Program Synthesis for the Modern Programmer

Jacob Van Geffen

May 1, 2018

Abstract

Program synthesis research has exploded over the past few years due to the increasing importance of safety and correctness in real-world systems. Synthesis allows programmers to generate code from specifications, such as input-output examples, natural language, or higher level code. There are two main selling points for synthesis. First, in many cases, synthesizing code can take less time than writing the same code. Second, generated code is *correct-by-construction* with respect to the specification, so there is no need for additional verification or tests. In this paper, I'll present two novel synthesis techniques that generate programs in two different domains. The first of these techniques targets data scientists, taking simple input-output examples and generating table transformations in R. The second technique targets concurrency programmers, automatically placing signal calls for concurrent monitor code. Though their target domains differ greatly, both of these techniques show the power of program synthesis to benefit many programmers.

1 Introduction

Programmers face difficulties when writing safe and correct programs in a variety of domains. One solution to these difficulties is program synthesis. Synthesis tools aim to generate code from some higher level specification. These specifications are usually much easier for programmers to supply than the actual code. For example, the user might give one or more input-output examples of what their desired program should do. Given this specification, the synthesis tool will generate a program that is correct by construction, meaning that the synthesis tool only considers candidate programs that are correct with respect to the specification. This allows programmers to worry only about the correctness of their higher level specification rather than the correctness of a more complex piece of code. Of course, synthesis isn't the solution

to every problem. For some domains, providing a high level specification may be just as difficult as writing the actual code. For others, a synthesis tool may take hours or days to generate a correct program. As a result, prior to the time of these works, there have been only a few domains where synthesis has made real improvements. Tools like Lambda Learner for Haskell programs and Flash Fill for spreadsheet transformations have demonstrated the power of synthesis for their respective domains, but there are many more domains left unexplored.

With these insights in mind, we present two contributions in this report. The first of these contributions is a new synthesis technique for synthesizing data cleaning programs in R from input-output examples. Unlike standard synthesis techniques, our technique assumes little about the domain structure. We avoid making these assumptions by allowing our tool to take as input a domain for synthesized programs. Thus, our technique can be easily adapted to many other domains. The second contribution is a synthesis technique that generates signal calls for monitor code. Here, the specification is a partially completed program. Although this specification is relatively complex, it allows our synthesis tool to take advantage of complex information from the programmer while still simplifying the task of monitor programming. Together, these two contributions demonstrate that synthesis techniques can benefit programmers in a wide variety of domains.

Our first contribution aims to improve the productivity with which R programmers organize their data. While writing data cleaning and data wrangling programs can take hours [7], synthesizing them takes only a few seconds with our technique. In order to efficiently synthesize these programs, we combine enumerative search with deduction, shrinking the search space of partial programs. Our technique also takes advantage of partial evaluation when searching for program completions, reducing the size of the overall search space. We implemented these ideas in a tool called Morpheus and evaluated our Morpheus over 80 benchmarks from popular Stack Overflow questions. Of these 80 benchmarks, Morpheus is able to synthesize code for 78, taking an average of 3.59 seconds for each benchmark. This evaluation shows that Morpheus can greatly improve the productivity of R programmers. Section 3 describes these contributions.

Our second contribution aims to help concurrent programmers write safe and correct code. Due to the *correct-by-construction* property of synthesized code, synthesized signal placements are guaranteed to be safe and bug-free. In a domain where bugs like deadlocks and race conditions can crash entire systems [12, 20], these guarantees are extremely valuable. The key idea that allows our technique to place signals correctly is to construct verification conditions corresponding to each possible signal placement. By proving these verification conditions, our technique can deduce where

to correctly place signals. In order to demonstrate the effectiveness of these ideas, we implemented this novel technique in a tool called Expresso and evaluated Expresso over a variety of real-world benchmarks. We found that Expresso can correctly place signals for all of these benchmarks, and that the performance of code synthesized by Expresso closely matches that of hand-written code. These results demonstrate that Expresso can help programmers write a large class of monitor programs without sacrificing efficiency. Section 4 describes these contributions.

2 Background

2.1 Hoare Logic

The core of any program verification or synthesis technique is automated logical reasoning. Whether generating code or proving the safety of some program, any formal methods tool will need to prove several intermediate properties. For example, in order to prove the overall safety of a system, a verification tool may need to show that each function in the system executes without leaking information. These properties often take the following form: given some block of code S and some precondtion P, Q holds after S terminates (assuming S terminates). This type of property is so common that it has the shorthand $\{P\}S\{Q\}$, called a Hoare Triple [16]. While Hoare triples give us an easy way to describe properties of programs, we still need some way to actually prove these properties. To do so, we need to translate these triples into provable logical statements called verification conditions. For a triple $\{P\}S\{Q\}$, this can be done by computing the weakest possible precondition P such that $\{P\}S\{Q\}$ holds, written $P = wp(Q, S)$. If P implies $wp(Q, S)$ (i.e. if P is stronger than $wp(Q, S)$), then $\{P\}S\{Q\}$ must also hold. Although computing $wp(Q, S)$ in practice is quite difficult, modern tools can usually make good approximations [4]. Thus, proving $\{P\}S\{Q\}$ boils down to proving that P implies some approximation of $wp(Q, S)$. Fortunately, this is exactly the type of problem handled by SMT solvers [8].

The Satisfiability problem for first order logic is an extension of the NP-Hard Satisfiability (SAT) problem. The SAT problem poses the following question: given a formula expressed in propositional logic, is there any variable assignment under which the formula evaluates to true? The SAT problem for first order logic, or Satisfiability Modulo Theories (SMT), also asks for the satisfiability of a formula. Unlike formulas in propositional logic however, formulas in first order logic may include quantifiers, predicates, variables with non-boolean values, and first-order functions. These additional functions and variable domains are called "theories."

For example, the theory of linear arithmetic allows formulas with the constants 0 and 1, predicate $\leq$, and functions $+$ and $-$ like the following:

$$\forall x.\forall y.(x \leq 1) \wedge (y \leq x - 1) \rightarrow (y \leq 0)$$

In general, the SAT problem for first order logic is undecidable, but many theories are decidable and most have decidable fragments. This means that for many domains, SMT solvers are quite reliable.

2.2 Relational Algebra

Relational algebra is an algebra that describes operations over tables in a relational database [6]. While the core of relational algebra is relatively small, there are many extensions that add more complex operations to the algebra. We first describe these core operations before discussing common extensions, including those used by R libraries.

The core operations of relational algebra are select, project, rename, and natural join. The select statement takes a single table and some first order predicate as input, removing any rows that do not obey the given predicate. Likewise, the project statement takes a table and a list of columns, removing columns not on the list. The rename operator takes a table, an old column name, and a new column name as input. The operator then renames the old column in the input table to the new column name. Finally, the natural join statement takes two input tables and outputs a single table with one row for each matched column value in the two tables. In R, the select operator is denoted by `filter`, the project operator by `select`, the rename operator by `rename`, and the natural join operator by `inner_join`.

While these core operations describe many common table transformations, some extensions of relational algebra are necessary for *data wrangling* tasks, or tasks that involve transforming and preparing data for analysis. Here, we describe a few popular extensions in R: `arrange`, `gather`, and `spread`. The libraries `tidyr` and `dplyr` contain many more data wrangling operations [24, 14], but these three are most important for understanding the work presented in Section 3.

The `arrange` command rearranges rows in a table based on the values in one or more columns. This extension allows programmers to reason about the order or rows in a table, which cannot be done with standard relational algebra. The `gather` command moves information from column names into table cells, creating a table with fewer columns but more rows. Although `gather` can be expressed in relational algebra, it requires a large number of commands. Since the `gather` transformation is fairly common, R libraries include it as a single command. The `spread` command

can be thought of as an inverse of `gather`, converting table cell value information into column names. As a result, `spread` creates a table with fewer rows and more columns. Like `gather`, `spread` can be expressed in relational algebra, but is included in R libraries to simplify data wrangling programs.

2.3 Concurrency and Monitors

Concurrent programs often need to manage shared resources between threads. Improper access between these shared resources may cause many types of erroneous behavior in concurrent systems, including race conditions and deadlocks. Constructs like monitors and semaphores allow systems to safely manage these resources [17, 15, 22].

Monitors are composed of a single lock, several condition variables, and several methods. These monitor methods are used by the overall system in order to gain or release access to various shared resources. Each of these methods is guarded by the monitor lock, ensuring that access to these resources is atomic. Condition variables, as the name implies, track the conditions under which resources become available. To do so, each condition variable may perform three functions: `wait`, `signal`, and `broadcast`, all of which are performed only within monitor methods. The `wait` command blocks the current thread until the waiting condition variable is signaled. This allows threads to wait for some condition to hold before continuing. Correspondingly, the `signal` command wakes a single waiting thread, usually called once some waiting condition is satisfied. The `broadcast` command wakes all waiting threads.

There are several implementations of these monitor semantics, the most popular of which is the Mesa monitor [15]. Under this implementation, each condition variable maintains a queue of waiting threads. On a `wait` call, the current thread releases the monitor lock and moves into the waiting queue. When woken up by either a `signal` or `broadcast` call, the previously waiting thread must contend with other threads for the monitor lock. As a result, the waiting thread may not execute directly after the signaling thread. In many cases, some thread may execute in between these two threads and alter the value of the waiting condition. If this happens, the condition may not hold when the waiting thread executes, even if the condition held when `signal` was called. Considering this pathological case, correct monitor implementations must call `wait` in a while loop, iteratively checking the waiting condition after each `wait` call. The waiting thread only exits this loop if it has both been signaled and the condition holds once the thread begins executing.

