Olympiads in Informatics, 2015, Vol. 9, 57–73
DOI: http://dx.doi.org/10.15388/ioi.2015.06

Introducing tcframe: A Simple and Robust Test Cases Generation Framework

Ashar FUADI
Indonesia Computing Olympiad Alumni Association
e-mail: fushar@gmail.com

Abstract. Preparing test cases is a vital step in a programming contest. Creating all test cases manually by hand is hard and error-prone, so they should be generated by programs. There have been several attempts at creating a framework for test cases generation, that involve writing a generator program that generates the test cases, and a validator program that validates whether the produced test cases conform to the constraints. This paper proposes a simpler yet robust framework, called tcframe, for generating test cases especially for programming contest problems. The proposed approach involves writing a single self-validating C++ generator program as opposed to writing two separate programs. The framework API is designed in such a way that the resulting generator program is easy to read and modify. Using this framework, programming contest organizers can produce generator programs with a consistent and similar structure across all problems.

Keywords: test case, test cases generator, test cases framework.

1. Introduction

In the past few years, competitive programming contests have been on the rise. Worldwide, the number of online programming contests has been increasing. Specifically in Indonesia, there have been more universities and high schools that have started organizing programming contests. Students have many opportunities for participating in programming contests in each school year.

In a competitive programming contest, contestants are given several problems to solve under a predetermined time. A contestant is considered to solve a given problem if he/she can write a program which produces correct output for each of the problem setter's secret input data (Halim and Halim, 2010). Therefore, the correctness of the secret input data (test cases) itself is really important and the programming contest organizers should put a lot of efforts in creating the test cases.

It is common that the people who prepare programming contest problems, including the test cases, are the ones who have participated in some programming contests in the past. The problem is that having experience in participating in a lot of contests usually

does not automatically make us a good in preparing test cases. It does help a bit, as for example, a seasoned contestant will be aware that many kinds of tricky cases should be included in the test cases. However, we feel that it is not enough to be a good test cases preparer.

There are at least two aspects in learning how to produce good test cases. The first is learning how to systematically write a test cases generator program. The second is learning how to come up with a strong set of test cases that catch as many bugs as possible in contestants' submissions. In this paper, we will be focusing on the former aspect.

We realize that there are currently very few learning resources on how to systematically create test cases for programming contests. Many people simply don't know how or where to even start. In our experience, this lack of knowledge usually results in each problem setter inventing their own test cases generator program.

The programs are usually not standard: some have parameters that are easy to modify; some do not; some write the test cases directly to files, some to the standard output, and so on.

The above situation is actually dangerous: it becomes hard for a person to modify or even understand other people's generator programs. Imagine a possible situation where we have to do some last-minute changes to a problem's constraints (for example: adding very easy subtask for beginners), but the person that wrote the generator program is not available at the moment. Other people will have to spend a considerable amount of time to understand and modify the program to produce the desired changes.

To eliminate the mentioned problems, we developed a test cases generator framework called **tcframe**. We did not create tcframe at once. Instead, we begin by using an existing library called testlib. It is a library designed for generating test cases for ICPC-style programming contests. It has been used for many Russian programming contests (Mirzayanov, 2008). Using testlib, we have to write a generator program that outputs exactly one test case, and a validator program that checks whether a particular test case conforms to the constraints.

We found that it was not quite suitable for IOI-style contests, so we created a wrapper around it, called tokilib (Fuadi, 2014). The wrapper essentially makes it possible to generate multiple test cases at once, and to check the constraints based on the assigned subtasks of each test case. We have been using it for preparing Indonesian IOI training camps throughout 2014 and the scientific committee members have been very satisfied with it.

Finally, we invented a quite significantly better approach that involves only writing a single self-validating generator program. We decided to rewrite the framework from scratch and give it a new name, tcframe. We are currently in the process of completing the development of tcframe.

The rest of the paper is organized as follows. In Section 2 we propose a formalization of test cases organization. Section 3 talks about the existing framework attempts and how we incrementally refined them to finally create tcframe. Section 4 explains our implementation of the proposed approach. In the last section, Section 5, we conclude the paper and offer many possible future works for tcframe.

2. Test Cases Organization

In order to create a framework, we have to formalize several aspects related to the organization of test cases. In this section, we propose such formalization. This section contains the definitions and relations between the aspects.

