ASAC: A Benchmark for Algorithm Synthesis

Zhao Zhang
Key Lab of HCST (PKU), MOE; SCS, Peking University
Beijing, China
zhangzhao2019@pku.edu.cn

Yican Sun
Key Lab of HCST (PKU), MOE; SCS, Peking University
Beijing, China
sycpku@pku.edu.cn

Ruyi Ji
Key Lab of HCST (PKU), MOE; SCS, Peking University
Beijing, China
jiruiy190387714@pku.edu.cn

Siyuan Li
SCS, Peking University
Beijing, China
1700017763@pku.edu.cn

Xuanyu Peng
SCS, Peking University
Beijing, China
dofypxy@pku.edu.cn

Zhechong Huang
SCS, Peking University
Beijing, China
willhuang@stu.pku.edu.cn

Sizhe Li
SCS, Peking University
Beijing, China
lisizhe@pku.edu.cn

Tianran Zhu
SCS, Peking University
Beijing, China
ztr@pku.edu.cn

Yingfei Xiong
Key Lab of HCST (PKU), MOE; SCS, Peking University
Beijing, China
xiongyf@pku.edu.cn

ABSTRACT
In this paper, we present the first benchmark for algorithm synthesis from formal specification: ASAC. ASAC consists of 136 tasks covering a wide range of algorithmic paradigms and various difficulty levels. Each task includes a formal specification and an efficiency requirement, and the program synthesizer is expected to produce a program that satisfies the formal specification and meets the efficiency requirement. Our evaluation of two state-of-the-art (SOTA) approaches in ASAC shows that ASAC exposes new challenges for future research on program synthesis.

ASAC is available at https://auqwqua.github.io/ASACBenchmark, and the demo video is available at https://youtu.be/JXVleCdBh8U.

CCS CONCEPTS
• Software and its engineering:

KEYWORDS
Program synthesis, benchmark

1 INTRODUCTION
Program synthesis is the task of automatically finding a program that satisfies the user-provided specification. The specification could be in the natural language form [1, 2], or could also be in formal logic [7]. In this paper, we focus on the latter because it allows us to guarantee the correctness of the synthesized program.

However, the existing program synthesis benchmarks based on formal specifications involve only small programs and writing such programs may not be noticeably more difficult than writing the formal specifications. Recently, some researchers have focused on synthesizing algorithms from formal specifications [9, 15, 16, 21]. Synthesizing algorithms do not have this problem because algorithms are generally difficult to contrive, and writing an algorithm is noticeably more difficult than writing a formal specification for the algorithm. However, so far all existing approaches were evaluated in specialized problem sets that were covered by the respective approaches, and the input formats of the problem sets are different. There still lacks a unified benchmark.

To motivate research on algorithm synthesis, we present the first unified benchmark for algorithm synthesis, ASAC. The build of the benchmark is as follows. First, we collected 136 problems from the National Olympiad in Informatics in Provinces (NOIP), one of the world’s most participated national programming contests. Second, we manually constructed the formal specification of the problems above in MiniZinc [18], a widely used constraint modeling language (See Section 4.2 for details). Third, we built the test suite and set the time and memory limits for each task. The total construction of the benchmark cost 410 man-hours.

For each task, a synthesizer is expected to synthesize a program that conforms to the formal specification. Similarly to an algorithm competition, the user can get the task score if and only if the synthesized programs passes all the tests within the time and memory limit. Besides, we preserved the natural language descriptions, making it possible to evaluate and compare tools that generate code from natural language.
As mentioned before, though approaches for algorithm synthesis have been proposed, the benchmarks used in their evaluation are collected in an adhoc way. On the one hand, only those algorithms in ASAC provide both formal specifications and natural language descriptions, allowing tools with different input formats to be used. On the one hand, only those algorithms have been proposed, the benchmarks used in their evaluation are collected in an adhoc way. On the one hand, only those algorithms can not exceed a given integer. To ensure that all the souvenirs are distributed in as short a time as possible, Lele wants to keep the number of groups to a minimum. Your task is to write a program to find the minimum number of groups among all the grouping schemes and outputs the minimum number of groups.

