Synthesizing Data Structure Refinements from Integrity Constraints

Shankara Pailoor
University of Texas at Austin
Austin, Texas, USA
spailoor@cs.utexas.edu

Xinyu Wang
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA
xwangsd@umich.edu

Yuepeng Wang
University of Pennsylvania
Philadelphia, Pennsylvania, USA
yuepeng@seas.upenn.edu

Isil Dillig
University of Texas at Austin
Austin, Texas, USA
idillig@cs.utexas.edu

Abstract

Implementations of many data structures use several correlated fields to improve their performance; however, inconsistencies between these fields can be a source of serious program errors. To address this problem, we propose a new technique for automatically refining data structures from integrity constraints. In particular, consider a data structure $D$ with fields $F$ and methods $M$, as well as a new set of auxiliary fields $F'$ that should be added to $D$. Given this input and an integrity constraint $\Phi$ relating $F$ and $F'$, our method automatically generates a refinement of $D$ that satisfies the provided integrity constraint. Our method is based on a modular instantiation of the CEGIS paradigm and uses a novel inductive synthesizer that augments top-down search with three key ideas. First, it computes necessary preconditions of partial programs to dramatically prune its search space. Second, it augments the grammar with promising new productions by leveraging the computed preconditions. Third, it guides top-down search using a probabilistic context-free grammar obtained by statically analyzing the integrity checking function and the original code base. We evaluated our method on 25 data structures from popular Java projects and show that our method can successfully refine 23 of them. We also compare our method against two state-of-the-art synthesis tools and perform an ablation study to justify our design choices. Our evaluation shows that (1) our method is successful at refining many data structure implementations in the wild, (2) it advances the state-of-the-art in synthesis, and (3) our proposed ideas are crucial for making this technique practical.


Keywords: Programming Languages, Program Synthesis, Data structure refinement

ACM Reference Format:

1 Introduction

It is well known that implementations of most data types use auxiliary fields to improve performance. For example, a program that needs to perform frequent bidirectional look-ups may explicitly store two different mappings $M : r_1 \rightarrow r_2$ and $M^{-1} : r_2 \rightarrow r_1$ even though one can be derived from the other. Similarly, programmers may store the same information as both a linked list and a hash map to efficiently implement different types of functionality. As a final example, Figure 1 shows a class called SpdySession from the Netty project [5] that maintains two auxiliary fields called a1s and ars. Even though these fields can be derived from the core data structure called actStream, doing so would hurt the application’s performance.

While it is quite common to maintain such auxiliary states for performance reasons, correctly maintaining all related copies of the same data can be a source of bugs and security vulnerabilities. For instance, several prior efforts study violations of consistency requirements in data structures and propose techniques for mitigating them [12–15, 27, 28]. In addition, The Common Vulnerabilities and Exposures (CVE) database reports several security vulnerabilities that
are caused by violations of integrity constraints between their fields [1–4].

Motivated by this problem, this paper proposes a new synthesis-based approach for automatically refining implementations of data types given integrity constraints between their fields. In particular, given a data structure $D$ consisting of fields $F$ and methods $M$ as well as a set of new fields $F'$ to be added to $D$, our approach generates a new data structure $D'$ with fields $F \cup F'$ and a set of updated methods $M'$ such that (1) $D'$ is functionally equivalent to $D$, and (2) the implementation of $D'$ satisfies a user-provided integrity constraint $\Phi(F,F')$ that expresses consistency requirements between these fields. Specifically, our method takes as input a Java class $D$ and an integrity constraint (expressed in Java as a consistency checking function) and produces a new class $D'$ that is a refinement of $D$. Thus, programmers can first focus on implementing functionally correct software and then use our approach to automatically refine their implementation in a way that ensures consistency.

In order to solve this problem, our approach needs to synthesize code that correctly updates the new fields such that the integrity constraints are satisfied. However, this synthesis task is quite challenging for several reasons: First, since these new fields may be complex user-defined data structures, the update logic is often non-trivial and requires synthesizing arbitrary Java code, including loops or introduction of new program variables. Second, data structures often contain multiple methods that update the relevant fields; thus, the synthesis problem is not localized to a single function. Finally, the update logic may require invoking other functions from libraries or the surrounding code base; hence, the synthesizer needs to consider an enormous number of building blocks from which to generate code.

In this paper, we address these challenges using a modular instantiation of the counterexample-guided inductive synthesis (CEGIS) paradigm powered by a novel inductive synthesis engine. In particular, our method is modular in the sense that it decomposes the data structure refinement problem into independent synthesis subproblems over a single procedure. For each subproblem, we use an inductive synthesizer based on top-down enumerative search to find code snippets that satisfy a given set of (counter-)examples and then check correctness of the synthesized code using a verifier. However, to make inductive synthesis tractable in this setting, we leverage the following three key ideas:

1. **PCFG construction via static analysis**: Our method performs static analysis of the integrity checking function and the surrounding code base to identify important syntactic and semantic clues that are useful for guiding the underlying search procedure. In particular, our method leverages the results of static analysis to construct a probabilistic context free grammar (PCFG) and uses this PCFG to prioritize promising candidates during the search.

2. **Pruning via necessary preconditions**: Our approach dramatically prunes its search space using the notion of necessary preconditions. In particular, given a partial program $P$ with missing statements or expressions, our method computes a formula $\phi$ that must be satisfied by every completion of $P$. Thus, if this formula is falsified by any of the counterexamples, our method can prune the partial program without sacrificing completeness.

3. **Deduction-guided grammar augmentation**: In addition to pruning the search space, necessary preconditions are also useful for identifying promising code snippets that are likely to appear in the desired solution. In particular, our synthesis procedure augments the grammar with new productions so that promising code snippets need not be synthesized from scratch. Our method also updates probabilities in the PCFG in order to prioritize programs that use these promising code snippets.

Based on the above ideas, we designed and implemented a new system called $\text{Volt}$ for automatically synthesizing data structure refinements from integrity constraints. To evaluate $\text{Volt}$, we identified data structures with correlated fields in widely-used Java projects and used $\text{Volt}$ to derive the full data structure from a version without the auxiliary fields. Our evaluation shows that $\text{Volt}$ can successfully synthesize 92% of these refinements with an average running time of 264.2 seconds per benchmark. We also compare $\text{Volt}$ against two state-of-the-art synthesis tools for Java and show that $\text{Volt}$ dramatically outperforms these tools. Finally, we also evaluate three different ablations of $\text{Volt}$ and show that our proposed ideas are indeed necessary for making this approach practical.

