Assuage: Assembly Synthesis Using A Guided Exploration

Jingmei Hu
Harvard University
Cambridge, MA, USA

Priyan Vaithilingam
Harvard University
Cambridge, MA, USA

Stephen Chong
Harvard University
Cambridge, MA, USA

Margo Seltzer
The University of British Columbia
Vancouver, BC, Canada

Elena L. Glassman
Harvard University
Cambridge, MA, USA

ABSTRACT

Assembly programming is challenging, even for experts. Program synthesis, as an alternative to manual implementation, has the potential to enable both expert and non-expert users to generate programs in an automated fashion. However, current tools and techniques are unable to synthesize assembly programs larger than a few instructions. We present Assuage: ASsembly Synthesis Using A Guided Exploration, which is a parallel interactive assembly synthesizer that engages the user as an active collaborator, enabling synthesis to scale beyond current limits. Using Assuage, users can provide two types of semantically meaningful hints that expedite synthesis and allow for exploration of multiple possibilities simultaneously. Assuage exposes information about the underlying synthesis process using multiple representations to help users guide synthesis. We conducted a within-subjects study with twenty-one participants working on assembly programming tasks. With Assuage, participants with a wide range of expertise were able to achieve significantly higher success rates, perceived less subjective workload, and preferred the usefulness and usability of Assuage over a state of the art synthesis tool.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**;
• **Software and its engineering** → *Automatic programming*.

KEYWORDS

Program synthesis; interactive synthesis; assembly programming

ACM Reference Format:
Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L. Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472749.3474740

1 INTRODUCTION

Although most software is written in higher level languages, there is still a large body of mission critical software that must be written in assembly, and assembly language, by comparison, is difficult and tedious to write. People write assembly code in many situations: when developing low level, machine-dependent parts of an operating system, when working on resource-constrained embedded platforms, when programming low-level device drivers, and while optimzing code that requires specific instructions that the compiler cannot produce. However, most programmers are neither professional assembly programmers nor familiar with the myriad assembly languages emerging with the end of Moore's law [23]. Most programmers never write assembly programs. It takes time and effort for people to learn different assembly languages and then to debug programs written in them. Assuage is designed for users who are familiar with the high-level concepts of assembly languages, but do not know the syntax for a given assembly language or the details of what each instruction does.

Assembly program synthesis has been proposed as an alternative to manual implementation [20, 28, 48]. The existing assembly synthesis systems [25, 28] leverage *CounterExample Guided Inductive Synthesis (CEGIS)* [46], which is a technique that iteratively generates candidate programs, i.e., sequences of assembly instructions, and then checks whether they satisfy a given specification. If the candidate violates the specification, CEGIS provides a counterexample demonstrating the violation. However, the major barrier to the adoption of assembly synthesis is the limitation of its scalability to unrestricted real-world problems; specifically, the search space of possible programs is combinatorial in the number of machine states and exponential in the number of instructions. This limits the feasibility of assembly synthesis to sequences of only a few instructions [2, 48, 49]. Current strategies for exploiting parallelism to speed up the search in this domain are limited.

Interactive synthesis techniques have been shown to improve synthesis scalability in other programming contexts [2, 10, 13, 21, 32, 41, 44, 54, 55]. Though this is a promising direction, there are two major concerns when applying interactive synthesis to assembly programming: First, assembly programs are hard to understand and tedious to write, possibly impairing users' ability to provide guidance, especially when they are unfamiliar with the specific assembly language being used. Second, for users to help guide the synthesis process, they need to understand details about the synthesis process as it unfolds [54]; however, relative to other synthesis methods, such as exhaustive enumeration, CEGIS has a complex structure, with candidates and counterexamples as intermediates. Moreover, for assembly synthesis, these intermediates contain sophisticated syntax and semantic information that dramatically increases complexity.

To address these concerns, we introduce Assuage (Figure 1), a novel interaction technique for CEGIS-based assembly synthesis that (1) allows users to provide multiple types of guidance during synthesis, (2) provides users with different representations of synthesizer feedback, and (3) enables parallel synthesis to decrease the penalty for users whose guidance is, at times, counterproductive. Assuage supports two types of *interventions*, i.e., user guidance: (1)

UIST '21, October 10–14, 2021, Virtual Event, USA
2021. ACM ISBN 978-1-4503-8635-7/21/10…$15.00
https://doi.org/10.1145/3472749.3474740

constraints enable users to explicitly constrain the instructions and elements of the machine model that programs reference, and (2) *decompositions*, where the user can opt to accept a system-proposed decomposition of the original specification into a set of smaller specifications that might be easier to solve. To help users gain more insight and provide better guidance, Assuage displays the on-going synthesis process with representations of synthesizer feedback, including a synthesis candidate ranking list, frequency analysis of different instructions explored during synthesis, debug information, and satisfaction analysis about the correct and incorrect parts of a candidate's behavior. Finally, in contrast to traditional synthesis, Assuage supports *parallel synthesis*, which spawns new synthesis *instances* whenever users apply interventions, *while allowing previous instances to continue executing*. All these instances run simultaneously and users can choose to apply interventions to any of them. This allows users to try interventions that may not work, without penalty. This design encourages exploration in the presence of uncertainty.

Though parallel synthesis should help users explore more interventions, we were concerned that the complexity of tracking multiple instances would overwhelm users and negatively effect usability. To better understand the effectiveness of Assuage's features, we conducted a within-subjects user study with 21 participants with various levels of expertise. When interacting with all of Assuage's features (including parallel synthesis), participants achieved a significantly higher success rate than when using a state-of-the-art non-parallel synthesis toolchain that provided only instruction constraints and synthesizer feedback. Furthermore, when using Assuage, participants found parallel synthesis intuitive and easy to use; many participants reported that parallel synthesis, combined with multiple types of intervention support, reduced the subjective workload of giving interventions and made Assuage more approachable. By removing the penalty of adding potentially incorrect interventions, we enabled the users to freely explore multiple ideas simultaneously.

In summary, this paper contributes the following:

- An interactive assembly synthesizer, Assuage, that allows the user and the synthesizer to collaboratively search a larger space of assembly programs
- A parallel synthesis approach that reduces the penalty of adding incorrect interventions and allows users to explore multiple ideas simultaneously
- A within-subjects study showing the usefulness and usability of Assuage compared to a interactive synthesis tool that implements the current state of the art.

2 RELATED WORK

Program synthesis is used to automatically generate target programs that satisfy a given specification [3, 6, 12, 34, 40]. There are two major categories of program synthesis: those with complete specifications and those without.

Synthesizing with partial specifications. Partial specification approaches include programming-by-example (PBE) and programming-by-demonstration (PBD) systems, which allow users to demonstrate desired program behavior with specifications consisting of input-output examples or desirable trace demonstrations [8, 37]. Gulwani et al. [17] show that PBE is an effective paradigm for industrial applications, such as string transformations [15], table transformations [16] and data extraction [31]. One of the drawbacks of PBE systems is the incompleteness of their specifications. These systems can provide a synthesized result quickly, but the result might exhibit incorrect behavior due to a case not covered by the incomplete specification [9]. Recent work [45] reveals a counter-intuitive disconnect between the efficiency of PBE systems and users' perceived utility of them.

Synthesizing with complete specifications. This area of program synthesis began in 2006 with the introduction of Sketch [47], in which a programmer provides an incomplete program with holes and synthesizes code to complete the holes. Following this work, syntax-guided synthesis generalizes the partial program with incomplete details yielding a syntactic template [1], and counterexample guided inductive synthesis (CEGIS) [46] generalizes this for infinite state programs. CEGIS uses an iterative process to perform inductive generalization for all possible inputs. The approach in this paper uses CEGIS, where specifications consist of two logical expressions: a *precondition* and a *postcondition*. The precondition is a predicate that holds for the initial state (before the assembly code executes), while the postcondition must hold for the final state (after the assembly code executes).

Compared to PBE systems, a CEGIS specification is complete and precise. However, the search space is large, and these techniques do not scale well for unrestricted real-world problems [9]. While Bornholt and Torlak [5] developed symbolic profiling techniques to identify symbolic execution performance bottlenecks, in synthesis, SMT solving is the performance bottleneck, not symbolic execution. Synthesizing assembly code is significantly harder than synthesizing high-level languages. Assembly programs manipulate untyped memory and global states, leading to enormous search spaces that are exponential in program length and combinatorial in the number of instructions, registers, memory locations, and immediate values [28]. State-of-the-art assembly synthesis tools, such as McSynth, can generate sequences of only about three to five instructions (i.e., 3–5-instruction programs) within a reasonable time period [28, 48, 50], while many assembly code blocks needed for essential operating system services require hundreds or thousands of lines of code (e.g., more than 500 lines for OS/161 [26], a simplified teaching operating system). Srinivasan et al. explored optimizations for synthesizing machine code from semantic specifications with a divide-and-conquer scheme [49, 50], but scalability is still limited. Prior work [24] requires other special tools to handle large cases; Assuage is an alternative, which can synthesize large programs using human guidance.