### Problem description
The school student union let Lele be responsible for the New Year party souvenir distribution work. To make the value of the souvenirs obtained by the students attending the party relatively balanced, he should group the souvenirs according to the price, but each group can only include two souvenirs at most, and the sum of the prices of each group of souvenirs can not exceed a given integer. To ensure that all the souvenirs are distributed in as short a time as possible, Lele wants to keep the number of groups to a minimum.

Your task is to write a program to find the minimum number of groups among all the grouping schemes and outputs the minimum number of groups.

### Input
The input file contains n+2 lines,

1. n: An integer, which represents the total number of souvenirs purchased.
2. w: An integer, which represents the total number of souvenirs purchased.
3. Price: A list of n integers, which represents the prices of n souvenirs.
4. Group: A list of n integers, which represents the grouping of n souvenirs.

### Output
The output file has only one integer which is the minimum number of groups.

### Figure 1: English Description
We evaluated two approaches to ASAC. First, we evaluated SynMem [21], a SOTA algorithm synthesizer focusing on dynamic programming (DP) programs. SynMem solved 21.1% dynamic programming tasks in ASAC. Second, we evaluated ChatGPT [2], a SOTA neural model that supports generating programs from natural language descriptions or formal specifications. ChatGPT solves 8.8% of the tasks under both settings. These results indicate that our benchmarks present a new challenge for future research.

## 2 RELATED WORK

### 2.1 General Program Synthesis Benchmarks
SyGuS (Syntax-Guided Synthesis) benchmarks are the public benchmarks used in SyGuS Competition [6] for program synthesis. Given a logic specification and grammar rules, a synthesizer is expected to provide a program conforming to the grammar rules and satisfying the specification. Compared with ASAC, there are the following differences. (i) The programs to be synthesized in SyGuS are small expressions, which are significantly different from complete algorithms in ASAC, which usually require tens of or hundreds of lines. (ii) SyGuS does not have efficiency requirements and its problems are also simple programming tasks that do not require the applications of algorithm paradigms.

### 2.2 Benchmarks for Synthesizing Algorithms from Natural Language Descriptions
There are several benchmarks based on competitive programming problems [5, 8, 11, 14, 17]. However, these benchmarks are based on natural language descriptions but not formal specifications. The inherent ambiguity of natural language makes it impossible to prove the correctness of the generated programs. Though some of the benchmarks [5, 17] are augmented with tests, tests specify only partial behavior and cannot be used as a full specification. ASAC provides both formal specifications and natural language descriptions, allowing tools with different input formats to be used and compared.

### 2.3 Benchmarks for Synthesizing Algorithms from Logic Specifications
As mentioned before, though approaches for algorithm synthesis have been proposed, the benchmarks used in their evaluation are collected in an adhoc way. On the one hand, only those algorithms in ASAC, which usually require tens of or hundreds of lines. (ii) SyGuS does not have efficiency requirements and its problems are simple programming tasks that do not require the applications of algorithm paradigms.

### 2.4 Benchmarks for Constraint Solving
There exist multiple constraint solving benchmarks, such as SMTCOMP [22] and MiniZinc challenges [20]. Program synthesis benchmarks aim to synthesize programs to solve a class of problems, while constraint solving benchmarks aim to solve one problem instance at a time. Nevertheless, constraint solving and program synthesis both use logic specifications to describe the problems and the logic specification language can be shared. The MiniZinc language we use is originally designed for constraint solving, and is recently used in the algorithm synthesis domain.

## 3 AN EXAMPLE
Let us begin with a sample task in our benchmark. This task is relatively simple and is selected to ease understanding.

The English description of the sample task is shown in Figure 1. It consists of three separate parts: the natural language description of the problem, the input format, and the output format. To implement an efficient program for this problem, a greedy algorithm is required.

The formal specification of this task is shown in Listing 1. This task is formalized as an optimization problem, and the correspondence between the specification and the natural language description is listed below.

