert (e.src == root) // it must start from root
    e.src.pol = 0
  } else { // enforce wavefront
    // pick edge whose src is assigned polarity already
    assert ( e.src.pol != −1) // −1 means uninitialized
  }
  e.dest.pol = choose(0,1) // angel assigns polarity
  e.handled = true
  assert (beta())
}
```

Note that functional correctness is already embedded in the partial program, as is some restriction on the order of traversing the edges (via the wavefront condition), but the exact order in which edges are picked is nondeterministic. Also nondeterministic is the assignment of polarities. A successful trace from this program will contain the selection of edges that the angels made, as well as the polarities that the angels assigned: *e.g.,* $e_2, 1, e_4, 1, e_5, 1, e_6, 0, \ldots$. The angelic program above produces 3,888 safe traces, each of length 18, on the representative input in Fig. 1.

The Problem In an ideal scenario, an examination of the safe traces would suggest a clearly discernible pattern that the angels are following. However, a challenge with angelic programming is that not all safe traces necessarily correspond to a plausible algorithm.

In our example, the safe traces contain standard graph traversals such as breath-first and depth-first, but they also contain several arbitrary traversals. Here we show a randomly chosen trace:

$$e_2, 1, e_4, 1, e_6, 0, e_7, 0, e_5, 1, e_8, 1, e_1, 0, e_0, 1, e_3, 1$$

The order is unpredictable and it is hard to see how to produce this order algorithmically. In some cases, a safe execution can even be abusing the clairvoyance of the oracle: a data structure is broken only to be corrected later.

Lacking *a priori* knowledge of which traces are desirable, the oracle cannot help the user in discriminating between desirable traces and arbitrary ones. However, since thousands of safe traces may exist even on small inputs—as is the case in our running example—it is desirable to mechanize the analysis of traces.

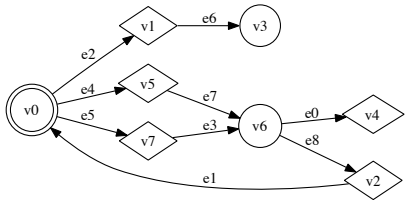

Figure 1. Example graph

Angelic communication and entanglement This paper presents a method for grouping angelic traces so that they need not be triaged individually. Our approach is based on the observation that undesirable traces tend to contain many angels that one can think of as "communicating." Angels communicate in the sense that the choice of one angel influences the choice of another. Communicating angels often compensate for each others' undesirable actions, for example, when one angel breaks the data structure and another corrects it in a way specific to the choice of the first angel.

Not all angelic communication is spurious. Sometimes, a plausible algorithm corresponds to multiple safe traces and angels must communicate to generate them all. The role of the analysis of traces is in grouping traces with similar angelic communication, but letting the user decide whether the communication is spurious or necessary.

We model angelic communication by an *entanglement partition* on the choose statements executed in the traces. Informally, the intent is that angels that are communicating are placed in the same equivalence class, and angels in different equivalence classes do not communicate.

We have designed an algorithm for computing entanglement partitions. Applied to the 3,888 traces produced in our example, our algorithm tells us the entanglement partition is as follows:

$$[\{o_0, o_1, \ldots, o_8, p_1 \ldots p_8\}\{p_0\}]$$

Here, o_i are the angels that pick edges, and p_i are angels that pick polarities. This information tells us that the angels that select edges, and most of the ones that select polarities must be communicating. It is not surprising that p_0 does not need to communicate with any other angel because it has no choice! The polarity of the destination of *any* edge out of root must be 1, as the polarity of the root is fixed to be zero.

The programmer might now realize that, in fact, the polarity angels should never have choice, because as soon as the root is assigned a polarity of 0, the polarity of all the vertices is predetermined (for a connected graph). If they have no choice, they need not be communicating either with the edge-order angels, or with other polarity angels. The reason they *appear* to be communicating is that not all vectors of polarity assignments, combinatorially speaking, are possible.

Entanglement-guided refinement A programmer might wish to pose the question: is there a refinement he could carry out so that entanglement on the remaining traces is as follows:

$$[\{o_0, o_1, \ldots, o_8\}\{p_0\}\{p_1\} \ldots \{p_8\}]$$

This partition reflects his understanding that polarity angels should not be communicating.

We have designed a second algorithm that answers such a query. In this case, we can inform the programmer that there are indeed such subsets of traces that will result in the desired unentanglement. In fact there are 37 such subsets, each of which has a distinct constant vector of polarity assignments to the destination vertices of selected edges.

One such set has polarity assignments in the pattern 111000110, which is suggestive of a breadth first order of traversing the graph. Zooming in on that subset of 72 traces, we now pay attention to the edge-order angels. Using the first algorithm again, we find that the finest entanglement among the order angels in this subset is

$$[\{o_0, o_1, o_2\}\{o_3, o_4, o_5\}\{o_6, o_7\}\{o_8\}]$$

Indeed, a perusal of some of the corresponding traces shows that in each case, first the angels choose some permutation of edges e_2, e_4 and e_5, then choose some permutation of edges e_3, e_6 and e_7, and so on.

Although it is now straightforward to determinize the program, methodologically the next step for a programmer would be to transform the program to produce only this desirable subset of 72 traces. We envision that in the future, suitable assertions can be automatically synthesized from a set of templates (see, e.g. [16]), but for now this is a manual process. A candidate program is the following:

Program $Bipartite_1$