Trust and performance when interacting with synthesizers. In addition to scalability challenges, modern user studies on program synthesis systems reveal several major usability issues and challenges [9, 18, 33, 39]. General black-box program synthesis leads to a lack of confidence and trust in synthesized programs due to the opaqueness of the synthesis process, even though this approach avoids overloading users with details [39, 54]. The ambiguity of incomplete user-given specifications, such as examples, also misleads the synthesizer to generate plausible, but incorrect programs [33]. Since writing the specification in a general purpose

specification language is sometimes harder than writing a program, the assistant approach with different levels of programming automation has been explored [43].

To the best of our knowledge, interactive CEGIS-based synthesis has not been investigated in the literature. Prior work has shown that for interactive PBE/PBD systems, annotating input-output examples [55] or synthesized program components [41] with include and exclude sets performed well, and participants achieved better performance when they had access to more actionable information about the on-going enumerative synthesis process [54]. Similarly, in the domain of CEGIS-based synthesis, Assuage both supports constraints that allow users to define include/exclude sets on both instructions and portions of the machine state used in a program and reveals actionable information about the underlying parallel CEGIS synthesis processes.

Techniques to clarify user intent. Many methods have been proposed to clarify user intent during program synthesis when the user-provided specification is incomplete. Most PBD and PBE systems allow users to actively provide additional examples to disambiguate their intent. There are several approaches that automate this process by generating examples that distinguish multiple plausible candidate programs [30, 35, 51]. REGAE [55] extends this work by supplementing user-provided examples with many additional strategically generated inputs that show the outputs of user-selected synthesized program(s) on the input space nearby user-provided examples as well as possible unanticipated corner cases far away. By looking at the outputs of synthesized programs on these many additional inputs, users can clarify their intent by selectively pulling input-output pairs into the set of user-provided examples describing their intent, preserving or changing their desired output for that input in the process. Unlike PBE/PBD, CEGIS-based assembly synthesis requires a complete specification of what the user wants in terms of pre-conditions and post-conditions that must be satisfied; therefore, instead of asking the user to provide additional examples or demonstrations to clarify what they want, the user provides additional constraints to reduce the synthesis search space.

Techniques to communicate synthesis progress. There have been many attempts to present synthesized program candidates to the user *after* the synthesizer finishes its given synthesis task. Topaz [38] generalized the cursor movement capabilities from text editors to a graphical domain for graphical program synthesis, while Rousillon [7] demonstrates synthesized scripts with a graphical interface showing hierarchical information for web scraping. FlashProg [35] introduced *program navigation* that allows user to navigate between all programs synthesized by the underlying PBE engine and pick the desired one. Zhang et al. [54] trace and visualize various views of all the programs enumerated during the synthesis process *as the synthesis process is running*, before the synthesis task is complete. For example, they show a line chart of how many examples each candidate satisfies and a tree view representing the space of programs explored so far. Assuage extends this work to CEGIS-based assembly synthesis. Analogous to Zhang et al.'s line chart, Assuage has a chart of how many test cases satisfy the specification after executing candidates from each of multiple parallel synthesis instances. Analogous to their tree view of enumerated programs, Assuage shows the top candidates generated by the CEGIS synthesizer along with their execution trace, which can help the user recognize additional, potentially helpful interventions to add.

Parallel synthesis. Jeon et al. [29] propose a synthesis technique to combine symbolic search and explicit search by partially concretizing a randomly chosen subset of unknowns. These random trials of the algorithm can be run in parallel—solving problems that were otherwise intractable. Instead of exploiting an inherently parallel algorithm, Assuage enables users to explore multiple possibilities under various interventions in parallel.

Mixed-initiative systems. Horovitz [27] introduced the principles of mixed-initiative user interfaces, which seek synergies between intelligent services and users. With mixed-initiative interaction, the intelligent services and users collaborate efficiently to achieve the user's goal. Assuage leverages this interaction approach using the best of human and computer abilities enabling both the computer and the human to take initiative and make decisions, i.e., the synthesizer does the computational work of producing candidate programs and automatically suggesting some interventions, while the human applies interventions that reflect higher-level insight into the final program structure and content. Assuage supports both multiple types of interventions to let users guide and expedite synthesis and multiple visualizations of synthesizer feedback about the proposed candidates and the counterexamples generated during synthesis.

3 PRELIMINARIES

We adopt a CEGIS-based assembly language synthesis toolchain from existing work [25, 28]. The goal of general assembly synthesis is to automatically produce assembly code from two inputs: a machine model and a specification. The machine model is the machine description, which provides an executable model of an instruction set architecture. It declares machine states, such as registers and memory locations, and defines the semantics of assembly instructions. The specification describes the intended functionality of the target program with pre- and postconditions. We refer the reader to prior work [25] for a detailed description of the synthesis algorithm.

The synthesizer uses a CEGIS technique, which iteratively produces assembly programs as candidates that are tested against an accumulated set of counterexamples (i.e., a set of initial machine states that satisfy the precondition and violate the postcondition). The conventional CEGIS-based synthesizer starts with 0-instruction program synthesis (called stage 0) and proceeds to stage $n+1$ when synthesis at stage n fails, iterating until synthesis succeeds. In each stage, the synthesizer iteratively suggests a candidate program that might satisfy the given specification, i.e., satisfies the specification in the presence of the current counterexample set, provides additional counterexamples that make the candidate program violate the specification (i.e., the postcondition), and adds them to the accumulated set. This procedure continues until the synthesizer fails to find either an appropriate candidate or more counterexamples.

Since we assume that our target user is familiar only with the high-level concepts of assembly language but not the exact syntax or behavior details, we categorize assembly instructions into type groups that make sense to users, shown in Table 1. These

Coarse-grained types	Fine-grained types	
ARITH General Arithmetic	ADD Addition	CMP Comparison
LOGIC General Logical	BIT Bitwise Logic	SHIFT Shift/Rotate
MEMOP Memory Handling	LOAD Load from	STORE Store into
DATAOP Data Transfer	MOV Data Move	
JMP General Branch		
COPROC Coprocessor Handling		

Table 1: Assembly Instruction Types. We categorize into coarse-grained and fine-grained types. Users can refer to both in Assuage.

type groups can, however, overlap. For example, an instruction that performs subtraction, e.g., a `sub` instruction, is both a general arithmetic instruction with type `ARITH` and an addition operation with type `ADD`. When specifying type information, users can both use coarse-grained categories (left column in Table 1) and more fine-grained categories (right columns in Table 1). Type groups provide high-level primitives that abstract away low-level syntax and semantics for assembly languages. These type groups cover all instructions declared in the machine model.

4 USER SCENARIO

The following scenario illustrates how an engineer, Alex, can use Assuage (Figure 1) to synthesize ARMv7 code that implements some exception handling code in the Barrelfish operating system [4]. Specifically, Alex wants to write assembly code that satisfies the following specification, expressed in pseudocode (shown in the left sidebar in Figure 1):

```
1. Mem: memory region with 4 slots
       ([Mem, 0], [Mem, 4], [Mem, 8], [Mem, 12])
2. cond: boolean = *R3 < load_from([Mem, 8])
3. precondition: *R2 == [Mem, 0]
4. postcondition: if cond then *R1 == 0x1 else *R1 == 0x0
```

Line 1 indicates that the code can use four locations (called slots) in memory `Mem`. The precondition (Line 3) requires that before the code implementing this specification executes, register `R2` must contain a pointer to the specific memory location (`[Mem, 0]`), while the postcondition (Line 4) requires that when the code finishes executing, register `R1` contains `0x1` if the variable `cond`, which is defined in Line 2, is `true` or `0x0` if `cond` is `false`. The variable `cond` (Line 2) is a boolean condition that stores the comparison result between the contents of a register (R3) and a value stored in a particular location in memory (`[Mem, 8]`). Alex has some basic understanding of MIPS assembly language but has never written ARM assembly programs, so she decides to use an assembly synthesizer instead.

Alex starts the synthesizer. In real time, Assuage generates candidate implementations of the specification and displays information on each candidate generated (shown in the middle in Figure 1). Using the specification, Assuage generates 20 test cases consisting of initial states that satisfy the precondition. For each candidate, it presents the user with a score, indicating the number of test cases for which the candidate satisfies the postcondition. In other words, the higher the score a candidate achieves, the closer it is to a correct implementation.

The synthesizer first starts trying single-instruction candidate programs. It quickly determines that no one-instruction program can satisfy the specification, so it moves on to two-instruction programs. Within about a minute, it starts generating candidates consisting of three instructions, after determining that there is no two-instruction solution. As this synthesis process proceeds, Alex watches the live-updated score chart (Figure 2A), which shows the highest score that any candidate has achieved so far, and examines the top candidates table (Figure 2B), which shows the five candidates with the highest scores.

While impressed that the best candidate so far is a two-instruction program, Alex suspects the synthesizer has wasted time evaluating poor candidates. Therefore, she decides to inspect patterns across high-scoring candidates for inspiration. To do so, Alex clicks the occurrence table (Figure 2C), which shows the frequency of each assembly instruction type that appeared in previous candidates and the average scores of those candidates. Although the synthesizer struggled with low-scoring candidates, Alex finds it helpful to see what kinds of instruction types produced consistently better average scores. She notices that `LOAD` and `MOV` have occurred frequently and that candidates with those instructions have higher scores than others. She also finds in the specification that the variable `cond` (Line 2) uses the `load_from` function to read a value from memory. These observations give Alex some clues, e.g., `LOAD` should be present in the program. She then clicks on promising candidates from the candidate table (Figure 2B) to get a detailed execution trace and the specification analysis for each candidate. Using the execution trace, she knows which parts of the specification were satisfied or unsatisfied by each candidate (Figure 2D). She notices that whenever there is a `LOAD`, part of the specification is satisfied.