- Lines 1–3 in our formal specification specify the input of the problem, which consists of parameter declarations without the var keyword. We use the same identifier as those in the input format in our natural language descriptions. A concrete input is provided by a MiniZinc data file in the standard dzn format, and the synthesized program is expected to read the dzn file.
- Lines 5–7 formalize the concepts. Variable “setIndex[i]” represents the index of the group that includes the i-th souvenir, variable “weight[i]” represents the weighted sum for the i-th group, and “num[i]” represents the number of souvenirs in the i-th group.
- Lines 8-11 formalize the constraints in the problem. Every constraint has its corresponding sentence in the natural description.
- Lines 13-14 formalize the objective function. The function returns the number of non-empty groups, and our target is to minimize the returned number.

### Listing 1: Specification
that fall into the target domain of the specific approach are covered. On the other hand, different approaches use different input formats (imperative [12, 13, 15] or functional programs [3] satisfying certain requirements, or constrained logic specification [16, 21]), making it difficult to cross-compare these approaches or combine them for synthesizing a larger class of algorithms. Compared with them, problems in ASCA are selected broadly from programming contests and are specified uniformly in a logic specification language.
We selected the constraint modeling language MiniZinc to construct well-simulate problems in real software development. Problems whose complexity is higher than expected would hardly meet the specifications are precise so that it is possible to verify, manually or automatically, the correctness of the synthesized programs. Second, the specifications are declarative. No algorithm has to be taken into account when modeling the problem. Third, many real-world problems can be naturally represented by constraint forms in a declarative, high-level, and solver-independent way [4, 10].

4.3 Formalization

Based on the above choices, we formalize the problems as follows:

- For each problem, we manually construct its formal specification based on the natural language description.
- Since each problem in algorithm contests has a test suite, we convert the original test suite into the format of a MiniZinc data file. We keep both versions of the test suite in our dataset, where the original one is used when synthesizing from the natural language description, and the converted one is used when synthesizing from the formal specification.

Seven authors conducted the formalization work, and the total time used was 410 man-hours. All the authors (i) are experienced MiniZinc users and (ii) have good backgrounds in algorithms. Since the same problem can be formalized in different ways, we define the following rules to guide the formalization process:

- New concepts can be defined based on their standard definitions in mathematical textbooks. For example, “prime number” is a concept that does not exist in MiniZinc. We can define it as “a natural number greater than 1 that is not a product of two smaller natural numbers”.
- When a standard definition cannot be represented in MiniZinc, an equivalent definition can be used. For example, one way to define the prime number is “a natural number greater than 1 that is not a product of any two natural numbers”. As the integers in MiniZinc are always defined in a range, we cannot represent “a product of any two natural numbers”, and thus we use the equivalent definition in the previous paragraph.
- No manual optimization can be used. Yet another way to define a prime number is to use the Ehrlich sieve method, which could translate to an efficient decision procedure. We avoid such manual optimization and stick to natural and simple definitions.

Furthermore, each time a new concept is introduced, its definition is discussed among the authors and is shared in the rest of the formalization process.

We use a peer-review procedure to ensure the correctness of the formalization. To reduce the workload, we first apply automated inspection and then apply manual inspection.

- Automated inspection. For each problem r and each test input i of r, we invoke Gecode [19], a widely-used constraint solver, on our formalization of r and i to obtain the output, and then check if the output is the same as the test oracle.
- Manual inspection. If Gecode does not produce an answer in 10 minutes for any test, we manually check if the specification of the problem is correct. In total, 12 out of 136 specifications were modified during this process.

4.4 Translation

To ease the use of our benchmark by a wide range of users, and to support more types of works about code generation from natural language, we recruit professional translators to translate the original Chinese natural language descriptions into English. One author

**THE CONSTRUCTION OF ASAC**

4.1 Task Selection

The problems in our benchmark are collected from competitive programming contests NOIP. We choose problems from competitive programming contests because (i) it is difficult to obtain tasks of designing algorithms in real software development because most of them are undocumented. (ii) Problems in competitive programming are designed based on algorithmic paradigms widely used in practice, such as D&C and dynamic programming, and thus can well simulate problems in real software development. (iii) Problems in competitive programming are well recognized as representative by the industry and are commonly used in job interviews for programmers. (iv) These problems have already been used in existing studies for evaluating program synthesizers based on LLMs.