```
for (j <− 0 to numedges − 1) {
  val e : Edge = choose(Edge) // angel selects an edge
  assert (! e.handled)
  if (j == 0) { // first edge processed
     assert (e.src == root) // it must start from root
     e.src.pol = 0
     curlevel = 0
  } else { // enforce wavefront
     assert ( e.src.pol != −1) // −1 means uninitialized
     // enforce non−decreasing level
     assert ( level(e.dest) >= curlevel)
     curlevel = max( e.dest.level, curlevel )
  }
  e.dest.pol = choose(0,1) // angel assigns polarity
  e.handled = true
  assert (beta())
}
```

Here level is temporary instrumentation for the purpose of constraining executions, and is not intended to appear in the final, deterministic program.

We have built a tool with a GUI in which a user can interact with traces using the operations described above. We carried out program development using this tool on several problems, some of which are the same ones that we handled without this tool previously. In our experience, entanglement analysis provides a useful means to quickly navigate through a large set of traces to find subsets that are interesting from the perspective of program development.

Contributions Our contributions are the following.

- *A formalization of angelic communication.* We propose entanglement as a means of capturing which choose statements communicate to produce safe traces and which do not in the setting of angelic programs. This information helps identify sets of traces that might plausibly correspond to algorithms. We provide a mathematical definition of entanglement, and define relationships between angelic partitioning and angelic traces that a programmer may wish to explore during program development.
- *Efficient algorithms.* We design novel algorithms for efficiently computing the results of entanglement queries on demand. The maximum runtime of the algorithms on any of the examples presented in this paper was only 10 seconds, attesting to the feasibility of using these in an interactive tool in developing small but subtle programs. Our algorithms may have applicability in other kinds of trace-based analyses.
- *Empirical evaluation.* We evaluate usefulness of the idea of entanglement in angelic methodology on several problems, including the Deutch-Schorr-Waite graph marking problem, and ListZipReverse, a problem due to Olivier Danvy.

Outline The rest of the paper is organized as following. In Section 2, we define entanglement and present its properties. In Section 3, we present a formal structure that connects entanglement partitions with sets of traces, and show how a programmer would use it for program refinement. Section 4 shows how to compute this formal structure. Section 5 presents a brief description of the tool. Sections 6 and 7 present case studies of using entanglement in program development. We conclude with a discussion of related work.

2. Entanglement

We begin by defining entanglement as a property of traces.

DEFINITION 1 (Entanglement). *Let $\mathcal{L}$ be a set of labels that uniquely identify the* choose *statements executed on an input. Let $\mathcal{T}$ be a set of traces, where each trace is mapping from labels to values. We say that labels $l_i, l_j \in \mathcal{L}$ are unentangled, if and only if, there exists a set $\mathcal{W} \subset \mathcal{L}$, such that $l_i \in \mathcal{W}$, $l_j \in \mathcal{L} - \mathcal{W}$, and*

$$\forall t, t' \in \mathcal{T}. \exists t'' \in \mathcal{T}. \forall l \in \mathcal{L}. t''(l) = ((l \in \mathcal{W})\ ?\ t(l)\ :\ t'(l))$$

Otherwise, l_i and l_j are entangled.

Traces are bounded executions of an angelic program on an input. Entanglement is an empirical property of the program that brings out the combinatorial structure latent in the set of traces. Unlike the notion of dependence, entanglement is *not* directly based on program semantics.

We illustrate the definition of entanglement with a synthetic example shown below, where $\mathcal{L} = \{a_0, a_1, a_2, a_3, a_4\}$.

Program $Synthetic$