Base on these observations, she decides that the program must read from memory before it does anything else. Though she is not quite sure whether `MOV` is necessary for the program, she decides to explore two ideas in parallel: one where `MOV` is included in the program and one where `MOV` is excluded from the program. Assuage allows the user to explore multiple ideas in parallel instead of committing to just one. Alex adds two interventions: first, she selects the radio button for the first instruction and clicks the "Include" button. In a pop-up window for "Include", she selects `LOAD` from a list of instruction types (shown in Table 1) to indicate that `LOAD` should be the first instruction of the program. She also selects the whole program radio button and marks `MOV` as included, indicating that `MOV` should appear somewhere in the program (Figure 3A); this kicks off a new synthesis instance running in parallel with the first, i.e., the original instance. Second, she selects the original synthesis instance and then adds a different intervention: she again marks that `LOAD` should be the first instruction in the program and instead clicks on "Exclude" and selects `MOV` from the pop-up window to indicate that `MOV` should not be used anywhere in the program (not pictured), kicking off a third synthesis instance running in parallel.

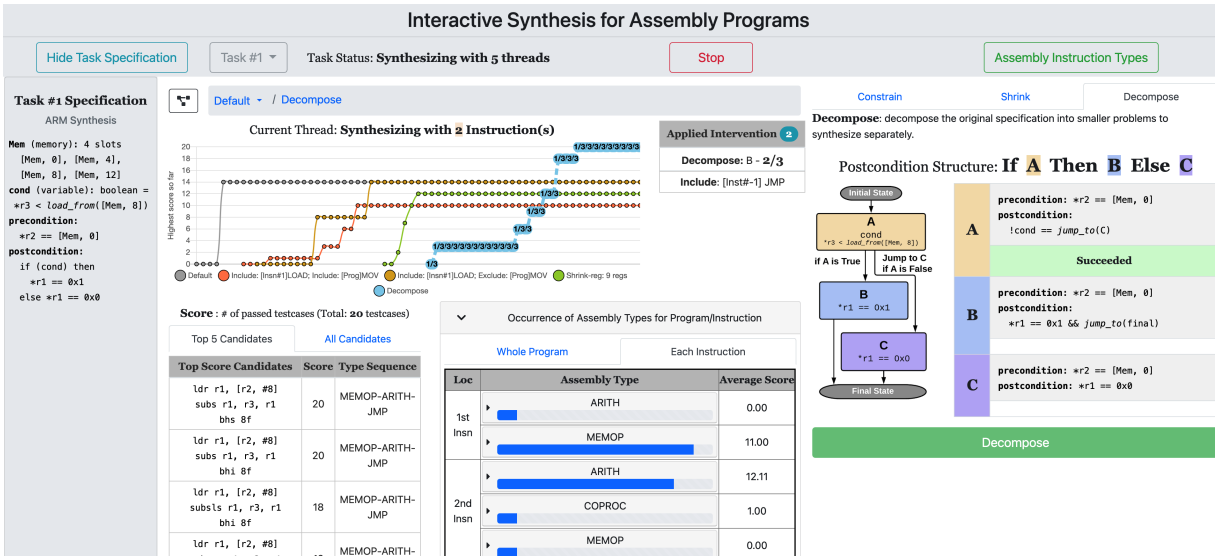

Figure 1: Assuage Interface. The side bar on the left shows the task specification, the middle panel contains visualizations of the synthesis process, and the right panel presents the user with interventions that can be used to guide synthesis. In this snapshot of Assuage, there are five instances running in parallel (described in Section 4); the currently selected instance is using a decomposition intervention in which the first part of the decomposed specification (part A) has succeeded, as indicated by the green highlighting.

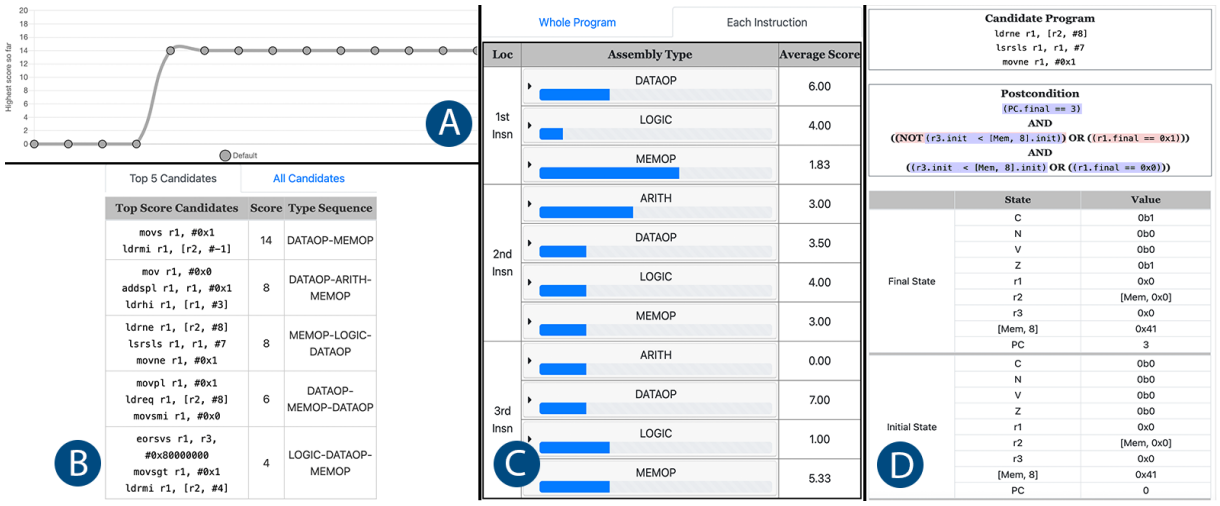

Figure 2: Pre-intervention synthesizer feedback. (A) displays the highest score among all generated candidates, (B) shows a list of the candidates with the top five highest scores, (C) indicates how frequently each instruction type appears across all candidates synthesized so far. Alex has selected the third candidate in B, which looks promising and (D) indicates which parts of that selected specification have been satisfied (blue) and which have not (red), and provides the values of multiple machine states at different program points.

This means that Assuage will not use MOV instructions in any future candidates for this third instance. In summary, when Alex adds each intervention, Assuage creates a new synthesis instance using the union of the currently selected instance's interventions and the newly added one. The live-updated score chart (originally shown in Figure 2A) now shows multiple lines, each representing one of the parallel instances labeled with the user-added interventions that caused their creation (gray, orange, and gold-colored lines in Figure 3D).

So far, Alex has only considered restricting the types of instructions that Assuage will use, but she is eager to try more types

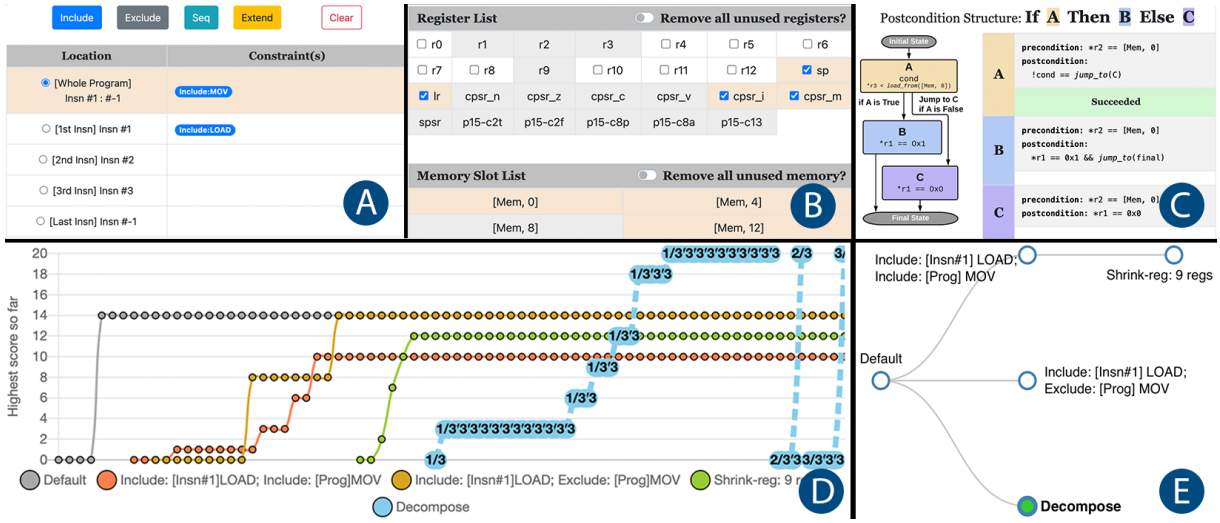

Figure 3: Parallel synthesis and user interventions in Assuage. To guide synthesis, Alex can either (A) constrain programs with instruction-level details, i.e., directly mark partial programs as desired or undesired in the final results, (B) eliminate irrelevant and unused registers and memory locations, or (C) ask the synthesizer to decompose the specification into a set of smaller, easier-to-solve sub-problems. With each of these user actions, new synthesis instance are initiated, running in parallel. Assuage is running with five synthesis instances with different interventions, shown by the five lines in D and the five nodes in the tree view E. Solid lines in D represent the synthesis instance without decomposition, while dashed lines represent multiple sub-instances under decomposition.

of interventions. With three synthesis instances running in parallel, Alex notices that some arbitrary registers (such as R5 and R6) appear in candidates but not in the specification; she thinks these other registers might be irrelevant for this code sequence. To guide Assuage towards her intuition, Figure 3B shows how she removes all the registers that do not appear in the specification, except for some flag registers that she thinks might affect control flow. Alex applies this intervention on top of one of her previous interventions (the one with LOAD and MOV both marked for inclusion), because after inspecting the progress of different synthesis instances by clicking on different lines in Figure 3D, she thinks that this synthesis instance looks most promising. At this point, all four instances are shown with solid lines in Figure 3D with different color annotations (gray, orange, gold, and green lines).

