Center for Open Access in Science ▪ https://www.centerprode.com/ojit.html
Open Journal for Information Technology, 2018, *1*(2), 25-36.
ISSN (Online) 2620-0627 ▪ https://doi.org/10.32591/coas.ojit.0102.01025a

Automated Checking in Education

Tatiana A. Andreyeva

A. P. Ershov Institute of Informatics Systems, Novosibirsk
Siberian Branch of the Russian Academy of Sciences

Received 13 July 2018 ▪ Revised 25 September 2018 ▪ Accepted 2 October 2018

Abstract

This work concerns the automated checking of solutions in education. The article discusses the structure of a problem as a whole body and of its parts, studies various check approaches, introduces problem complexes, suggests methods for creating accurate and consistent problem statements and check sets, and touches the automation of the preparation of problem sets and of the checking processes.

Keywords: automated correctness checking, test case generation, text analysis.

1. Introduction

Systems for the automated checking of solutions have their origin in programming contests. But today they become the means of automation of the teacher's work. They can be useful not only at programming classes. With their help, any teacher can organize a mini-contest or check own tests and students' works faster and easier.

Having the experience of using and creating the automated checking systems for the programming problems, the author extends the corresponding approach onto the non-programming problems and shows that they also can be checked automatically. In order to make the automated checking easier for a wider range of users, its preparation also should be automated.

- Our aim is to make the automated checking of tests and contests easier for teachers.
- Problems from various fields can be checked automatically like the programming ones.
- The preparation of problems for a contest or a quiz should also be automated.
- Automated systems for preparation and checking are described (not very profoundly).
- Automation reduces the number of possible errors, ambiguities, and inconsistencies.

Section 1 contains the necessary mathematical notes along with glosses on what a generalised problem, its solution method and results are. It also shows how the number of check cases and the check approach depend on the amount and the type of the problem's variable and constant data, its conditions and restrictions.

Section 2 discusses various approaches to the checking of solutions.

In order to create full, explicit and consistent problems (not only for contests with the automated checking of solutions but for all quizzes as well) it is necessary to realize the structure

Correspondence: Tatiana A. Andreyeva, VLSI CAD Laboratory, A. P. Ershov Institute of Informatics Systems, 6, Lavrentiev st., Novosibirsk, 630090, RUSSIAN FEDERATION. E-mail: ata@iis.nsk.su.

of a problem as a union of a statement, specifications and checking means. These are considered in **Section 3**. The important notion of the *problem complex* is also introduced here.

Section 4 is dedicated to questions of the automated preparation of the problem complexes.

2. Mathematical and common-sense basis

2.1 *Problem, solution and result*

Let a problem be represented by a triad $< D, C, R >$, where D is known data, C is conditions necessary for creating a correct solution, and R is unknown values to be found out during solving. Then the solving *method* M is a function, which creates the result R from the initial data D and the conditions C.

$$M: D \times C \rightarrow R.$$

Solution method (*algorithm*) is a sequence of inter-connected and inter-dependable components (R_1, … R_N); each *step* R_i is based upon steps $R_{[0...i-1]}$. The sequence (R_1, … R_N) is not bound to be strictly linear, on the contrary, in most cases it only can be depicted by a graph structure.

Some people think that "to solve a problem" means "to find a correct answer" and sometimes, if the solving method is obvious, they are almost right. Still, the method itself can be the point to find. For example, in IQ tests (see, for example, Eysenck, 1962) and the like, an often task is "find the rule and then apply it" and figuring out the rule is much more difficult than retrieving the answer with the help of this rule. In this case, the outcome is not the result but only the means to judge about the correctness of the invented method.

2.2 *Power and dimension of the input*

Domain D of all known data can be divided into two parts: invariables D_{inv} and variables D_{var}. *Invariable data* are stated in the problem's description and never change. On the contrary, *variable data* are provided by the *check cases* (see Section 2.3.3. Check cases) and can change from case to case not only in value but in number, too.

2.2.1 *No variables*

Obviously, there exist problems with no variable input at all. Almost all non-programming problems are like this. In such a case, the *power* $| D_{var} | = 0$.

If such a problem is correctly stated, it must have the unique correct answer. As a rule, if there are several correct answers, it means that the problem description is not full or consistent. For example, a partial case is not correctly pointed out[1] or no explicit specification of a suitable output is provided[2].

[1] For illustration, consider the problem: "Two segments on a plane are given by coordinates of their ends. Define their mutual position: (a) do not touch; (b) intersect; (c) partly overlap; (d) one of them contains another". It is obvious, that the partial case of a zero-length segment can meet both b and d variants. It would be better to divide the output domain not into four but into three equivalence classes: "… (a) do not touch (no common points); (b) intersect (1 common point); (c) partly or fully overlap (more than 1 common point)".