2.1. *Test Case*

We define a *test case* as a pair of input values and the corresponding output values. The input must satisfy the *constraints* (defined later in this section). We will talk mostly about test case inputs. Therefore, for simplicity, if not explicitly clarified, a test case will mean a test case input.

Test cases should be hidden from contestants. However, there are some test cases that are given in the problem statements. They are given so that the contestants will not misunderstand the input/output formats. Such test cases are called *sample test cases*. In contrast, test cases that are not given in problem statements are called *official test cases*.

2.2. *Input Variables*

We define *input variables* as variables which compose test cases inputs. They are usually declared in the input format section of a problem statement. For each value that appears in a test case input, it must be either a constant or (part of) an input variable.

For example, consider the following problem:

Statement. Given an integer N, and an array consisting of N integers, compute the product of all integers, modulo 10007!"

Input format. The first line consists of a sentence “Dengklek has N integers", where N is an integer. The next line contains N space-separated integers A_1, A_2, ..., A_N."

Output format. Output a single line containing a single real value: the mean, printed with two digits after the decimal point.

In this problem, for this input:

```
Dengklek has 3 integers
10 20 30
```

- The strings “Dengklek”, “has”, and “integers” are constants. If we consider whitespaces, they are constants as well.
- The integer 3 is N.
- The integer 10 is A_1.
- The integer 20 is A_2.
- The integer 30 is A_3.

2.3. *Constraints*

We define a *constraint* as a boolean predicate which limits the possible values of the input variables in test case inputs. The value of the predicate must be completely determined by the values of input variables only. For example, $1 \leq N \leq 100$ is a valid constraint for the previous problem.

2.4. *Subtasks and Test Groups*

We define a *subtask* as a set of one or more constraints. Subtasks are numbered with consecutive integers starting from 1. A test case is said to satisfy a subtask if it satisfies all constraints in the subtask.

One purpose of introducing subtasks in a problem is to create a nice score distribution (van der Vegt, 2009). Since 2010, all IOI problems have used subtasks in their constraints. Besides having a good distribution, the scores are also predictable since the points allotted to each subtask are usually fixed.

Let's consider a typical problem, which has only N as an input variable, and has three subtasks as follow:

1. $1 \leq N \leq 100$
2. $1 \leq N \leq 1000$
3. $1 \leq N \leq 10000$

We must assign a set of test cases to each subtask. The most common way is to consider each subtask independently. This means that the test cases generation for each subtask is independent to the other subtasks. In this way, it is quite unlikely that two subtasks have a test case with exactly the same content.

However, this has a serious problem. If each subtask is considered independently, then theoretically it is possible that a submission solves subtask 2, but not subtask 1. This can happen if subtask 1 has a tricky case that is not present in subtask 2. However, this situation does not make sense: if a solution fails a test case having $1 \leq N \leq 100$, then logically we should not let it pass subtask 2, since that test case satisfy $1 \leq N \leq 1000$ as well.

Our proposed solution to the above problem is as follows. Instead of each subtask having a set of independent test cases, we *assign* each test case to a set of subtasks. A submission is then considered to solve a subtask if it solves all test cases assigned to it. For convenience, we define a *test group* as a set of test cases, numbered with consecutive integers starting from 1. If two test cases are assigned to the same set of subtasks, then they are in the same test group.

Formally, we propose the following assignment rules:

- Each test case is assigned to a set of subtasks.
- For each test group, all official test cases in it have the same set of subtasks.
- If a test case is assigned to a set of subtasks S, then:
  - The test case must satisfy all subtasks $s \in S$.
  - The test case must not satisfy any of the subtasks $s \notin S$.

In the proposed rules, instead of generating independent set of test cases for each subtask, we organize the test cases generation as follows.

- Test group 1: generate test cases that satisfy $1 \le N \le 100$. Assign them to subtasks $\{1, 2, 3\}$.
- Test group 2: generate test cases that satisfy $101 \le N \le 1000$. Assign them to subtasks $\{2, 3\}$.
- Test group 3: generate test cases that satisfy $1001 \le N \le 10000$. Assign them to subtasks $\{3\}$.

Let's consider another problem, this time with two input variables N and K.

