A Sparse Coding Method for Specification Mining and Error Localization

Wenchao Li
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-163
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-163.html

December 25, 2011

Copyright © 2011, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. This work was also supported in part by a Hellman Family Faculty Fund Award.

A Sparse Coding Method for Specification Mining and Error Localization

Wenchao Li and Sanjit A. Seshia

University of California at Berkeley

Abstract. Formal specifications play a central role in the design, verification, and debugging of systems. We consider the problem of mining specifications from simulation or execution traces of reactive systems with a special focus on digital circuits. We propose a novel sparse coding method that can extract specifications in the form of a set of *basis subtraces*. For a set of finite subtraces each of length p, we introduce the *sparse Boolean basis problem* as the decomposition of each subtrace into a Boolean combination of only a small number of *basis subtraces* of the same dimension. The contributions of this paper are (1) we formally define the *sparse Boolean basis problem* and propose a graph-based algorithm to solve it; (2) we demonstrate that we can mine useful specifications using our sparse coding method; (3) we show that the computed bases can be used to do simultaneous error localization and error explanation in a setting that is especially applicable to post-silicon debugging.

1 Introduction

Formal specifications play a central role in system design. They can serve as abstract models from which a system is to be synthesized. They can encode key properties that the system must and must not exhibit, and thus find use in formal verification, testing and simulation. Additionally, formal specifications are valuable as contracts for use in code maintenance. Finally, they are also useful in debugging and error localization: e.g., if relevant assertions widely cover the program or design, a failing assertion can be used to localize the source of the bug. In this paper, we are mainly concerned with the use of formal specifications for error localization and debugging in *reactive systems*.

Unfortunately, in practice, comprehensive formal specifications are rarely written by human designers. It is more common to have instead a comprehensive test suite used during simulation or testing. There has therefore been much interest in automatically deriving specifications from simulation or execution traces (e.g. [12, 4]). It is important to note that, since traces are rarely exhaustive, the properties generated from traces are only *likely* specifications or *behavioral signatures* of a design.

Different kinds of formal specifications provide different tradeoffs in terms of ease of generation from traces, generality, and usefulness for error localization. Büchi automata provide a very general formalism, and are typically inferred by learning a finite automaton from finite-length traces and interpreting it over infinite traces. However, such automata tend to "overfit" the traces they are mined from, and do not generalize well to unseen traces — i.e., they are very sensitive to the choice of traces $\mathcal{T}$ they are

mined from and can easily exclude valid executions outside of the set $\mathcal{T}$. Linear temporal logic (LTL) formulas are an alternative. One typically starts with templates for common temporal logic formulas and learns LTL formulas that are consistent with a set of traces. If the templates are chosen carefully, such formulas can generalize well to unseen traces. However, the biggest challenge is in coming up with a suitable set of templates that capture all relevant behaviors.

In this paper, we introduce a third kind of formal specification, which we term as *basis subtraces*. To understand the idea of a subtrace, consider the view of a trace as a two-dimensional table, where one dimension is the space of system variables and the other dimension is time. A *subtrace* is a finite window, or a snapshot, of a trace. Thus, just as a movie is a sequence of overlapping images, a trace is a sequence of overlapping subtraces. Restricting ourselves to Boolean variables, each subtrace can be viewed as a binary matrix. Given a set of finite-length traces, and an integer p, the traces can be divided into subtraces of time-length p. The set of all such subtraces constitutes a set of binary matrices. The basis subtraces are simply a set of subtraces that form a basis of the set of subtraces, in that every subtrace can be expressed as a superposition of the basis subtraces.

The form of superposition depends on the type of system being analyzed. In this paper, we focus on digital systems, and more concretely on digital circuits. In this context, one can define superposition as a "linear" combination over the semi-ring with Boolean OR as the additive operator and Boolean AND as the multiplicative operator. The coefficients in the resulting linear combination are either 0 or 1. The problem of computing a basis of a set of subtraces turns out to be equivalent to a Boolean matrix factorization problem which decomposes a Boolean matrix into the product of two Boolean matrices. If we seek the basis of the smallest size, the problem is equivalent to finding the *ambiguous rank* [13] of the Boolean matrix, which is known to be NP-complete [28].

Given a set of subtraces, several bases are possible. Following Occam's Razor principle, we seek to compute a "simple" basis that generalizes well to unseen traces. More concretely, we seek to find a basis that is minimal in that each subtrace is a linear combination of only a small number of basis subtraces. This yields the *sparse basis problem*. In this paper, we formally define this problem in the context of Boolean matrix factorization and propose a graph-theoretic algorithm to solve the sparse-version of the problem. Such a problem is often referred to as a *sparse coding* problem in the machine learning literature, since it involves encoding a data set with a "code" in a sparse manner using few non-zero coefficients.

We apply the generated basis subtraces to the problem of error localization. In digital circuits, an especially vexing problem today is that of post-silicon debugging, where, given an error trace with potentially only a subset of signals observable and no way to reproduce the trace, one must localize the problem in space (to a small collection of error modules) and time (to a small window within the trace). Error localization is of course very relevant to "pre-silicon" verification as well. Our approach is to attempt to reconstruct k-windows of an error trace using a basis computed from slicing a set of good traces into subtraces of length p. The hypothesis is that the earliest windows that cannot be reconstructed are likely to indicate the time of the error, and the portions that cannot be reconstructed are likely to indicate the signals (variables) that are the source

of the problem. The technique can thus be applied for *simultaneous* error localization and explanation. We apply this technique to representative digital circuits.