With Alex's guidance, the synthesizer starts to synthesize programs of four instructions. While waiting, Alex notices that the postcondition has an *if-then-else* structure, suggesting the possibility of breaking this synthesis problem into small pieces to be synthesized separately. Assuage proposes a possible decomposition (Figure 3C). Alex confirms that it seems reasonable and decides to give it a try. She adds this decomposition as a new intervention, shown as the blue dashed line in Figure 3D. At this point, five synthesis instances are running in parallel as shown in Figure 3E, and Alex can sit back and continue to watch the best scores associated with each instance rise or inspect any individual instance to better understand why its best candidates are not yet full solutions. Figure 1 shows a screenshot of Assuage with all five interventions applied by Alex.

Before any of the other instances are able to reach a full solution, the blue dashed line representing the instance with the decomposition intervention climbs up to a full score once for each smaller synthesis challenge that makes up the full decomposition. That means that all components of the decomposition have been synthesized completely, and together they satisfy the original specification. Assuage returns this sequence of assembly instructions to Alex for final inspection. Alex is surprised that this interactive synthesizer Assuage took about 15 minutes to produce an assembly program that satisfied the given specification, because without her guidance, a traditional synthesizer takes hours to finish.

5 DESIGN AND IMPLEMENTATION

Developing a collaborative system that seamlessly combines human intuition and expertise with automated CEGIS-based assembly synthesis requires an interaction model: we first describe how users can intervene to guide synthesis, then how Assuage gives feedback to users for better understanding, and finally how Assuage exposes parallelism.

5.1 User Interventions

Assuage provides two types of interventions that allow users to provide information to the synthesizer: *Constraints* and *Decomposition*. *Constraints* include *instruction constraints*, which allow users to constrain specific instructions, and *location constraints*, which allow users to eliminate irrelevant registers and memory locations to expedite synthesis. *Decomposition* breaks the specification into smaller problems. Assuage automatically provides suggestions for *location constraints* and *decomposition* based on the specification;

if the user likes these suggestions, she can create a new instance including the new intervention, either by adding it to the set of interventions present in an already running instance or by creating an instance with only the suggested interventions.

Constraints. Assuage allows users to introduce constraints on instructions for the target program and the machine model that are used for assembly synthesis.

Instruction Constraints enable users to place constraints on instruction choice and ordering as shown in Figure 3A. We refer to both coarse-grained and fine-grained instruction types in Table 1 as $t_1, t_2, \ldots, t_n$, and their corresponding grouped instruction sets as $G_{t_1}, G_{t_2}, \ldots, G_{t_n}$ for convenience. Given a candidate assembly program P with x instructions, i.e., $final(P) = I_x(I_{x-1}(\ldots(init(P))\ldots))$, where $init(P)$ and $final(P)$ represent the initial and final machine states, respectively, and I_i represents the i-th instruction in the program P, we introduce the following constraints, which impose restrictions on either parts of the program or the entire program behavior.

- *Include* (G_{t_i}) where $i \leq n$: will hold for programs P' with y ($y \geq x$) instructions where $\exists k\ (k \leq y), I_k \in G_{t_i}$.
- *IncludeLoc* (G_{t_i}, j) where $i \leq n \wedge j \leq x$: will hold for programs P' with y ($y \geq x$) instructions where $I_j \in G_{t_i}$.
- *Exclude* (G_{t_i}) where $i \leq n$: will hold for programs P' with y ($y \geq x$) instructions where $\forall k\ (k \leq y), I_k \notin G_{t_i}$.
- *ExcludeLoc* (G_{t_i}, j) where $i \leq n \wedge j \leq x$: will hold for programs P' with y ($y \geq x$) instructions where $I_j \notin G_{t_i}$.
- *Seq* ($G_{t_i}, \ldots, G_{t_j}$) where $(i \leq n) \wedge (j \leq n)$ and $Length(G_{t_i}, \ldots, G_{t_j}) = l$: will hold for programs P' with y ($y \geq l$) instructions where $\exists k\ (k \leq y - l), I_k \in G_{t_i} \wedge \cdots \wedge I_{k+l} \in G_{t_j}$.
- *Extend* (y) where ($y > x$): will extend the program length to y instructions.

Inclusion, i.e., *Include* and *IncludeLoc*, requires that one or one specific group of instructions appears in the target program. In Section 4, through interacting with the Assuage interface, Alex implicitly added *IncludeLoc* (`LOAD`, *1*) to force the first instruction to be a `LOAD`-like instruction. *Exclusion*, i.e., *Exclude* and *ExcludeLoc*, rules out a specific group of instructions for the whole program or for some specific location in the program. Likewise, in Section 4, Alex implicitly added *Exclude* (`MOV`) to rule out any data movement instructions. *Seq* defines a partially ordered sequence that must appear in the target program. For example, Alex could click the "Seq" button in Figure 3A and select `LOAD` and `MOV` in order, to implicitly add *Seq* (`LOAD`, `MOV`), which requires that a partial sequence containing `LOAD` and `MOV` must appear, consecutively and in order, in the synthesized program. *Extend* directly extends the program to the user-specified length.

Location Constraints enable users to eliminate use of certain registers or memory locations in the synthesized program, as shown in Figure 3B. Machine models contain more information than is strictly necessary to facilitate synthesis. The complete machine model contains the descriptions of all registers, memory locations and instructions for the entire system. Removing registers and memory locations that are not needed to produce a correct implementation for a specific specification will speed up synthesis and reduce a user's cognitive load. Assuage collects all related registers and memory locations that a specification might access (explicitly or implicitly), compares them to the complete machine model, and recommends a shrinkable location set to the user. This shrinkable location set includes both registers and memory entries that contain arbitrary values without specification restrictions. Removing those arbitrary location elements should not affect program correctness. For example, a specification might access only a subset of the slots inside a memory region. Eliminating the irrelevant and unused head and tail slots in memory regions can reduce the search space and expedite synthesis. While Assuage analyzes the specification and proposes a reduced machine model, we rely on user guidance to apply the reduction. While these reductions frequently work, they are not guaranteed to be correct when, for example, an implementation requires temporary storage (e.g., producing a swap function). One could imagine letting Assuage automatically create new instances for these suggested location reductions, but we leave investigation of this approach for future work.

Decomposition. Since the search space and synthesis time grows exponentially in the number of instructions, our intuition was that breaking the problem down and solving smaller problems would improve synthesis performance considerably [42]. As it is overwhelming to consider the specification as a whole, we distinguish three statement structures that might comprise the postcondition: *if-then-else (ITE)*, *conjunction (AND)*, and *disjunction (OR)*. Given a specification with precondition *Pre* and postcondition *Post* with one of these structures, Assuage proposes one of the following decompositions:

- *Post = if A then B else C*: The *ITE*-like specification can be decomposed into three blocks as shown in Figure 3C. Each block has its own specification, referred to as *sub-specification*. These three blocks share the same precondition *Pre*. The first block contains two exit points, which allow it to branch into different following blocks based on the condition *A*. The second and third blocks synthesize for *B* and *C*, respectively, and they both exit to the ultimate exit point of the entire program. Concatenating the three blocks produces three smaller synthesis problems that satisfy the specification *Post*.
- *Post = A and B*: The *AND*-like specification can be decomposed into two blocks. Unlike the *ITE* format, these two blocks are coherent and no control flow is necessary. The first block achieves the partial postcondition *A*, while the second block takes the synthesized program for *A* as a prefix and synthesizes the program for the entire *Post (A and B)*.
- *Post = A or B*: The *OR*-like specification cannot simply be separated into *A* and *B*, but we can rearrange *Post* into the *ITE* format, *if A then exit else B*, and decompose it into two blocks. Similar to the *ITE* format, the first block contains two exit points, one to the ultimate exit point of the entire program and the other to the entry of the second block, based on the condition *A*. The second block achieves the partial postcondition *B*. Concatenating the two blocks produces two smaller synthesis instances the satisfy the original specification *Post*.

These proposed decompositions may be less efficient or produce an even more difficult synthesis problem. We rely on user guidance to determine of the proposed decomposition is a good avenue of

exploration. As suggested in the discussion of machine model reduction, this is another area where letting Assuage automatically generate instances using the proposed decompositions could prove fruitful.