```
a0 = choose(0,1)
a1 = choose(0,1)
a2 = choose(0,1)
a3 = choose(0,1)
a4 = choose(0,1)
if (a1) {
  assert a2 + a3 <= 1
} else {
  assert a2 + a3 == 2
}
assert a4 == 1 − a1
assert a0 + a1 + a2 > 0
```

Variables a_0 through a_4 take values in $[0, 1]$, as evident from the choose statements in the code. Our system produces the following safe traces (and no more) for this example.

$$\mathcal{T} = \left\{ \begin{array}{|ccccc|} \hline a_0 & a_1 & a_2 & a_3 & a_4 \\ \hline 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ \hline \end{array} \right.$$

Entanglement analysis on $\mathcal{T}$ reveals that a_1, a_2, a_3 and a_4 are entangled with each other, but a_0 is not entangled with any other variable. (Here, by entanglement among variables we mean entanglement between invocations of choose operators that are assigned to those variables.) Referring to the definition of entanglement, if we had to show a_0 is unentangled with a_1, the set $\{a_0\}$ serves the role of $\mathcal{W}$ in the definition. Intuitively, we can interpret this as telling us that angelic statements for a_1, a_2, a_3 and a_4 must communicate to make an execution avoid assert failures, but a_0 can act independently.

Some features of entanglement are as follows:

- The absence of entanglement can be computed conclusively only when the full set of traces is available.
- Removing a trace from a set of traces does not necessarily reduce entanglement, and can even increase it. For example, if the last trace were dropped, a_0 would lose its independence. Thus, entanglement is not monotonic in the size of the set $\mathcal{T}$.
- Entanglement is modeled as a symmetric relation. In the example, although the code seems to suggest a_2 is "dependent" on a_1 and not the other way around as well, we consider both entangled with the other.
- Entanglement is modeled as a transitive relation. In the example, since a_2 is entangled with a_1, which in turn is entangled with a_4, we say a_2 is entangled with a_4, even though the program does not seem to suggest that directly. A transitive formulation of entanglement allows its representation as a partition and simplifies our algorithms. Here, partition $L = [\{a_0\}\{a_1, a_2, a_3, a_4\}]$.
- The definition implies that if an angel has a constant value in the set of traces, it is unentangled with other angels.
- The definition implies that in a singleton set of traces, all angels are unentangled.

3. Entanglement-guided program refinement

We propose to use entanglement to help with refinements in the angelic development methodology. An angelic program Q is a refinement of an angelic program P if the set of safe traces in Q is a subset of the safe traces in P, i.e., Q specializes the behavior of P. While two angels can be entangled on the set $\mathcal{T}$ of all safe traces, there may exist a subset T' of traces on which their entanglement disappears. T' is thus a refinement of the program that removes potentially undesirable angelic communication and could be the plausible algorithm. Indeed, the search for a plausible algorithm is application of refinement until the remaining entanglement is deemed necessary by the programmer.

The relationship between entanglement of angels and program refinement can be described with a two-lattice structure: the *trace lattice*, which is defined over the safe traces ($\mathcal{T}$) of a program, and the *angel lattice*, which is defined over its angelic labels ($\mathcal{L}$). The trace lattice orders the power set of $\mathcal{T}$ via the subset inclusion relation $\subseteq$. The angel lattice orders the partitions of $\mathcal{L}$ via the refinement relation $\sqsubseteq$. The refinement relation $\sqsubseteq$ has the usual meaning: $L \sqsubseteq L'$, where both L and L' are sets of sets that partition $\mathcal{L}$, if and only if for every $L_i \in L$, there is an $L'_j \in L'$ such that $L_i \subseteq L'_j$. Moving down the angel and trace lattices simultaneously brings us to more refined programs with less entanglement.

To facilitate entanglement-guided browsing of the lattices, we define two relations that connect them in a meaningful way: the *finest entanglement partitioning* (FEP) for a set of traces and a *locally maximal support* (LMS) for an angel partition. Both are derived from the *entanglement partitioning* relation induced by Def. 1, which maps sets of traces to partitions of angels entangled in those traces. FEP and LMS summarize the edges of this relation by taking a point in one lattice to its lowest or highest counterpart(s) in the other lattice, as specified below.

DEFINITION 2 (Entanglement partitioning). *A partition $L = \{L_1, \ldots, L_n\}$ of $\mathcal{L}$ is an entanglement partitioning for a set of traces $T \subseteq \mathcal{T}$ if and only if T can be expressed as a concatenation of projections onto the sets $L_1, \ldots, L_n$: i.e., $T = T \downarrow L_1 \oplus \ldots \oplus T \downarrow L_n$.*

The projection $T \downarrow L_i$ of a set of traces T onto a set of labels L_i is defined as $\{t \downarrow L_i \,|\, t \in T\}$, where $t \downarrow L_i :=$

$\{\langle l, v\rangle \mid l \in L_i \wedge t[l] = v\}$. The concatenation of the sets T and T' is the pairwise concatenation of their elements, $T \oplus T' := \{t \oplus t' \mid t \in T \wedge t' \in T'\}$. The trace concatenation operation $t \oplus t'$ yields $t \cup t'$ on traces with disjoint labels and is undefined otherwise. By extension, $T \oplus T'$ is defined only if $t \oplus t'$ is defined for every $t \in T$ and $t' \in T'$.

DEFINITION 3 (Finest entanglement partitioning). *A partition L is the finest entanglement partitioning (FEP) for a set of traces T iff L is the lowest point on the angel lattice that is an entanglement partitioning for T.*

DEFINITION 4 (Locally maximal support). *A set of traces $T \subseteq \mathcal{T}$ is a locally maximal support (LMS) for a partition L iff L is an entanglement partitioning for T but not for any proper superset of T.*

To illustrate Definitions 2-4, let us revisit the sample program from Sec. 2. The FEP for all safe traces of this program is the partition $L = \{\{a_0\}\{a_1, a_2, a_3, a_4\}\}$, which precisely captures the entanglement relationships in $\mathcal{T}$. All partitions coarser than the FEP (in this case, only $\{\{a_0, a_1, a_2, a_3, a_4\}\}$) are entanglement partitionings for $\mathcal{T}$. All partitions finer than FEP have one or more LMSs in $\mathcal{T}$. For example, consider the following refinement of $\mathcal{T}$'s FEP:

$$L' = [\{a_0\}\{a_2\}\{a_1, a_3, a_4\}]$$

This partition has three LMSs, shown here compactly in regular expression notation, with a_2 highlighted:

$$\begin{aligned} T_1 &= (0|1).0.\mathbf{1}.1.1 + (0|1).1.\mathbf{1}.0.0 \\ T_2 &= (0|1).1.\mathbf{0}.(0|1).0 \\ T_3 &= (0|1).1.(\mathbf{0}|\mathbf{1}).0.0 \end{aligned}$$

T_1, T_2 and T_3 demonstrate several important properties of LMSs:

- FEPs induced by the LMSs will necessarily disentangle a_2, but may split other equivalence classes as well. In fact, both T_2 and T_3 induce the FEP $L'' = \{\{a_0\}\{a_1\}\{a_2\}\{a_3\}\{a_4\}\}$.