1. $N = 1$; $1 \le K \le 100$
2. $1 \le N \le 100$; $K = 1$
3. $1 \le K, N \le 100$

Generate the test cases as follow:

- Test group 1: consists of only one test case $N = K = 1$. Assign it to subtasks $\{1, 2, 3\}$.
- Test group 2: generate test cases that satisfy $N = 1$; $2 \le K \le 100$. Assign them to subtasks $\{1, 3\}$.
- Test group 3: generate test cases that satisfy $2 \le N \le 100$; $K = 1$. Assign them to subtasks $\{2, 3\}$.
- Test group 4: generate test cases that satisfy $2 \le N,K \le 100$. Assign them to subtasks $\{3\}$.

Perhaps it will be understood better by drawing the Venn diagrams of the subtasks, as depicted in Fig. 1. Our advice is to have a test group for each closed region in the resulting Venn diagram.

Note that the above solution still depends on the grader system used for the contest. If the grader system only support independent set of test cases for each subtask, then a possible workaround is to have multiple copies of a test case for each of the assigned subtasks. This will, however, result in a test case being evaluated multiple times. In the

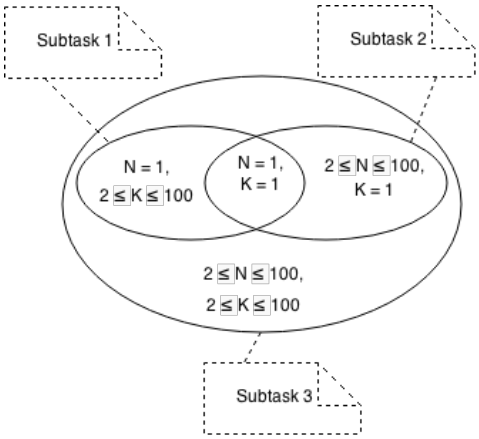

Fig. 1. Example test groups Venn diagram.

current implementation of grader mentioned in (Fernando and Liem, 2014), the above idea has been enforced. Each test case will never be evaluated more than once. Each subtask result is then deduced from the test case evaluation results.

Having formalized the test cases generation, we are now ready to discuss how to design a framework that supports it.

3. Test Cases Generation Framework Designs

There have been several previous efforts related to test cases generation that we are aware of. For each of them, we will explain the key designs, the problems that arise, and proposed solutions. Eventually, the proposed solutions are then used as foundations for writing the tcframe framework.

For each problem, there should be a set of test cases, a generator program that generates the test cases (if any), and a validator program that validates the correctness of the test cases, as proposed in (Diks *et al.*, 2008). All previous works require us to write a generator and a validator program. tcframe takes a little step further and only requires us to write a self-validating generator program.

When explaining a framework, we will assume that an official solution to the problem we are currently considering is already available. This means that our job is only generating test cases (not writing the solution).

3.1. *testlib*

testlib is a library created by Mike Mirzayanov *et al.* To generate test cases using the testlib library, we need to write two programs: a *generator program*, which outputs a test case input, and a *validator program*, which validates the produced input.

Generator program. This program, when run, will produce a single test case input file. To have some variations in the values of the input, the generator program usually contains some randomizations. The randomization parameters and seed can be passed as the program's command line arguments.

testlib provides a bunch of functions related to randomization that we can use.

For example, generating weighted random values, generating random strings based on a regex pattern, etc.

For example, here is a sample pseudocode of a generator program using testlib.

```
int maxN = the program's 1st argument

int N = randomize between 1 and maxN
println N
for 0 <= i < N:
    int a = randomize between 0 and 1000
    println a
```

Validator program. After a test case input file has been produced by the generator, we need to validate whether the values in the input file conform to the constraints. This is done by the validator program. It takes the input file as standard input and validates the values using the provided functions.

For example, here is a sample pseudocode of a validator program using testlib.

```
read an integer N and verify that 1 <= N <= 100000
for 0 <= i < N:
    read an integer a, and verify that 0 <= a <= 1000
    if i + 1 < N:
        read a space
    else:
        read a newline
verify that it is EOF
```

We found that testlib is missing several features. First, the generator program only produces a single input file in a single execution. So, to produce test data that has many test cases, we have to write a script and call the generator program several times using different arguments. Second, the validator program does not support subtasks: it is not possible to validate a test case on a specific subtask.