5.2 Synthesizer Feedback

Empowering users to intelligently apply all these interventions requires that the synthesizer provides information that helps the user understand how synthesis is progressing. Assuage presents the following dynamically updated information during the synthesis process:

Candidate Scoring. To visualize approximately how close various candidates are to satisfying the specification, Assuage generates M arbitrary initial states satisfying the precondition as test cases and evaluates the candidate programs P against them on every synthesis iteration. If the corresponding final states of N of the M initial states verify successfully after executing a program P, i.e., they satisfy the postcondition, Assuage grades the candidate P with a score N/M (*Score (P)* = N/M). This scoring mechanism introduces a trade-off between the time overhead needed to run all the test cases and the user's perception of a candidate's quality. We discuss the selection of $M = 20$ in Section 6.

During synthesis, the score chart (Figure 2A) gives user real time updates about the synthesis process. The x-axis of the score chart shows the total number of candidates that have been tried by the synthesizer across all parallel synthesis instances; the y-axis shows the highest score among the generated candidates. This view allows a user to monitor synthesis progress over time. For example, if one line suddenly increases dramatically, it indicates that synthesis has produced some better candidates, and the user might want to inspect the corresponding candidates to understand what is preventing them from fully satisfying the specification; the right information should help them translate their observations into helpful interventions. Assuage also displays a ranking list of the candidates with the top five highest scores for a given synthesis instance (Figure 2B), so the user can determine the cause of a given instance's higher scores.

Type Frequency. To provide a more holistic view at the instruction level, the frequency table (Figure 2C) shows, for a selected instance, how frequently each assembly instruction type appears across all candidates synthesized so far. For each individual synthesis instance, Assuage shows the type frequency for the whole program and for different instruction locations (e.g., "1st Insn" and "2nd Insn" in Figure 2C) and the average score of candidates that include each instruction type. For example, if the frequency of `LOAD` is extremely high and its average score looks good, it suggests that the final program should contain a `LOAD`. By default, Assuage shows coarse-grained type information (left column in Table 1) and when the user clicks on a specific type, it shows the frequency of its fine-grained types (right columns in Table 1). Compared to the score information, type frequency indicates a more concrete view of the target program and provides a sense of the connection between instructions and their behaviors against the specification. However, it also requires a deeper understanding of assembly language. Our assumption is that instructions that are frequently chosen by the synthesizer or get high scores on average have a higher probability of occurring in the target sequence. Type frequency information with candidate scores gives users intuition about the potential target sequence and the possible constraints to apply.

Specification Analysis. To let users constrain programs (as mentioned in Section 5.1), Assuage applies each candidate P and its corresponding counterexamples CE (a set of initial machine states) to the postcondition expression and creates an execution trace by concretizing all immediate states after each instruction with P and CE. This is the debug information shown to the users, i.e., execution values at every program point and a specification analysis in which Assuage highlights *specification-satisfying* and *specification non-satisfying* parts of P's behavior during each iteration, i.e., the parts that lead to postcondition violations with different colors (blue and red highlights in Figure 2D). Highlighting the unsatisfied parts helps a user make further suggestions. For every specification non-satisfying part, Assuage also explicitly presents the way it was generated from the initial machine states (`.init` postfix notations in Figure 2), which helps the user understand why that part violates the postcondition . The specification analysis and execution trace provide a detailed view, which requires a good understanding of assembly language concepts to appreciate and exploit. It helps users confirm their guess about some specific candidates. Experts will be more likely able to benefit from this particular feature than non experts.

5.3 Parallel Synthesis

Assuage provides an integrated representation of the interventions that users have performed so far during synthesis. When the user adds an intervention, Assuage creates a new synthesis instance using the union of the currently selected instance's interventions and the new one. The tree view (Figure 3E) shows all existing instances, where every child instance contains all the interventions of its parent. Assuage also plots multiple lines in the score chart, as shown in Figure 3D, where each line represents the progress of one synthesis instance. Assuage highlights the currently selected instance with a thicker border in Figure 3E and thicker lines in Figure 3D.

Compared with traditional synthesis, Assuage's support for parallelism allows users to explore multiple possible interventions simultaneously. By spawning new synthesis instances when users add interventions, while continuing to run all existing instances, parallel synthesis reduces the penalty of making mistakes.

6 USER STUDY

To validate that assembly synthesis can be scaled with user interventions and to evaluate the usefulness and effectiveness of Assuage, we conducted a within-subject study with 21 participants. As a baseline, we implemented a CEGIS-based synthesizer that represents the state of the art in traditional synthesis interface affordances; Table 2 illustrates the comparison between the control and experimental synthesizers' capabilities. Instead of using a traditional CEGIS-based synthesizer, we compared against a baseline condition where participants completed tasks using a traditional synthesizer that provided feedback on the synthesis process and allowed users to apply instruction constraints only. The experimental

	Synthesizer Feedback			User Interventions			
	Candidate Scoring	Type Frequency	Specification Analysis	Instruction Constraints	Location Constraints	Decomposition	Parallel Synthesis
Control	✓	✓	✓	✓			
Experiment (Assuage)	✓	✓	✓	✓	✓	✓	✓

Table 2: Controlled user study design of the control and experimental synthesizers.

synthesizer, i.e., Assuage, had all the features described in Section 5 enabled. We ask the following research questions:

- **RQ1**: *Compared to the prior state of the art, can Assuage help a user more efficiently propose interventions and more quickly arrive at a specification-satisfying program?*
- **RQ2**: *How does Assuage affect users' subjective workload and experience during assembly synthesis?*
- **RQ3**: *How do users respond to Assuage holistically?*
- **RQ4**: *How do users respond to each feature of Assuage, including parallel synthesis?*
- **RQ5**: *What obstacles do users encounter when using Assuage for interactive (assembly) synthesis?*

6.1 Participants and Settings

We recruited 21 participants (3 female and 18 male). Seventeen were recruited through mailing lists of several research groups at two R1 universities, and four were reached through professional networks. Of these four, three participants knew at least one author, but were not involved in the project. Participants received a $25 Amazon gift card as compensation for their time. Nine participants were graduate students, five were undergraduate students, and the other six were professional developers. Participants had a diverse range of prior experience with assembly language. Ten participants said they knew assembly basics but only used it several times, seven said they were familiar with assembly languages and have used them many times, and four said they were experts in assembly and remembered most of the syntax and semantic details. The majority of participants (15/21) said, when writing assembly programs, they often had to search online for the specific instruction set architecture. All non-expert participants considered writing in assembly more difficult than writing in other familiar languages. We conducted all studies using a Ubuntu 18.04 LTS computer with 32G of memory.

6.2 Tasks

To design realistic programming tasks for assembly programs, we selected two tasks derived from the Barrelfish operating system [4] and the book *Hacker's Delight* [52], which is commonly referred to as the "Bible of bit twiddling hacks" [19]. The specifications of these two tasks in pseudo code and their solutions in ARM assembly sequence are listed below, Note that these tasks might have multiple correct solutions. In our study, the specifications are written by a domain expert and given to users, who are not allowed to alter them. We restrict the synthesizers to searching for assembly programs of up to 4 instructions for one specification.

Task 1. This task is the same as the example mentioned in Section 4. Without any interventions, the CEGIS-based synthesis for this task finishes in about 6.1 hours on the same machine used in the user study. Using Assuage, an omniscient user, who knows which interventions should be applied, i.e., the correct decomposition and constraints, can successfully synthesize a 12-instruction program with control flow (4 instructions for each decomposed sub-specification and 3 decomposed sub-specifications in total) in about 5 minutes.

```
Specification:
    Mem: memory region with 4 slots
        ([Mem, 0], [Mem, 4], [Mem, 8], [Mem, 12])
    cond: boolean = *R3 < load_from([Mem, 8])
    precondition: *R2 == [Mem, 0]
    postcondition: if cond then *R1 == 0x1
        else *R1 == 0x0

ARM assembly sequence:
        ldr r1, [r2, #8]
        cmp r3, r1
        movlo r1, #1
        movhs r1, #0
```

Task 2. This task is derived from two benchmark examples: turning on the rightmost 0-bit and turning off the rightmost 1-bit in a 32-bit vector. Without any interventions, the CEGIS-based synthesis for this task finishes in about 1.2 hours. The omniscient user using Assuage can successfully synthesize a 4-instruction program (2 decomposed sub-specifications) in about 3 minutes.

```
Specification:
    val: 32 bit = *R1
    precondition: true
    postcondition: ( *R2 == (val + 0x1) & val )
        && ( *R3 == (val - 0x1) | val )

ARM assembly sequence:
        add r2, r1, #1
        and r2, r1, r2
        sub r3, r1, #1
        orr r3, r1, r3
```

6.3 Methodology

We conducted a 75-min study session with each participant and, with permission, recorded the session. In each session, with a think-aloud protocol, participants completed one of the two tasks using the synthesizer in the control condition and the other task using the experimental synthesizer (i.e., Assuage). To mitigate any learning effects, both the order of tasks and of interactive synthesizers were counterbalanced across participants through random assignment. Before each task, participants were given a tutorial video of the features of the synthesizer they would have access to during that

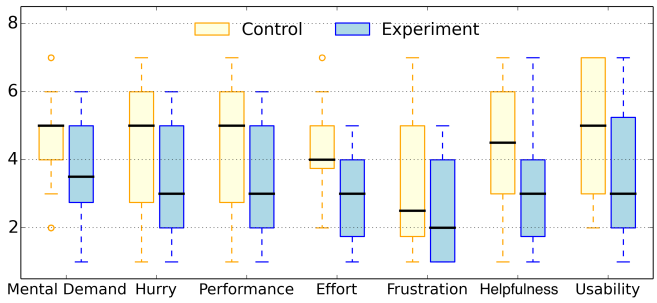

Figure 4: Subjective workload and usability measurement. The answers for helpfulness and usability are reverse scored. Users perceived less mental demand, felt less time pressure, spent less effort, and gave themselves better performance ratings with Assuage. Users considered Assuage more helpful and usable.