To summarize, the main contributions of the paper are:

- We introduce the idea of *basis subtraces* as a formal way of capturing behavior of a design as exhibited by a set of traces;
- We formally define the *sparsity-constrained Boolean basis problem* and propose a graph-based algorithm to solve it.
- We demonstrate with experimental results that we can mine useful specifications using our sparse coding method;
- We show that the computed bases can be used to do simultaneous error localization and error explanation in a setting that is especially applicable to post-silicon debugging.

The rest of the paper is organized as follows. We begin in Sec. 2 with basic terminology and preliminaries. Sec. 3 introduces our approach to finding a sparse basis. In Sec. 4, we show how we can use our approach for performing error localization. Experimental results are presented in Sec. 5. Related work is surveyed in Sec. 6 and we conclude in Sec. 7.

2 Preliminaries

In this section, we introduce basic notation used in the rest of the paper. Sec. 2.1 introduces notation representing traces of a reactive system as matrices, and Sec. 2.2 connects the matrix representation with a graph representation.

2.1 Traces and Subtraces

We model a reactive system as a transition system (V, Σ_0, δ) where V is a finite set of Boolean variables, Σ_0 is a set of initial states of the system, and δ is the transition relation. In general V contains input, output and (internal) state variables. A state of the system σ is a Boolean vector comprising valuations to each variable in V. For clarity, we restrict ourselves in this paper to synchronous systems in which transitions occur on the tick of a clock, such as digital circuits, although the ideas can be applied in other settings as well.

Let the state of the system at the ith cycle (step) be denoted by σ_i. A *complete trace* of the system of length l is a sequence of states $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{l-1}$ where $\sigma_0 \in \Sigma_0$, and $(\sigma_{i-1}, \sigma_i) \in \delta$ for $1 \leq i < l$. Note however that the full system state and/or inputs might not be observed or recorded during execution. We therefore define a *trace* τ as a sequence of valuations to a subset of the variables in V; i.e., $\tau = \sigma'_0, \sigma'_1, \sigma'_2, \ldots, \sigma'_{l-1}$ where $\sigma'_i \subseteq \sigma_i$. A *subtrace* $\tau_{i,j}$ of length j in τ is defined as the segment of τ starting at cycle i and ending at cycle $i + j - 1$, such that $i \geq 0$ and $i + j \leq l$, i.e. $\tau_{i,j} = \sigma_i, \sigma_{i+1}, \ldots, \sigma_{i+j-1}$. We only consider subtraces of length at least 2; i.e., containing at least one transition.

For example, Equation 1 shows a trace τ of length 4 where each state comprises a valuation to two Boolean variables. We depict the trace in matrix form, where the rows correspond to variables and the columns to cycles.

$$\begin{matrix} 1\ 0\ 1\ 1 \\ 1\ 0\ 1\ 1 \end{matrix} \tag{1}$$

The subtrace $\tau_{0,2}$ of τ is

$$\begin{matrix} 1\ 0 \\ 1\ 0 \end{matrix}$$

Let $\mathcal{T}_p$ be the set of all subtraces of length p in τ, i.e. $\mathcal{T}_p = \{\tau_{i,p} | 0 \leq i \leq l - p\}$. For any $\tau_{i,p} \in \mathcal{T}_p$, we can view it as a Boolean matrix of dimension $|V| \times p$. We can also represent it using a vector $v_i^p \in \mathbb{B}^{|V| \times p}$ by stacking the columns in $\tau_{i,p}$ (i.e., using a column-major representation). For example, v_0^2 as shown below represents the subtrace $\tau_{0,2}$.

$$v_0^2 = \begin{bmatrix} 1\ 1\ 0\ 0 \end{bmatrix}^T$$

For brevity, we use v_i for v_i^p when the length of each subtrace p is obvious from the context. Hence, we can represent $\mathcal{T}_p$ as a Boolean matrix with $|V| \times p$ rows and $l - p + 1$ columns. For example, we can represent all the subtraces of length 2 for the trace in Equation 1 as the following matrix.

$$\begin{bmatrix} 1\ 0\ 1 \\ 1\ 0\ 1 \\ 0\ 1\ 1 \\ 0\ 1\ 1 \end{bmatrix} \tag{2}$$

2.2 Boolean Matrices and Bipartite Graphs

A Boolean matrix can be viewed as an adjacency matrix for a *bipartite graph* (*bigraph*, for short). Recall that a bipartite graph $G = \langle U, V, E \rangle$ is a graph with two disjoint non-empty sets of vertices U and V and such that every edge in $E \subseteq U \times V$ connects one vertex in U and one in V. For a Boolean matrix $M \in \mathbb{B}^{k_1 \times k_2}$, denote $M_{i,j}$ as the entry in the i^{th} row and j^{th} column of M. Then, M can be represented by a bigraph G_M with $U = \{u_1, u_2, \ldots, u_{k_1}\}$ and $V = \{v_1, v_2, \ldots, v_{k_2}\}$, such that there is an edge connecting $i \in U$ and $j \in V$ if and only if $M_{i,j} = 1$. For example, the matrix X in Equation 2 can be represented by the bigraph G_X in shown in Figure 1.

A *biclique* is a complete bipartite graph; i.e., a bipartite graph $G' = \langle U', V', E' \rangle$ where $E' = U' \times V'$. Given a bigraph G, a *maximal edge biclique* of G is a biclique $G_1 = \langle U_1 \subseteq U, V_1 \subseteq V, E_1 = U_1 \times V_1 \rangle$ if it is not contained in another biclique of G, that is, there does not exist another biclique $G_2 = \langle U_2 \subseteq U, V_2 \subseteq V, E_2 = U_2 \times V_2 \rangle$ and either $U_1 \subset U_2$ or $V_1 \subset V_2$. A *biclique edge cover* Cov of G is a set of bicliques such that all the edges E in G are covered by the set, that is, $\forall e \in E, \exists\, G' = \langle U' \subseteq U, V' \subseteq V, E' \rangle \in Cov,$ s.t. $e \in E'$. Denote E_{Cov} as the set of edges covered by Cov. The smallest number of bicliques needed is called the *bipartite dimension* of G. For example, a biclique cover for the bigrah in Figure 1 is shown in Figure 2.