Among these contests, we select problems from NOIP, which is an annual competition in China for junior and senior high school students. It has the following advantages. First, it is one of the largest algorithm competitions for students (e.g. 24781 participants in 2018.) Second, the competition is divided into junior and senior groups, covering a wide range of difficulty levels. Third, competition problems include various algorithms, such as dynamic programming and greedy algorithms.

We collect 136 problems from the 10th to the 25th NOIP, involving a wide range of algorithmic paradigms and difficulty levels.

4.2 Formalization Language Selection

We selected the constraint modeling language MiniZinc to construct the specification. The MiniZinc language is originally designed to describe constraint satisfaction/optimization problems. It supports separating inputs, where different inputs represent different constraint modeling problems that need to be solved.

We can also comprehend the MiniZinc language from another perspective: when the input is unknown, the entire file describes a function from an input to an output that meets constraints. This essentially describes a programming problem. Therefore, we can use the MiniZinc language to describe algorithm synthesis problems.

The use of the MiniZinc language has multiple benefits. First, the specifications are precise so that it is possible to verify, manually or automatically, the correctness of the synthesized programs. Second, the specifications are declarative. No algorithm has to be taken into account when modeling the problem. Third, many real-world problems can be naturally represented by constraint forms in a declarative, high-level, and solver-independent way [4, 10].

7 authors conducted the formalization work, and the total time used was 410 man-hours. All the authors (i) are experienced MiniZinc users and (ii) have good backgrounds in algorithms. Since the same problem can be formalized in different ways, we define the following rules to guide the formalization process:

- New concepts can be defined based on their standard definitions in mathematical textbooks. For example, “prime number” is a concept that does not exist in MiniZinc. We can define it as “a natural number greater than 1 that is not a product of two smaller natural numbers”.
- When a standard definition cannot be represented in MiniZinc, an equivalent definition can be used. For example, one way to define the prime number is “a natural number greater than 1 that is not a product of any two natural numbers”. As the integers in MiniZinc are always defined in a range, we cannot represent “a product of any two natural numbers”, and thus we use the equivalent definition in the previous paragraph.
- No manual optimization can be used. Yet another way to define a prime number is to use the Ehrlich sieve method, which could translate to an efficient decision procedure. We avoid such manual optimization and stick to natural and simple definitions.

Furthermore, each time a new concept is introduced, its definition is discussed among the authors and is shared in the rest of the formalization process.

We use a peer-review procedure to ensure the correctness of the formalization. To reduce the workload, we first apply automated inspection and then apply manual inspection.

- Automated inspection. For each problem r and each test input i of r, we invoke Gecode [19], a widely-used constraint solver, on our formalization of r and i to obtain the output, and then check if the output is the same as the test oracle.
- Manual inspection. If Gecode does not produce an answer in 10 minutes for any test, we manually check if the specification of the problem is correct. In total, 12 out of 136 specifications were modified during this process.

4.4 Translation

To ease the use of our benchmark by a wide range of users, and to support more types of works about code generation from natural language, we recruit professional translators to translate the original Chinese natural language descriptions into English. One author

**THE CONSTRUCTION OF ASAC**

4.1 Task Selection

The problems in our benchmark are collected from competitive programming contests NOIP. We choose problems from competitive programming contests because (i) it is difficult to obtain tasks of designing algorithms in real software development because most of them are undocumented. (ii) Problems in competitive programming are designed based on algorithmic paradigms widely used in practice, such as D&C and dynamic programming, and thus can well simulate problems in real software development. (iii) Problems in competitive programming are well recognized as representative by the industry and are commonly used in job interviews for programmers. (iv) These problems have already been used in existing studies for evaluating program synthesizers based on LLMs.

Among these contests, we select problems from NOIP, which is an annual competition in China for junior and senior high school students. It has the following advantages. First, it is one of the largest algorithm competitions for students (e.g. 24781 participants in 2018.) Second, the competition is divided into junior and senior groups, covering a wide range of difficulty levels. Third, competition problems include various algorithms, such as dynamic programming and greedy algorithms.

We collect 136 problems from the 10th to the 25th NOIP, involving a wide range of algorithmic paradigms and difficulty levels.