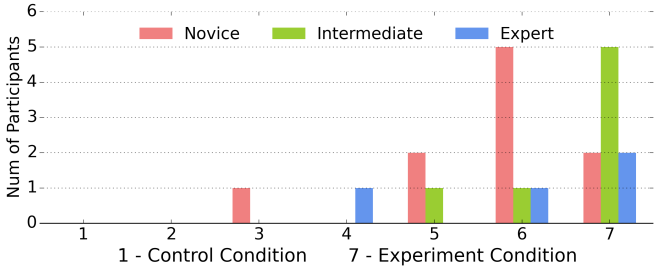

Figure 5: The preference of different expertise levels. Users with different level of expertise preferred Assuage for assembly synthesis.

task. They were then given 20 minutes to finish the assigned task. The task was considered failed if participants did not guide the synthesizer to a specification-satisfying assembly sequence within that time limit. After each task, participants filled out a survey about their experience using the assigned synthesizer. The survey included questions shown in Table 3, i.e., five NASA Task Load Index questions [22] to rate their perceived subjective workload during the task and two questions about the usefulness of the assigned synthesizer. After finishing both tasks, participants answered a final survey to directly compare their experiences using each synthesizer. We open-coded participants' responses with themes and used them to shed light on the underlying reasons for the quantitative results in the following section.

7 USER STUDY RESULTS

7.1 User Performance

In the experiment condition (i.e., using Assuage), 20 of 21 participants successfully guided the synthesizer to a correct assembly solution, while only 8 participants finished the task in the control condition. Fisher's exact test [11] on the performance comparison shows that the difference is statistically significant ($p < 0.001$). To compute average task completion time, we assigned the time-out limit of 20 minutes to those users who were unable to complete a task. The average task completion time with Assuage was 9.28 minutes, while the average task completion time in the control condition was 17.57 minutes. Welch's t-test [53] shows the mean difference of completion time is statistically significant as well ($p < 0.001$). As for different tasks, all eleven participants using Assuage for Task 1 finished successfully (8.31 minutes on average) and nine out of ten participants using Assuage for Task 2 finished successfully (10.25 minutes on average). By contrast, in the control condition, five out of ten participants finished Task 1 (17.24 minutes on average) and three out of eleven participants finished Task 2 (17.89 minutes on average).

Qualitative data speaks to four main reasons why participants performed significantly better with Assuage (*RQ1*). First, Assuage afforded participants more choices to prune the search space during synthesis, while participants in the control interface had very few options other than constraining instruction usage. P18 complained, *"it felt like the number and precision of constraints required to get an answer in a reasonable time were barely sufficient."* We also noticed that participants were more inclined to wait for some active synthesis progress and analyze both the specification and candidates to add the corresponding constraints. Assuage provides more information to help them analyze and apply interventions. Second, parallel synthesis allowed participants to more freely apply interventions. P9 wrote, *"the fact that you lost previous programs when adding new constraints meant it was difficult to decide whether to add more."* P2 explained, *"it's really nice when you have these different kinds of experiments that you can do in parallel and you're not forced to directly have the perfect constraints from the beginning."* Third, parallel synthesis created high fault tolerance. P1 mentioned, *"if you ever make a mistake, then it won't have a hard reboot for the whole tool."* P12 also complained about the control interface, *"if I want to change my mind about something, I would have to basically stop all the progress. That was very punishing and a huge burden."* Fourth, participants gained more engagement during synthesis and showed more trust in the synthesized result. P19 explained, *"the fact that there was more information there certainly kept me engaged in the tool."* P12 also said, *"I like the tree view and the concurrent synthesis. It felt like more stuff was happening. It was kind of reassuring."*

Figure 4 shows participants' responses to the questions in Table 3 (*RQ2*). With Assuage, participants perceived less mental demand, felt less time pressure, spent less effort, and gave themselves better performance ratings. Welch's t-test on the comparisons in Figure 4 shows that the mean differences of mental demand, performance and effort are statistically significant ($p = 0.045$, $p = 0.039$, $p = 0.008$). However, there is no significant improvement in participants' response to perception of hurry and frustration ($p = 0.062, p = 0.224$). The mean differences of helpfulness and usability are also statistically significant ($p = 0.016$, $p = 0.034$), which indicates the perception of usability preference is consistent across participants.

Adding up t-tests across all our experiments, we ran 8 statistical tests (including five NASA-TLX questions, two helpfulness and usability measurements, and completion time), giving us a Bonferroni-corrected threshold of 0.00625 for an initial α of 0.05. Note that those eight tests are dependent, which shows the Bonferroni correction is conservative in our study setting. After correction, the mean difference of completion time between two synthesizers

Q1. How mentally demanding was this task with this tool? (1—Very Low, 7—Very High)
Q2. How hurried or rushed were you during this task? (1—Very Low, 7—Very High)
Q3. How successful would you rate yourself in accomplishing this task? (1—Perfect, 7—Failure)
Q4. How hard did you have to work to accomplish your level of performance? (1—Very Low, 7—Very High)
Q5. How insecure, discouraged, irritated, stressed, and annoyed were you? (1—Very Low, 7—Very High)
Q6: How helpful was this tool for writing assembly programs? (1—Not Helpful, 7—Very Helpful)*
Q7: How likely to use this tool in the future of possible assembly programming? (1—Not Likely, 7—Very Likely)*

Table 3: Participants rated on a 7-point scale. Q1-Q5: rating the subjective workload under different aspects: mental demand, hurry, performance, effort, and frustration; Q6-Q7: rating the usefulness of the assigned synthesizer. Asterisks designate the statements that are reverse scored.

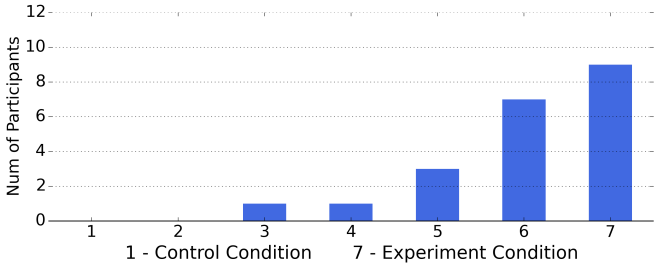

Figure 6: Which interface did you prefer to use?

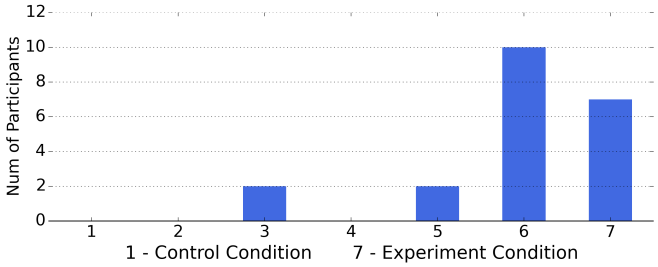

Figure 7: Which interface was more useful?

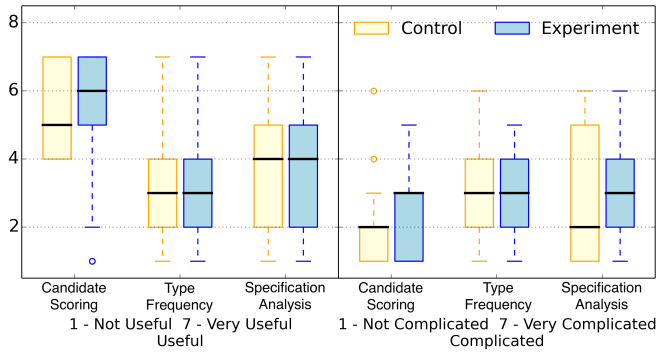

Figure 8: How useful vs. complicated was each synthesizer feedback information?

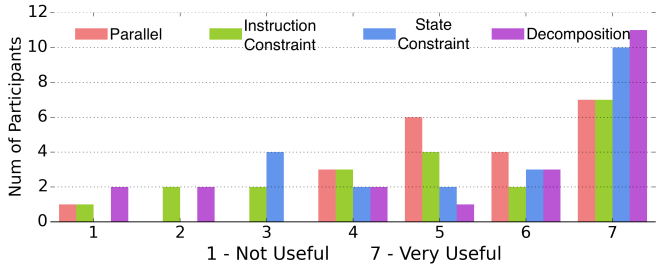

Figure 9: How useful was each intervention and parallel synthesis in Assuage?

is still statistically significant and the mean difference of effort participants spent is marginally significant. Yet there is no significant difference for other questions after correction.