- No subset of traces from $\mathcal{T}$ can be added to any one these subsets while keeping a_2 unentangled (maximality).
- The LMSs cover all of $\mathcal{T}$; *i.e.,* $\mathcal{T} = T_1 \cup T_2 \cup T_3$.

These traces also show that to unentangle a_2, some of the other angels may have to be held to a constant value.

Figure 2 shows the lattice structures for the sample program. On the left is the angel lattice and on the right is the trace lattice for this example, each shown only in part. The dashed arrows show the result of computing FEP. The dotted arrows show the result of computing LMS.

4. Computing lattice relationships

To support interactive usage of our tool, we have designed efficient algorithms to compute the FEP and LMS relations.

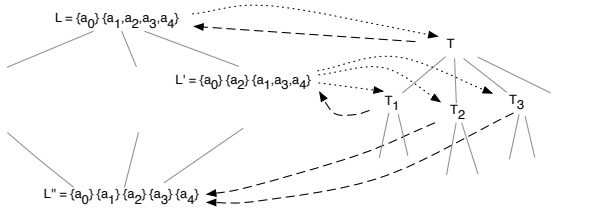

Figure 2. The 2-lattice structure

Both algorithms require that the traces in a given input set are of the same length, and that each label gives a unique identity to a dynamically occurring choose statement.

To satisfy these requirements, our tool encodes angelic labels and traces as follows. We first attach unique static names to each choose statement, drawn from a set S. A label $l \in \mathcal{L}$ for a dynamically occurring choose statement then becomes a finite string of symbols from S, as they are encountered in a particular execution. A trace $t \in \mathcal{T}$ is described as a mapping $\mathcal{L} \mapsto \mathcal{V}$ from labels to values. A mapping $\langle l, v\rangle \in t$ represents the value v generated by the non-deterministic choice at the label l. If a given execution does not perform the choice labeled by l (*i.e.,* it does not traverse the path encoded by l), then the trace t of that execution maps l to bottom: $t[l] = \perp$. As a result, all traces generated for a given program on a given input are defined over the same set of labels, each of which uniquely identifies a dynamically occurring choose statement.

4.1 Finding the finest entanglement partitioning

Given a set of traces T, we compute its FEP efficiently using the algorithm in Fig. 3. The top-level procedure, FEP, is straightforward. Lines 1 initializes the variable L, which holds discovered partitions, to the empty set; line 2 initializes $rest$, which holds unpartitioned labels, to the set of all labels mapped by the traces in T. The main loop then computes the partitions by choosing some label l that has not yet been placed in a partition (line 4); finding the finest partition $part$ that contains l (line 5); and updating $rest$ to exclude, and L to include, $part$ (lines 6-7).

The key step in the algorithm—finding the finest partition that contains a given label—is performed by the procedures ENTANGLED and WITNESS. Given a label l and traces T such that $l \in labels(T)$, ENTANGLED computes the labels entangled with l in T as follows. We first hypothesize that l is in a partition $part$ of its own (line 1). This hypothesis is then tested by invoking WITNESS on $part$ and T (line 2). The WITNESS procedure, as explained below, returns the empty set if $part$ is an entanglement partition for T. Otherwise, it returns some, but not necessarily all, labels that are entangled with $part$. Because the set of labels returned by WITNESS may be incomplete, the main loop of ENTANGLED (lines 3-5) keeps expanding $part$ with WITNESS labels until $part$ becomes an entanglement partition for T.

The WITNESS procedure works by first checking if $part$ is an entanglement partition for T. Lines 1-3 implement this check as a straightforward application of Def. 2. If the check succeeds, the procedure returns the empty set (line 4). Otherwise, we choose (line 5) some *unsafe* trace of the form $x \oplus y$, where x is in the projection of T onto $part$ and y is in the projection of T onto the complement of $part$. The chosen trace is a witness of entanglement between $part$ and some labels in $part$'s complement. The rest of the procedure (lines 7-11) collects and returns these labels, which comprise the *smallest* subset of $part$'s complement that the witness $x \oplus y$ maps differently than a *safe* trace $x \oplus y'$.

To see that any non-empty set returned by WITNESS contains only labels entangled with $part$, suppose that, at the end of the loop, w contains one or more labels that are not entangled with $part$. Denote these labels with b. Then, by Def. 2, there is a partitioning $\{L_1, L_2\}$ of $labels(T)$ such that $T = T \downarrow L_1 \oplus T \downarrow L_2$, $part \subseteq L_1$ and $b \subseteq L_2 \subseteq (labels(T) \setminus part)$. This and line 5 imply that for any witness $x \oplus y$, the following equalities must hold: $x \oplus y = ((x \oplus y) \downarrow L_1) \oplus ((x \oplus y) \downarrow L_2) = (x \oplus (y \downarrow L_1)) \oplus (y \downarrow L_2)$. Since $L_2 \subseteq (labels(T) \setminus part)$ and y is chosen from $T \downarrow (labels(T) \setminus part)$, we know that $y \downarrow L_2 \in T \downarrow L_2$. Furthermore, since $T = T \downarrow L_1 \oplus T \downarrow L_2$ and $x \in T \downarrow part$ (where $part \subseteq L_1$), there must be a safe trace $t \in T$ such that $t \downarrow part = x$ and $t \downarrow L_2 = y \downarrow L_2$. The existence of t, however, forces the computation on lines 6-10 to yield $w \subseteq L_1 \setminus part$. This contradicts the initial assumption that w contains a non-empty subset $b \subseteq L_2$.

Correctness of the algorithm as a whole follows easily from the correctness of WITNESS. The running time is at most cubic in the number of traces: this cost can be seen from line 5 in procedure WITNESS. The number of traces is the dominant cost since the number of labels (*i.e.,* the length of traces) is negligible in comparison.

Example. Figure 4 illustrates an execution of the FEP algorithm on the example from Section 2. Each column in the table represents one iteration of the main loop of FEP. During the first iteration, the algorithm checks if a_0 is entangled with any other angelic statement by trying to find a witness. Because no witness exists (*i.e.,* $(X \oplus Y) = T$ on line 3 of WITNESS), we conclude that a_0 is unentangled with all other angelic labels and place it into its own partition.

The loop in FEP then moves on to the next angelic statement, a_1. This time, we can find a witness which shows that a_1 is entangled with some of the remaining angelic statements. One such witness is $x \oplus y = \{a_1 = 1, a_2 = 1, a_3 = 1, a_4 = 1\}$. This value is a witness because in some safe trace, $\{a_1 = 1\}$, and in some other safe trace, $\{a_2 = 1, a_3 = 1, a_4 = 1\}$, but there does not exist a single trace with both of these properties. The algorithm then chooses a minimal set of angels, $w = \{a_2, a_4\}$, such that the values of these angels in $x \oplus y$ can be changed in order to get a trace in $T \downarrow rest$ (*e.g.,* $\{a_1 = 0, a_2 = 1, a_3 = 0, a_4 = 1\}$).