The view of Boolean matrices as bigraphs is relevant for decomposing a set of traces into a set of basis subtraces. The following problem is important in this context.

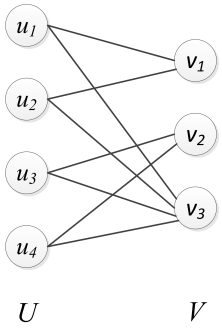

Fig. 1. Bipartite graph for the matrix in Equation 2

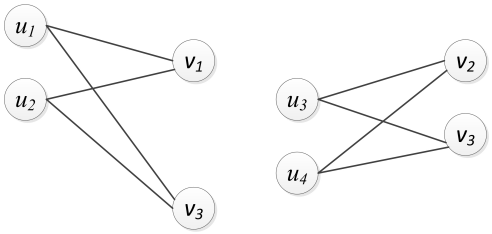

Fig. 2. Biclique edge cover for the bigraph in Figure 1

Definition 1. *Consider a Boolean matrix $X \in \mathbb{B}^{m \times n}$, the* Boolean matrix factorization problem *is to find k and Boolean matrices $B \in \mathbb{B}^{m \times k}$ and $S \in \mathbb{B}^{k \times n}$ such that*

$$X = B \circ S \tag{3}$$

That is, X is decomposed into a Boolean combination (denoted by the operator $\circ$) of two other Boolean matrices, in which scalar multiplication is the Boolean AND operator $\wedge$, and scalar addition ("$+$") is the Boolean OR operator $\vee$. In other words, we perform matrix/vector operations over Boolean semi-ring with $\wedge$ as the multiplicative operator and $\vee$ as the additive operator. For example, the matrix in Equation 2 can be factorized in the following way.

$$\begin{bmatrix} 1\,0\,1 \\ 1\,0\,1 \\ 0\,1\,1 \\ 0\,1\,1 \end{bmatrix} = \begin{bmatrix} 1\,0 \\ 1\,0 \\ 0\,1 \\ 0\,1 \end{bmatrix} \circ \begin{bmatrix} 1\,0\,1 \\ 0\,1\,1 \end{bmatrix}$$

We use M_i to denote the i^{th} column vector of a matrix M. Thus, the columns of matrix X are $X_1, X_2, \ldots, X_n$. We will refer to X as the *data matrix* since it represents the traces which are the input data for error localization. We call the matrix B the *basis matrix* because each B_i can be viewed as some basis vector in $\mathbb{B}^m$. We call the matrix S the *coefficient matrix*. Each S_i is a Boolean vector in which a 1 in the j^{th} entry indicates that the j^{th} basis vector is used in the decomposition and 0 otherwise.

We can also rewrite the factorization in the following way as a Boolean sum of the matrices formed by taking the tensor (outer) product of the i^{th} column in B and the i^{th} row in S.

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Notice that the two matrices on the right hand side are essentially the bicliques in Figure 2.

Remark 1 *Clearly, a solution always exists for the problem in Definition 1. This is because one can always pick $k = n$ such that $B = X$ and $S = I$ (where I is the identity matrix). Hence, it is more interesting to find B and S such that k is minimized. The smallest k for which such a decomposition exists is called the ambiguous rank [13] of the Boolean matrix X. It is also equal to the bipartite dimension of the bigraph G_X corresponding to matrix X. The problem of finding a Boolean factorization of X with the smallest k is equivalent to finding a biclique edge cover of G_X with the minimum number of bicliques. Both problems are NP-hard [28].*

3 Specification Mining via Sparse Coding

In this section, we describe how specifications are mined via sparse Boolean matrix factorization. The specifications we mine, basis subtraces, can be viewed as temporal patterns over a finite time window.

3.1 Formulation as Sparse Coding Problem

The notion of sparsity is borrowed from the wealth of literature in machine learning such as sparse coding [20] and sparse principal component analysis (PCA) [33]. The key insight is that sparsity often times provides a better interpretation of the data in terms of the underlying concepts (albeit greater in number). In the setting of mining specification from a trace, we argue that each subtrace of a trace can be viewed as a superposition of patterns, and a potential specification is a pattern that is commonly shared by multiple subtraces. These patterns are the so-called *basis subtraces*.

We present the *sparse Boolean basis problem* for computing basis subtraces below. A few different options are presented for formulating the problem and we pick one with a notion of sparsity that seems well-suited to our context.

Definition 2. *Given $X \in \mathbb{B}^{m \times n}$ and a positive integer C, the* sparsity-constrained Boolean matrix factorization problem *is to find k, $B \in \mathbb{B}^{m \times k}$, and $S \in \mathbb{B}^{k \times n}$ such that*

$$\begin{aligned} X &= B \circ S \\ \textit{and} \quad |S_i|_1 &\leq C, \forall_i \end{aligned} \tag{4}$$

Let us reflect on the above problem formulation. The constraint $X = B \circ S$ imposes the requirement that the input data (subtraces) represented by X must be reconstructed as a superposition of the subtraces represented by B, with S encoding the coefficients in the superposition. The second constraint $|S_i|_1 \leq C, \forall_i$ encodes the sparsity constraint, which ensures that each subtrace in X is a sparse superposition of the subtraces in B.