In short, this paper makes the following key contributions:

- We propose automatically refining data structures from integrity constraints in order to correctly implement efficient data structures that utilize many correlated fields.
- We propose a modular algorithm that decomposes the overall data structure refinement task into procedure-level synthesis tasks.
- We propose a novel inductive synthesis algorithm that uses three key ideas, namely (1) PCFG construction using static analysis, (2) pruning via necessary preconditions, and (3) dynamic PCFG augmentation and update.
- We conduct an experimental evaluation on 25 real-world Java programs and compare our method against state-of-the-art synthesis/repair tools as well as against several ablations of $\text{Volt}$ itself.

## 2 Overview

In this section we provide a high level overview of our technique with the aid of the motivating example shown in Figure 1. The code snippet in Figure 1 belongs to a data structure called $\text{SpdySession}$ used in Netty [5], a framework for developing high-performance protocol servers in
Java. SpdySession tracks active streams using a ConcurrentHashMap called actStreams (line 2) where keys correspond to stream identifiers and values are the stream states (encapsulated in another data structure called StreamState).

In one of the Github commits, Netty developers added a new method called numActiveStreams (line 11) to the class SpdySession. This new method takes as input a boolean argument called remote and returns the number of remote or local streams depending on the value of the boolean. A naive implementation of numActiveStreams would iterate over the hashmap (actStreams) to count the number of remote/local streams; however, this is obviously quite inefficient. Thus, Netty developers add two new fields called als and ars to keep track of the number of local and remote streams respectively and modify the implementation of SpdySession to correctly update these fields. In particular, the commented out (green) lines in Figure 1 are the new code added to the original code.

**Volt usage scenario.** Our proposed approach is useful in a scenario like this for automatically updating the implementation of SpdySession in a correct-by-construction manner. To use Volt, the user only needs to provide the original implementation of SpdySession, the names of the new fields (ars and als), and an integrity constraint expressed as a function called check in Figure 2. This check function iterates over the actStream hashmap and counts the number of remote and local streams. If the remote (resp. local) stream count is equal to ars (resp. als), check returns true; otherwise, it returns false. Given this integrity checking function, Volt automatically generates the refined data structure in under one minute. The new implementation involves changes to 5 of the 42 methods in SpdySession and is verified to both preserve existing functionality and satisfy the provided integrity constraint.

**Challenges.** We now highlight some of the challenges involved in automating data structure refinement. First, when adding new fields to a data structure, we may need to synthesize new code in multiple methods. For instance, in the SpdySession example, 10 of the 42 methods in the original data structure manipulate streams, and 5 of these 10 need to be modified in order to correctly maintain the new als and ars fields. Second, modifications to existing methods are non-trivial: they require adding several lines of code with nested conditionals (e.g., see lines 16-22) and calling functions like addAndGet that are implemented elsewhere in the code base. Finally, observe that modifications to each method are not identical and cannot be simply “copy-pasted”.

**The Volt solution.** We solve these challenges using a modular application of the CEGIS paradigm coupled with
a novel inductive synthesizer. In particular, Volt performs synthesis in a modular way by independently updating the implementation of each method using CEGIS. The inductive synthesizer used in Volt is based on top-down enumerative search but takes advantage of three important observations that allow it to succeed in this setting:

**Observation #1:** The integrity checker and existing methods in the code base provide important syntactic clues to the synthesizer. For instance, looking at the implementation of check from Figure 2, one would expect that isRemoteInitiated would be useful during synthesis, as the return value of check depends on that of isRemoteInitiated. In addition, performing deeper static analysis of existing methods in the code base can also provide important clues. For example, addAndGet and subAndGet methods update variables of type AtomicInteger, and since the new fields als and ars also have the same type, these two methods may appear as part of the solution. To leverage such useful clues during synthesis, our method attaches probabilities to productions in the context-free grammar. Then, our search algorithm expands non-terminals according to these probabilities, thereby biasing search towards program that use various clues present in the integrity checker and the surrounding code base.

**Observation #2:** We can deductively prune away partial programs using necessary preconditions. To gain some intuition, suppose that the inductive synthesizer generates the following partial implementation of acceptStream during search:  

```java
1 public void acceptStream(int sid, ...) {
2   StreamState newS = new StreamState(...);
3   StreamState oldS = actStreams.put(sid, newS);
4   if (??s) {
5     this.als.addAndGet(??e);
6   } else {
7     this.als.addAndGet(??e);
8   }
9 }
```

Here, ??s is a hole that can be filled by any arbitrary statement. This partial program is infeasible because, no matter how we complete the hole, the resulting program will not satisfy the integrity constraint. In particular, the code snippet above always sets the value of this.als to 0 (the last line); however, this will violate the integrity constraint if acceptStream is ever called with a local stream (as it is added to actStreams on the second line).

Volt can prune such infeasible partial programs by computing a necessary precondition for correctness. A necessary precondition ϕ for a partial program P is a constraint on P’s inputs that must be satisfied by any completion P in order for P to satisfy the integrity constraint. Thus, if the current set of counterexamples includes an input that violates ϕ, we can conclude that the partial program is infeasible — i.e., no completion of the holes can satisfy the specification.

Going back to our example, the necessary precondition computed by Volt for the partial program above logically implies that sid must correspond to a remote (not local) stream. Thus, if the current counterexamples include a local stream, Volt can immediately reject this partial program. Since there are approximately 756k completions of ??s (up to size 5), the computation of necessary preconditions allows our technique to dramatically reduce the search space.