3 Automating Data Wrangling Tasks

In this section, we explore a new domain for program synthesis: data wrangling. Data analysis has become increasingly important in a variety of fields. Unfortunately, raw tabular data is rarely formatted properly for data analysis. As a result, data scientists spend over 80% of their time organizing their data [7]. While languages like R offer a variety of libraries to perform data wrangling tasks, writing programs with these libraries can take hours even for expert R programmers. To address these problems and improve the productivity of data scientists, we introduce a new technique for automating data wrangling tasks using program synthesis.

Due to the difficult nature of writing data wrangling programs, the question-answer site Stack Overflow contains many questions about writing R programs for specific data preparation tasks. Interestingly, when programmers pose these questions on Stack Overflow, they almost always include one or more small input-output examples with their problem description [2, 3, 1]. This implies that while creating a correct data wrangling program may be difficult, specifying an example is quite easy. In addition, although these tasks are complex, they can often be decomposed into some number of key library calls.

Motivated by this reasoning, we created a synthesis algorithm that generates data wrangling programs in R from input-output examples. Our synthesis algorithm is *component-based*, meaning that our algorithm constructs programs by combining some number of specified program components [19, 13, 10]. The key idea that allows us to synthesize these programs is our novel combination of enumerative search and deductive reasoning. To synthesize some data wrangling program, we search over the space of partially completed programs, pruning invalid branches of our search tree with SMT deduction. Since each partial program represents many concrete programs, our deduction-based pruning technique greatly reduces the size of the overall search space.

To demonstrate the effectiveness of our technique, we implemented our ideas in a tool called Morpheus and evaluated Morpheus over many data transformation and consolidation tasks from Stack Overflow. In our evaluation, we show that Morpheus can efficiently synthesize a wide range of real-world data wrangling programs (78 out of 80 programs synthesized, with an average runtime of 3.59 seconds). We also demonstrate the generality of our technique by modifying Morpheus to synthesizing SQL and comparing it to a similar synthesis tool, SQLSynthesizer [25]. Over the SQLSynthesizer benchmarks, Morpheus outperforms SQLSynthesizer in both number of programs synthesized and in median runtime. These results demonstrate that our technique can efficiently synthesize a wide variety of data wrangling programs.

3.1 Overview of Technique

We now discuss the problem that our technique addresses and some of the challenges in solving this problem. The goal of this work is to create a synthesis tool that takes as input

1. a set of sample input tables $\overrightarrow{T_{in}}$
2. a single sample output table T_{out}
3. a list of components Θ and their specifications

and outputs a program P that satisfies these inputs. Each of the terms used in this definition is formally defined below.

Definition 1. *(***Table***)* *A* table T *is a tuple* (r, c, τ, ς) *where:*

- r, c *denote number of rows and columns respectively*
- $\tau : \{l_1 : \tau_i, \ldots, l_n : \tau_n\}$ *denotes the type of* T*. In particular, each* l_i *is the name of a column in* T *and* τ_i *denotes the type of the value stored in* T*. We assume that each* τ_i *is either* `num` *or* `string`*.*
- ς *is a mapping from each cell* $(i, j) \in ([0, r) \times [0, c))$ *to a value* v *stored in that cell*

Definition 2. *(***Component***) A component* $\mathcal{X}$ *is a triple* (f, τ, ϕ) *where* f *is a string denoting* $\mathcal{X}$*'s name,* τ *is the type signature,and* ϕ *is a first-order formula that specifies* $\mathcal{X}$*'s input-output behavior.*

Definition 3. *(***Component Specification***) A component specification* ϕ *for* $\mathcal{X}$ *is a first-order formula over variables* $t^1_{in}, \ldots, t^n_{in}, t_{out}$*, where* t^i_{in} *denotes some input for* $\mathcal{X}$ *and* t_{out} *denotes the output of* $\mathcal{X}$*. Moreover,* ϕ *must soundly describe* $\mathcal{X}$*, but not necessarily precisely.*

Definition 4. *(***Program***) A program* P *is a tree whose interior nodes are components and whose leaf nodes are concrete inputs. These concrete inputs may either be constants or input tables* T^i_{in}*. The* i^{th} *child of each interior node represents the* i^{th} *input of the corresponding component.*

Note that a program P satisfies the program specification $(\overrightarrow{T_{in}}, T_{out}, \Theta)$ if P outputs T_{out} over input $\overrightarrow{T_{in}}$ and only uses components in Θ. As convention, we'll used

the notation T_{in}^i and T_{out} to denote input and output tables from the user given example, and use t_{in}^i and t_{out} to denote the inputs and output of single components.

In order to generate such programs, there are several challenges that our program synthesis technique needs to overcome. First, even when generating small programs and considering a small number of components, the search space of program sketches quickly becomes large. Here, a program sketch refers to a program whose inputs are left unspecified. With 20 components, the number of sequential 5-command program sketches reaches the millions. Second, many of these library calls require auxiliary inputs that users might not provide. One example of this is the `filter` statement, which removes all rows from a table that do not satisfy some given predicate. When the space of input predicates is considered, the possibility of brute force search over the program space quickly becomes intractable. Third, new functions may be added to the libraries in the future. If our synthesis technique relies heavily on the semantics of old library calls, it will not be compatible with new ones.

To handle the first main problem, our technique uses SMT-based deduction to prune the search space of program sketches. By converting a program sketch into a checkable first order formula, our technique can determine whether some candidate sketch is a feasible solution without considering any auxiliary inputs. Since any program sketch represents a large number of concrete programs, this technique greatly reduces the size of the overall search space.

For the second problem, our technique applies partial evaluation in conjunction with SMT-based deduction to quickly search for correct auxiliary inputs. To complete a sketch with multiple auxiliary inputs, a naïve approach would need to check every combination of inputs. Partial evaluation circumvents this by partially evaluating the program sketch after a single auxiliary input has been specified. SMT-based deduction can then be reapplied, pruning the resulting sketch if the sketch is not a feasible solution. Combined, these two techniques greatly reduce the overall search space of programs.

Finally, to allow for generality and to overcome the third challenge, our technique takes as input a list of components and their specifications. Doing so means that our technique does not depend on any specific components. These specifications describe the semantics of the program components in first order logic, allowing for SMT-based deduction over program sketches. In order to reason about programs correctly, these specifications must be sound. However, completeness of these specifications is not required. More precise specifications will strengthen the power of SMT-based deduction and thus speed up our search, but totally precise specifications are often too complicated to create. In this work, we provide specifications for a number of library calls in R. Though, as we demonstrate later in this section, our technique can

easily be extended to other domains like SQL.

Together, these three techniques are key in creating an expressive and efficient synthesis tool.

3.2 Motivating Example

In this section, we present an example transformation that our technique aims to synthesize. This comes from a real world question found on Stack Overflow [1].

For this transformation, the user has specified two input tables:

frame	X1	X2	X3
1	0	0	0
2	10	15	0
3	15	10	0

frame	X1	X2	X3
1	0	0	0
2	14.53	12.57	0
3	13.90	14.65	0

And one output table:

frame	pos	carid	speed
2	X1	10	14.53
3	X2	10	14.65
2	X2	15	12.57
3	X1	15	13.90

These tables all describe data from a vehicle simulator, where strings (e.g. X1, X2, and X3) represent the position of a vehicle, integers (e.g. 10 and 15) represent vehicle identifiers, and real numbers (e.g. 14.53) represent vehicle speed. The first input table contains information about both the vehicle position and the vehicle identifier. The second contains information about vehicle position and speed. As specified by the output table, the user would like to create a transformation that consolidates position, identifier, and speed information into a single table, removing any 0 entries.

In this example and in the rest of our R benchmarks, we assume that the list of components comes from the popular `tidyr` and `dplyr` R libraries. These libraries contain functions for data tidying and data manipulation, respectively.

With these assumptions and inputs, we would like to synthesize the following program

```
df1=gather(table1,pos,carid,X1,X2,X3)
df2=gather(table2,pos,speed,X1,X2,X3)
df3=inner_join(df1,df2)
```

```
df4=filter(df3,carid != 0)
df5=arrange(df4,carid,frame)
```

Here, the gather command converts column name information into table value information. New column names must be specified during this command, as do the column names that should be converted to values. These inputs are not specified by the user in this example. After executing, the first gather command generates the following table, stored in `df1`

frame	pos	carid
1	X1	0
2	X1	10
3	X1	15
1	X2	0
2	X2	10
3	X2	15
1	X3	0
2	X3	0
3	X3	0

Next, `inner_join` takes two tables and combines them, using the columns common between both tables to organize information. As a result, this single table is stored in `df3`

frame	pos	carid	speed
1	X1	0	0.00
2	X1	10	14.53
3	X1	15	12.57
1	X2	0	0.00
2	X2	10	13.90
3	X2	15	14.65
1	X3	0	0.00
2	X3	0	0.00
3	X3	0	0.00

The `filter` command then removes all rows that do not satisfy some predicate. Here, the predicate `carid` $\neq 0$ specifies that any rows whose `carid` is 0 should be removed. Notice that the predicate was not specified by the user. After the command, this table is stored in `df4`

frame	pos	carid	speed
2	X1	10	14.53
3	X1	15	12.57
2	X2	10	13.90
3	X2	15	14.65

Finally, the `arrange` command sorts rows based on the values of specified columns. Again, notice that the auxiliary inputs `carid` and `frame` were not specified by the user. The result is the desired output table, displayed earlier in this section.