```
FEP(T)
1  L ← {}
2  rest ← labels(T)
3  while rest ≠ ∅ do
4    l ← choose(rest)
5    part ← ENTANGLED(l, T ↓ rest)
6    rest ← rest \ part
7    L ← L ∪ {part}
8  return L

ENTANGLED(l, T)
1  part ← {l}
2  w ← WITNESS(part, T)
3  while w ≠ ∅ do
4    part ← part ∪ w
5    w ← WITNESS(part, T)
6  return part

WITNESS(part, T)
1  X ← T ↓ part
2  Y ← T ↓ (labels(T) \ part)
3  if (X ⊕ Y) = T then
4    return {}
5  x ⊕ y ← choose((X ⊕ Y) \ T)
6  w ← (labels(T) \ part)
7  for y' ∈ Y such that x ⊕ y' ∈ T do
8    w' ← {l | y[l] ≠ y'[l]}
9    if |w'| < |w| then
10     w ← w'
11 return w
```

Figure 3. Algorithm for finding the finest entanglement partitioning of angels for a given set of traces. The algorithm requires that all traces in the input set are defined on the same set of labels.

At this point, the algorithm has found that $\{a_1, a_2, a_4\}$ are all entangled, but must repeat the loop in ENTANGLED to ensure that no other angelic statement has been left out. In this second pass, we find that a_3 also belongs in this partition. Since there are no remaining angels, the execution terminates.

4.2 Enumerating all locally maximal supports

Given a partitioning L of labels and a set of traces T defined over those labels, we enumerate the LMSs of L in T using the LMS algorithm in Fig. 5. If L consists of a single partition, then its set of maximal supports is simply $\{T\}$ (lines 2-3). If L consists of two or more partitions, then the problem of computing its LMSs reduces to the problem of

FEP(T)	FEP(T)		FEP(T)
$L = \emptyset$	$L = [\{a_0\}]$		$L = [\{a_0\}\{a_1, a_2, a_3, a_4\}]$
$rest = \{a_0, a_1, a_2, a_3, a_4\}$	$rest = \{a_1, a_2, a_3, a_4\}$		$rest = \{\}$
ENTANGLED(a_0, T)	ENTANGLED($a_1, T \downarrow rest$)		
$part = \{a_0\}$	$part = \{a_1\}$	$part = \{a_1, a_2, a_4\}$	
WITNESS($\{a_0\}, T$)	WITNESS($\{a_1\}, T \downarrow rest$)	WITNESS($\{a_1, a_2, a_4\}, T \downarrow rest$)	
$X = \{0, 1\}$	$X = \{0, 1\}$	$X = \{100, 110, 011\}$	
$Y = \{1000, 1010, 1100, 0111\}$	$Y = \{000, 010, 100, 111\}$	$Y = \{0, 1\}$	
	$x \oplus y = 1111$	$x \oplus y = 0101$	
	$w = \{a_2, a_4\}$	$w = \{a_3\}$	

Figure 4. Execution trace of the FEP algorithm as applied to the example in Section 2.

enumerating all maximal bicliques [1, 10] in the bipartite graph representation of T. In particular, given a partitioning $\{L_1, L_2\}$ of its labels, a set of traces T can be encoded directly as a bipartite graph $(V_1 \cup V_2, E)$ using the procedure $\mathcal{G}$ in Fig. 5. A maximal biclique in this graph is a maximal subgraph of the form $(V \cup V', V \times V')$, where the subgraph relation is defined in the usual way: *i.e.,* $V \subseteq V_1$, $V' \subseteq V_2$ and $V \times V' \subseteq E$. It is easy to see that the trace representation of a maximal biclique in $\mathcal{G}(T, \{L_1, L_2\})$ satisfies the definition of an LMS (Def. 4). Hence, line 6 correctly enumerates all LMSs of $\{L_1, L_2\}$ in T. The correctness of the algorithm in the case of an L with more than two partitions (lines 7-10) follows by induction from the base cases.

Since there may be exponentially many maximal bicliques in a given graph, the worst case running time of the LMS algorithm is exponential. In practice, however, graphs that correspond to traces have a small number of bicliques for any given partitioning. Our current implementation enumerates them quickly using a SAT-based constraint solver [17], but we plan to implement a dedicated biclique enumeration algorithm (*e.g.,* [1, 10]) in the future.

Example. Revisiting the example from Sec. 2, let us now walk through calculating the LMSs of the partitioning $L = \{\{a_0\}\{a_2\}\{a_1, a_3, a_4\}\}$ using the LMS algorithm. Since the maximal biclique subroutine only works on two partitions at a time, the algorithm first finds the maximal bicliques of $\{a_0\}$ and $\{a_1, a_2, a_3, a_4\}$. This returns only one biclique which encompasses the entire graph. A recursive call to LMS is made with $L = \{\{a_2\}\{a_1, a_3, a_4\}\}$ and $T = \{1000, 1010, 1100, 0111\}$, which returns the three maximal supports shown in Sec. 2. Figure 6 illustrates the graphs and maximal bicliques created during the execution of the algorithm.

5. Implementation

We have extended the tool from [5] to allow the programmer to use angelic entanglement to gain insight about about the angelic program. The tool embeds the angelic choose construct into the Scala programming language [12]. The program passes to the angelic choose operator a list of values from which a parallel backtracking solver selects one

```
LMS(L, T)
1  switch L
2    case {L_1} :
3      return {T}
4    case {L_1, L_2} :
5      B ← MAXBICLIQUES(G(T, {L_1, L_2}))
6      return ⋃_{(V∪V', V×V')∈B} {V ⊕ V'}
7    case {L_1, L_2, ..., L_n} :
8      L' ← L \ {L_1}
9      B ← MAXBICLIQUES(G(T, {L_1, ⋃ L'}))
10     return ⋃_{(V∪V', V×V')∈B} ⋃_{M∈LMS(L', V')} {V ⊕ M}

G(T, {L_1, L_2})
1  V_1 ← T ↓ L_1
2  V_2 ← T ↓ L_2
3  E ← {⟨v_1, v_2⟩ | v_1 ∈ V_1 ∧ v_2 ∈ V_2 ∧ v_1 ⊕ v_2 ∈ T}
4  return (V_1 ∪ V_2, E)
```

Figure 5. Algorithm for enumerating all locally maximal supports for a given partitioning of angels in a given set of traces. The algorithm requires that the input set L partitions the set of labels on which the given input traces are defined.

that leads to a safe trace. During the entanglement analysis, this list of values serves as the finite domain for the angelic choose operator. The solver computes all safe executions, which can then be browsed through the GUI. A trace is created for each execution by mapping each angelic choose operator to the index in the list of values that is chosen for that execution. We have used this implementation to develop the examples in this paper.

The GUI presents the programmer with a list of safe executions. Once the programmer selects an execution, the main window displays that execution by showing the source code annotated with decisions made by the angelic choose operators during that execution.

The entanglement extension of the tool has been designed to allow the programmer to quickly understand the safe traces through queries about the FEP and LMS relations.

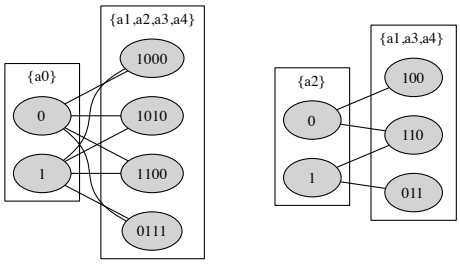

Figure 6. Graph of $\mathcal{G}(T, [\{a_0\}, \{a_1, a_2, a_3, a_4\}]$ and $\mathcal{G}(T, [\{a_2\}, \{a_1, a_3, a_4\}])$

Program	#Traces	#Labels	FEP	LMS
dsw	11679	37	-	1665
dsw	8040	37	1023	3473
dsw	6760	37	438	348
dsw	200	37	15	9
listzip	460	30	136	145
listzip	368	30	386	42
bipartite	3888	18	594	10076

Table 1. Runtime of our algorithms on selected inputs (ms)

Initially, he sees all safe traces returned by the synthesizer. He can ask queries or refine this set of traces through various commands. For example, if the programmer wanted to know the finest angelic partitioning, he would use the *entanglement* command. Another useful command is to view the maximal supports for a finer angelic partitioning. The *supports* command takes in an angelic partitioning and creates subsets of traces. The programmer can then choose one of the subsets as the current set of traces and continue with the refinement. Whenever the user chooses a support, the GUI is updated to only show the executions in the chosen support. This allows the programmer to quickly browse the traces in a support in order to find patterns among the traces. There are also commands to allow backtracking in case a misstep is taken. The time taken (in milliseconds) for various trace sets on FEP and LMS queries are shown in Table 1.

6. Deutsch-Schorr-Waite

The Problem. This section considers angelic development of the Deutsch-Schorr-Waite (DSW) algorithm for marking of reachable nodes in a directed graph. Unlike graph marking with an explicit stack, DSW uses constant memory thanks to cleverly reversing pointers in the graph.

Methodology. One of co-authors developed the angelic program shown below as part of [5]. As part of work on this paper, a different co-author analyzed the safe traces using entanglement.