More precisely, the definition above imposes a constraint on the number of 1s per column of S. Similar to the Boolean matrix factorization problem in Definition 1, a solution always exists by setting $B = X$ and $S = I$ (and $k = n$). Clearly, one must restrict the size of k in order to get meaningful results for B and S. We therefore consider the variant of the above problem that seeks the smallest k, terming this as the *sparsity-constrained Boolean basis problem*. In this variant, sparsity is defined both in terms of the restriction on the S matrix and on the size of k. We describe how we address this problem in Section 3.2. Note that, if we allow C to be a function of k, then this problem is also NP-hard since the *Boolean matrix basis problem* is a special case of this when $C = k$.

One might also consider defining sparsity in a somewhat different manner. Instead of imposing a L_1-norm constraint on the columns of the coefficient matrix S, we can seek B and S such that the total sparsity is minimized.

Definition 3. *Given $X \in \mathbb{B}^{m \times n}$ and a positive integer k, the* sparsity-optimized Boolean matrix factorization problem *is the following optimization problem.*

$$\begin{aligned} \underset{B,S}{\textit{minimize}} \quad & \sum_{i}^{n} |S_i|_1 \\ \textit{subject to} \quad & X = B \circ S \end{aligned} \tag{5}$$

The main issue with this problem definition is that k is fixed; in other words, one has to "guess" a suitable k for which B and S can be computed. While modifications of this problem that restrict or minimize k could potentially be useful, we leave an investigation of these to future work.

3.2 Solving the Sparse Coding Problem

In this section, we describe an algorithm that solves the *sparsity-constrained Boolean matrix factorization problem* and tries to minimize k. Our solution is guaranteed to satisfy the sparsity constraint but may not be minimal in terms of k. The algorithm exploits the connection between the matrix factorization problem and the biclique edge cover problem described in Sec. 2. Specifically, it is based on growing a biclique edge cover Cov for the bigraph $G_X = \langle U, V, E \rangle$ corresponding to matrix X. At each step, a maximal edge biclique that covers some number of previously uncovered edges is added to Cov until Cov covers all the edges. The sparsity constraint is then a constraint on the number of maximal bicliques that can be used to cover the edges that connect each vertex in V. (Recall that each vertex in V corresponds to a column S_i of S.)

Notice that this algorithm implicitly assumes that the set of all maximal edge bicliques are given *a priori*. Computing this set is not easy: for instance, the closely-related problem of finding a maximum edge biclique in a bigraph is NP-complete [26].

Additionally, the number of maximal bicliques in a bigraph can be exponential in the number of vertices [15], and so enumerating these is worst-case exponential in the number of vertices. However, there do exist enumeration algorithms that are polynomial in the combined input and output size [2]. We use an off-the-shelf implementation [1] of the enumeration algorithm in [2] to compute the set of all maximal bicliques. We have found that in practice the high complexity of deriving this set does not prevent us from finding a good solution.

We now formally describe the algorithm which computes Cov, the biclique edge cover.

(1) Initialize Cov to $\emptyset$. Associate each vertex $v \in V$ with a positive integer count α_v and initialize each α_v to C. Intuitively, α_v will ensure that the sparsity constraint for the column of S corresponding to v is maintained.
Denote by E_v the set of edges that connect a vertex $v \in V$. Let $E_{v'} = E_v \setminus E_{Cov}$ and $G_{v'}$ be the subgraph that is induced by $E_{v'}$.
(2) Given the set of maximal bicliques Γ, for each maximum biclique $G_i = \langle U_i, V_i, E_i \rangle \in \Gamma$, associate with it a value β_i that is equal to the number of edges in E_i but not in E_{Cov}. In other words, $\beta_i = |\{e_i | e_i \in E_i \setminus E_{Cov}\}|$. Initially, $\beta_i = |U_i| \times |V_i|$.
(3) Iterate until $E_{Cov} = E$:
   (a) $\forall v \in V$ such that $\alpha_v = 1$ and $E_{v'} \neq \emptyset$, Add $G_{v'}$ to Cov. Update the β values for all $G_i \in \Gamma$.
   (b) Add a $G_i \in \Gamma$ that has the highest positive β_i and satifies the condition $\forall v_i \in V_i, \alpha_{v_i} > 1$ to Cov. In case of a tie, pick G_i that has the largest $|U_i|$. Remove G_i from Γ and update the β values for the rest to the bicliques in the set. Decrease all α_{v_i} by 1.

The intuition for Step 3(a) is that if we have a vertex v with uncovered edges whose sparsity count α_v has reached 1, we must cover it (with the default biclique that includes all the uncovered edges). In 3(b), we have the general case, where there exists a maximal biclique with vertices with sparsity counts greater than 1, in which case we use a greedy heuristic based on the β value. The theoretical guarantees provided by our algorithm are as follows.

Proposition 1. *The above algorithm terminates in a finite number of steps.*

Proof. (sketch) It is clear that at each step of the iteration, at least one new biclique that contains an extra edge is added to the edge cover. Since the number of edges is finite, the algorithm will terminate in a finite number of steps. □

Proposition 2. *The above algorithm finds a biclique edge cover for the bigraph G.*

Proof. (sketch) Since the termination condition of the loop is when the set E_{Cov} contains all the edges in G, Cov at all times contains bicliques, and the loop terminates in a finite number of steps, the algorithm computes a biclique edge cover for G. □

Proposition 3. *The above algorithm solves the sparsity-constrained Boolean matrix factorization problem.*

Proof. (sketch) Recall that the i^{th} biclique G_i added to Cov corresponds to the i^{th} column in the basis matrix B multiplied with the i^{th} row in the coefficient matrix S. Also note that the cardinality of the biclique cover is equal to the number of bases which is k. Since X can be written as a Boolean sum of the matrices formed by multiplying the i^{th} column of B with the i^{th} row of S, we essentially solves the Boolean matrix factorization problem by finding a biclique edge cover Cov of G_X. The sparsity constraint for the j^{th} column of S is then the maximum number of bicliques in Cov that can contain the vertex $v_j \in V$. Since each time when a biclique containing v_j is added to Cov, the count at v_j is decreased by 1 and it never goes below 0, the sparsity contraint is satisfied. Thus, the algorithm solves the *sparsity-constrained Boolean matrix factorization problem*. □

4 Application to Error Localization

The key idea in our approach is to localize errors by attempting to reconstruct the error trace from basis subtraces generated from correct traces. Our hypothesis is that the earliest section (subtrace) of the error trace that cannot be reconstructed contains the likely cause of the error. Our localization algorithm is presented in this section, along with some theoretical guarantees. We begin with the problem definition.