As this example demonstrates, reasoning about transformations can be quite difficult. Although only five library calls are used, choosing which functions to use and where to place them is overly burdensome for many R programmers. Even expert programmers may spend hours creating similar transformations [7]. Thus, by synthesizing these transformations in a matter of seconds, my technique can greatly improve the productivity of data scientists using R.

3.3 Synthesis Algorithm

Here, we describe our algorithm for generating data-wrangling programs. As discussed previously, a program can be viewed as a tree whose interior nodes are some component and whose leaves are inputs. Thus, our algorithm can be viewed as a search algorithm over trees of this form. In this search, there are three distinct tasks that our algorithm performs.

First, our algorithm searches for a program tree whose leaf nodes (i.e. inputs) are left as holes, which we denote as a *program sketch*. Our technique creates these sketches in an iterative process called *sketch generation*.

Next, after generating a sketch, our algorithm checks the whether or not the sketch may be part of a correct program with respect to the given example tables. As discussed earlier, this is done via SMT-based deduction. If the deduction procedure

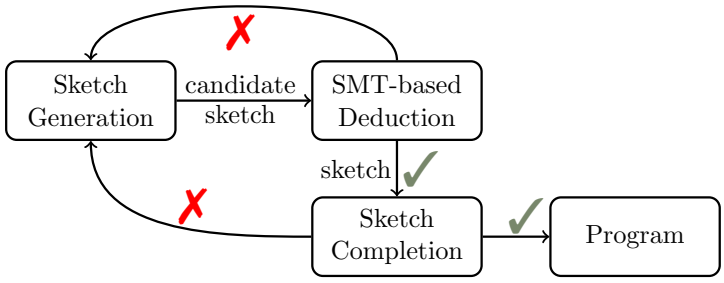


Figure 1: Illustration of the top-level synthesis algorithm

Algorithm 1 Synthesis Algorithm

```
1:  procedure SYNTHESIZE(𝒯, Λ)
2:      input: Input-output example 𝒯 and components Λ
3:      output: Synthesized program or ⊥ if failure
4:      W := {?0:tbl}                                          ▷ Init worklist
5:                                                             ▷ SMT-based Deduction
6:      while W ≠ ∅ do
7:          choose 𝒮 ∈ W;
8:          W := W\{𝒮}
9:          if DEDUCE(𝒮, 𝒯) = ⊥ then
10:             goto sketchgen;
11:                                                            ▷ Sketch Completion
12:         for 𝒮 ∈ SKETCHES(𝒮, 𝒯_in) do
13:             𝒫 := FILLSKETCH(𝒮, 𝒯)
14:             for p ∈ 𝒫 do
15:                 if CHECK(p, 𝒯) then return p
16:         sketchgen:                                         ▷ Sketch Generation
17:         for 𝒳 ∈ Λ_T, (?i: tbl) ∈ LEAVES(𝒮) do
18:             𝒮' := 𝒮[?j^𝒳(?j : τ⃗)/?i]
19:             W := W ∪ 𝒮'
20:     return ⊥
```

determines that no completion of the program sketch can satisfy the input-output example, then our algorithm backtracks to the sketch generation stage and generates a new program sketch. Otherwise, our algorithm performs its final phase: *sketch completion*.

In this final phase, our algorithm searches for possible completions of the given program sketch, using partial evaluation to speed up this search. These sketch completions are sketches whose holes have been filled in with given input tables and auxiliary inputs. If no complete program that satisfies the example tables is found, then our algorithm backtracks to the sketch generation phase. If some valid program is found, then our algorithm finishes and outputs the resulting program.

This process will continue until either some correct program is output or the sketch generation stage cannot generate any new sketches. Since the list of possible sketches may be infinite, our algorithm is not guaranteed to terminate.

3.3.1 Sketch Generation

We now give an in-depth description of each phase of our algorithm, the first of which is sketch generation. This phase finds some candidate program sketch by iteratively refining holes in previous candidate program sketches. We refer to the unspecified inputs of a sketch as *hole* and use the notation $?_i : \tau$ to denote a hole of type τ.

Using the reasoning of Occams razor, we would like to generate the simplest program sketches first. If those fail, we then generate more complex sketches. To do this, we maintain a queue of program sketches, beginning as a singleton queue containing the sketch

$$?_0 : \texttt{table}$$

When the SMT deduction phase tests a sketch, it removes this sketch from the queue. If the sketch fails here, then sketch generation adds new sketches to the queue, each generated by refining a single hypothesis in the sketch. Here, our algorithm refines a sketch by replacing a single hole of the form $?_i : \textit{table}$ with some function $\mathcal{X} = (f, \texttt{table}, \phi)$, where χ is some component that outputs a single table. The sketch may have multiple holes of type table, and there are multiple components to choose for each hole, so sketch generation will add multiple new sketches to the queue in a single iteration.

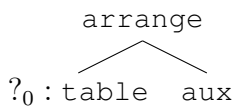
To illustrate this process, we look back at our motivating example. Clearly, the identity transformation, with structure

$$?_0 : \texttt{table}$$

will not satisfy the input-output example. As a result, our sketch generation procedure will refine the single hole of type `table`, namely $?_0$. This process will add every single-component sketch to our queue, including

$$\texttt{arrange}(?, \texttt{aux})$$

which has the following tree structure. Here, `aux` represents the set of auxiliary inputs for `arrange`.



This process of iteratively refining rejected sketches continues until our algorithm finds a correct program. For our motivating example, the following sketches will all

Algorithm 2 SMT-based Deduction Algorithm

1: **procedure** DEDUCE($\mathcal{S}, \mathcal{T}$)
2: **input:** Program Sketch $\mathcal{S}$, input-output example $\mathcal{T}$
3: **output:** $\bot$ if cannot be unified with $\mathcal{T}$; $\top$ otherwise
4: $\Phi(\mathcal{S}) := \Big(\bigwedge_{\mathcal{X}^i \in \mathcal{S}} (\phi^i)\Big) \wedge \Big(\bigwedge_{(\mathcal{X}^i, \mathcal{X}^j) \in \mathcal{S}} (t^{i,j}_{in} = t^k_{out})\Big)$
5: $\Phi(\mathcal{T}) := \alpha(\mathcal{T}_{out}) \wedge \bigwedge_{\mathcal{T}^i_{in} \in \overrightarrow{\mathcal{T}_{in}}} (\alpha(\mathcal{T}^i_{in}))$
6: $\varphi_{in} := \bigwedge_{?_j \in \mathcal{S}} \bigvee_{0 \le i < |\overrightarrow{\mathcal{T}_{in}}|} (?_j = \mathcal{T}^i_{in})$
7: $\varphi_{out} := ROOT(\mathcal{S}) = \mathcal{T}_{out}$
8: $\psi := \Phi(\mathcal{S}) \wedge \Phi(\mathcal{T}) \wedge \varphi_{in} \wedge \varphi_{out}$
9: **return** SAT(ψ)

be added to the queue during this process. The last of these sketches is the correct sketch for our program.

$$\texttt{arrange}(?, \texttt{aux})$$
$$\texttt{arrange}(\texttt{filter}(?, \texttt{aux}), \texttt{aux})$$
$$\texttt{arrange}(\texttt{filter}(\texttt{inner\_join}(?, ?), \texttt{aux}), \texttt{aux})$$
$$\texttt{arrange}(\texttt{filter}(\texttt{inner\_join}(\texttt{gather}(?, \texttt{aux}), ?), \texttt{aux}), \texttt{aux})$$
$$\texttt{arrange}(\texttt{filter}(\texttt{inner\_join}(\texttt{gather}(?, \texttt{aux}), \texttt{gather}(?, \texttt{aux})), \texttt{aux}), \texttt{aux})$$

3.3.2 SMT-based Deduction

As we have illustrated, sketch generation adds many new sketches to the queue each iteration. Without some way to prune these sketches, our search problem quickly becomes intractable. This is why the next phase — SMT-based deduction — plays a crucial role in our algorithm.

At a high level, SMT-based deduction combines information about the input-output example and the program sketch to determine if any completion of that sketch can satisfy the given example. In order to do so, our deduction procedure creates a formula ψ that encodes this information and checks the satisfiability of ψ. If ψ is satisfiable, then some completion of the given sketch may satisfy the input-output example. Otherwise, no completion can possibly satisfy the example, allowing our algorithm to prune the sketch.

To construct this formula ψ, our deduction procedure constructs four subformulas, $\Phi(\mathcal{S})$, $\Phi(T)$, φ_{in}, and φ_{out}. The first of these formulas, $\Phi(\mathcal{S})$, describes the information of the given program sketch $\mathcal{S}$. The second, $\Phi(T)$, describes both the input and output tables from the example. Finally, φ_{in} and φ_{out} assert that each input of $\mathcal{S}$ should be some table in T_{in} and that the output of $\mathcal{S}$ should be T_{out}, respectively. The overall formula ψ is a conjunction of all of these four formulas.

Now we demonstrate how to create each of these subformulas of ψ. To construct $\Phi(\mathcal{S})$, our deduction procedure combines the specifications of each component in $\mathcal{S}$ to produce one overall specification for $\mathcal{S}$. Recall that the specification for a component $\mathcal{X}^i$ is a first order logic formula over the variables $t^1_{in}, \ldots, t^n_{in}, t_{out}$. To distinguish the variables of different component specifications, we add i to the superscript of each variable, giving us a formula over $t^{i,1}_{in}, \ldots, t^{i,n}_{in}, t^i_{out}$. Here $t^{i,j}_{in}$ is an input and t^i_{out} is the output of $\mathcal{X}^i$. We denote this new specification for $\mathcal{X}^i$ as ϕ^i.