**Observation #3:** We can use the feedback from the feasibility checking engine to dynamically augment the PCFG. Consider the following partial implementation for acceptStream:

```java
1 public void acceptStream(int sid, ...) {
2   StreamState newS = new StreamState(...);
3   StreamState oldS = actStreams.put(sid, newS);
4   if (??s) {
5     this.ars.addAndGet(??e);
6   } else {
7     this.als.addAndGet(??e);
8   }
9 }
```

where lines 4–8 correspond to a sketch — i.e., synthesized code that does not contain any unknown statements or left-hand-side expressions. Even though this sketch is infeasible, its necessary precondition is consistent with many (but not all) of the counterexamples due to a missing edge case. We refer to such sketches as partially feasible. Our key observation is that such partially feasible program sketches often appear as sub-fragments of the desired solution. For example, observe that lines 4–8 above actually appear as a code snippet within the correct implementation of acceptStream shown in Figure 1 (lines 17–21). Based on this observation, our method augments the PCFG with new non-terminals that correspond to partially feasible program sketches. Since the addition of the new non-terminals is driven by necessary preconditions, we refer to this method as deduction-guided grammar augmentation.

### 3 Problem Formulation

We formulate the data structure refinement problem with respect to a Java-like statically-typed object-oriented language. A program in this language is a collection of data structures, and a data structure is of the form $D = (F, M)$ where $F$ is a set of fields and $M$ is a mapping from method signatures to their bodies. Without loss of generality, we assume that every data structure $D$ has a unique method called $D_0$ that corresponds to its initialization method (constructor). Fields and variables in $D$ are either of type int or Ref($D'$) where int denotes integers and Ref($D'$) denotes the address of some heap-allocated data structure of type $D'$. The program heap $\mathcal{H} : \text{Ref} \times \text{Field} \rightarrow \text{Value}$ maps reference and field pairs to values.

**Definition 3.1. (State)** A state $\sigma$ for a data structure $D = (F, M)$ is a pair $(o, \mathcal{H})$ where $o$ is an instance of $D$ and heap $\mathcal{H}$ determines the values of $o$’s fields.
We say that a state $\sigma$ is an initial state for data structure $D = (F, M)$ if it is obtained immediately after calling the constructor $D_0$ of $D$.

**Definition 3.2. (State equality modulo fields)** Given a pair of data structure states $\sigma = (o, \mathcal{H})$ and $\sigma' = (o', \mathcal{H}')$, we say that $\sigma$ and $\sigma'$ are equivalent modulo fields $F$, written $\sigma \equiv_F \sigma'$, if for every $f \in F$, we have:

1. $\mathcal{H}(o, f) = \mathcal{H}'(o', f)$ if $f$ has type int
2. If $f$ has type $\text{Ref}(D)$ for $D = (F', \_)$, then $\sigma_1 \equiv_{F'} \sigma_2$ where $\sigma_1 = (\mathcal{H}(o, f), \mathcal{H})$ and $\sigma_2 = (\mathcal{H}'(o', f), \mathcal{H}')$

In other words, the $\equiv_F$ relation indicates deep equality with respect to fields $F$.

**Definition 3.3. (State refinement)** Let $\sigma$ and $\sigma'$ be two states for data structures $(F, M)$ and $(F', M')$ respectively where $F \subseteq F'$. We say $\sigma'$ is a refinement of $\sigma$, written $\sigma' \leq \sigma$, iff $\sigma \equiv_F \sigma'$.

Intuitively, a data structure state $\sigma'$ refines another state $\sigma$ if they are equivalent modulo the fields defined in $\sigma$.

**Definition 3.4. (Action)** Let $D = (F, M)$ be a data structure. An action $\alpha$ on $D$ is a pair $(m, \overline{x})$ where $m \in \text{Domain}(M)$ and $\overline{x}$ is a list of arguments for $m$.

In other words, an action corresponds to invoking a method of the data structure.

**Definition 3.5. (Transition)** Let $D = (F, M)$ be a data structure, $\sigma$ a state of $D$, and $\alpha = (m, \overline{x})$ an action on $D$. We write $\sigma \xrightarrow{\alpha} \sigma'$ to denote that $\sigma'$ is the resulting state after executing method $m$ on state $\sigma$ with arguments $\overline{x}$. We refer to $\langle \sigma, \alpha, \sigma' \rangle$ as a transition of $D$.

Informally, a transition represents a change in the data structure’s state after calling a method.

**Definition 3.6. (State transition system)** The state transition system $T_D$ for a data structure $D$ is a tuple $(S, I, A, \rightarrow)$ where $S$ is the set of all possible states of $D$, $I \subseteq S$ are the set of initial states, $A$ is the set of all possible actions on $D$, and $\rightarrow$ is the transition relation for $D$ (i.e., $\sigma \rightarrow \sigma'$ iff $\exists \alpha \in A$ such that $\langle \sigma, \alpha, \sigma' \rangle$ is a transition of $D$).

We write $\rightarrow^*$ to denote the reflexive transitive closure of $\rightarrow$, and we say that $\sigma$ is a reachable state of $D$ if there exists a $\sigma_0 \in I$ such that $\sigma_0 \rightarrow^* \sigma$.

**Definition 3.7. (Method refinement)** Let $D = (F, M)$ and $D' = (F', M')$ be two data structures with corresponding transition systems $T_D = (S, I, A, \rightarrow)$ and $T_{D'} = (S', I', A', \rightarrow')$. The implementation of $m$ in $D'$ refines that of $m$ in $D$, written $m_{D'} \leq m_D$, if, for every action $\alpha = (m, \overline{x})$, and reachable state $\sigma_b \in S$, we have: If $\sigma'_b \leq \sigma_b$ and $(\sigma'_b, \alpha, \sigma_a) \in \rightarrow'$, then $\sigma'_a \leq \sigma_a$.

Intuitively, the implementation of a method in $D'$ is a refinement of the corresponding method in $D$ if it preserves the refinement relation between the states of the data structures when called with the same arguments.

**Definition 3.8. (Data structure refinement)** We say a data structure $D' = (F', M')$ is a refinement of $D = (F, M)$, written $D' \leq D$, if the following conditions are satisfied: (1) $F \subseteq F'$ and (2) $\text{Domain}(M) = \text{Domain}(M')$ and (3) For every method $m \in \text{Domain}(M)$, $m_{D'} \leq m_D$.

In other words, a data structure $D'$ refines $D$ if the set of fields of $D'$ is a superset of those of $D$ and, for every method $m$ in $D$ and its corresponding method $m'$ in $D'$, $m'$ refines $m$.