These features are missing because testlib was designed for ICPC-style problems.

Fig. 2 shows the diagram of steps for generating test cases using testlib.

3.2. *tokilib*

Recall that in summary, testlib does not support generating multiple test cases at once for problems with subtasks. To fill this missing feature, we wrote a wrapper around testlib, called *tokilib*. We call it wrapper because it is using testlib's generation and validation functions.

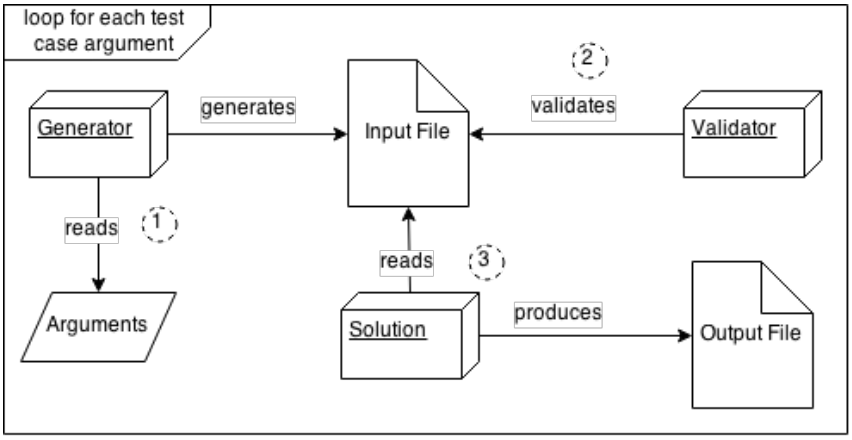

Fig. 2. testlib generation diagram.

Improved generator program. A generator program now consists of several functions, each of which defines a test group (called *batch* in tokilib). Each test group can be assigned a set of subtasks. Then, *test case definitions* follow, in the format as described in the next section below.

Component-based test cases. We introduce the concept of *components* in a test case as follows. Suppose we have a labeled tree data structure of N nodes as the test case. We can break down the structure into two *independent* components: a tree consisting of N nodes, and a list of N labels for the nodes. This way, we can generate each component independently, and then mix them to produce strong test cases.

For example, for the first component, we can have the following two variations: a binary tree and a random tree. For the second component, we can also have two variations: a list consisting of equal labels, and a list consisting of random labels.

For each component, we choose a suitable representation, which might be different from the input format section. For example, the tree can be represented as follows. Consider the tree as a rooted tree, with node 1 as the root. The representation is a vector of N elements. The i-th (one-based) element is the number of the parent of node i, or 0 (no parent) if i = 1 (Manev *et al.*, 2010).

Each component is declared as a global variable in the desired representation.

Then, a test case definition can be defined as a sequence of statements that assign the correct values to the components. Finally, we must implement a print() function, which prints the components according to the input format. Here is a pseudocode of a sample generator program in tokilib:

```
variables:
    int N
    int[] parents

print():
    println(N)
    for 0 <= i < N:
        if parents[i] != 0:
            println(i + " " + parents[i])

batch1():
    assignToSubtasks({1, 2})
    beginTC(); N = 7; binaryTree(); equalLabels(); endTC()
    beginTC(); N = 10; randomTree(); equalLabels(); endTC()
    beginTC(); N = 100; binaryTree(); randomLabels(); endTC()
    beginTC(); N = 100; randomTree(); randomLabels(); endTC()

batch2():
    assignToSubtasks({2})
    beginTC(); N = 500; randomTree(); randomLabels(); endTC()
    beginTC(); N = 1000; randomTree(); randomLabels(); endTC()
```

Improved validator program. A validator program in tokilib combines the schemes used in testlib's validator and tokilib's generator. It consists of a list of *subtask definitions*. A subtask definition consists of a list of *constraint definition*s, each of which is an assignment to a *constraint boundary variable*. A constraint boundary variable is just a

variable that holds the varying values of a constraint across subtasks, for example, min/max possible value of *N*. It is declared as a global variable. Finally, we must implement a validate() function, which validates a test case according to the assigned subtasks. Here is a pseudocode of a sample validator program in tokilib:

```
variables:
    int maxN

validate():
    read N and verify that 1 <= N <= maxN
    read tree edges
    verify that the edges form a tree

main():
    beginSubtask(); maxN = 100; endSubtask()
    beginSubtask(); maxN = 1000; endSubtask()
```

Improved usage. testlib requires us to manually call the validator program on the generated input file, and then run the solution to produce an output file.
This is not the case in tokilib: after the generator program produces an input file, the validator program will be automatically run against it. If it passes, the solution will be automatically run against it to produce an output file.