To act as a specification of $\mathcal{S}$, $\Phi(\mathcal{S})$ is a conjunction of all of these individual specifications ϕ^i. In addition, $\Phi(\mathcal{S})$ also needs some information about how the components of $\mathcal{S}$ connect. To provide this, $\Phi(\mathcal{S})$ also contains statements of the form $t^{i,j}_{in} = t^k_{out}$ in this conjunction, specifying that the output of X^j should serve as the input of X^i. Thus, we can construct $\Phi(\mathcal{S})$ with the following conjunction

$$\Big(\bigwedge_{\mathcal{X}^i \in \mathcal{S}} (\phi^i) \Big) \wedge \Big(\bigwedge_{(\mathcal{X}^i, \mathcal{X}^j) \in \mathcal{S}} (t^{i,j}_{in} = t^k_{out}) \Big)$$

To illustrate this, we again look at our motivating example. Suppose our deduction procedure needs to check the sketch

$$\mathcal{S} := \texttt{arrange}(\texttt{filter}(?, \texttt{aux}), \texttt{aux})$$

The specifications for arrange and filter are

$$t^0_{in}.rows > t_{out}.rows \wedge t^0_{in}.cols = t_{out}.cols$$

And

$$t^0_{in}.rows = t_{out}.rows \wedge t^0_{in}.cols = t_{out}.cols$$

Respectively. As a result, $\Phi(\mathcal{S})$ would combine these to create the overall specification

$$\begin{aligned} & t^{0,0}_{in}.rows = t^0_{out}.rows \ \wedge \ t^{0,0}_{in}.cols = t^0_{out}.cols \\ \wedge\ & t^{1,0}_{in}.rows > t^1_{out}.rows \ \wedge \ t^{1,0}_{in}.cols = t^1_{out}.cols \\ \wedge\ & t^{0,0}_{in} = t^1_{out} \end{aligned}$$

Next, to construct $\Phi(T)$, our procedure needs to encode each input and output table in first order logic. To do this, we define the operator α, which takes as input a table T and outputs a corresponding SMT formula. The exact definition of $\alpha(T)$ depends on the component specifications of our domain. If component specifications only reasoned about the number of rows and columns in their input and output tables, then $\alpha(T)$ would be a formula specifying the number of rows in T and the number of columns in T. To demonstrate, we provide the formula $\alpha(T_{out})$ below, where T_{out} is the output table from our motivating example.

$$T_{out}.rows = 4 \ \wedge \ T_{out}.cols = 4$$

This definition of α assumes that components only reason about the number of rows and columns in a table. In reality, $\alpha(T)$ contains information about the values T as well. Using α, we can define $\Phi(T)$ as follows

$$\alpha(T_{out}) \wedge \bigwedge_{T_{in}^i \in \overrightarrow{T_{in}}} (\alpha(T_{in}^i))$$

Finally, our procedure constructs φ_{in} and φ_{out}. Creating φ_{out} is fairly simple, since $\mathcal{S}$ will only has one output variable. In this case, we say

$$\varphi_{out} := ROOT(\mathcal{S}) = T_{out}$$

Where $ROOT(\mathcal{S})$ is the output of the last component in $\mathcal{S}$ (i.e. the root in our tree structure). Constructing φ_{in} is not as trivial. At a high level, φ_{in} should encode the following phrase: For each unspecified input in $\mathcal{S}$ of type table, some concrete input table T_{in}^i fills this spot. In first order logic, we can describe φ_{in} with the following conjunction of disjunctions.

$$\bigwedge_{?_j \in \mathcal{S}} \bigvee_{0 \leq i < |\overrightarrow{T_{in}}|} (?_j = T_{in}^i)$$

With a fully constructed formula ψ, our deduction procedure then checks the satisfiability of ψ using Z3. If the formula is not satisfiable, then no completion of $\mathcal{S}$ can satisfy the examples. If it is satisfiable, then our algorithm continues onto the next stage: sketch completion.

3.3.3 Sketch Completion

Once a program sketch passes the SMT-based deduction phase, our algorithm begins sketch completion by filling holes in the sketch. To fill a hole $?_i : \tau$, our algorithm

enumerates all valid inputs of type τ. When $\tau = \texttt{table}$, these valid inputs are the set of example input tables T_{in}^i. For other types (i.e. `int` and `string`), we only consider those whose constants can be found in some T_{in}^i or T_{out}. This restriction greatly reduces the search space for these auxiliary inputs without losing expresitivity.

After a hole is filled in, our sketch completion procedure attempts to perform partial evaluation over the resulting sketch. Partial evaluation is possible if all of the inputs for some component have been specified. In this case, our algorithm can simply compute the output of that component on the specified inputs. This process generates a new program sketch, replacing the component and its inputs with the generated output. The new program sketch is then checked by our deduction procedure. If the sketch is rejected, our algorithm backtracks with two possibilities. First, our algorithm will try backtracking to some previously filled hole (including the one that was just filled). Here, the algorithm will try a new valid input, assuming one exists. Otherwise, if all valid inputs have been exhausted for each hole, our algorithm backtracks to the hypothesis refinement phase in order to generate a completely new sketch.

3.4 Implementation and Evaluation

Finally, we describe the implementation of Morpheus and our evaluation.

Implementation details The techniques described in this paper have been implemented in a tool called Morpheus, written in C++. In the implementation of our SMT-based deduction, we used the Z3 SMT solver to check the satisfiability of our generated constraints [8].

In addition to our basic algorithm, we used a few insights to improve the performance of Morpheus. First, we order our candidate program queue based on an n-gram model heuristic (using the 2-gram model in SRILM) [23]. This model assigns a score to each candidate program based on the components used in the hypothesis, ignoring control flow. To generate the model, we trained over 15,000 code snippets founds in Stack Overflow answers, all of which involved the use of the `tidyr` or `dplyr` libraries. This allows Morpheus to search through fewer candidate programs. Second, we parallelize our computations, running several threads that check candidate programs of different lengths. This means that in order to generate a large program, Morpheus avoids the time overhead of checking smaller programs.

Category	Description	#	No deduction		Deduction	
			#Solved	Time	#Solved	Time
C1	*Reshaping* dataframes from either "long" to "wide" or "wide" to "long"	4	2	198.14	4	6.70
C2	*Arithmetic computations* that produce values not present in the input tables	7	6	5.32	7	0.59
C3	Combination of *reshaping* and *string manipulation* of cell contents	34	28	51.01	34	1.63
C4	*Reshaping* and *arithmetic computations*	14	9	162.02	12	15.35
C5	Combination of *arithmetic computations* and *consolidation* of information from multiple tables into a single table	11	7	8.72	11	3.17
C6	*Arithmetic computations* and *string manipulation* tasks	2	1	280.61	2	3.03
C7	*Reshaping* and *consolidation* tasks	1	0	✗	1	130.92
C8	Combination of *reshaping*, *arithmetic computations* and *string manipulation*	6	1	✗	6	38.42
C9	Combination of *reshaping*, *arithmetic computations* and *consolidation*	1	0	✗	1	97.3
Total		80	54 (67.5%)	95.53	78 (97.5%)	3.59

Figure 2: Summary of experimental results. All times are median in seconds and ✗ indicates a timeout (> 5 minutes).

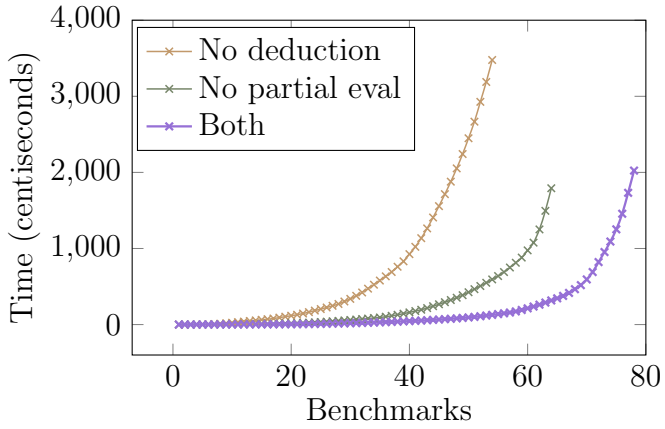


Figure 3: Cumulative running time of EXPRESSO

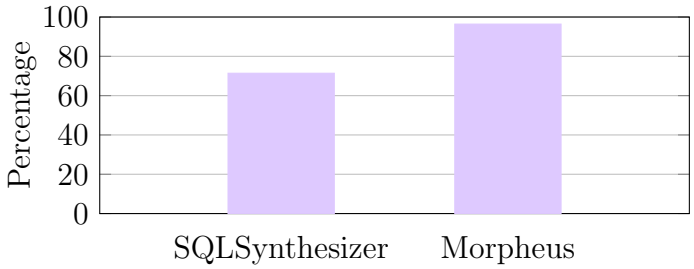


Figure 4: Comparison with SQLSYNTHESIZER

Evaluation Methodology With our evaluation, we aimed to answer three main questions

1. Is Morpheus able to synthesize a wide variety of complex programs in a reasonable amount of time?

2. Do SMT-based deduction and partial evaluation improve the Morpheus's runtime?

3. How does Morpheus compare to other similar synthesis tools?

To answer the first two questions, we collected 80 benchmarks from Stack Overflow questions. For each benchmark, we evaluated Morphues under three configurations: one with no deduction or partial evaluation, one with deduction but without partial evaluation, and one with both deduction and partial evaluation. The measurements for these experiments are displayed in Figure 3, summarized over categories. The overall results are also summarized in the graph below. As these figures demon-

strate, both deduction and partial evaluation play a key role in allowing Morpheus to efficiently synthesize a large class of programs.