**Definition 3.9. (Integrity constraint)** Let $D$ be a data structure with fields $F \cup F'$. An integrity constraint $\Phi$ for fields $F'$ is a function that takes as input a data structure state $\sigma$ of $D$ and returns a boolean value such that:

\[ \forall \sigma, \sigma' \in \text{Reachable}(D). \quad \Phi_c(\sigma) \land \Phi_c(\sigma') \land \sigma \equiv_{F'} \sigma' \Rightarrow \sigma \equiv_F \sigma' \]

We say the integrity constraint holds on state $\sigma$, denoted $\sigma \models \Phi_c$, if $\Phi_c$ returns true on input $\sigma$.

In other words, the integrity constraint can be used to check whether the values of auxiliary fields $F'$ are correct based on the values of other fields. For instance, the check function from Figure 2 conforms to our definition: given any value for field `actStream`, there is only one possible value for the fields `a1s` and `a1s` for which check returns true.

**Definition 3.10. (Integrity constraint satisfaction)** We say that a data structure $D$ satisfies integrity constraint $\Phi_c$, denoted $D \models \Phi_c$, iff, for every reachable state $\sigma$ of $D$, we have $\sigma \models \Phi_c$.

**Problem statement.** Given a data structure definition $D = (F, M)$ and an integrity constraint $\Phi_c$, our goal is to synthesize a new data structure $D' = (F', M')$ such that (1) $D' \leq D$ and (2) $D' \models \Phi_c$.

## 4 Data Structure Refinement Algorithm

In this section we present our algorithm for synthesizing data structure refinements from integrity constraints. We start by introducing some terminology and then give an overview of our modular refinement procedure. Afterwards, we describe the novel aspects of our inductive synthesis approach in more detail, including necessary precondition computation for pruning partial programs as well as deduction-guided grammar augmentation.

### 4.1 Preliminaries

Figure 3 shows the context-free grammar (CFG) used by our synthesis algorithm for generating method bodies. It contains loops, conditionals, assignments, stores, loads, method calls, etc. Some of the non-terminals $K$ in this CFG (e.g., $L'$, $E'$) are parametrized by types to prevent enumeration of ill-typed programs during synthesis.
We now describe our modular data structure refinement where Derivs $\mathcal{P}$

### Modular Refinement Algorithm

**Algorithm 1.** Top-level refinement procedure

```plaintext
1: procedure Refine($D, F', \Phi_c$)
   input: Data structure $D = (F, M)$; new fields $F'$
   input: Integrity constraint $\Phi_c$
   output: $D'$ such that $D' \preceq D$ and $D' \models \Phi_c$
2: $M' \leftarrow \emptyset$
3: for all $m \in \text{Domain}(M)$ do
4:   $P \leftarrow \text{InsertHoles}(M(m))$
5:   $S \leftarrow \text{UpdateMethod}(M[m], P, \Phi_c)$
6:   if $S = \perp$ then return $\perp$
7:   $M' \leftarrow M' \cup \{(m, S)\}$
8: return $(F \cup F', M')$
```

**Algorithm 2.** CEGIS loop for updating a method

```plaintext
1: procedure UpdateMethod($S, P, \Phi_c$)
   input: Original body $S$ for some method $m \in D$
   input: Partial program $P$ for new body of $m \in D'$
   input: Integrity constraint $\Phi_c$
   output: New method body $m_{D'} = S'$ such that (1) $m_{D'} \preceq m_D$, and (2) $\{\Phi_c\} S' \{\Phi_c\}$ is a valid Hoare triple.
2: $C \leftarrow \emptyset$; $\triangleright Counterexamples$
3: $G \leftarrow \text{IntPCFG}(\Phi_c, f)$;
4: while true do
5:   $(S', G') \leftarrow \text{Synthesize}(P, C, G, S, \Phi_c)$;
6:   if $S' = \perp$ then return $\perp$; $\triangleright$ No feasible solution
7:   $S'' \leftarrow \text{Instrument}(S', \Phi_c)$;
8:   $C' = \text{Verify}(S'')$;
9:   if $C' = \emptyset$ then return $S'$; $\triangleright$ Verification successful
10: $C \leftarrow C \cup C'$; $G \leftarrow G'$;
```

defining data structure $D' = (F \cup F', M')$ such that $D' \preceq D$ and $D' \models \Phi_c$.

At a high level, the Refine procedure constructs $D'$ in a modular way by updating each method $m \in M$ independently. Specifically, for each method $m$, Refine invokes a procedure called InsertHoles (line 3) which uses lightweight static analysis to identify code fragments that modify $F$ and inserts statement holes (i.e., nonterminals $S$) at all relevant program points. Hence, the output of InsertHoles is a partial program $P$ which is then refined into a complete program using the UpdateMethod procedure at line 5. If UpdateMethod is successful (i.e., returns $S \neq \perp$), then the new implementation $m_{D'} = S$ of $m$ in $D'$ is guaranteed to satisfy the refinement relation $m_{D'} \preceq m_D$ as well as the integrity constraint (i.e., $\{\Phi_c\} m_{D'} \{\Phi_c\}$ is a valid Hoare triple).

The UpdateMethod procedure (Alg. 2) synthesizes $m_{D'}$ using a CEGIS loop but first initializes a PCFG $G$ by assigning probabilities to each production in the grammar (line 3). We describe how to initialize the PCFG in more detail in Section 4.4. Then, in each iteration of the CEGIS loop, our algorithm invokes the Synthesize method (line 5) to generate a candidate implementation $S'$ that satisfies all counterexamples $C$.
encountered so far. A key novelty of our inductive synthesis procedure is that, in addition to producing a candidate implementation, it also produces a new grammar \( G' \) that can be leveraged in subsequent iterations of the CEGIS loop.

Next, given a candidate implementation \( S' \) for \( \mathcal{P}' \), UpdateMethod checks whether \( S' \) satisfies the conditions (1) \( S' \leq S \) and (2) \( \{ \Phi_c \} S' (\Phi_c) \). To do so, it invokes the INSTRUMENT procedure to construct the code snippet shown in Figure 4 which is then verified using an off-the-shelf assertion checker. Observe that the instrumented code in Figure 4 works as follows: First creates an instance \( d \) of the original data structure \( D \) by calling GetInstance (Figure 5), which simply invokes the constructor of \( D \) followed by an invocation of an arbitrary sequence of \( D \)'s methods. Then, it initializes \( d' \) to be an arbitrary object of type \( D' \) and stipulates that (1) \( d \) and \( d' \) obey the refinement relation and (2) \( d' \) satisfies the integrity constraint. Next, it invokes the methods \( m,m' \) for \( d,d' \) with the same arguments, and finally asserts that \( d,d' \) continue to satisfy the refinement relation and that \( d' \) obeys the integrity constraint. If the assertions in this instrumented program can be verified using an off-the-shelf assertion checker (line 8 of Algorithm 2), UpdateMethod returns \( S' \) as a valid refinement of \( S \) (line 9). Otherwise, it adds the counterexamples returned by the verifier to \( C \) and continues the CEGIS loop with updated grammar \( G' \) and additional counterexamples \( C' \).