[2] For example, in an English language test, one can give answers "I cannot…" or "I can't…" Both are correct, but if the output description is not worded accurately, a mistake can be reported erroneously.

2.2.2 *Variables as the means to split a problem*

In programming, any "good" algorithm has to be *mass-oriented*, i.e. it must be potent to solve not a single problem but a whole class of similar problems. Therefore, presence of a variable input is characteristic for programming problems.

Still, problems with variable input data can be met not only in programming but in other fields, too. For example, several variants of a quiz may include the same task with different numeric values[3].

If there are several variables $V_1, \dots V_N$, each of them having its own domain D_i; then the whole variable domain D_{var} can be represented as a direct product of these domains:

$D_{var} = [D_1 \times \dots \times D_N]$.

And the dimension of the domain D_{var} is the sum of dimensions of $D_1, \dots D_N$:

$\dim D_{var} = \dim D_1 + \dots + \dim D_N$.

If we take one value for each variable, we make a section of the domain D_{var}. The initial problem with a restricted input domain is also a problem, but with no variables now. Thus, we can reduce a problem with a variable input to a problem with the invariable input.

2.2.3 *Constant and known number of variables*

Now let us consider one variable V_i. The dimension of its domain can vary in a wide range from 1 to any value having the practical sense. In simple words, this *dimension* denotes the number of variable's components.

As a rule, the upper bound for possible dimensions of a variable's domain is specified explicitly for all variables in the problem's description, it is known before the checking process is started. It can be considered as a constant belonging to the invariable input D_{inv}.

Also, the upper bound for some variable can be specified not in the description but in check cases. Then it belongs to D_{var}, is a variable itself and, thus, must have an upper bound, too. Let us call a changeable upper bound the *sub-bound*.

An example of such a situation is "N integers $A_1, \dots A_N$ are given ($1 < N < 100$)…". Here variable A consists of N components $A_1, \dots A_N$ and, therefore, its dimension equals to N. Variable N is the sub-bound of the current dimension, and 100 is the invariable upper bound common for all possible sub-bounds. Note that each A_i must have its own upper limit too, but this example does not mention any.

A sub-bound is variable but becomes known when the check starts.

2.2.4 *Indefinite length of the input*

Now let us consider the case when the actual dimension of a variable's domain stays unknown until the end of the checking.

Example is "No more than 100 integers are given…" Here we do not know how many components $A_1, \dots A_N$ the actual input has. We can preliminary write all of them down and count them; then we will know the current sub-bound N (not given but calculated). Thus, we return to

[3] In such a case, each variant can be checked as an independent one and, therefore, can represent the single-answer case.

the case of a known sub-bound. On the other hand, we can process these components not using the value of ***N*** at all. The difference can be illustrated by cycles:

for 1 to N do… and *do… until <the end is detected>*.

In programming, the second way is preferable since it means that the input file must be read through only once. And the first way means that reading is performed twice, which is inefficient if the file is large.

Another example of the situation when the number of variables must be retrieved from the input is "A graph is specified by the list of its edges, which are pairs of vertex numbers".

Let us note that theoretically both the number of variants and their range can be infinite. Nonetheless, practically it is impossible.

2.3 *Power of the output*

The number of possible correct answers $|R|$ is very important for our discussion of the automated checking. Here we only mention the possible variants. And the influence of the multiplicity of possible correct answers onto the checking process is discussed in Section 2.3.4 *Checking and judging*.

$|R| = 0$ means that the problem is stated erroneously. No correct answer is possible. Nothing is to be checked. Such problems must not appear in any test, quiz or contest. Some ways to avoid such errors are discussed in Section 3.1 *Problem's description*.

$|R| = 1$ means that the solution exists and it is unique. In this case, the checking is obvious and easy: it is sufficient to ascertain that the output coincides with the given exemplar answer.

$|R| > 1$ means that the problem has several correct answers. There can be three cases:

- $|R|$ is finite. For example, "The anterior part of a shoulder is called a collar bone or a clavicle".
- $|R|$ is infinite but denumerable. For example, "Any even integer".
- $|R|$ is infinite and non-denumerable. For example, "Any real value from the interval $[0 .. 1]$" or "Any point on the plane within the circle with the centre in $(0, 0)$ and radius 1".

Let us note that restrictions of the computer data representation obviously reduce the case of infinite (denumerable or non-denumerable) $|R|$ to the case of finite $|R| > 1$. The only difference between them is in approaches to the checking.