```
def DSW(g) {
  val vroot = new Node(g.root)
  var current = g.root
  ParasiticStack.push(vroot, List(vroot,g.root))

  while (current != vroot) {
    if (!current.visited) current.visited = true
    if (current has unvisited children) {
      current.idx = index of first unvisited child
      val child = current.children[current.idx]
      ParasiticStack.push(current, List(current, child))
      current = child
    } else {
      current = ParasiticStack.pop(List(current))
    }
  }
}
```

Figure 7. The DSW algorithm with a parasitic stack.

Angelic Programs for DSW. We developed DSW with angelic refinement in [5]. A key idea was to avoid mixing graph traversal code with the pointer reversal code by expressing DSW with a *parasitic stack*, a data structure that behaves like a stack but borrows storage from the host data structure. Since it behaves like a stack, it allows us to write DSW cleanly, as if it was regular dfs traversal (see Figure 7). Thanks to the parasitic stack, we achieved modularization: pointer reversal is hidden in the parasitic stack under the metaphor of borrowing and restoring memory locations.

The programming challenge is to parameterize the parasitic stack. There are three questions:

1. Which location to borrow from the graph? The traversal must not need the location until it is returned.
2. How to restore the value in the borrowed location? The stack does not have enough locations to remember values from all borrowed locations.
3. How to use the borrowed location to provide the stack semantics?

A generic parasitic stack is implemented in Figure 8. The parameterization of this stack has been delegated to angels.

The parasitic stack in Figure 8 keeps only a single memory location (e). The push method first angelically selects which memory location to borrow from the host. This is done by selecting a suitable node n and a child slot c in that node. The borrowed location is n.children[c]. The stack (deterministically) stores the selected child slot index in the node itself, as that is allowed by the constraints of the DSW problem. Next, push reads the value in the borrowed location since it will need to be restored later and so may need to be saved. Finally, there is a hard decision. The stack has four values that it may need to remember: the pushed value x, the reference to the borrowed location n, the value in that location v, and the value in the extra location e. However,

```
ParasiticStack {
    e = choose(node in g) // initialize one extra storage location

    // 'nodes' is list of nodes we can borrow from
    push(x,nodes) {
        // borrow memory location n.children[c]
        n = choose(nodes)
        c = choose(n.children.length)

        // value in borrowed location will need to be restored
        v = n.children[c]

        // we are holding 4 values but have only 2 locations
        // select which 2 values to remember, and where
        e, n.children[c] = angelicallySemiPermute(x, n, v, e)
    }
    // values are pointers to nodes that may be useful
    pop(values) {
        // ask angels which location we borrowed in push()
        n = choose({e} ∪ values)
        c = choose(0 until n.children.length)

        // v is the value stored in the borrowed location
        v = n.children[c]

        // (1) select return value
        // (2) restore value in the borrowed location
        // (3) update the extra location e
        r, n.children[c], e = angelicallyPermute(n,v,e,values)
        return r
    }
}
```

Figure 8. Angelic implementation of the parasitic stack.

there are only two locations available to the stack: the borrowed location n and the extra location e. Clearly, only two of the four values can be stored. Perhaps the value of e is not needed, but the remaining three values are essential.

A little reasoning reveals that the parasitic stack is plausible only if the value that push must throw away is available at the time of pop from the variables of the enclosing traversal code. Therefore, we decided to make the environment of the client available to pop. The pop method first guesses which location was borrowed in the corresponding push. This location is either n or is in nodes; no other alternatives exist. Next, pop reads the value from the borrowed location. Finally, pop angelically decides (i) which value to return, (ii) how to update the extra locations, and (iii) how to restore the borrowed location. As in the case of push, it must select from among four values.

Entanglement Analysis. Recall that entanglement typically starts with a set of traces. In this study, we started with the set of all safe traces and partitioned them based on trace properties. The specific property we used was trace length. We then chose the set with equal number of pushes and pops. (Angels were indeed able to to form safe traces without matching pushes and pops.)

Next, we computed the partitioning L of angels induced by this set of traces. We observed that the very first angel executed in the trace—the angel that initialized the extra location e—was entangled with other angels. This contradicted our hypothesis that the initialization value should be a constant across all traces, whereas entanglement told us that there were traces in which later angels depended on a *particular* initialization value; in these traces, the initialization could not be changed arbitrarily without some other angel compensating. We posited that our hypothesis was correct and hence that traces with entanglement on the first angel were undesirable. We asked the tool to filter them out by requesting the maximal supports in which this first angel was completely disentangled. These angelic partitions corresponded to traces that *did* match our belief.

The tool found four such maximal supports, of sizes roughly 2,000 traces (three supports) and 6,000 traces (one support). Quick examination showed that in the three smaller supports, the angel was set to a constant value (the same value across all traces of the set). In contrast, in the large support, the angel could take on any value. This subset of traces matched a stronger hypothesis: not only the initialization value could be a constant, it in fact did not matter. This was because in all these traces, the location e was overwritten before it was read. We proceeded with this subset of traces.

Finally, we requested a subset of traces that removed some further entanglement. Specifically, we requested to keep only entanglement within the same function invocation. If entanglement does not cross procedure boundaries, then the function may have a simpler interface (decisions in one invocation do not depend on decisions in callers or callees). Removing inter-function-invocation entanglement created eight maximal supports. After manually examining the supports, we found that one support, of 200 traces, exhibited had a lot of regularity, i.e., function invocations made mostly the same angelic decisions. Specifically, the angels in the first push were all entangled, and the angels in the last pop were all entangled but the remaining angels were completely disentangled. And more so, most dynamic invocations of the same syntactic angel took on only one value. The entanglement in the first push and last pop were entangled because in these invocations, the angels could find the necessary values also in other locations. However, these two invocations could also use the same angelic decisions made in the other invocations.

After doing some more analysis, we concluded that this set contained the traces that we had previously deemed to be the algorithm. In summary, we arrived at the algorithm in a mere two requests for disentanglement, which reduced 11679 traces to 200 traces.

7. ListZipRev

The Problem. The ListZipRev problem, posed at a summer school, asks to take two lists of strings and return a single list, created by zipping the first list with the reversed second list. The crucial restriction is that each list must be traversed at most once, and only operations cons(head,tail), list.head, and list.tail can be used.

Methodology. One of us had developed a solution to this problem via angelic refinement about a year ago, without the benefit of the theory of entanglement. A different co-author re-analyzed the angelic program with entanglement; the results are reported below. To compare the two experiences, the manual examination of traces necessary for arriving at a correct solution took more than an hour; an incomplete examination of traces necessary for deeper understanding of the angelic program took more than half a day. In contrast, entanglement-based analysis produced deeper understanding in about one hour.

Angelic Programs for ListZipRev. Before we arrive at the final angelic program, let us review its development. The first angelic program quite literally encodes the specification. The program iterates as many times as necessary, traversing each list at most once. (Note that in each iteration, the index variables a and b have a choice of advancing or not advancing along their respective lists. Since these variables cannot reset to the beginning of the list, the angelic program ensures that each list is traversed at most once.) This program encodes all possible traversals over the lists yet it has no angelic trace, which means that the problem cannot be solved under the given resource constraints.

Program $ListZipRev_0$