**Remark.** Note that the GetInstance call in the instrumented code in Figure 4 ensures that \( d \) is a reachable state of data structure \( D \). Furthermore, based on Def. 3.9 of integrity constraint, this \( d \) uniquely determines \( d' \) (modulo memory addresses). Thus, any counterexample corresponds to a reachable state of \( D' \) and ensures that we do not reject valid solutions despite using a modular strategy. Lastly, note that after calling GetInstance we can safely assume the integrity constraint holds on \( d' \). This is because if \( \Phi_c \) does not hold prior to \( m' \), then one of the methods invoked in GetInstance must violate \( \Phi_c \). However, we verify (inductively) that none of the methods are the first to break the integrity constraint and so it is safe to assume \( \Phi_c \) holds. This is known in the verification literature as circular compositional reasoning [32, 35].

We state and prove the following theorems under the assumption that the verification oracle is sound and complete.

**Lemma 4.2.** Suppose that UpdateMethod rejects a candidate body \( S' \) for method \( m \). Then, there does not exist a solution \( D' \) to the problem where \( M' [ m ] = S' \).

This lemma is important for the completeness of the end-to-end approach (Theorem 4.4) because it states that UpdateMethod does not reject valid solutions.

**Theorem 4.3. (Soundness)** Suppose that Refine\((D,F',\Phi_c)\) returns \( D' \). If \( D' \neq \bot \) then \( D' \leq D \) and \( D' \models \Phi_c \).

\[
d = \text{GetInstance}(D); \quad d' = \star
\]

\[
\text{assume}(d \equiv_f d'); \quad \text{assume}(\Phi_c(d'));
\]

\[
\text{args} = \star; \quad m(d, \text{args}); \quad m'(d', \text{args});
\]

\[
\text{assert}(d \equiv_f d'); \quad \text{assert}(\Phi_c(d'));
\]

**Figure 4.** Code generated by the Instrument procedure. \( \star \) denotes a random value.

**Algorithm 3.** Inductive synthesis algorithm

**Theorem 4.4. (Completeness)** Let Refine\((D,F',\Phi_c)\) return \( D' \). If \( D' = \bot \) then there does not exist a \( D' = (F', M') \) such that \( D \leq D' \) and \( D' \models \Phi_c \).

### 4.3 Inductive Synthesis Algorithm

In this section, we describe our inductive synthesis procedure summarized in Algorithm 3. This algorithm uses top-down enumerative search with deduction-based pruning and grammar augmentation. In particular, it maintains a worklist \( W \) of partial programs (initialized to \( \{ \mathcal{P} \} \) at line 2) and iteratively explores partial programs until it finds a complete program that satisfies all the counterexamples (line 6). In each iteration of the loop, it invokes the SelectBest function (line 4) to identify the most promising partial program in the worklist according to the PCFG. Here, the best program is defined as follows:

\[
\text{SelectBest}(W, G) = \arg \max_{\mathcal{P} \in W} \Pr_G(\mathcal{P})
\]

Next, given a partial program \( \mathcal{P}' \), Synthesize invokes the Deduce function (line 5) to check whether \( \mathcal{P}' \) is infeasible.
Algorithm 4. Checking feasibility of partial programs

In particular, DEDUCE returns a set $C_{\text{SAT}} \subseteq C$ such that $P'$ is consistent with the specification for every $C \in C_{\text{SAT}}$. Thus, if $C_{\text{SAT}} \neq C$, this means that all completions of $P'$ violate the specification for at least one input $C \in C$; hence, $P'$ can be pruned from the search space without sacrificing completeness. Therefore, we only expand a partial program if $C_{\text{SAT}} = C$ (see line 12). On the other hand, if $C_{\text{SAT}} = C$ and $P'$ is a complete program, then $P'$ is a solution to our inductive synthesis problem and is returned at line 6.

Lines 8–10 of the SYNTHESIZE algorithm perform deduction-guided grammar augmentation. In particular, if $P'$ is a sketch that is consistent with a non-empty subset of the counterexamples, it iterates over all the synthesized fragments in $P'$, and for each synthesized fragment $P_i$, it invokes the AUGMENTGRAMMAR procedure to add a new production $A \to P_i$ to the grammar. This is based on the observation that program sketches that satisfy some of the counterexamples often tend to occur as sub-components of the final solution. Thus, to avoid re-synthesizing these (potentially large) program sketches from scratch in future iterations, we directly add them as productions to the grammar. In essence, such deduction-guided grammar augmentation allows combining the benefits of top-down and bottom-up search in a goal-directed way.

In what follows, we explain the DEDUCE and AUGMENT-GRAMMAR procedures in more detail.

Pruning via necessary preconditions. Our technique for checking feasibility of partial programs is presented in Algorithm 4. Given a partial program $P$, the DEDUCE procedure returns a subset $C_{\text{SAT}}$ of $C$ that $P$ is consistent with. The high level idea is to compute a necessary precondition $\Phi$ for correctness that any instantiation of $P$ must satisfy and test whether $\Phi$ is satisfied by the counterexamples. If any $C \in C$ violates this necessary precondition, $P$ is guaranteed to be infeasible.

In more detail, the DEDUCE procedure first inlines the method calls in the $P$ \footnote{We inline method calls here to simplify presentation; our implementation does not perform inlining} and calls APPROXIMATE to generate a completion $S'$ of $P$. Here, $S'$ contains symbolic variables that represent “environment choices”, and it is constructed in such a way that, if there exists a completion of $P$ that satisfies the specification, then it is possible to find values of symbolic variables in $S'$ so that $S'$ satisfies the specification.