To answer the final question, we compared Morpheus to a similar tool called SQLSynthesizer [25]. This tool also takes sample input and output tables in order to synthesize a program, but targets SQL instead of R. In order to allow Morpheus to synthesize SQL code, we created a list of 8 SQL commands for Morpheus to use as components. Then, we evaluated Morpheus over the 28 benchmarks used to evaluate SQLSynthesizer. Over these benchmarks, Morpheus outperforms SQLSynthesizer by a significant margin. While SQLSynthesizer solves 71.4% of benchmarks with a median runtime of 11 seconds, Morpheus solves 96.4% of these benchmarks with a median runtime of 1 second.

4 Automatic Signal Placement for Monitors

In this section, we explore another new domain for program synthesis: concurrent programs. Concurrent programs, with the need for explicit communication between threads, are notoriously difficult to write without bugs [20]. Managing shared resources between these threads is a common challenge in many systems. For this reason, monitors help manage shared resources in a standard way [17, 15, 22]. They allow threads to wait for resources to become available by using the `wait` command, relying on other threads to signal waiting threads once these resources are available by calling either `signal` (for waking a single thread) or `broadcast` (for waking all associated threads). Unfortunately, signaling both correctly and efficiently requires reasoning about all possible interleavings of thread executions. Furthermore, improper signal placement can easily cause serious bugs in concurrent systems [12]. A lack of signals may result in some threads waiting indefinitely, while abundant signaling may wake up threads spuriously, causing unnecessary context switches. To address these problems, we introduces a new synthesis technique that generates correct-by-construction signal placements for monitors.

Some runtime systems have also addressed these problems by automatically signaling threads without requiring explicit signals from the programmer. These systems are said to implement *implicit signal monitors*. In implicit signal monitors, there are no signal and broadcast constructs. Instead, threads use `waituntil`(P) statements to wait until their associated predicate P evaluates to true. These implicit signal monitors allow users to specify the semantic meaning of a monitor without dealing with signal placement. As such, errors created by improper signal placement are impossible with implicit signals.

Although these runtime systems remove the signaling burden from the programmer, they often add significant runtime overhead [5]. In order to determine when to wake up each thread, these systems store predicates in complex data structures and regularly check these predicates at runtime [18]. Expert hand written code often takes advantage of the monitor's structure to avoid unnecessary runtime checks, so hand written explicit signal monitors are often much more efficient than their implicit signal counterparts. Since efficiency plays a large role in real-world concurrent systems, implicit signal monitors have not been widely adopted. However, by reasoning about predicates statically, we can implement implicit signal monitors without such an unruly overhead.

The key idea in our technique is to generate some set of verification conditions that, when proven, allow us to transform implicit signal monitors into efficient, semantically equivalent explicit signal implementations. Namely, for each atomic segment in the monitor and each `waituntil`(P) statement, we generate a set of Hoare triples whose validity allows us to determine

1. whether to signal threads waiting on P at all,
2. whether to signal conditionally or unconditionally, and
3. whether to signal or broadcast.

To augment the strength of our theorem prover, we introduce the concept of a monitor invariant. Similar to a loop invariant, a monitor invariant is a logical formula I over monitor variables that

1. holds at the construction of a monitor, and
2. holds after the execution of every atomic segment in the monitor.

We specify that I only contains monitor variables to ensure that only methods in the monitor may change the truth value of I. Along with this definition, we also present a novel algorithm for automatically inferring strong monitor invariants. Since real world monitors may have complex `waituntil` predicates, monitor invariants are essential for reasoning about these predicates.

In addition, this section also presents an implementation of these techniques, called Expresso. We evaluate Expresso over a variety of benchmarks, comparing our transformation to both a similar runtime technique and to hand written explicit signal code. On average, Expresso gets a significant (1.56x) speed-up over this runtime system. In addition, Expresso closely matches the performance of hand-optimized code in nearly all cases. These results demonstrate that Expresso can synthesize a large class of programs without adding unruly runtime overhead.

```
1   class RWLock {
2     unsigned int readers = 0;
3     boolean writerIn = false;
4
5     atomic void enterReader() {
6       waituntil(!writerIn);
7       readers++;
8     }
9     atomic void exitReader() {
10      if(readers > 0) readers--;
11    }
12    atomic void enterWriter() {
13      waituntil(readers == 0 && !writerIn);
14      writerIn = true;
15    }
16    atomic void exitWriter() {
17      writerIn = false;
18    }  }
```

Figure 5: Implicit-signal monitor for readers-writers lock.

4.1 Motivating Example

To illustrate our technique, lets consider the case of the Readers-Writers problem. The access control for the Readers-Writers problem has been implemented with implicit signaling in Figure 5 and corresponding explicit signaling in Figure 6. Notice that in these implementations, any reading or writing thread must enter the monitor to gain access to the resource. Only the monitor code is shown in these figures. In this section, we will illustrate how our technique works for the Readers-Writers problem, highlighting the most important aspects of our algorithm.

Before the analysis begins, Expresso generates a monitor invariant. In the case of `RWLock`, Expresso generates the invariant `readers` ≥ 0. This holds because

1. readers begins at value 0, and
2. readers is only decremented when `readers` > 0.

Next, Expresso will determine what type of signaling needs to be done in each atomic segment. In implicit signal code, a global monitor lock is acquired at the start of each method and released at the end of each method. This lock is also released and reacquired at every `waituntil` statement. An atomic segment is the block of code that sits in between a lock acquire and lock release. Since there are two `waituntil(`P`)`

statements with different predicates, Expresso will consider signaling at every atomic segment for each of these two predicates.

EnterReader. Let's discuss how Expresso will transform the single atomic segment within the `enterReader` method, which includes only the statement

$$\text{readers} + +$$

Normally, we would first consider signaling threads waiting on the predicate

`!writeIn`

However, since this atomic segment follows the `waituntil` statement with this predicate, we won't need to signal this predicate. Instead, we only need to consider signaling threads waiting on $P :=$ readers $== 0 \wedge$!writerIn. To determine whether or not we need such a signal, let's consider a thread t_r that calls `enterReader`, and assume some thread t_w is blocked on P. If P never holds after t_r executes `enterReader`, then we never need to signal t_r. This check can be written as a Hoare triple in the following way:

$$\{readers \geq 0 \wedge \neg writerIn \wedge \neg P_w\} \texttt{ readers++ } \{\neg P_w\}$$

Here, the precondition of this Hoare triple captures several important facts. First, it includes the invariant we mentioned earlier, which is guaranteed to hold at the start of the atomic segment. Second, it includes the predicate of the preceding `waituntil` statement. This is because after executing `waituntil(`P`)`, P is guaranteed to hold. Lastly, it captures our assumption that some thread t_w is blocked on P_w by including $\neg P_w$ in the precondition. In this case, the triple is indeed valid, so no signal is necessary here. Notice that the validity of the triple would not be provable without the invariant.

ExitReader. First, to check whether we need to signal threads waiting on

$$P_r := \texttt{!writeIn}$$

Expresso checks the triple

$$\{readers \geq 0 \wedge \neg P_r\} \texttt{ if(readers > 0) readers-- } \{\neg P_r\}$$

Since the code doesn't alter `writeIn` in any way, this triple is trivially valid. Next, Expresso will determine whether it needs to signal threads waiting on P_w. As in `enterReader`, Expresso will check the following triple:

$$\{readers \geq 0 \wedge \neg P_w\} \texttt{ if(readers > 0) readers--} \{\neg P_w\}$$

In this case, the triple is not valid, which means some signaling is necessary. Expresso will then determine whether to signal or broadcast. To understand how this is done, let's consider the atomic segment that follows the statement `waituntil(`P_w`)` and thread t_w that executes this segment. If P_w becomes false after t_w's execution, then broadcast is not necessary, as no other thread waiting on P_w should be woken up. We can make this check by evaluating the following triple:

$$\{readers \geq 0 \wedge P_w\} \texttt{ writerIn = true } \{\neg P_w\}$$

Since this triple is valid, a broadcast is not necessary. Finally, Expresso will determine whether the signal should be conditional or unconditional. For this, Expresso must again reason about the truth value of P_w at the end of `exitReader`. If P_w always holds after `exitReader` executes, then Expresso should signal unconditionally. Otherwise, Expresso needs to signal conditionally in order to avoid spurious wake-ups. This reasoning is encoded in the following triple:

$$\{readers \geq 0 \wedge P_w\} \texttt{ if(readers > 0) readers--} \{P_w\}$$

Since this triple is not valid, Expresso must signal conditionally here.