```
val x = List("a", "b", "c", "d")
val y = List("1", "2", "3", "4")
var a = x, b = y, r = Nil

while (choose) {
  a = choose(a, a.tail)
  b = choose(b, b.tail)
  if (choose) r = cons(a.head + b.head, r)
}
assert r == List("a4","b3","c2","d1")
```

It is obviously necessary to relax the specification a little. By using recursion, we give the traversal memory, which allows visiting a node that has been previously visited. The first angel decides whether to construct the list when descending into the recursion (up=false) or when ascending from the recursion (up=true). For the sake of presentation, this program is not concerned with how to traverse the lists: instead of using pointers into a list, it angelically selects a suitable element from the list, with the operation choose(x). This program is more nondeterministic than the version we want to understand with entanglement, but this version is a useful sanity check: if no angelic traces exist, the program is buggy, which we discover before progressing too far in the development process.

Program $ListZipRev_1$

```
var r = Nil
val up = choose // pick true or false

def descent() = {
  if (choose) return

  if (!up) r = cons(choose(x) + choose(y), r)
  descent()
  if (up) r = cons(choose(x) + choose(y), r)
}
descent()
```

We now implement the angels choose(x) and choose(y) with angelic expressions that actually traverse the lists, in all possible ways, rather than pick an element from the list. Note that $ListZipRev_2$ is a refinement of $ListZipRev_1$ in that the set of traces of the former is a subset of those of the latter. This is because the implementation of choose(x) and choose(y) restricts the values these two angels can produce.

When it comes to list traversal, the angels in this program make several decisions. When invoking the recursive function, the pointers into each list are passed in; here the angels decide whether the pointer should be advanced or not (line 1). When returning from recursion, we must return a pointer into the list. If the pointer can be computed either from the argument (a) or from the pointer returned from recursion (aa), in line 3. Finally, we have an analogous choice when constructing the list.

Program $ListZipRev_2$

```
var a = x, b = y, r = Nil
val up = choose

def descent(a, b) : (List,List) = {
  if (choose) return (a,b)

  if (!up) r = cons(a.head + b.head, r)

  // 1: advance the pointers a, b to the next element?
  val (aa, bb) = descent(choose(a,a.tail), choose(b,b.tail))

  // 2: construct the list from the arg (a) or ret val (aa)
  if (up) r = cons(choose(a,aa).head + choose(b,bb).head, r)