Figure 6 presents the APPROXIMATE procedure as inference rules deriving judgments of the form $\phi \vdash a \sim \beta$ where $a$ is a sequence of symbols in the grammar and $\beta$ is its corresponding replacement (i.e., over-approximation). If $\phi$ represents an expression or predicate, we obtain its replacement by recursively replacing any non-terminals nested inside it with fresh (unconstrained) variables. For assignments, we replace the assignment $a \leftarrow \beta$ with $a' \leftarrow \beta'$ where $a', \beta'$ are replacements for $a, \beta$ respectively and where $a$ does not contain the non-terminal $L$ (Assign-1). On the other hand, if $a$ contains a non-terminal $L$, we do not know which memory location is being written to; thus, we model it as a non-deterministic assignment to any of the new variables (Assign-2). Similarly, we model an unknown statement $S$ as a write to all possible memory locations that may be modified by $S$ (Statement).

Example 4.5. APPROXIMATE produces the following complete program for Listing 2 in Section 2.

```java
public void acceptStream(int sId, ...) {
    StreamState newS = new StreamState(...);
    StreamState oldS = actStreams.put(sId, newS);
    AtomicInteger v1, v2; // fresh
    this.als = new AtomicInteger();
    this.als = v1; this.ars = v2; // added
}
```

Given the output $S'$ of APPROXIMATE, the DEDUCE procedure computes the weakest pre-condition of $S'$ with respect to the integrity constraint $\Phi$. \footnote{Since $\Phi$ is a boolean function, $WP(S', \Phi)$ can be computed as $WP(S', \Phi, true)$. The verifier in our implementation is a bounded model checker, so we unroll and compute weakest preconditions in a standard way.} As stated by the following theorem, the weakest precondition $\Phi$ of $S'$ is a necessary precondition for $S$ (and therefore $P$):

Theorem 4.6. Let $S'$ be an over-approximation of code $S$. Then, if $\phi$ is a necessary condition for $S'$ to be correct, then $\phi$ is also a necessary precondition for $S$.

Thus, based on this theorem, if there is a counterexample that is inconsistent with the computed necessary condition $\Phi$, this means that we can prune partial program $P$ from the search space.

Grammar augmentation. The final piece of our inductive synthesis technique is the AUGMENTGRAMMAR procedure presented in Algorithm 5 for adding new productions to the PCFG. This algorithm takes as input the current PCFG $G$, a sketch $P$, the current set of counterexamples $C$, and it returns an augmented grammar that contains new productions. In particular, the augmented grammar contains a new
Algorithm 5. Grammar augmentation procedure. NumSatisfied(α, C) returns the average number of counterexamples in C satisfied by previously explored sketches containing α.

non-terminal S that represents sketches. It also contains two new productions:

- The production A → S allows using sketches as atomic building blocks when constructing partial programs. (Recall that non-terminal A represents atomic statements.)
- The production S → P adds sketch P as a new building block in the grammar.

In more detail, if the input grammar G does not contain the non-terminal symbol S, we add S to the set of non-terminals and add A → S to the set of productions. We also call InitProbabilities (discussed in Section 4.4.2) to update the probabilities of all productions starting with non-terminal A. If the grammar already contains productions of the form S → α, the probabilities associated with all of these productions also need to be updated. Thus, the loop in lines 10–12 adjusts the probabilities for each production r1 = (S → αi) to n1/Z where n1 is the average number of counterexamples satisfied by sketches containing αi and Z is a normalization term. Here, the intuition is to assign higher probabilities to productions associated with sketches that satisfy more counterexamples.

4.4PCFG Initialization

As stated earlier, an important observation underlying our solution is that the integrity checker and the surrounding code base contain useful clues that can be used to guide search. Thus, our technique (1) performs static analysis to identify features that are likely to be used in the desired solution, and (2) initializes PCFG probabilities to prioritize programs that use these features.

4.4.1 Mining Features via Static Analysis. Our static analysis extracts three types of code elements (namely, functions, types, and operators) for assigning probabilities to productions. These code elements are extracted by analyzing the integrity checking function and the surrounding code base.

**Analysis of integrity checking function.** Our method statically analyzes the integrity checker Φc to identify (1) types of expressions used in Φc, (2) all operators (e.g., bitshift, addition, etc.) that syntactically appear in Φc, and (3) functions that are invoked by Φc. Such code elements that syntactically appear in the integrity checker often also tend to appear in the synthesized code; thus, our technique assigns a higher probability to productions involving these code elements. (The mechanism for increasing probabilities is discussed in Section 4.4.2.)

**Analysis of existing functions.** Recall from Section 2 that existing functions in the code base may be useful for
updating the new fields. Thus, our method statically analyzes all existing functions to identify a subset of methods that return or update values of type τ, where τ is also the type of one of the new fields. In particular, our technique uses an off-the-shelf pointer analysis to identify all memory locations of type τ that are updated by some function f. If τ is also the type of a new field, then function f is considered to be a promising candidate and the probability of the corresponding production is increased.

4.4.2 Initializing Probabilities. Given a set of “interesting” productions $R'$ identified using static analysis, our method initializes PCFG probabilities as follows. First, let $R_N$ denote the set of productions whose left-hand-side is non-terminal $N$, and let $R_N'$ be $R_N \cap R'$. To initialize probabilities for productions in $R_N$, we first define a normalization constant $Z$ as follows:

$$Z = |R_N \setminus R_N'| + c_N|R_N'|$$

where $c_N$ is a constant strictly greater than 1. Then, we assign probabilities to productions $r \in R_N$ as follows:

$$\mathbb{P}(r) = \begin{cases} 
1/Z & \text{if } r \in (R_N \setminus R_N') \\
(c_N/Z) & \text{if } r \in R_N'
\end{cases}$$

**Theorem 4.7.** For each non-terminal $N$ in the grammar, $\mathbb{P}(N)$ defines a valid probability distribution over $R_N$.

5 Implementation

We have implemented our proposed approach in a tool called Volt. The inputs to Volt include (1) the source code of a Java data structure $D$, (2) an integrity constraint implemented as a Java function, and (3) a set of new fields to be added to $D$. In addition, Volt also takes a time limit $t$ (in seconds) indicating the maximum time it has to synthesize the desired function. Volt itself is implemented in Java and leverages the Z3 SMT solver [10] to check the feasibility of partial programs in the Deduce procedure and the JBMC assertion checker [9] for verification in the Verify procedure. In what follows, we discuss some key optimizations for the synthesis algorithm from Section 4.