EnterWriter. Since `enterWriter` contains the `waituntil` statement for writers, `enterWriter` will not need to signal any writers. Also, using similar reasoning as in `enterReader`, Expresso can establish that `enterWriter` does not need to signal any readers because the following Hoare triple is valid:

$$\{readers \geq 0 \wedge P_w \wedge writerIn\} \texttt{ writerIn = true } \{writerIn\}$$

ExitWriter. With similar reasoning as in `exitReader`, Expresso can determine that `exitWriter` doesn't need to signal any writers. Furthermore, Expresso also determines that `exitWriter` needs to signal (but not broadcast) to readers, again with similar reasoning as in `exitReader`. To determine whether this signal should be conditional or unconditional, Expresso will check this triple:

$$\{readers \geq 0 \wedge writerIn\} \texttt{ writerIn = false } \{\neg writerIn\}$$

In this case, this triple is valid, which means that `exitWriter` should signal unconditionally.

```
1   class RWLock {
2     unsigned int readers = 0;
3     boolean writerIn = false;
4     Lock l = new ReentrantLock();
5     Condition readers = l.newCondition(),
6               writers = l.newCondition();
7     void enterReader() {
8       l.lock();
9       while(writerIn) readers.await();
10      readers++;
11      l.unlock();
12    }
13    void exitReader() {
14      l.lock();
15      if (readers > 0) readers--;
16      if (readers == 0) writers.signal();
17      l.unlock();
18    }
19    void enterWriter() {
20      l.lock();
21      while(readers != 0 || writerIn) writers.await();
22      writerIn = true;
23      l.unlock();
24    }
25    void exitWriter() {
26      l.lock();
27      writerIn = false;
28      if (readers == 0) writers.signal();
29      readers.signalAll();
30      l.unlock();
31    } }
```

Figure 6: Explicit-signal monitor for readers-writers lock.

4.2 Signal Placement Algorithm

In this section, we describe our general algorithm for placing signals. This algorithm can be broken up into two main steps. First, our system generates a strong monitor invariant. Next, it uses this monitor invariant to prove several verification conditions, determining where signals are needed, whether to signal or broadcast at these points, and whether to do so conditionally or unconditionally. For simplicity, we will first describe the second part of our algorithm, assuming that a monitor invariant has already been generated. This part of our algorithm, called PlaceSignals, takes as

Algorithm 3 Signal Placement Algorithm

```
1:  function PlaceSignals(M, I)
2:      input: M, an implicit signal monitor
3:      input: I, a monitor invariant
4:      output: M', an explicit signal monitor
5:      Σ ← [w ↦ ∅ | w ∈ CCRs(M)]
6:      for (w, p) ∈ CCRs(M) × Guards(M) do
7:          if ⊢ {I ∧ Guard(w) ∧ ¬p} Body(w) {¬p} then
8:              continue;
9:          if ⊢ {I ∧ Guard(w) ∧ ¬p} Body(w){p} then
10:             cond ← ✓
11:         else
12:             cond ← ?
13:         if ∀(p, s') ∈ CCRs(M). ⊢ {I ∧ p} s'{¬p} then
14:             bcast ← false
15:         else
16:             bcast ← true
17:         Σ(w) ← Σ(w) ∪ {(p, cond, bcast)}
18:     return Instrument(M, Σ)
```

input an implicit signal monitor M, a monitor invariant I, and outputs an equivalent explicit signal monitor. In addition, we will also assume that `waituntil` statements don't contain any thread local variables for our general algorithm.

Our algorithm works by creating a mapping from atomic segments, or Condition Critical Regions (CCRs), to signals. This mapping represents what signals should be performed after execuing each CCR but before exiting the monitor. We represent a signal by the triple $(p, cond, bcast)$, where p represents the predicate to signal, *cond* represents whether the signal should be conditional, and *bcast* represents whether the signal should be a `broadcast` statement.

The first step in PlaceSignals is to create an empty mapping over all CCRs in the monitor M (line 5). Then, our algorithm iterates over each pair (w, p) of CCRs and `waituntil` predicates in M. Each iteration determines if a signal for p at w is necessary, and if so, also determines *cond* and *bcast*.

The first check in the loop decides whether a signal to p at w is necessary at all.

This is done by checking the triple

$$\{I \wedge \mathit{Guard}(w) \wedge \neg p\}\ \mathit{Body}(w)\ \{\neg p\}$$

Here, our algorithm asks the following question: "Assuming the monitor invariant I, the guard of the CCR $\mathit{Guard}(w)$, and $\neg p$ hold before the execution of $\mathit{Body}(w)$, after it's execution, is $\neg p$ guaranteed to hold?" When some thread is blocked on p, these assumptions are guaranteed to hold before executing $\mathit{Body}(w)$. Thus, if this triple is provable, then there should be no signal for p at w. In this case, our algorithm can continue onto the next iteration. Otherwise, it must perform two more checks.

Next, our algorithm checks whether the necessary signal should be conditional or not. To do so, we check this triple at line 9

$$\{I \wedge \mathit{Guard}(w) \wedge \neg p\}\ \mathit{Body}(w)\ \{p\}$$

By checking this, our algorithm asks the question: "Assuming the monitor invariant I, the guard of the CCR $\mathit{Guard}(w)$, and $\neg p$ hold before the execution of $\mathit{Body}(w)$, after it's execution, is p guaranteed to hold?" Again, when a thread is blocked on p, these assumptions are guaranteed to hold before execution $\mathit{Body}(w)$. If this triple is provable, then p will always hold after executing $\mathit{Body}(w)$ when some thread is blocked on p. Thus, we can signal unconditionally in this case. If not, then the predicate isn't guaranteed to hold after executing $\mathit{Body}(w)$, so we must signal conditionally.

Finally, our PlaceSignals procedure determines whether the signal should be a broadcast or not. Our algorithm does so by checking triples of the following form, where w' is some CCR such that $\mathit{Guard}(w') = p$

$$\{I \wedge p\}\ \mathit{Body}(w')\ \{\neg p\}$$

Unlike the last two checks, this check requires reasoning about the CCR executed by the blocked thread. That's because in order to determine whether a broadcast is necessary, the algorithm must reason about the truth value of the predicate p after the blocked thread executes. If the predicate remains true, then a broadcast is necessary, as other threads may be blocked on p as well. If not, then any remaining threads blocked on p shouldn't be woken up, so a broadcast isn't necessary. Thus, for this check, the algorithm asks the question: "For every CCR w' whose guard is p, assuming the monitor invariant I and predicate p hold before executing $\mathit{Body}(w')$, after it's execution, is $\neg p$ guaranteed to hold?" As discussed, if any of these triples are not provable, then a broadcast is necessary, since some thread blocked on p may

Algorithm 4 Monitor Invariant Inference

```
1: function InferMonitorInv(M, Θ)
2:     input: M, an implicit signal monitor
3:     input: Θ, set of Hoare triples of the form {P} s {Q}
4:     output: I, a monitor invariant
5:     Φ ← ∅
6:     for {P} s {Q} ∈ Θ do
7:         Φ ← Φ ∪ abduce(P, wp(s, Q))
8:     do
9:         numPreds ← |Φ|
10:        for ψ ∈ Φ do
11:            if ⊬ {true} Ctr(M) {ψ} then
12:                Φ ← Φ \ {ψ}
13:                continue;
14:            I ← ⋀_{ψ_i ∈ Φ} ψ_i
15:            if ∃w ∈ CCRs(M). ⊬ {I ∧ Guard(w)} Body(w) {ψ} then
16:                Φ ← Φ \ {ψ}
17:    while numPreds ≠ |Φ|
18:    return I
```

need to be woken up. Otherwise, if all triples are provable, then broadcast is not necessary.

Once PlaceSignals finishes this loop, it has created a list of all signals that need to be placed in the explicit implementation. Finally, it returns the original monitor annotated with these signals.

Now we describe the first part of our algorithm: monitor invariant generation. InferMonitorInv takes as input both an implicit signal monitor M and a set of Hoare triples Θ, and outputs an invariant of the monitor, I. This procedure is property-directed, meaning that it only generates invariants that are helpful to prove some given Hoare triples. Specifically, InferMonitorInv takes as input Θ, the set of Hoare triples that we described the procedure PlaceSignals, but with invariant I set to *true*. At a high level, our algorithm first generates a set of candidate predicaes by performing abductive inference over each Hoare triple in Θ. Then, InferMonitorInv prunes this list using predicate abstraction, removing any candidates that aren't invariants in the monitor M. Finally, InferMonitorInv returns a conjunction of all candidates that are invariants in M.

We now give a more detailed description of each of these high level steps. In

the first phase of this procedure, InferMonitorInv performs abductive inference over each triple $\{P\}s\{Q\}$ in Θ by making a call to the sub-procedure *abduce* [9]. Intuitively, *abduce* takes some Hoare triple $\{P\}s\{Q\}$ that is not necessarily valid and finds some hypothesis ψ such that the new triple $\{P \wedge \psi\}s\{Q\}$ is valid. More formally, this procedure operates over verification conditions that corresponds to these Hoare triples. It takes as input the antecedent P and the consequent $wp(s, Q)$, which together represent the verification condition $P \Rightarrow wp(s, Q)$. Here, $wp(s, Q)$ represents the weakest precondition of s with respect to Q. Then the hypothesis ψ output by *abduce* obeys the following constraints:

$$(1)\ P \wedge \psi \models \mathtt{wp}(s, Q) \qquad (2)\ P \wedge \psi \not\models \mathit{false}$$

The first of these conditions states that by assuming ψ, the verification condition $P \Rightarrow \mathtt{wp}(s, Q)$ should be provable. The second condition states that by assuming ψ and the antecedent P, the statement *false* should not be provable. InferMonitorInv saves the result of each of these *abduce* calls in a list Φ representing all candidate predicates. It's also important to note that the *abduce* procedure may return several predicates $\psi_1, \psi_2, ...$ that satisfy these conditions. In this case, all predicates are added to Φ.