Fig. 3 shows the diagram of steps for generating test cases using tokilib.

3.3. *tcframe*

After using tokilib for generating test cases for many problems, we noticed that the part of validator that parses and checks whether the values are printed according to the input format looks similar and repetitive across all problems. We then wondered whether checking the input format in the validator can be eliminated.

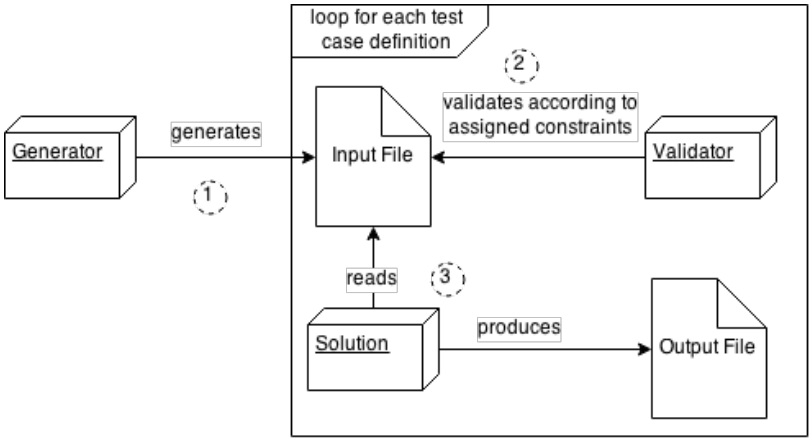

Fig. 3. tokilib generation diagram.

Let's begin with the reason why input format validation has been necessary in the first place. It has been because it is us humans who type the actual code for printing the values, and humans are error-prone. In addition, there are many ways to print the input variable values, because we may be manipulating the test case components using a different representation that defined in the input format section. For example, suppose the test case consists of an undirected graph.

The input format section might state that the graph is given as a list of edges, but we might be manipulating the graph using an adjacency list instead. One implementation for printing the edges is to print them while traversing the graph using, for example, breadth-first search. Then, if we make mistakes in the traversal code, we might print duplicate edges, or we might miss some edges.

So, we had an idea. If we hand over the input variables printing part to the framework, then the input variables parsing and input format validation should not be necessary anymore. To make this possible, we need to formalize how to declare the input variables and specify the input format. Remember that this has not been formalized before: we are free to decompose the structure into any components we want.

We propose the following formalization:

- The framework only works on "primitive" input variables. For example: scalars, vectors, and matrices of basic types (integers, floating-points, strings). Note that these are the most commonly used structures in input format sections.
- The framework provides an API to arrange the input variables in certain formats. For example, space-separated scalars in a single line, space-separated elements of a vector in a single line, multiple lines each containing an element of a vector, etc.

In this way, we only need to specify the input variables arrangement, and to assign values to the input variables for each test case. Then, the framework will take care of printing the values according to the specified input format. Assuming the framework is always correct, we now don't need to parse the variables and validate the input format in the validator program anymore.

Here is the pseudocode of the desired scheme:

```
inputVariables:
    int N
    int[] u, v

inputFormat():
    singleLine(N)
    multipleLines(N-1, {u[i], v[i]})
```

The interesting thing about this scheme is that the pseudocode is really similar to the input format section in the problem statement, which is nice because it will be easy to verify.

Now, the only thing left in validator program is constraints checking. Previously, this is done after we parse the input variables from the input file produced by the generator. Since the parsing part is now eliminated, we can do the check directly on the input variables declared in the generator, as a list of *subtask definitions*, each of which consists of a list of *constraint definitions*. As the input format validation is already similar to the

corresponding section in the problem statement, it would be nice if this part can also be made similar to constraints/subtasks section in the problem statement.