  // 3: return the pointer based on the arg or ret val
  return (choose(a,aa).tail, choose(b,bb).tail)
}
descent(x,y)
```

The angelic program $ListZipRev_2$ has 30 angel invocations in each trace and produces 460 safe traces. Our goal now is to find the desired algorithm within these traces. The first thing we observe is that up=true in all traces, which means that a tail-recursive algorithm does not exist, as the real work happens when returning from recursion.

Now, to find the algorithm in the 460 traces, we turn to entanglement analysis. The analysis discharges two angels directly: they are unentangled but have multiple values, causing an explosion in the number of traces. After reviewing these angels, we observed that the values of these angels are not used during the execution; they are the two angels in the last invocation of return(choose(a,aa).tail, choose(b,bb).tail). These are don't-care angels because they can return any value. Such angels are always unentangled and cause a multiplier effect in the number of traces.

We are still left with more than 100 traces. Here entanglement helps by revealing two entanglement properties:

- Entanglement of angels crosses procedure invocations. That is, the choices made by a procedure invocation influence the choices of other procedure invocations.
- Angels that manipulate the list x (there are three in each procedure, or 12 in total) are not entangled with the angels that manipulate the list y (again, there are 12 in total).

It follows that we can reason about each list in isolation, but we need to do it by crossing procedure boundaries. Furthermore, we can implement (refine) the two lists in isolation because we can constrain the angels for one list without restricting the safe behaviors (in our case, traversals) over the other list.

A plausible strategy is to remove all entanglement that crosses procedures. This however refines the program too much, meaning that it creates many support sets, too many to analyze. Instead, it is more beneficial to perform a more gentle refinement, one that removes only some entanglement but creates a large set of traces that is easier to understand.

We thus proceeded to understand the entanglement in the list traversal code, separately for each list. The b list is almost disentangled except for three angels. Disentangling one of them—the angel in descent(..., choose(b,b.tail))—yields two support sets. In the first set, this angel chooses the first value, while the second set permits both values. Analyzing the traces, we discovered that this was a case of conditional aliasing: under some conditions (which held in the second set of traces), the values of variables were such that either choice of an angel led to a safe trace. In short, this situation manifested itself as entanglement between the angel that controlled the condition and the angel that was allowed to make both choices. In more detail, this situation arose in the last recursive call, where bb takes on the value chosen by this angel. So when the angel chooses the first value, bb = b, and when the angel chooses the other value, bb = b.tail. The value that makes the program execute correctly in all function invocations is b in cons(choose(a,aa).head + choose(b,bb).head, r), but when b is chosen for choose(b,b.tail), b and bb become aliased, which causes the entanglement. After realizing this insight, it is becomes simple to replace the angels with deterministic code that works in all invocations.

In the angels that traverse the list a, one particular angel, choose(a,a.tail), is mostly disentangled, except for one invocation. This angel chooses whether to use a or aa as the value to pass into the next recursive invocation. After it is disentangled and forced to be the same value as the rest of the invocations of the syntactic angel expression—most likely the desired value since it makes the angel constant—it breaks the original set of entangled angels into two. Observing that making an angel constant removes entanglement is a heuristic clue that we have removed many undesirable communications between angels. The remaining entangled angels can be refined easily since only local reasoning needs to be made.

8. Related Work

In concept analysis [18], there exists a finite set of attributes and a finite set of objects which have these attributes. A partial order is defined using the subset relation on sets of objects and sets of attributes. This partial order induces the concept lattice which shows a hierarchy of object and attribute clusters. Ammons used concept analysis to cluster traces used specification mining [3]. At first, it looks like concept analysis can be applied to entanglement. But because of the lack of monotonicity in entanglement, concept analysis was inapplicable in our setting. Specifically, if a trace is added to a set of traces, then the FEP of this new set could be finer, coarser or remain the same.

Others have proposed statistical methods that find properties inherent to a program. Specification mining [2] uses machine learning and observations of program executions to create a state machine which represents implicit dependencies. Another related problem is the so called light bulb problem [13] posed by Valiant that again tries to find objects whose attributes are statistically correlated. Using a statistical approach to find entanglement would reduce the number of traces needed to be seen, but would not be able to detect the complete set of correlations as shown by our tool. By leveraging the complete set of safe traces, we are able to show all correlations which exists in the angelic program, along with possible ways of breaking these correlations.

Additionally, there has been much work done in trace analysis and in trace clustering. In most of this work, traces are debug statements of a program which give insight into the runtime state of the program. The work that has been done in trace analysis usually detects anomalies, eliminates redundant traces [7], or clusters similar traces together [11].

The problem posed as FEP is closely related to boolean formula decomposition. If each angel ranges over boolean values, then each trace is essentially represents a solution for some boolean formula. FEP poses the question: is it possible to break this formula into a conjunction of two formulas with non-overlapping variables? The decomposition problem is

more difficult since it requires the synthesis of formula for each conjunct. We instead solve for the simpler question, can two variables exist in different conjuncts?

Others have proposed different uses of angelic nondeterminism during program development. Floyd [8] used it as an executable construct, much like our use. Back and von Wright [4] used angelic nondeterminism for problem decomposition. Angels are used as substitutes which satisfy intermediate conditions which the programmer could fill in later on in the development process. Celiku and Wright [6] refined angelic nondeterminism to demonic in order to refine each independently.

The work in this paper builds off the programming methodology described in [5] and can be viewed as another step in the evolution of the SKETCH project. In SKETCH [15, 14], the programmer leaves holes in the program, which are later filled in with a synthesizer. One of the limitations of SKETCH is that these holes can be filled in with a limited set of expressions. Angelic programming aims to tackle this criticism by allowing more user-defined choices for expressions.

9. Conclusion

We have defined a notion of entanglement in angelic programs that formalizes the idea of angelic communication: i.e. whether choose statements in an angelic program somehow must collaborate to produce a safe trace. Entanglement analysis helps a programmer triage a large set of traces into subsets that might correspond to plausible algorithms, and those that might contain too much spurious angelic communication. We have also developed two novel algorithms for efficient answering of entanglement queries. We have embedded these algorithms in an interactive tool, and have demonstrated its usefulness in carrying out refinement steps in angelic development methodology. Our experience on two case studies has been very positive, especially when compared to handling the same programming problems without the aid of the entanglement tool.

References

[1] G. Alexe, S. Alexe, Y. Crama, S. Foldes, P. L. Hammer, and B. Simeone. Consensus algorithms for the generation of all maximal bicliques. *Discrete Appl. Math.*, 145(1):11–21, 2004.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[3] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. *SIGPLAN Not.*, 38(5):182–195, 2003.

[4] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 1998.

[5] R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 339–352, New York, NY, USA, 2010. ACM.

[6] O. Celiku and J. von Wright. Implementing angelic nondeterminism. In *APSEC*, pages 176–185. IEEE Computer Society, 2003.

[7] M. Diep, S. Elbaum, and M. Dwyer. Trace normalization. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 67–76, Washington, DC, USA, 2008. IEEE Computer Society.

[8] R. W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, oct 1967.

[9] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. Simple inductive synthesis methodology and its applications. In *SPLASH 2010 (to appear)*, 2010.

[10] J. Li, G. Liu, H. Li, and L. Wong. Maximal biclique subgraphs and closed pattern pairs of the adjacency matrix: A one-to-one correspondence and mining algorithms. *IEEE Trans. on Knowl. and Data Eng.*, 19(12):1625–1637, 2007.

[11] A. V. Miranskyy, N. H. Madhavji, M. S. Gittens, M. Davison, M. Wilding, and D. Godwin. An iterative, multi-level, and scalable approach to comparing execution traces. In *ESEC-FSE '07*, pages 537–540, New York, NY, USA, 2007. ACM.

[12] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[13] S. Rajasekaran, J. Reif, R. P. Sanguthevar, and R. Paturi. The light bulb problem, 1989.

[14] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 167–178, New York, NY, USA, 2007. ACM.

[15] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGPLAN Not.*, 41(11):404–415, 2006.

[16] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, 2010.

[17] E. Torlak and D. Jackson. Kodkod: a relational model finder. In *TACAS'07: Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 632–647, Berlin, Heidelberg, 2007. Springer-Verlag.

[18] R. Wille. Restructuring lattice theory: an approach based on hierarchies of concepts. In I. Rival, editor, *Ordered sets*, pages 445–470, Dordrecht–Boston, 1982. Reidel.