**Grammar productions.** The grammar presented in Figure 3 is simple but unnecessarily permissive. In our implementation, we use a more fine-grained grammar that disallows enumerating obviously useless programs. In particular, our implementation restricts left-hand-side grammar expressions to new fields and fresh (temporary) variables. Second, it restricts method invocations to those that do not have side effects on existing fields. Third, it disallows atomic statements that call pure functions (since they are essentially no-ops). Fourth, it restricts loops to range-based for loops as arbitrary while loops are fairly uncommon compared to range-based counterparts. Finally, it restricts the set of local variables used in right-hand-side expressions to those that are in scope at the relevant program point. Observe that some of these restrictions require source code analysis; therefore our implementation leverages the Soot program analysis infrastructure [26] and the SPARK pointer analysis [31] to perform these optimizations.

**Additional pruning strategies.** Beyond using a grammar with auxiliary nonterminals, our implementation performs a few other optimizations to reduce search space. First, since multiple updates of a program variable along the same execution path are redundant, Volt disallows enumerating such partial programs. Second, it disallows loops that do not use the iterator in the body. Finally, since writing to temporary variables is only useful if there is a read afterwards, our implementation also avoids enumerating programs that write to, but do not read from, temporary variables.

6 Evaluation

In this section we describe a series of experiments that are designed to answer the following research questions:

1. (RQ1) Can Volt be used to refine data structures in real-world Java applications?
2. (RQ2) How does Volt compare against other approaches that could be used for solving the same problem?
3. (RQ3) How important is each of the three design decisions in Volt (i.e., use of PCFGs, necessary precondition computation, and grammar augmentation)?

**Benchmarks.** To evaluate Volt on real-world applications, we used a Github crawler to identify popular Java projects that use correlated fields for performance reasons. The crawler looks for commits with messages that match certain relevant keywords such as “performance”, “cache”, “new fields”, etc. We then manually inspected projects returned by the crawler and retained the first 25 classes that indeed have multiple correlated fields. To evaluate Volt on these benchmarks, we manually removed all the declarations and statements involving correlated fields (except for one field), wrote an integrity checking function, and used Volt to automatically derive the original implementation. To determine which fields to remove we used three criteria. First, if fields $f_1, \ldots, f_n$ could be derived from $f$, we removed each $f_i$. Second, if the fields could be derived from each other we marked those added in later commits as the auxiliary fields and removed them. Finally if the fields were added in the same commit we used syntactic hints e.g. name of the field to break the tie. For example in our benchmark PersistentSequentialDictionary, one of the correlated fields was called “cache” and the other was called “reverseCache” and we marked the latter as auxiliary.

**Setup.** In our evaluation, we use a time limit of 1 hour and run all of our experiments on the Google Cloud Engine (GCE) on an 8 vcpu instance with 128GB of memory.
Table 2. Baseline results. The Avg. Time is the average time over all the solved benchmarks (ignoring verification time) so timeouts do not contribute to the average time.

<table>
<thead>
<tr>
<th>Tool</th>
<th># Solved</th>
<th>Avg. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volt</td>
<td>23</td>
<td>264.2</td>
</tr>
<tr>
<td>FrAngel</td>
<td>5</td>
<td>160.21</td>
</tr>
<tr>
<td>JSketch</td>
<td>2</td>
<td>1033.3</td>
</tr>
</tbody>
</table>

6.1 Main Results

To answer our first research question, we evaluated whether Volt is able to automatically refine the benchmarks and how long it takes to do so. The results of this evaluation are summarized in Table 1. Here, the first two columns show the name of the class to be refined and the project it is taken from. The third column indicates the lines of code in the class and the fourth column shows the number of initial productions in our PCFG. The next three columns provide information about the number of new fields to be added, the total number of functions defined in the class, and the number of functions that need to be updated. Finally, the last column shows the running time of Volt. Overall, Volt is able to automatically refine 23 out of the 25 benchmarks (92%) within the provided time limit, and its average synthesis time is 264.2 seconds. Furthermore, we manually inspected the synthesized code and compared it against the human-written version. In all cases, we confirmed that the synthesized code is correct and matches the human-written code except for minor variations (e.g., if(b) S1 else S2 vs. if((b) S2 else S1).

Analysis of failed benchmarks. As shown in Table 1, there are two benchmarks that Volt failed to solve within the 1 hour time limit. In particular, Volt is unable to synthesize the desired update for classes GeocodingLookupService and DemonicParserState because the required update logic is very complex for some functions. For instance, a function in DemonicParserState requires adding 15 lines of code with over 150 AST nodes.

6.2 Comparison Against Baselines

To put these results in context and answer our second research question, we also evaluated Volt against existing tools. While there is no existing technique that addresses exactly our problem, we adapted two program synthesis tools to our problem setting:

- FrAngel: FrAngel is a component-based synthesis tool that can synthesize code with loops and conditionals [37]. However, since FrAngel only handles input-output examples, we cannot directly use it to solve our problem. Thus, to adapt FrAngel to our setting, we used Volt’s modular refinement algorithm (Algorithm 2) but replaced its inductive synthesis engine with FrAngel instead.
- JSketch: Our third baseline is JSketch [23], which is another state-of-the-art synthesizer for Java. Since JSketch can also not be used to directly solve our problem, we also perform this comparison by replacing Volt’s inductive synthesis engine with JSketch.

Note that both of these baselines are not quite apples-to-apples comparisons as they actually utilize Volt’s modular refinement procedure. Nonetheless, they serve as useful baselines for evaluating our proposed inductive synthesis algorithm (Algorithm 3).

As we can see from Table 2, JSketch can only solve 2 of the 25 benchmarks and is more than an order of magnitude faster than FrAngel.
we consider the following three ablations of Volt: variants perform significantly worse than are important for making this technique useful in practice. Among the three ablations. Overall, these results demonstrate computation of necessary preconditions has the most impact.\[\text{Cumulative Synthesis Time (s)}\]

Figure 7. Comparing Volt to baselines.

slower than Volt for these two benchmarks. FrAngel solves 5 of the benchmarks but fails to solve the remaining 20 within the 1 hour time limit. These results indicate that state-of-the-art synthesis tools are not sufficient for solving our problem even when leveraging the modular refinement idea proposed in this paper.