Here is the pseudocode of the proposed scheme of this part:

```
subtask1():
    declareConstraint({1 <= N and N <= 100})
    declareConstraint({graph is valid tree})

subtask2():
    declareConstraint({1 <= N and N <= 1000})
    declareConstraint({graph is valid tree})
```

Therefore, the validator program is now eliminated completely. As the generator program now contains both test cases generation and validations, we call it with another name: *runner program*.

Finally, the test groups/test cases definitions are similar to those in tokilib, except that we also make the syntax more declarative.

```
testGroup1():
    assignToSubtasks({1, 2})
    declareTestCase({N = 7; binaryTree(); equalLabels()})
    declareTestCase({N = 100; randomTree(); randomLabels()})

testGroup2():
    assignToSubtasks({2})
    declareTestCase({N = 200; binaryTree(); equalLabels()})
    declareTestCase({N = 1000; randomTree(); randomLabels()})
```

In summary, Fig. 4 depicts the flow of test cases generation using the mentioned idea so far.

We felt that this change is very significant compared to tokilib, so we decided to consider this as a brand new framework and give it another name, *tcframe*.

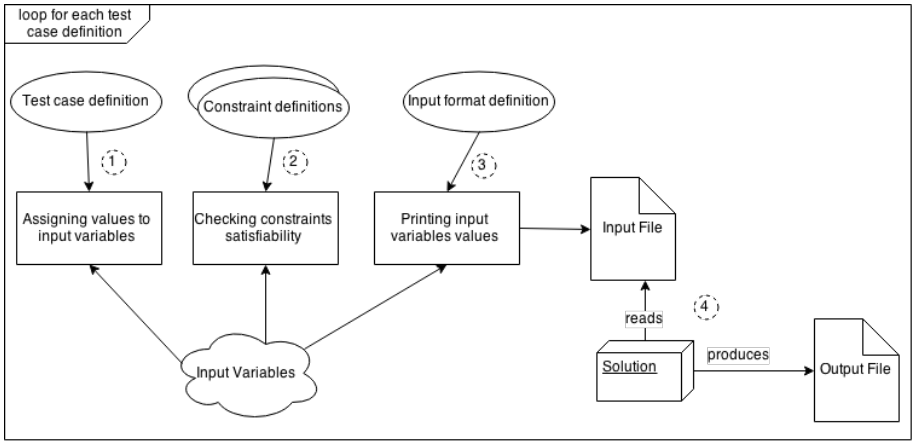

Fig. 4. tcframe generation diagram.

4. Implementation Details

In the previous section, we described motivations and ideas behind tcframe. This section will explain how tcframe is implemented in more detail.

The code is hosted on GitHub (`https://github.com/ia-toki/tcframe/`). At the time of writing, it is on version 0.3.0 and it is not ready for public use.

We chose to write tcframe in C++. One of the main reasons is because it is a popular language used by competitive programmers. In addition, C++ has preprocessor macros, which will be used extensively for producing concise but powerful code.

Let's begin with some philosophies that we want to achieve in the implementation of tcframe.

- It should be possible to write only a single runner program as opposed to writing two programs: generator and validator programs.
- The resulting runner program code should be concise, declarative, and should resemble the problem statement.
- The overall syntax should be well structured so that it is easy to extend in the future.

Fig. 5 depicts a proposed high-level class diagram to meet the above requirements.

Let's discuss each part of the class diagram in more details. Note that tcframe is not final yet so some part definitions may change in the future.

4.1. *Specification Classes: BaseProblem and BaseGenerator*

Recall that testlib and tokilib both require writing two separate programs: generator and validator. This scheme has an advantage that the validator cannot access any user-defined functions in the generator. This is important because to test a system (the generator), we