4.1 Problem Definition

Consider the problem of localizing an error given a set of correct traces and a single error trace. Our goal is to identify a small interval of the timeline at which the error occurred. What makes the problem especially challenging is that the input sequence that generated the error trace is either unknown (or only partially known) or it is extremely slow to re-simulate the input sequence (if known) on the correct design (also sometimes referred to as a "golden model"). This means that a simple anomaly detection technique which checks the first divergence of the error trace and the correct trace obtained by simulating the golden model on the same input sequence does not work. One has to use the set of correct traces to help localize the bug in the error trace. This setting is especially applicable to post-silicon debugging where the bugs are often difficult to diagnose due to limited observability, limited reproducibility and susceptibility to environmental variations.

More formally, the error localization problem we address in this section can be defined as follows.

Definition 4. *Given an error trace of length l, and an integer p, consider partitioning it into non-overlapping subtraces each of length p (without the loss of generality, we assume l is an integer multiple of p; otherwise, the last subtrace can be treated specially).*

Then, the error localization problem *is to identify the subtrace containing the first point of deviation of the error trace from the correct trace on the same input sequence.*

Figure 3 illustrates the problem.

One might note that the problem we define is not the only form of error localization that is desirable. For instance, one might also want to narrow down the fault to the signals/variables that were incorrectly updated. Also, there might be more than one source

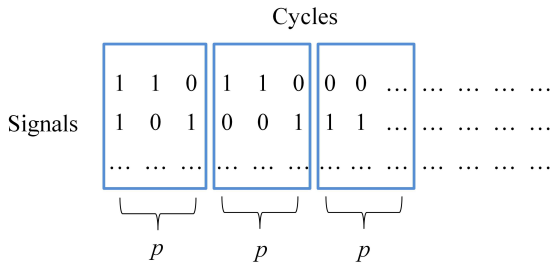

Fig. 3. Localizing the error to one of the subtraces of length p

of an error, in which case one might want to identify all of the sources. While these goals are important, we contend that our algorithm to address the problem defined above can also be used to achieve these additional objectives. For example, the error explanation technique we present below can be used to identify which variables were incorrectly updated and how. Similarly, one can apply our reconstruction-based localization algorithm iteratively to identify multiple subtraces that cannot be reconstructed from the basis subtraces, and could potentially be used to identify multiple causes of an error.

4.2 Localization by Reconstruction

As described above, the key hypothesis underlying our approach is that the earliest section (subtrace) of the error trace that cannot be reconstructed contains the likely cause of the error.

Our error localization algorithm operates in the following steps:

1. Given a set of correct traces $\mathcal{T}$, first obtain the set of all *unique* subtraces of length p in $\mathcal{T}$. Denote this set by $\mathcal{T}_p$. Using the approach described in Section 2, convert the set $\mathcal{T}_p$ to a data matrix X.
2. Solve the *sparsity-constrained Boolean basis problem* for X for a given constant C.
3. Given an error trace τ', partition it into an ordered set of q subtraces of length p. Denote this set by $\mathcal{T}'_p$. The elements in $\mathcal{T}'_p$ are ordered by their positions in τ'. Convert $\mathcal{T}'_p$ to a data matrix X'.
4. Starting from X'_0, try to reconstruct X'_i using the basis computed above with the same sparsity constraint C. Return i as the location of the bug if the reconstruction fails. In case all reconstructions succeed, return $\perp$ indicating inability to localize the error.

Algorithm 1 describes the above approach in more detail using pseudo-code. It uses the following subroutines:

– **dataMatrix** is the procedure that converts a set of subtraces to the corresponding data matrix described in Section 2.

Algorithm 1 Error localization in time

Input: Set of subtraces $\mathcal{T}_p$ from set of correct traces $\mathcal{T}$, $\mathcal{T}'_p$ from error trace τ'
Input: Constant $C > 0$
  $X =$ **dataMatrix**$(\mathcal{T}_p)$
  $X' =$ **dataMatrix**$(\mathcal{T}'_p)$
  $B =$ **sparseBasis**(X, C)
  for $i := 0 \rightarrow q - 1$ **do**
    E = **reconstructTrace**(X'_i, B, C)
    if $E \neq$ **0 then**
      return i
    end if
  end for
  return $\perp$

– **sparseBasis** solves the *sparsity-constrained Boolean basis problem* using the graph-theoretic algorithm presented in Section 3 for X with a given C, and returns the computed basis B.
– **reconstructTrace** solves the following minimization problem.

$$\begin{aligned} \underset{S_i}{\text{minimize}} \quad & |X'_i \oplus (B \circ S_i)|_1 \\ \text{subject to} \quad & |S_i|_1 \leq C \end{aligned} \tag{6}$$

where $\oplus$ is the bit-wise Boolean XOR operator, and is interpreted to apply entry-wise on matrices.
Notice that for fixed C, this problem is *fixed-parameter tractable* because we can use a brute-force algorithm that enumerates all the $\sum_{1 \leq i \leq C} \binom{k}{i}$ possible S_is. It can also be solved using a pseudo-Boolean optimization formulation, where the Boolean variables in the optimization problem are the entries in S_i.