7.2 User Preference

Figures 6 and 7 show the overall preference and usefulness ratings (*RQ3*). Sixteen of 21 participants preferred or strongly preferred synthesis with Assuage, and 19 of 21 participants thought Assuage more useful. We also categorize different kinds of users with assembly expertise and investigate their performances and preferences in the user study. Figure 5 shows that the strong preference using Assuage did not vary much across participants with different degrees of expertise.

In the survey for each task, we asked participants to rate the usefulness and subjective workload of each synthesis feature available to them during the task (*RQ4*). Figure 8 shows the comparison of each type of synthesizer feedback information that was present in both conditions. The candidate scoring was most preferred; participants' comments indicate that it provides a holistic view of the back-end synthesis. For example, P10 described that *"it provides a nice visualization of how the synthesis is doing."* Specification analysis was considered slightly more helpful than the type frequency, while their complexities were rated similarly. Unsurprisingly, participants with higher self-reported levels of expertise rated themselves higher in understanding what the candidates were doing and how close they were to satisfying the specification.

Figure 9 indicates the perceived usefulness of each intervention type as well as parallel synthesis in Assuage. The majority of participants (15/20) expressed a strong preference towards both location constraints and decomposition features. Participants hesitated to try decomposition at the beginning as P2 explained, *"the decomposition interface is more intimidating than the other interfaces since there is more going on; the other things are super easy and clear about what's happening."* As for parallel synthesis, all but one participant

found parallelism useful and thought that it provided more possibilities to explore. We will discuss more qualitative observations in Section 7.3.

7.3 Qualitative Observations: Obstacles and Lessons

Having Parallel instances requires less activation energy to add interventions and encourages simultaneous exploration. P15 expressed the common sentiment well: *"I was able to put up new possibilities, but I didn't have to commit all or nothing to them. The other ones would still work and [Assuage] allows me to explore the space of things that I could do."* P2 also explained, *"it's really nice when you have these different kinds of experiments that you can do in parallel and you're not forced to directly have the perfect constraints from the beginning." "The fact that you don't have to restart but just keep intervening in the meantime both saves time and also saves mental energy,"* P7 exclaimed. Though we were wary of revealing the underlying parallel synthesis process and exposing users to the associated additional complexity, users loved leveraging parallel synthesis and were able to wield it more effectively to reach their goals compared to the non-parallel (control) condition. Twelve out of 21 participants had to restart the synthesis process in the control condition due to incorrect interventions; some of them even restarted the process four times in a span of 20 minutes. Assuage reduced the penalty of adding incorrect interventions, so users were less hesitant to add them. As a result, users explored multiple ideas simultaneously. Several participants expressed that the thought process was different with parallel synthesis: P8 said, *"[Assuage] was much more open to experimenting, [control interface] was very streamlined and it had a single thought process."* P6 also explained, *"In [Assuage], you're able to quickly try a lot of different things at the same time, while I think [control interface] forces you to slow down a lot and think carefully about what constraints you would [want to add], so you want to be sure that you're not adding constraints that could be wrong or not optimal, because you have to restart all over again."*

Involvement increases trust. Participants trust the synthesizer's result when they are involved more in the synthesis process. P4 said, *"I'm involved enough in the process that I trust the results. I can see the intermediate steps, I can look at the programs, and I can help it along, so I feel like given seeing so much of what's going on makes me trust the tool more."* Nine participants reported appreciation for being involved in the synthesis process. P12 said *"I like the tree view and the concurrent synthesis. It felt like more stuff was happening. It was kind of reassuring."* In contrast, for some elements that were "too automatic," users expressed concern. P7 expressed his concerns specifically about decomposition: *"I also wonder how much we can trust the automatic decomposition like if there could be any errors because that's automatic, so I'm less likely to fully rely on it."*. On the other hand, participants also did not like too much involvement. P13 complained, *"[control interface] is pretty easy to use, but I don't think it is really helpful to fully synthesize instructions automatically, it involves a lot of user interactions."*

Idle time increases doubts in interventions (RQ5). P12, P13, P16, and P21 restarted the synthesis process in the control condition after few minutes of idling, even though the set of constraints provided by them were correct! The think-aloud strategy revealed that they started questioning their constraints while sitting idle as the synthesis process progressed, eventually resulting in their decision to restart synthesis with different constraints. P16 explained, *"The score started to go down when I added [Exclude Coproc] constraint, so I may have to restart the process"* after idling for a few minutes, even though the constraint she added and the intuition behind it were right. This did not happen when participants were using Assuage.

Participants concern about solution quality (RQ5). Expert users frequently expressed concern about solution quality. Three participants (2 experts, 1 intermediate user) reported that there is a trade-off between finding a solution quickly and synthesizing optimal solutions. For example, using *decomposition* in Assuage, they were able to produce correct solutions efficiently, but the solutions were different than what they would have written manually. P11 explained, *"this actually generated a worse solution using jump. The optimal solution will not use jump, since ARM has conditional instructions."* Experts frequently optimize for some metric: e.g., fewest lines of code, most performant, or most easily understood by a person, while the synthesizer does not.

On the other side, by adding *instruction constraints*, P18 observed that at one point, they were *"now just guiding the synthesizer to generate programs that I would write myself given this specification; this prevented the synthesizer from generating interesting solution."* Some participants were interested in obtaining eccentric solutions, like obscure bit manipulation programs for a given specification. By adding constraints they sometimes prevented the synthesizer from generating such solutions.

Expectation violations were not rare (RQ5). There were a few instances of expectation violation when the participant expected the synthesizer to "understand" the specification better. P5 was confused when they remarked, *"why is the synthesizer exploring conditional instructions [when] there [are] no conditional elements in the specification?"* Some participants also expected additional "intelligence" from the synthesizer such as knowing when to use conditional instructions, when to move data, and what registers to use.

More information increases learning curve (RQ5). Several participants explained that it was hard to inspect all the information provided within a limited time. Three participants (P4, P11, P14) explicitly mentioned they would perform better after getting more familiar with the interface (both control and Assuage), showing that there was a steep learning curve for assembly synthesis.

Participants expressed the need for more granular constraints (RQ5). Two experts (P5, P15), as well as two intermediate users (P2, P11) and four novices (P6, P7, P10, P12), expressed a preference for being able to express constraints at a finer granularity, such as directly assigning some particular instructions or registers in the target sequence instead of only manipulating with high-level type abstractions. One possible explanation is that with synthesizer feedback, participants gained more insight about the synthesis process, which allowed them to provide more detailed guidance to help the synthesizer make progress. However, we are worried that enabling

finer-grained interventions could increase the complexity of the interface, thereby imposing more subjective workload on users.

8 DISCUSSION AND FUTURE WORK

While significant effort has been devoted to optimizing synthesis algorithms, there are still limits to what can be accomplished without human intervention. This within-subjects study of Assuage, a novel interactive synthesis tool, demonstrates how much more the human and the synthesizer were able to accomplish together with sufficient interface design.

We are well aware that, compared to traditional interactive synthesis, Assuage's parallel support reveals more code-rich information and synthesis process details to users, increasing complexity and potentially burdening users mentally. Surprisingly, when using parallel synthesis, participants felt less mental demand overall. One explanation is that although Assuage showed more information and had a more complex UI, users could gain more information about the synthesis process, making it easier for them to give effective feedback. This also supports the fact that participants requested support for more specific constraints. Moreover, participants typically guided the synthesis with a trial-and-error procedure, which is more demanding when the cost of errors is high. Since the parent instance with no intervention is always running, participants can easily roll back and apply new interventions directly to the parent instance, letting the current instance fail if the interventions are counterproductive. Having multiple parallel instances at the same time also prevents long stretches of idle time, which introduced uncertainty in users' interventions, as mentioned in Section 7.3.

Expectation violations were not rare in our user study; most of them were confused about the capabilities and the scope of the synthesizer. Grimes et al. [14] show that expectation violations lead to mistrust and distrust in the system. In the future, we plan to investigate ways to establish appropriate trust by improving the communication of the synthesizer's capabilities to users.

Myers and McDaniel [36] point out that one major obstacle while using PBE/PBD systems was the lack of confidence and trust in synthesized programs, since users were not able to inspect the synthesis process or understand the synthesized programs. When using Assuage, especially with parallel instances, users are more involved in the synthesis process, which increased their trust in the results. However, users also did not prefer too much involvement. Balancing between the complexity and the usability of the interface remains an important avenue of investigation, to identify "sweet spots" where (1) users are involved enough to trust the outcome but not so involved that they feel the system is not helping them at all, and (2) without being overwhelmed by too much information, users have enough information to take informed actions with confidence, or at least without fear of the consequences of messing up.

While experts who are already capable of producing correct, highly-optimized assembly code may not be inclined to adopt a synthesizer, synthesis is beneficial to less expert programmers or those unfamiliar with a required assembly language. The combination of synthesis and Assuage can augment or replace the assembly code composition process, so developers can focus on other aspects of their system. It is a happy side effect if a user's interaction with Assuage teaches them something about writing assembly programs, much like Googling for Stack Overflow answers to a programming question sometimes teaches us new knowledge but usually just helps us get a job done.