6.3 Ablation Study

In this section, we present the results of an ablation study for answering our third research question. In this evaluation, we consider the following three ablations of Volt:

- **Volt-NoPCFG**: This is a variant of Volt that uses a CFG (instead of a PCFG). In particular, Volt-NoPCFG uses the integrity constraint to generate an initial CFG without probabilities, and augments it with new productions without probabilities during synthesis. In addition, it still computes preconditions to prune infeasible programs.

- **Volt-NoAugment**: This is a variant of Volt that does not perform grammar augmentation. That is, Volt-NoAugment starts with the same initial PCFG and performs deduction to prune infeasible programs; however, it does not use deduction to augment the grammar. In other words, the PCFG is not changed throughout.

- **Volt-NoDeduce**: This is a variant of Volt that does not compute necessary preconditions. As a result, it cannot perform pruning or grammar augmentation, and it only uses the initial PCFG during the entire synthesis process.

The results of this ablation study are summarized in Figure 7. Here, the x-axis shows the number of solved benchmarks (sorted in increasing order of synthesis time), and the y-axis denotes cumulative running time. As we can see, all variants perform significantly worse than Volt; however, the computation of necessary preconditions has the most impact among the three ablations. Overall, these results demonstrate that all three ideas used in our inductive synthesis algorithm are important for making this technique useful in practice.

7 Related Work

**Data structure Repair and Verification.** There is an extensive body of research on runtime detection and repair of data structures from arbitrary boolean constraints \[12–15\] starting from Demsky and Rinard \[13\]. Our work is similar to this line of research in that we expect users to provide integrity constraints over the fields of the data structure. However, our problem is fundamentally orthogonal as we want to statically update the data structure so that the integrity constraint holds whereas these approaches mutate the state of the data structure at runtime to satisfy the integrity constraints. There is a parallel line of research \[16, 27, 28, 39\] on statically verifying properties of data structures. Our implementation of Volt uses a bounded model checker instead of these verifiers since it needs to obtain counterexamples in the CEGIS loop.

**Data Invariants.** Our work is also related to prior research on maintaining data invariants \[30, 36\]. The most similar work to ours is Spyder \[36\]. Like Volt, Spyder requires the developer to specify an invariant over fields of the data structure and afterwards it automatically synthesizes a new data structure where the invariants are maintained after each basic block. Unlike Volt, Spyder requires the invariants to be iterator-based and alias-free. In particular, the correlated fields must be iterator-based data structures that are structurally similar (i.e. all are the same length) and the contents of the data structures cannot alias each other. The benefit of such restrictions is that it allows Spyder to perform powerful and efficient transformations such as updating an existing loop header to simultaneously iterate over multiple structures. Volt, on the other hand, allows developers to specify arbitrary invariants so long as they can be encoded as a boolean function. As such, it can handle a much larger class of invariants that Spyder cannot. For example, Spyder cannot synthesize the desired update in Figure 1 as the correlated fields are not iterator-based. In addition, the technical details of synthesis algorithms are completely different. Spyder’s technique is based on deductive synthesis whereas Volt performs inductive synthesis.

**Data representation synthesis.** Our work is related to a line of research on so-called data representation synthesis \[11, 21, 22, 33, 41\], where the goal is to synthesize a complete data structure from a specification. For example, Hawkins et al. \[21\] allow developers to specify data structures as a series of query and update relations, and their synthesis procedure uses rewrite rules along with deduction to generate the concrete implementation. Loncaric et al. \[33\] follow a similar procedure; however, they use enumerative search to generate the implementation for update relations. Unlike this line of research, our work focuses on cases where developers refine an existing implementation by adding correlated fields. Thus, developers only need to provide a simple integrity
checking function as opposed to a complete implementation in a specification language. Furthermore, our approach can perform an in-place update as opposed to synthesizing the complete data structure from scratch.

**Synthesis using probabilistic models.** There are several prior techniques that use probabilistic models to guide their search [6–8, 25, 29, 38]. Most of these techniques learn a static PCFG through offline training [6, 29] whereas our approach uses static analysis to initialize probabilities and updates them on the fly. Barke et al. [7] also update PCFG probabilities by identifying programs that satisfy some of the input-output examples. However, in contrast to Barke et al. [7], our approach uses deduction to identify promising program sketches and augments the grammar with those productions in addition to updating probabilities. In addition, Concod [8] also combines probabilistic models with deduction; however, it uses deduction to compute a reward for reinforcement-learning-guided synthesis, whereas our approach uses deduction to augment the PCFG.

**Component-based synthesis.** There has been a long line of research on synthesizing programs from a set of components such as library functions [17–20, 24, 34, 37, 40]. Among these component-based synthesis approaches, the most related one is FrAngel [37], which can also synthesize control flow constructs like conditionals and loops. FrAngel is particularly related to our approach in that it learns new components at synthesis time by composing existing components. This is similar to our approach in that we also augment the grammar with new productions. However, a key difference from FrAngel is that Volt uses program analysis and deductive reasoning to learn these productions; furthermore, the new “components” Volt learns are program sketches rather than complete programs. As we show experimentally in Section 6, our proposed inductive synthesis approach significantly outperforms FrAngel in this context.

8 Conclusion

We have presented Volt, a tool for refining data structure implementations from integrity constraints. Volt is based on program synthesis and uses a modular instantiation of the CEGIS paradigm powered by a novel inductive synthesizer that incorporates three key ideas: (1) pruning using necessary preconditions, (2) deduction-guided grammar augmentation, and (3) PCFG construction using static analysis. We have evaluated Volt on 25 real-world Java classes with correlated fields and show that Volt can successfully refine 23 out of these 25 (92%) benchmarks. We also compared Volt against other state-of-the-art synthesis tools for Java and showed that our closest competitor can only solve 20% of the benchmarks (despite already incorporating the modular aspect of Volt). We also present several ablations of Volt and show that our main ideas are all crucial for making the proposed approach feasible in practice.

Acknowledgments

We thank the anonymous reviewers for the helpful feedback. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1908304, CCF-1811865, and CNS-1514435. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding sources.

References