Error Explanation. Denote S_i^* as the optimal solution to the minimization problem in Equation 6. If the minimum value is non-zero, then $E = X'_i \oplus (B \circ S_i^*)$ is the minimum difference between the error subtrace X'_i and the reconstructed subtrace $B \circ S_i^*$. Notice that E is also a subtrace, and can be interpreted as a finite sequence of assignments to system variables. In our experience, E is a pattern that explains the error; we expand further on this point using our experiments in Sec. 5.

4.3 Theoretical Guarantees

We now give conditions under which our error localization approach is *sound*. By sound, we mean that when our algorithm reports a subtrace as the cause of an error, it is really an erroneous subtrace that deviates from correct behavior.

Since our approach mines specifications from traces, its effectiveness fundamentally depends on the quality of those traces. Specifically, our soundness guarantee relies on the set of traces $\mathcal{T}$ satisfying the following *coverage metrics* defined over the transition system (V, Σ_0, δ) of the golden model:

1. *Initial State Coverage:* For every initial state $\sigma_0 \in \Sigma_0$, there exists some trace in $\mathcal{T}$ in which σ_0 is the initial state.
2. *Transition Coverage:* For every transition $(\sigma, \sigma') \in \Sigma_0$, there exists some trace in $\mathcal{T}$ in which the transition (σ, σ') occurs.

While full transition coverage can be difficult to achieve for large designs, there is significant work in the simulation-driven hardware verification community on achieving a high degree of transition coverage [29]. If achieving transition coverage is challenging for a design, one could consider slicing the traces based on smaller module boundaries and computing tests that ensure full transition coverage within modules, at the potential cost of missing cross-module patterns.

Our soundness theorem relates test coverage with effectiveness of error localization.

Theorem 1. *Given a transition system Z for the golden model and a set of finite-length traces $\mathcal{T}$ of Z satisfying initial state and transition coverage, if Algorithm 1 is invoked on $\mathcal{T}$ and an arbitrary error trace τ', then Algorithm 1 is sound; viz., if it reports a subtrace of τ' as an error location, that subtrace cannot be exhibited by Z.*

Proof. (sketch) The proof proceeds by contradiction. Suppose Algorithm 1 reports a subtrace of τ' as the location of the error. Recall that a subtrace must be of length at least 2. Thus, if we compute basis subtraces of length 2, any transition of the golden model Z can be expressed as a superposition of these basis subtraces and hence reconstructed from the basis subtraces B, since $\mathcal{T}$ contains all transitions of Z. A subtrace reported as an error location, in contrast, is one that cannot be expressed as a superposition of the basis subtraces and hence **reconstructTrace** will report that it cannot be reconstructed. Thus, any subtrace reported as an error location by Algorithm 1 cannot be a valid transition of the golden model Z. □

We also note that, in theory, it is possible for Algorithm 1 to miss reporting a subtrace that is an error location, if that subtrace is expressible as a superposition of basis subtraces. However, experiments indicate that it is usually accurate in pinpointing the location of the error. Details of our experiments are provided in Sec. 5.

5 Experimental Results

In this section, we evaluate our sparse coding approach to generate specifications and localize error based on the following criteria.

(1) Are the computed "basis subtraces" meaningful? That is, do they correspond to some interesting specifications of the test circuit.
(2) Do the "basis subtraces" capture sufficient underlying struture of a trace? That is, can they be used to reconstruct traces that are generated from unseen input sequences?
(3) How accurately can we localize an error in an unseen trace (generated by unseen input sequences)?
(4) How good are the error explanations?

We first use a 2-port arbiter as an illustrative example to evaluate our approach. The 2-port arbiter is an arbiter that takes two Boolean inputs corresponding to two potentially competing requests, and produces two Boolean outputs corresponding to the two grants. It implements a round-robin scheme such that it will give priority to the port at which a request has not been most recently granted. Let r_0, r_1 denote the input requests and g_0, g_1 denote the corresponding output grants. Figure 4 shows part of a trace of the arbiter over the request and grant signals. The input requests were randomly generated and the trace was 100 cycles long.

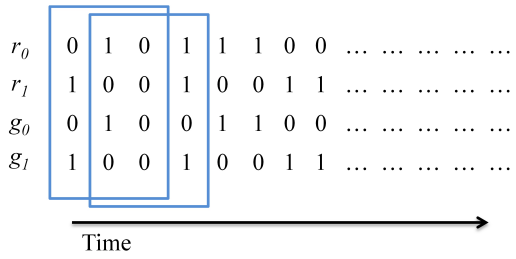

Fig. 4. A normal trace of a 2-port round-robin arbiter

We used a sliding window of length 3 to collect a set of subtraces. We then applied our sparse coding algorithm described in Section 3.2 to extract a set of "basis subtraces". We chose a sparsity of 4 for this experiment.

(1) Figure 5 shows some of the basis subtraces computed. We can observe that basis (a) and (b) correspond to the correct behavior of the arbiter granting a request at the same cycle when there is no competing request. Basis (c) shows that when there are two competing requests at the same cycle, the arbiter first grants one of the requests and the ungranted request will stay asserted the next cycle and then gets granted.