Overall, Assuage is an instantiation of mixed-initiative interactive synthesis, a promising class of interactive synthesizers that may generalize to additional complex, unrestricted, real-world programming challenges beyond assembly programming.

9 CONCLUSION

This paper presents a novel interactive assembly synthesis tool, Assuage, that communicates the progress explored by the synthesizer to users, so users can operate on generated candidate programs, intervene from different aspects, explore various possibilities in parallel, and provide more valuable guidance to the synthesizer. It allows the user and the synthesizer to work collaboratively towards generating a program that satisfies a complete specification of what the program should do. We evaluated its usefulness and usability in a within-subjects lab study with twenty-one participants and showed that, compared to prior state of the art interactive synthesis affordances, the availability of multiple types of interventions and parallel synthesis processes enabled more users, regardless of their level of expertise, to complete realistic assembly programming tasks.

ACKNOWLEDGMENTS

Thank you to David A. Holland, Eric Lu, and Ming Kawaguchi for their work on assembly language synthesis toolchain development and their extraordinarily useful feedback and advice on this work. Thank you to Tianyi Zhang for his useful suggestions on the interface design of Assuage. We also thank the anonymous reviewers for their useful feedback, which greatly improved this paper. This material is based upon work supported by the National Science Foundation under Grant No. 2123965. We acknowledge the support of Intel and Microsoft. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Nous remercions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de son soutien.

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo Martin, Mukund Raghothaman, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis, In 2013 Formal Methods in Computer-Aided Design. *2013 Formal Methods in Computer-Aided Design, FMCAD 2013*, 1–17. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Shaon Barman, Rastislav Bodik, Satish Chandra, Emina Torlak, Arka Bhattacharya, and David Culler. 2015. Toward Tool Support for Interactive Synthesis. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA) *(Onward! 2015)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/2814228.2814235

[3] David Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson. 2004. *Synthesis of Programs in Computational Logic*. Springer Berlin Heidelberg, Berlin, Heidelberg, 30–65. https://doi.org/10.1007/978-3-540-25951-0_2

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[5] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes under Symbolic Evaluation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 149 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276519

[6] J. Richard Buchi and Lawrence H. Landweber. 1969. Solving Sequential Conditions by Finite-State Strategies. *Trans. Amer. Math. Soc.* 138 (1969), 295–311. http://www.jstor.org/stable/1994916

[7] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 963–975. https://doi.org/10.1145/3242587.3242661

[8] Allen Cypher. 1995. EAGER: PROGRAMMING REPETITIVE TASKS BY EXAMPLE. In *Readings in Human–Computer Interaction*, RONALD M. BAECKER, JONATHAN GRUDIN, WILLIAM A.S. BUXTON, and SAUL GREENBERG (Eds.). Morgan Kaufmann, 804–810. https://doi.org/10.1016/B978-0-08-051574-8.50083-2

[9] Cristina David and Daniel Kroening. 2017. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20150403.

[10] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376442

[11] R. A. Fisher. 1922. On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P. *Journal of the Royal Statistical Society* 85, 1 (1922), 87–94. http://www.jstor.org/stable/2340521

[12] Pierre Flener and Derek Partridge. 2001. Inductive Programming. *Automated Software Engg.* 8, 2 (April 2001), 131–137. https://doi.org/10.1023/A:1008797606116

[13] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 653–663. https://doi.org/10.1145/2568225.2568250

[14] G Mark Grimes, Ryan M Schuetzler, and Justin Scott Giboney. 2021. Mental models and expectation violations in conversational AI interactions. *Decision Support Systems* (2021), 113515.

[15] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[16] S. Gulwani. 2012. Synthesis from Examples: Interaction Models and Algorithms. In *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 8–14. https://doi.org/10.1109/SYNASC.2012.69

[17] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. https://doi.org/10.1145/2240236.2240260

[18] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. 2015. Inductive Programming Meets the Real World. *Commun. ACM* 58, 11 (Oct. 2015), 90–99. https://doi.org/10.1145/2736282

[19] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose). Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

[20] Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages.

[21] Tihomir Gvero and Viktor Kuncak. 2015. Interactive Synthesis Using Free-Form Queries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 689–692.

[22] Sandra G. Hart and Lowell E. Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Human Mental Workload*, Peter A. Hancock and Najmedin Meshkati (Eds.). Advances in Psychology, Vol. 52. North-Holland, 139–183. https://doi.org/10.1016/S0166-4115(08)62386-9

[23] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[24] David A. Holland. 2020. *Toward Automatic Operating System Ports via Code Generation and Synthesis*. Ph.D. Dissertation. Cambridge, MA, USA. Advisor(s) Margo I. Seltzer and Stephen Chong.

[25] David A. Holland, Jingmei Hu, Ming Kawaguchi, Eric Lu, Stephen Chong, and Margo I. Seltzer. 2020. Aquarium: Cassiopea and Alewife Languages. arXiv:1908.00093 [cs.PL]

[26] David A. Holland, Ada T. Lim, and Margo I. Seltzer. 2002. A New Instructional Operating System. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 111–115. https://doi.org/10.1145/563340.563383

[27] Eric Horvitz. 1999. Principles of Mixed-Initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Pittsburgh, Pennsylvania, USA) *(CHI '99)*. Association for Computing Machinery, New York, NY, USA, 159–166. https://doi.org/10.1145/302979.303030

[28] Jingmei Hu, Eric Lu, David A. Holland, Ming Kawaguchi, Stephen Chong, and Margo I. Seltzer. 2019. Trials and Tribulations in Synthesizing Operating Systems. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (Huntsville, ON, Canada) *(PLOS'19)*. Association for Computing Machinery, New York, NY, USA, 67–73. https://doi.org/10.1145/3365137.3365401

[29] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 377–394.

[30] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[31] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

[32] Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. arXiv:1703.03539 [cs.PL]

[33] Tak Yeon Lee, Casey Dugan, and Benjamin B. Bederson. 2017. Towards Understanding Human Mistakes of Programming by Example: An Online User Study. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces* (Limassol, Cyprus) *(IUI '17)*. Association for Computing Machinery, New York, NY, USA, 257–261. https://doi.org/10.1145/3025171.3025203

[34] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (Jan. 1980), 90–121. https://doi.org/10.1145/357084.357090

[35] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Alex Polozov, Rishabh Singh, Ben Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *28th ACM User Interface Software and Technology Symposium (UIST 2015)* (28th acm user interface software and technology symposium (uist 2015) ed.). ACM – Association for Computing Machinery.

[36] Brad Myers and Richard McDaniel. 2001. Demonstrational interfaces: sometimes you need a little intelligence, sometimes you need a lot. (01 2001), 45–60.

[37] Brad A. Myers. 1990. Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints. *ACM Trans. Program. Lang. Syst.* 12, 2 (April 1990), 143–177. https://doi.org/10.1145/78942.78943

[38] Brad A. Myers. 1998. Scripting Graphical Applications by Demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Los Angeles, California, USA) *(CHI '98)*. ACM Press/Addison-Wesley Publishing Co., USA, 534–541. https://doi.org/10.1145/274644.274716

[39] Brad A. Myers and Richard McDaniel. 2001. *Demonstrational Interfaces: Sometimes You Need a Little Intelligence, Sometimes You Need a Lot.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 45–60.

[40] D. Partridge. 1997. The case for inductive programming. *Computer* 30, 1 (1997), 36–41. https://doi.org/10.1109/2.562924

[41] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1114–1124. https://doi.org/10.1145/3180155.3180189

[42] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[43] C. Rich and R. C. Waters. 1988. The Programmer's Apprentice: a research overview. *Computer* 21, 11 (1988), 10–25. https://doi.org/10.1109/2.86782

[44] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 661–676.

[45] Mark Santolucito, Drew Goldman, Allyson Weseley, and Ruzica Piskac. 2019. Programming by Example: Efficient, but Not "Helpful". In *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018) (OpenAccess Series in Informatics (OASIcs), Vol. 67)*, Titus Barik, Joshua Sunshine, and Sarah Chasins (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:10. https://doi.org/10.4230/OASIcs.PLATEAU.2018.3

[46] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ,

USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 136–148. https://doi.org/10.1145/1375581.1375599

[47] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 404–415. https://doi.org/10.1145/1168919.1168907

[48] Venkatesh Srinivasan and Thomas Reps. 2015. Synthesis of Machine Code from Semantics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 596–607. https://doi.org/10.1145/2737924.2737960

[49] Venkatesh Srinivasan, Tushar Sharma, and Thomas Reps. 2016. Speeding up Machine-Code Synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 165–180. https://doi.org/10.1145/2983990.2984006

[50] Venkatesh Srinivasan, Ara Vartanian, and Thomas Reps. 2017. Model-Assisted Machine-Code Synthesis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 61 (Oct. 2017). https://doi.org/10.1145/3133885

[51] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1631–1634. https://doi.org/10.1145/3035918.3058738

[52] Henry S. Warren. 2012. *Hacker's Delight* (2nd ed.). Addison-Wesley Professional.

[53] B. L. Welch. 1938. The Significance of the Difference Between Two Means when the Population Variances are Unequal. *Biometrika* 29, 3/4 (1938), 350–362. http://www.jstor.org/stable/2332010

[54] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery.

[55] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900