Next, InferMonitorInv finds the strongest conjunction of all candidate predicates using predicate abstraction [11, 21]. It does so by computing a fix-point on the conjunction of candidates, pruning all candidates from Φ that are not inductive in monitor M. At each iteration in the fix-point computation, InferMonitorInv makes the following checks to test whether or not candidate ψ inductively holds in M:

1. Does the candidate ψ hold initially in M? (checked at line 11)
2. For each atomic segment in M, does ψ always hold after executing the atomic segment? (checked at line 15)

If the answer to either of these questions is "no", then InferMonitorInv removes ψ from the list of candidates. Notice that a fix-point computation is necessary since the conjunction I of all candidate predicates is used as an assumption in the second check, at line 15. When this fix-point finishes (i.e. when no candidates are removed from ψ in one iteration), InferMonitorInv returns this conjunction I.

4.3 Implementation and Evaluation

In this section, we describe the implementation of Expresso and our evaluation.

Implementation details Expresso operates over a Java-like language and outputs code in Java. We say our input language is "Java-like" because we allow `waituntil(p)` calls, but do not allow these calls to be placed within loops or if-statements. We've described how Expresso determines which signals to place in the output monitor, but Expresso also makes a few more transformations that are trivial but necessary for a correct explicit monitor implementation. In addition to placing signals, Expresso also adds locking mechanisms, condition variables, and converts the implicit signal construct `waituntil(p)` into the corresponding construct for explicit signal monitors.

To ensure that each method in the monitor executes atomically, Expresso creates a single global lock for the monitor. Expresso adds a lock acquire statement at the beginning of each method and a lock release statement at the end. Although this transformation doesn't allow for complex locking patterns, it enforces the standard monitor paradigm (namely, that each monitor method should execute atomically). Expresso also transforms `waituntil(p);s` statements into statements of the form

```
while(!p) {c.await();}; s
```

Here, `c` is a condition variable added via instrumentation to the original monitor. Expresso adds one condition variable to the monitor for each unique `waituntil` predicate. This means that different methods waiting on the same predicate will call `await` using the same condition variable.

Evaluation Methodology In our evaluation, we aim to determine

1. if Expresso could successfully transform a large class of implicit signal monitors,
2. how Expresso compares to similar runtime based approaches, and
3. how Expresso compares to expert hand-written code.

Towards these goals, we gathered a large set of benchmarks from both GitHub and previous works, then evaluated Expresso, AutoSynch, and hand-written code over each of these benchmarks.

We compare these three monitor implementations using saturation tests [18]. These are tests that measure only the amount of time spent in the monitor, not including execution time spent outside the monitor. This allows us to perform a

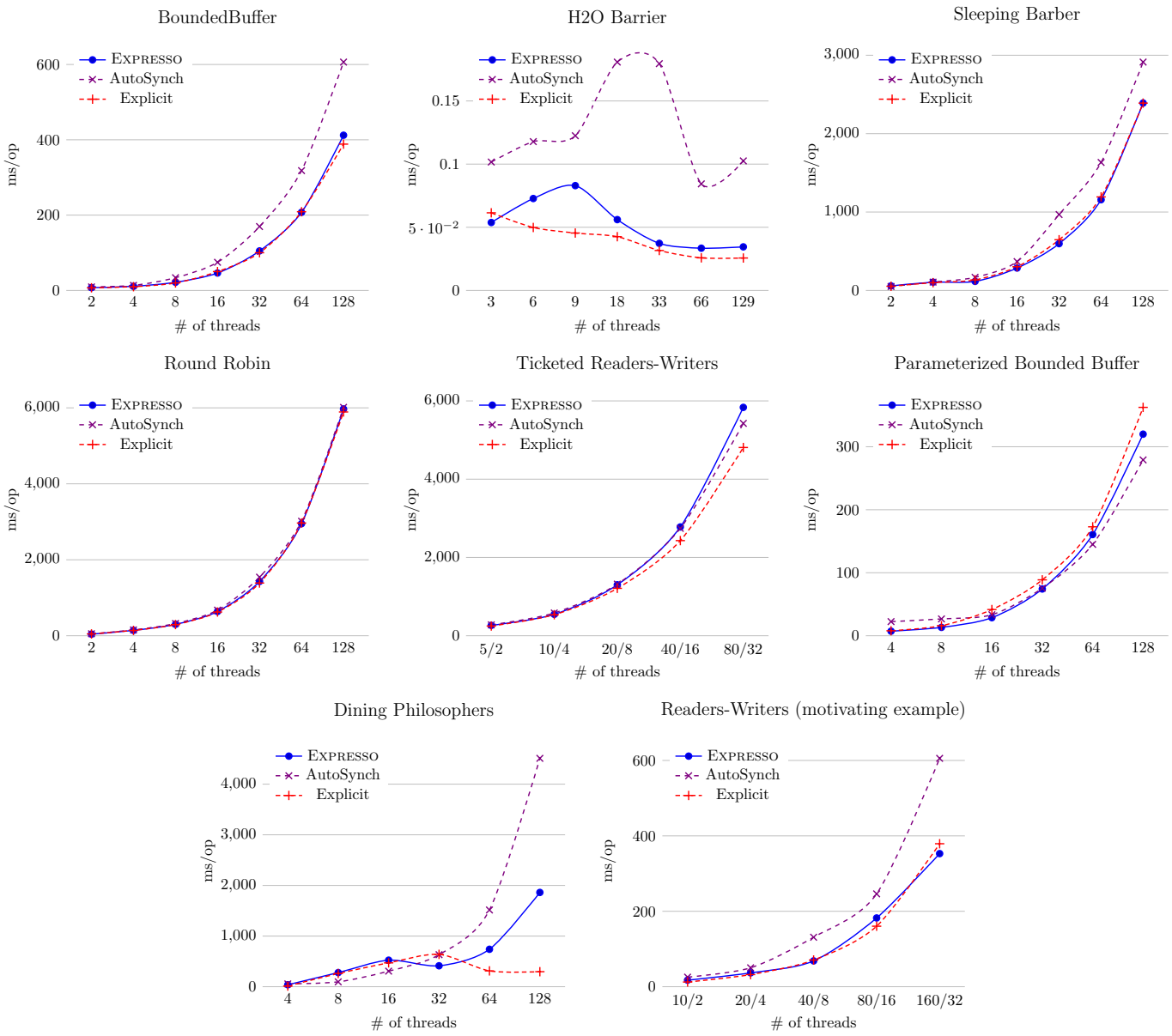


Figure 7: Performance over AutoSynch benchmarks and readers-writers example.

more direct comparison. For each benchmark, we perform a saturation test for each monitor over a varying number of threads. By doing so, we can measure how the performance of each monitor changes as the number of threads increases. To ensure that our measurements are accurate, we run each test 25 times and take the average as our measurement.

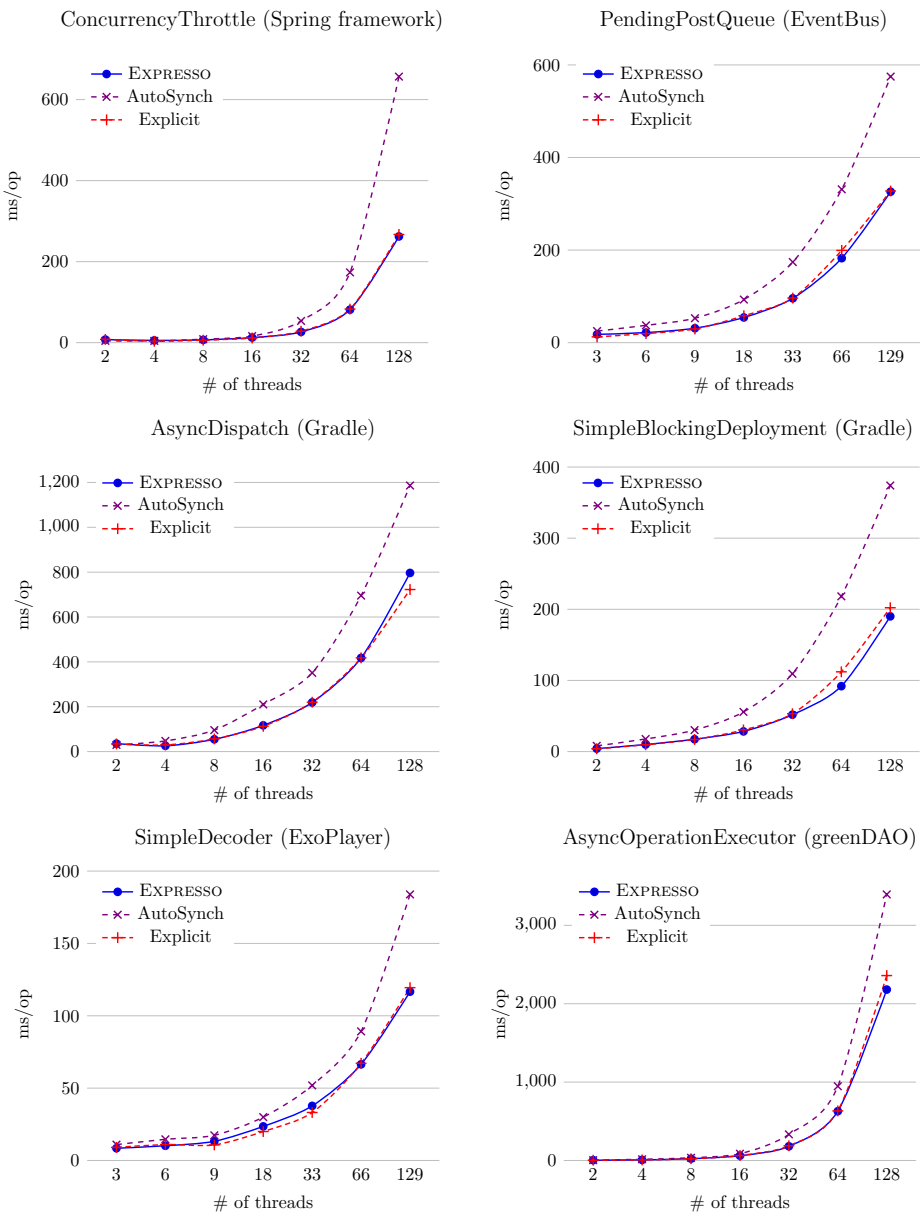


Figure 8: Performance over monitor code in popular GitHub projects.