4.2 Formalization Language Selection

We selected the constraint modeling language MiniZinc to construct the specification. The MiniZinc language is originally designed to describe constraint satisfaction/optimization problems. It supports separating inputs, where different inputs represent different constraint modeling problems that need to be solved.

We can also comprehend the MiniZinc language from another perspective: when the input is unknown, the entire file describes a function from an input to an output that meets constraints. This essentially describes a programming problem. Therefore, we can use the MiniZinc language to describe algorithm synthesis problems.

The use of the MiniZinc language has multiple benefits. First, the specifications are precise so that it is possible to verify, manually or automatically, the correctness of the synthesized programs. Second, the specifications are declarative. No algorithm has to be taken into account when modeling the problem. Third, many real-world problems can be naturally represented by constraint forms in a declarative, high-level, and solver-independent way [4, 10].

4.3 Formalization

Based on the above choices, we formalize the problems as follows:

- For each problem, we manually construct its formal specification based on the natural language description.
- Since each problem in algorithm contests has a test suite, we convert the original test suite into the format of a MiniZinc data file. We keep both versions of the test suite in our dataset, where the original one is used when synthesizing from the natural language description, and the converted one is used when synthesizing from the formal specification.

Seven authors conducted the formalization work, and the total time used was 410 man-hours. All the authors (i) are experienced MiniZinc users and (ii) have good backgrounds in algorithms. Since the same problem can be formalized in different ways, we define the following rules to guide the formalization process:

- New concepts can be defined based on their standard definitions in mathematical textbooks. For example, “prime number” is a concept that does not exist in MiniZinc. We can define it as “a natural number greater than 1 that is not a product of two smaller natural numbers”.
- When a standard definition cannot be represented in MiniZinc, an equivalent definition can be used. For example, one way to define the prime number is “a natural number greater than 1 that is not a product of any two natural numbers”. As the integers in MiniZinc are always defined in a range, we cannot represent “a product of any two natural numbers”, and thus we use the equivalent definition in the previous paragraph.
- No manual optimization can be used. Yet another way to define a prime number is to use the Ehrlich sieve method, which could translate to an efficient decision procedure. We avoid such manual optimization and stick to natural and simple definitions.

Furthermore, each time a new concept is introduced, its definition is discussed among the authors and is shared in the rest of the formalization process.

We use a peer-review procedure to ensure the correctness of the formalization. To reduce the workload, we first apply automated inspection and then apply manual inspection.

- Automated inspection. For each problem r and each test input i of r, we invoke Gecode [19], a widely-used constraint solver, on our formalization of r and i to obtain the output, and then check if the output is the same as the test oracle.
- Manual inspection. If Gecode does not produce an answer in 10 minutes for any test, we manually check if the specification of the problem is correct. In total, 12 out of 136 specifications were modified during this process.

4.4 Translation

To ease the use of our benchmark by a wide range of users, and to support more types of works about code generation from natural language, we recruit professional translators to translate the original Chinese natural language descriptions into English. One author
checked all the translations to ensure that there was no language error caused by the lack of expertise.

5 BENCHMARK STATISTICS

In this section, we quantitatively measure the benchmark in three aspects. First, we measure the lengths of the problem descriptions and specifications. The problem description in our benchmark contains 325 words on average and 822 words at maximum. A formal problem specification in MiniZinc contains 90 tokens on average and 399 tokens at maximum. These numbers indicate that understanding the problem description / analyzing the problem specification is a non-trivial task.

Second, we measure the difficulty of the problems in the benchmark. To measure difficulty, we refer to a large online community for competitive programming, huogu.com.cn, where these problems are available and the users could submit their solutions to test whether their solutions are correct and efficient enough. The website reports the pass rate of each problem, which is defined as the ratio of the number of users whose solutions are passed to the number of users who submitted a solution. The pass rates of problems in our benchmark are summarized in Table 1a. As we can see, the pass rates follow a normal distribution, indicating the coverage of a wide range of difficulty levels.