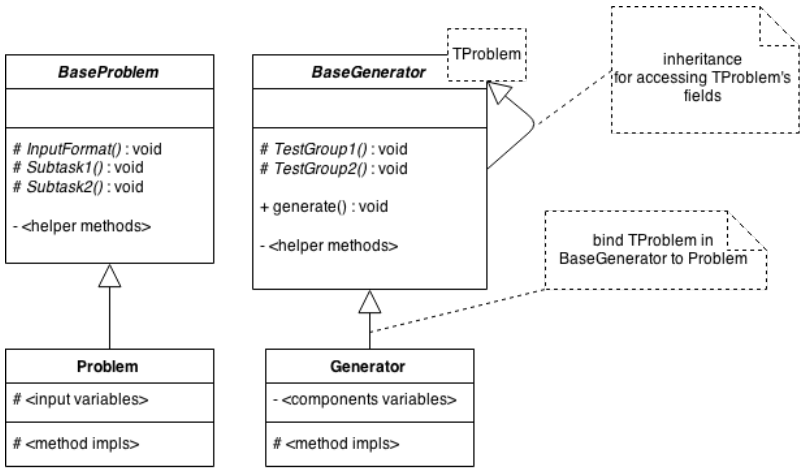

Fig. 5. tcframe class diagram.

should only use things outside the system. However, to implement tcframe, this cannot be the case: both the generation and validation steps need to access the input variables. However, we also want to prevent the validation step from using any user-defined functions used in the generation step.

To satisfy both seemingly contradicting requirements, we restructure parts of the generator and validator programs into two classes: input variables, input formats, and subtask definitions go into *problem specification* class, while test cases definitions go into *generator specification* class. Both classes then go to just a single program. The definitions are implemented as virtual (abstract) methods in these classes, which must be overridden in concrete classes. By convention, let's name them Problem and Generator. Any user-defined functions then should be declared private in Generator.

As mentioned above, the input variables go into the Problem class, as member variables. The Generator class then must be given access to the member variables.

We notice that there are two ways to implement this requirement: by composition and by inheritance.

By composition. The Generator will hold an object of type Problem, whose member variables are declared public. This is nice, but now whenever we want to assign values to input variables in the Generator class, we must type the object name (e.g., problem.N = 100). We feel that this way is not concise and pleasant.

By inheritance. To avoid writing an object name before each input variable, we chose to use inheritance. The input variables are to be declared as protected member variables in Problem. Then, we make BaseGenerator inherit Problem. BaseGenerator now can access the input variables. Generator is also able to access them as well, since it inherits BaseGenerator.

Finally, to make the above scheme work for any Problem class, we use templates in the following way. Here is the declaration of BaseGenerator:

```
template<typename TProblem>
class BaseGenerator : public TProblem;
```

And here is the declaration of Problem and Generator:

```
class Problem : public BaseProblem;
class Generator : public BaseGenerator<Problem>;
```

4.2. *Definitions: Input Format, Constraints/Subtasks, Test Cases/Test Groups*

The definitions are realized as virtual member functions in the base specification classes. We must implement those functions in the concrete specification classes, and put the definition items inside them by making API calls. They should have something in common: if any step in the whole generation process fails, then the definition item that causes the failure should be presented to the users so that the users can fix it. For example, if

an input variable value does not satisfy a constraint, something like “Error: constraint $1 \le N \le 100$ not satisfied" should be output.

To support the above requirement, the definition items will be implemented using C++ macros, which supports the stringization trick of the input parameters.
The next section will show the planned syntax for the macros.

Input format. It consists of one or more *input segments*, which will be printed one after another. Currently, the following input segment types are supported: space-separated scalars/vectors in a single line, lines each containing an element of vectors, and grid. For example, the definition of a line segment (the first type) is made by calling this macro:

```
LINE(A, B);
```

which then expands to something like:

```
inputFormat.addLineSegment("A, B", A, B);
```

The above call defines a line that consists of input variable A, followed by a space, followed by input variable B.

Constraints/subtasks. BaseProblem declares virtual methods Subtask1() ... SubtaskX() for a finite number X (currently it is set to 10). To define a subtask, implement any of the mentioned methods in the Problem class. Inside the method, we can define one or more constraints, by calling this macro:

```
CONS(1 <= N && N <= 100);
```

which then expands to something like:

```
constraints.add("1 <= N && N <= 100",
        [this] { return 1 <= N && N <= 100; });
```

In tokilib, we implement a constraint definition as an ordinary C++ statement.

In tcframe, we will use a new feature in C++11: lambda closure, for each constraint. This has several advantages. For example, each constraint is now independent from any other and they can be called in any order. It also makes the framework more extensible; for example, it becomes possible to write a plugin based on tcframe that only prints all constraint descriptions.