Benchmark	Time (sec.)
BoundedBuffer	2.5
H2OBarrier	2.3
Sleeping Barber	1.6
Round Robin	1.2
Ticketed Readers-Writers	3.8
Param. Bounded Buffer	2.5
Dining Philosophers	5.4
Readers-Writers	1.5
ConcurrencyThrottle	1.0
PendingPostQueue	0.5
AsyncDispatch	28.3
SimpleBlockingDeployment	0.4
SimpleDecoder	10.7
AsyncOperationExecutor	2.1

Table 1: Compilation time for benchmarks.

Results Above is a graph for each benchmark, displaying the average time spent in each monitor operation (in milliseconds) versus the number of threads. The first set of graphs shows results over the AutoSynch benchmarks, while the second set show results over monitors from popular GitHub projects.

In nearly all of these benchmarks, Expresso outperforms AutoSynch and closely matches the performance of hand written code. On average over all benchmarks and all numbers of threads, Expresso outperforms AutoSynch by 1.56x. Despite this, there are a couple cases where hand written code noticeably outperforms Expresso. The most significant difference can be seen in the Dining Philosophers benchmark. Here, hand written code avoids spurious wakeups by taking advantage of the program structure, only waking threads that can make progress. AutoSynch also outperforms Expresso in some cases, specifically the Ticketed Readers Writers problem and the Parameterized Bounded Buffer problem. For these benchmarks, AutoSynch is able to take advantage of complex data structures to more efficiently check predicates.

Notice that over all GitHub benchmarks, Expresso closely matches hand written code. Since these benchmarks were taken from widely used projects, we consider these results to be a more accurate depiction of the practical capabilities of Expresso. Over these benchmakrs, Expresso outperforms Autosynch by 1.62x on average.

In Table 1, we show the compilation time for each benchmark with Expresso. In nearly all cases, compilation finishes within a few seconds. The noticeable outliers are AsyncDispatch and SimpleDecoder, whose predicates rely on Java library calls that Expresso must analyze. Overall though, there is little noticeable overhead.

5 Related Work

Several other works share similar goals or methodology to the contributions described in this report. In this section, we describe several of these works.

5.1 Programming by Example

Our work is not the first to show a synthesis technique that takes input-output examples as specification. Most closely related to our work, SQLSynthesizer and Scythe take input-output examples and generate SQL queries. FlashExtract and FlashRelate also synthesize table transformations from examples, aiming to automate the process of extracting spreadsheet data. Each of these techniques takes advantage of a specific domain in order to efficiently synthesize code for that domain. Morpheus differs from these works by considering a list of components as input along with the user-defined example. By doing so, the techniques we've described in our work are not limited to only synthesizing data wrangling tasks in R.

Synthesis for domains outside of table transformations have also used programming by example. λ^2 uses programming by example to synthesize transformations for data structures such as trees and lists. Like Morpheus, λ^2 is a component-based synthesis tool and uses deduction to prune parts of the search space. However, unlike Morpheus, λ^2 fixes the set of program components and can only perform deduction for programs with these components. Our technique instead takes a specification for each component as input, and thus is able to perform deduction for programs with any components.

Since this work, there have been additional contributions to synthesis for data wrangling. Neo is a synthesis tool for various domains, including data wrangling, that builds upon Morpheus by adapting ideas from conflict-driven clause learning to synthesis. By learning information from past mistakes, or "conflicts," Neo is able to prune a large amount of the program search space. This additional insight allows Neo to outperform Morpheus over a suite of data wrangling benchmarks.

5.2 Implicit Signal Monitors

Various other works have introduced techniques for automatically signaling threads in concurrent systems. Serializers are a programming language construct that, like monitors, contain queues of threads waiting on a condition. Serializers automatically perform signal operations at runtime, but require a significant runtime overhead compared to explicit signal monitors. Unlike Serializers, Expresso generates signal calls statically and avoids a large runtime overhead.

Several other runtime-based mechanism also implement implicit signal monitors, but as we've previously discussed, the need to dynamically check predicates limits the efficiency of these systems. AutoSynch is one such runtime-based technique, but aims to avoid a large runtime overhead by storing local state information for each waiting thread. When some shared monitor value is updated, AutoSynch uses this local state information to determine which `waituntil` conditions might be affected, restricting the number of necessary predicate checks. Expresso differs from these runtime-based techniques by reasoning about `waituntil` conditions statically. By doing so, Expresso generates explicit signal monitor code without significant runtime or memory usage overhead.

6 Conclusion and Future Work

In this report, we've described two main contributions. First, we introduced a new synthesis technique that synthesizes data wrangling programs in R from input-output examples. This technique takes advantage of SMT-based deduction and partial evaluation in order to synthesize these programs efficiently. Our novel solution also allows us to take in a list of components and specifications as input, making our technique easily adaptable to new domains. Second, we presented a novel technique for synthesizing signal calls in monitor code. With this new synthesis tool, programmers no longer need to reason about complex predicates in their monitor code. Together, these two contributions demonstrate the power of synthesis techniques across domains.

We have identified several directions in which researchers may build upon these contributions. Although Morpheus and Expresso automate a large part of data wrangling and concurrent programming respectively, both fail to automate some parts of these tasks. Morpheus requires some incomplete specification of components, and more precise specifications lead to increased performance. Automatically generating these specifications would further automate the process of synthesis for a variety of domains. While Expresso automates signal placement for monitors, improper shared variable updates or incorrect `waituntil` predicates may also cause concurrency bugs like deadlocks. Statically checking for incorrect updates and predicates would help further limit the introduction of bugs into concurrent systems. Finally, while these works apply synthesis to two new domains, many other domains have yet to be explored. One area especially in need of exploration is synthesis for low-level systems. A synthesis tool that automates part of operating systems programming, for example, would both improve the productivity of systems programmers and reduce the number of bugs introduced into real-world operating systems.

Acknowledgements

I would first like to thank Dr. Isil Dillig for her continuous support throughout my undergraduate career. She was an important guide for all of the work presented in this report, and without her, I would never have explored programming languages research.

I would also like to thank Yu Feng, Dr. Ruben Martins, and Dr. Swarat Chaudhuri for their contributions to Morpheus. Without their help, this work would not have been possible. Additionally, I would like to thank Kostas Ferles and Dr. Yannis Smaragdakis for their work on Expresso. Their guidance and contributions were essential in creating this work.

References

[1] Motivating Example. `http://stackoverflow.com/questions/32875699/how-to-combine-two-data-frames-in-r-see-details`. Accessed 15-Nov-2016.

[2] Stack Overflow Question 1. `http://stackoverflow.com/questions/30399516/complex-data-reshaping-in-r`. Accessed 15-Nov-2016.

[3] Stack Overflow Question 2. `http://stackoverflow.com/questions/33207263/finding-proportions-in-flights-dataset-in-r`. Accessed 15-Nov-2016.

[4] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 243–262. ACM Press, Oct. 2009.

[5] P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Computing Surveys (CSUR)*, 27(1):63–107, 1995.

[6] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, pages 377–387, 1970.

[7] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*, volume 479. John Wiley & Sons, 2003.

[8] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] I. Dillig and T. Dillig. Explain: a tool for performing abductive inference. In *International Conference on Computer Aided Verification*, pages 684–689. Springer, 2013.

[10] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. Reps. Component-Based Synthesis for Complex APIs. In *Proc. Symposium on Principles of Programming Languages*. ACM, 2017.

[11] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/-java. In *International Symposium of Formal Methods Europe*, pages 500–517. Springer, 2001.

[12] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 426–438. ACM, 2015.

[13] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Component based synthesis applied to bitvector circuits. Technical Report MSR-TR-2010-12, February 2010.

[14] L. H. Hadley Wickham, Romain Francois and K. M˙ Technical report.

[15] P. B. Hansen. *Operating system principles*. Prentice-Hall, Inc., 1973.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–583, 1969.

[17] C. A. R. Hoare. Monitors: An operating system structuring concept. In *The origin of concurrent programming*, pages 272–294. Springer, 1974.

[18] W.-L. Hung and V. K. Garg. Autosynch: An automatic-signal monitor based on predicate tagging. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 253–262, New York, NY, USA, 2013. ACM.

[19] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. International Conference on Software Engineering*, pages 215–224. IEEE, 2010.

[20] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, volume 12, pages 221–236, 2012.

[21] S. K. Lahiri and S. Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *CADE*, pages 214–229. Springer, 2009.

[22] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley, 2nd edition, 1999.

[23] A. Stolcke. SRILM - an extensible language modeling toolkit. In *Proc. International Conference on Spoken Language Processing*, pages 901–904. ISCA, 2002.

[24] H. Wickham and L. Henry. Easily tidy data with 'spread()' and 'gather()' functions. Technical report, RStudio, 2018.

[25] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *Proc. International Conference on Automated Software Engineering*, pages 224–234. IEEE, 2013.