Third, we measure the algorithm paradigms and other knowledge required to solve the problem. In huogu.com.cn, each problem is associated with a set of labels, such as “dynamic programming” or “mathematics”, indicating the algorithm paradigms and other knowledge required to solve the problem. These labels are maintained by the administrators of the website, and users can comment on whether the labels are accurate or not. Since the problems we collected are among the most popular problems, we believe the labels accurately reflect the required algorithm paradigms and knowledge. On ASAC, a total of 70 labels are used, indicating that ASAC covers a wide variety of algorithms. We have listed the labels used for more than 10 problems in our benchmark in Table 1b. As we can see, the classic general algorithms paradigms, such as dynamic programming and greedy methods, are widely used in ASAC.

(a) Pass Rate  (b) Algorithm Paradigm

<table>
<thead>
<tr>
<th>Pass Rate</th>
<th>Proportion</th>
<th>Label</th>
<th>Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%-10%</td>
<td>0.7%</td>
<td>DP</td>
<td>30</td>
</tr>
<tr>
<td>10%-20%</td>
<td>8.1%</td>
<td>Mathematics</td>
<td>23</td>
</tr>
<tr>
<td>20%-30%</td>
<td>27.9%</td>
<td>Greedy</td>
<td>21</td>
</tr>
<tr>
<td>30%-40%</td>
<td>30.1%</td>
<td>Enumeration</td>
<td>17</td>
</tr>
<tr>
<td>40%-50%</td>
<td>20.6%</td>
<td>Search</td>
<td>17</td>
</tr>
<tr>
<td>50%-60%</td>
<td>11.0%</td>
<td>Sorting</td>
<td>16</td>
</tr>
<tr>
<td>60%-70%</td>
<td>1.5%</td>
<td>Graph Theory</td>
<td>10</td>
</tr>
</tbody>
</table>

| DP = dynamic programming |

Table 1: Pass rate and algorithm paradigm of problems

6 EVALUATION

Among existing algorithm synthesis approaches, some [3, 12, 13, 15] require imperative or functional programs as input, and thus it is difficult to adapt them for ASAC, which uses logic specifications. A series of approaches for synthesizing dynamic programming algorithms [9, 16, 21] supports logic specification. SynMem [21] is the SOTA approach among them and is the only one with a publically available implementation. We run SynMem [21] on 19 DP tasks on the benchmark (30 tasks have DP labels, but 19 of them have DP as the dominant algorithm). Finally, SynMem solved 4 of them (passing all the tests), which accounts for 21.1% of the dynamic programming tasks and 2.9% of all the tasks.

Since ASAC also supports generating programs from natural language descriptions, we also evaluate ChatGPT [2] on our benchmark. ChatGPT is also able to take the MiniZinc specification as text input, so we evaluate both for each task. Please note that different from algorithm synthesizers, ChatGPT does not ensure the correctness of the generated program. We used the prompt "Please write a program to solve the following problem (described by specification in MiniZinc) below efficiently", followed by the specification or the English problem description. The sentence in parentheses is removed for the English description case.

For the generated programs, we evaluated (1) the percentage of tasks where the synthesized programs pass all the tests, and (2) the percentage of tests where the synthesized programs pass on all tasks (each task has 10 tests). When the second percentage is higher than the first, either some generated programs are not fully correct, or the efficiency does not satisfy the efficiency requirement.

The results are shown in Table 2. Each row corresponds to an algorithm label in ASAC. Each column corresponds to a tool being tested, where ChatGPT(N) means ChatGPT with natural language input and ChatGPT(S) means ChatGPT with MiniZinc input. All numbers are represented as pass rates in percentage. As we can see from the table, ChatGPT solves 8.8% for both input formats.

All experiments in this section are conducted on Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz with 16GB memory. The version of ChatGPT is 3.5 accessed on Jan 16, 2023.

7 CONCLUSION

In this paper, we present the first benchmark for algorithm synthesis. ASAC. ASAC consists of 136 problems, covering a wide range of algorithmic paradigms and various difficulty levels. For each task in ASAC, we constructed a formal specification based on the original natural language description and translated the original Chinese description into English. The construction of ASAC took in a total of 410 man-hours. We conducted experiments on ASAC. The results suggest that ASAC is challenging even for the SOTA tools and calls for new research.

ACKNOWLEDGMENT

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2022YFB4501902.
REFERENCES