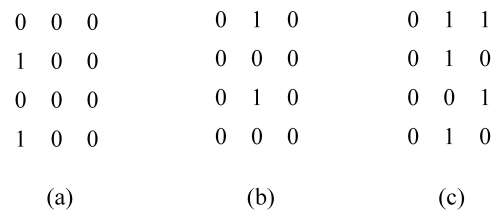

Fig. 5. Three basis subtraces computed via sparse coding

(2) We further simulated the arbiter with random inputs another 100 times each for 100 cycles. For each of these traces, we also use a sliding window to partition them into subtraces of length 3. Using the basis computed from Figure 4, we tried to

reconstruct these subtraces and succeeded in every attempt. This was because all the sub-behaviors were fully covered in the trace from which the bases were computed, even though unseen subtrace exists in the new traces.

(3) For each of the 100 trace in (2), we randomly injected a single bit error (flipping its value) at a random cycle to one of the four signals in the trace. Our task was to test if we could localize the error to a subtrace of length 3 that contained it. The traces used in the experiments can be found in `http://www.eecs.berkeley.edu/~wenchaol/arbiter.tar.gz`.

The following example illustrates one of the experiments. Figure 6 shows a snapshot of the trace in `err1.txt`.

	95	96	97	98	99
r_0	0	0	0	1	0
r_1	0	0	0	0	0
g_0	0	0	0	1	0
g_1	0	0	1	0	0

Fig. 6. Bit flip at r_1 at cycle 97

Using the approach described in Algorithm 1, the subtrace containing the error was correctly identified. Figure 7 shows the error subtrace.

	96	97	98
r_0	0	0	1
r_1	0	0	0
g_0	0	0	1
g_1	0	1	0

Fig. 7. Error subtrace as identified

Following Equation 6, Figure 8 shows the subtrace $X'_i \oplus (B \circ S_i)$ that minimizes $|X'_i \oplus (B \circ S_i)|_1$ and serves as an error explanation.

Clearly, this subtrace reveals the injected error. Behaviorally, the reqest at r_1 was not granted as it should had been. Note that multiple error explanations (solutions to the minimization problem in Equation 6) can exist. Figure 9 shows another error explanation subtrace produced for the example above.

	96	97	98
r_0	0	0	0
r_1	0	1	0
g_0	0	0	0
g_1	0	0	0

Fig. 8. Error explanation subtrace

	96	97	98
r_0	0	0	0
r_1	0	0	0
g_0	0	0	0
g_1	0	1	0

Fig. 9. Alternative error explanation subtrace

This subtrace does not directly reveal the injected bit flip. However, it still pinpoints the bug behaviorally – a grant was produced at g_1 when no corresponding request was made.

(4) In Section 4.2, we argue that the minimum difference between an error subtrace and any possible reconstructed subtrace using the computed basis can serve as an explanation for the error. In the 83 traces for which the error was correctly localized, the injected bit error was also uncovered by solving the optimization problem in Equation 6.

While these results are preliminary, they indicate that our sparse coding approach provides a completely new way to do specification mining and error localization.

6 Related Work

We survey related work along two dimensions: work on mining specifications from traces, and approaches to perform error localization.

6.1 Specification Mining

The study of automatically generating specifications goes back as early as 1974 [6][30]. Many techniques have been recently proposed to automatically reverse-engineer specifications from programs [27][14][4][12]. These specifications can be simple predicates or temporal specifications which specify the ordering of events, or rules of API usage. The generated specifications can then be used to formally verify a program's correctness, to assist in bug finding [31], or to detect malicious behaviors [7].

Many techniques seek to learn specifications dynamically from an execution trace (or a set of traces). Daikon [12] is one of the earliest tools that mine single-state in-

variants or pre-/post-conditions in programs. In contrast, we focus on mining (temporal) properties over a finite window for reactive (hardware) designs in this work. Most existing mining tools produce temporal properties in the form of automata. Automata-based techniques generally fall into two categories. The first class of methods learn a single complex specification (usually as a finite automaton) over a specific alphabet, and then extract simpler properties from it. For instance, Ammons et al. [4] first produce a probabilistic automaton that accepts the trace and then extract from it likely properties. However, learning a single finite state machine from traces is NP-hard [16]. To achieve better scalability, an alternative is to first learn multiple small specifications and then post-process them to form more complex state machines. Engler et al. [10] first introduce the idea of mining simple alternating patterns. Several subsequent efforts [31][32][14] built upon this work. For example, Javert [14] locates all instances of the alternating pattern $(a\ b)^*$ and a resource usage pattern $(a\ b^*\ c)^*$. The tool then composes these patterns into larger ones by using a set of inference rules. In previous work, we proposed a specification mining approach similar to Javert that focuses on patterns relevant for digital circuits [21] and showed how this can be applied to error localization. However, such approaches are limited by the set of patterns. The present work seeks to remove this limitation by inferring design-specific patterns in the form of basis subtraces.

Specifications can also be generated by reasoning about the program statically. For example, Alur et al. [3] proposes the use of predicate abstraction together with automata learning to automatically synthesize interface specifications for Java classes. Static and dynamic analyses complement each other. We refer the readers to [11] for a detailed comparison of the two techniques.

6.2 Error Localization

The problem of error localization and explanation has been much studied in literature, both in the software testing and model checking communities. Groce et al. [17, 18] present an approach based on distance metrics which, given a counterexample (error trace), finds a correct trace as "close" as possible to the error trace according to the distance metrics. Ball et al. [5] present an approach to localizing errors in sequential programs. They use a model checker as a subroutine, with the core idea to identify transitions of an error trace that are not in any correct trace of the program, and use this for error localization. Both of these approaches operate on error traces generated by model checking, and thus have full observability of the inputs and state variables. In contrast, in our context of post-silicon debugging, the error trace is only partially observed and not reproducible.