Test cases/test groups. BaseGenerator declares virtual methods TestGroup1() ... TestGroupX() for a finite number X (similar to subtasks, currently it is just set to 10). To define a test group, implement any of the mentioned methods in the Generator class. Inside the method, we can define one or more test cases, by calling this macro:

```
CASE(N = 100, randomArrayElements());
```

which then expands to something like:

```
testCases.add("N = 100, randomArrayElements()",
        [this] { N = 100, randomArrayElements(); });
```

Similar to constraint definitions, test case definitions also make use of lambda closures. Note that we choose comma operators rather than semicolons for separating input variable assignments. This way, the test case definition looks more "declarative": it consists a *list* of assignments to the input variables, rather than *statements*.

Finally, we also want to be able to use component-based in tcframe. This can be achieved by declaring the component variables in the Generator class, use them in test case definitions, and then convert them to the actual input variables in the Problem class before the end of each test case definition.

5. Conclusion

We have presented the ideas behind creating tcframe, and how we are currently implementing it. We also suggested a formalization on test cases organization.

We hope that tcframe will allow more people to be able to create good test cases systematically. For programming contests with multiple authors, we hope that tcframe allows the authors to be able to work together creating test cases more collaboratively.

6. Future Works

We are aware of several possible other improvements and new features that can be implemented in tcframe.

6.1. *Output Validation*

testlib, tokilib, and tcframe all share a problem: the produced test case outputs, obtained by running the solution against the generated test case inputs, are not validated. This is actually very dangerous because the solution might have some mistakes and print the output not according the output format, or have some invalid values.

We can validate the outputs in a similar way as we do to the inputs:

- Declare *output variables*. For example, in a single-integer answer, we can call it $answer$.

- Declare *output format*. The framework can then use this format for parsing the produced outputs, validating the format, and storing the values to output variables.
- Declare *output constraints*. For example, if a problem requires the answer modulo 10007, we have an output constraint $0 \leq answer \leq 10006$. The framework can then use it for validating the output variables.

6.2. *Answer Checker*

For problems with several possible solutions, we need a checker that compares the judge's and contestant's answers. Our class structures make it easy to add this functionality. We can build a BaseChecker class on top of Problem class, similar to BaseGenerator. The check can be then somehow implemented as follows.

```
class Checker : public BaseChecker<Problem> {
public:
    bool check(const Problem& contestant, const Problem& judge) {
        return fabs(contestant.answer - judge.answer) < 1e-9;
    }
};
```

6.3. *Offline Solution Checker*

Instead of just producing test cases, we can let tcframe take another solution as input, then report whether the outputs produced by that other solution match the outputs produced by the official solution. This way we can effectively "submit" a solution without using the online judge. We can limit memory and time limit using, for example, **ulimit** UNIX command.

6.4. *Public Library for Commonly Used Structures*

We want to provide a public place where people can contribute by writing generators that generate common test cases structures. For example, polygons for convex hull problems, coin values for coin change problems, etc. The submitted generators should generate strong test cases. For example, there should be convex polygons, the coin values should be in such a way that greedy solutions will fail, etc.

This will make it even easier for beginners to generate test cases. They can browse the library for the structures they need and modify the generator for their problems.

References

Diks, K., Kubica, M., Radoszewski, J., and Stencel, K. (2008). A proposal for a task preparation process. *Olympiads in Informatics*, 2, 64–73.

Fernando, J. and Liem, I. (2014). Components and architectural design of an autograder system family. *Olympiads in Informatics*, 8, 69–79.

Fuadi, A. (2014). *tokilib*. `https://github.com/fushar/tokilib/`

Halim, S., Halim, F. (2010). *Competitive Programming*. `Lulu.com`

Manev, K., Yovcheva, B., Yankov, M., Petrov, P. (2010). Testing of programs with random generated test cases. *Olympiads in Informatics*, 4, 76–86.

Mirzayanov, M. (2008). *testlib*. `https://github.com/MikeMirzayanov/testlib/`

van der Vegt, W. (2009). Using subtasks. *Olympiads in Informatics*, 3, 144–148.

A. Fuadi is a fellow in Indonesian Computing Olympiad Alumni Association. He participated and obtained a silver medal in IOI 2010. He has been a scientific committee member for IOI training camps for Indonesian teams 2011–2014. Graduated from Faculty of Computer Science, Universitas Indonesia in 2014.