In the software testing community, researchers have attempted to use predicates and mined specifications to localize errors [22, 9]; however, these rely on human insight in choosing a good set of predicates/templates. In contrast, our approach automatically derives specifications in the form of basis subtraces, which can be seen as temporal properties over a finite window. Program spectra [19], which include computing profiles of program behavior such as summaries of the branches or paths traversed, have also been proposed as ways to separate good traces from error traces; however, these

techniques are of limited use for digital circuits since they rely on the path structure of sequential programs and give no guarantees on soundness.

In the area of post-silicon debugging (see [23] for a recent survey), the problem of error localization has received wide attention, but few solutions are available. The IFRA approach [24, 25], which has proved effective for processor cores, is based on adding on-chip recorders to a design to collect "instruction footprints" which are analyzed offline with some input from human experts. However, this approach relies heavily on knowledge of processor designs and is not easily extensible to other kinds of designs such as communication and interface logic. Li et al. [21] have proposed the use of mined specifications to perform error localization; however, this approach relies on human insight in supplying the right templates to mine temporal logic specifications and provides no guarantees on soundness. The Backspace [8] system addresses the problem of reproducibility by attempting to reconstruct one or more "likely" error traces by performing backwards reachability guided by recorded signatures of system state; such a system is complementary to the techniques proposed herein for error localization.

7 Conclusion and Future Work

In this paper, we have presented *basis subtraces*, a new formalism to capture system behavior from simulation or execution traces. We showed how to compute a *sparse* basis from a set of traces using a graph-based algorithm. We further demonstrated that the generated basis subtraces can be effectively used for error localization and explanation.

In terms of future work, we envisage two broad directions: improving scalability and applying the ideas to other domains. Since the Boolean basis problem and its sparse variants can be computationally expensive to solve, the scalability of the approach is somewhat limited. In this context, it would be interesting to use slightly different definitions of a basis (for example, using the field of rationals rather than the semi-ring we consider) so that the problem of computing a sparse basis is polynomial-time solvable.

Moreover, the ideas introduced in this paper should be be applicable beyond digital circuits to software, cyber-physical systems, and analog/mixed-signal circuits. Exploring these application domains could provide a rich source of problems for future work.

Acknowledgement

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. This work was also supported in part by a Hellman Family Faculty Fund Award.

References

1. Maximal biclique enumeration. http://genome.cs.iastate.edu/supertree/download/biclique/readme.html.
2. 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.*, 145:11–21, December 2004.

3. R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.
4. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
5. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, pages 97–105, 2003.
6. M. Caplain. Finding invariant assertions for proving programs. In *Proceedings of the international conference on Reliable software*, pages 165–171, 1975.
7. M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE'07*, pages 5–14, 2007.
8. F. M. de Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang. Backspace: Formal analysis for post-silicon debug. In *FMCAD*, pages 1–10, 2008.
9. N. Dodoo, L. Lin, and M. D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, 2003.
10. Engler, D. et al. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.
11. M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA*, pages 24–27, 2003.
12. Ernst, M. et al. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
13. V. Froidure. *Rangs des relations binaires, semigrollpes de relations non ambigues*. PhD thesis, June 1995.
14. M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *FSE*, pages 339–349, 2008.
15. S. Gaspers, D. Kratsch, and M. Liedloff. On independent sets and bicliques in graphs. In H. Broersma, T. Erlebach, T. Friedetzky, and D. Paulusma, editors, *Graph-Theoretic Concepts in Computer Science*, volume 5344 of *Lecture Notes in Computer Science*, pages 171–182. Springer Berlin / Heidelberg, 2008.
16. E. M. Gold. Complexity of automatic identification from given data. 37:302–320, 1978.
17. A. Groce. Error explanation with distance metrics. In *TACAS*, LNCS 2988, pages 108–122, 2004.
18. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Software Tools for Technology Transfer (STTT)*, 8(3):229–247, 2006.
19. M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Softw. Test., Verif. Reliab.*, 10(3):171–194, 2000.
20. H. Lee, A. Battle, R. Raina, and A. Y. Ng. Efficient sparse coding algorithms. In *In NIPS*, pages 801–808. NIPS, 2007.
21. W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. In *Proceedings of the Design Automation Conference (DAC)*, pages 755–760, June 2010.
22. B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.
23. S. Mitra, S. A. Seshia, and N. Nicolici. Post-silicon validation: Opportunities, challenges and recent advances. In *Proceedings of the Design Automation Conference (DAC)*, pages 12–17, June 2010.
24. S. Park and S. Mitra. Ifra: Instruction footprint recording and analysis for post-silicon bug localization in processors. In *DAC*, 2008.
25. S. B. Park, A. Bracy, H. Wang, and S. Mitra. Blog: Post-silicon bug localization in processors using bug localization graphs. In *DAC*, 2010.
26. R. Peeters. The maximum edge biclique problem is NP-complete. *Discrete Applied Mathematics*, 131(3):651–654, 2003.

27. S. Sankaranarayanan, F. Ivanči, and A. Gupta. Mining library specifications using inductive logic programming. In *ICSE*, pages 131–140, 2008.
28. D. J. Siewert. *Biclique covers and partitions of bipartite graphs and digraphs and related matrix ranks of 0,1 matrices*. PhD thesis, 2000.
29. S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 18(4):36–45, 2001.
30. B. Wegbreit. The synthesis of loop predicates. *Commun. ACM*, 17(2):102–113, 1974.
31. W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *TACAS*, pages 461–476, 2005.
32. Yang, J. et al. Perracotta: mining temporal api rules from imperfect traces. In *ICSE*, pages 282–291, 2006.
33. H. Zou, T. Hastie, and R. Tibshirani. Sparse principal component analysis. *Journal of Computational and Graphical Statistics*, 15:2006, 2004.