3. Checking and correctness

3.1 *Solution: Method or result?*

Now let us return to the difference between an outcome and a solution. What should we check? Should the result or the method be of most interest?

The well-known joke (unfortunately, its author is unknown to us) illustrates that an erroneous method also can produce a correct answer:

– Reduce the fraction 16/64.

– 1/4.

– OK! How do you count?

– I've crossed out the 6's above and below.

Thus, we have to remember that the aim of checking is to form a judgement about the method not the outcome.

3.2 *Method is a White box*

So, how can we check a solution method?

One way is to check the description of the method. For example, "To find a Fibonacci number, sum two previous Fibonacci numbers, starting from two units". Or, mathematically, "For any natural $k > 1$, $Fib_k = Fib_{k-1} + Fib_{k-2}$, while $Fib_0 = Fib_1 = 1$". These are two equivalent descriptions of the same method.

Still, there can be equivalent but different methods. For example, in order to find the greatest common divisor (G.C.D.) of two natural numbers, one can a) use the Euclidean algorithm[4] or b) find all natural devisors for both numbers separately (by trying to divide them by each natural), compare these sets of divisors, and find the greatest common one. These methods are obviously equivalent, while the first one is much more efficient than the second.

If it is important that the solution method be a particular one, the author of the problem can shift the focus from the result to the method: not "Find the G.C.D. of two naturals" but "Describe the Euclidean algorithm of finding the G.C.D.". In this new problem, what earlier was a method (one of several possible ones) became the result.

Although there exist methods for *automated verification* (starting from Floyd (1967), Hoare (1969) and Anderson (1979), these methods have been developed by their followers), our aim is to make the checking easier for a wider circle of teachers. Therefore, we seek for less complicated and laborious way of checking. The mentioned verification theory and methods are the instruments for designers of the automated checking systems not for users of these systems.

So, let us consider another way of checking.

3.3 *Method is a Black box*

The other way to check a method is to inspect its behaviour: "if we feed a valid input, what output shall we get?" This approach is called *Black-box testing* (was introduced by Ashby, 1956).

The theory of *Black-box testing* is well developed for computer programs (see, for example, Ashby, 1956, Beizer, 1995 or Ponrod, 2014); we shall try to adopt some of its methods for developing the theory of the automated checking in education.

3.3.1 *Correct or incorrect?*

How can we decide that the method under examination is correct? For the behavioural approach, the answer is "a method is correct if and only if it always produces a correct outcome". But what is the *correct outcome*? It is the result of applying a correct method M to the valid input data:

[4] The Wikipedia (https://en.wikipedia.org/wiki/Euclidean_algorithm) gives its detailed description.

$$M: D \times C \rightarrow R.$$

Therefore, we need some *exemplar* method. We declare this exemplar method M_{ex} *correct* and check whether the method under examination is *equivalent* to M_{ex}. In other words, we believe that method M is *correct* if and only if, being applied to the same inputs, methods M and M_{ex} produce the same results.

To make the matter more intricate, there is a situation with multiple correct answers (see Section 1.3 *Power of the output*). In the case of $|R| > 1$, the exemplar method should provide all possible correct outputs while the method under examination may produce only one of them. So, the equivalence should be not between two methods M and M_{ex} but between the method M and only a sub-method of the method M_{ex}.

Note that we cannot prove correctness both of a method and of its outcome simultaneously. We only can prove that two methods are or are not equivalent. Here is the source of unavoidable difficulties: if the exemplar method is erroneous (intentionally or unintentionally), it produces the erroneous outcome, which nonetheless is declared "correct". Therefore, henceforth we call the exemplar method's outcome not *correct outcome* but *exemplar outcome*.

Irrespectively to its actual correctness or erroneousness, the exemplar outcome is the base for judging about correctness of a method under examination. Therefore, it is important to eliminate the possibility of errors in the exemplar method and the exemplar outcome. And here an automated system for preparation of problem complexes can be of great use (see Section 4.2 *Automated systems for preparation of problem complexes*).

3.3.2. *Exemplar input and output*

How can we get an exemplar outcome? Should we apply the exemplar method to all possible inputs? Obviously, this way is too generous. It is sufficient to apply the exemplar method only to some characteristic representatives.

The domain of valid inputs D can be split into equivalence classes. Input data belong to the same equivalence class if they generate (with the help of exemplar method) the same (or equivalent) outcomes. Some of these outcomes should be "good", some "bad". Each equivalence class is considered to be a *partial case*. Only one representative from each partial case is sufficient for the exemplar input (see, for example, Beizer, 1995, or Myers, 1979 or 2011).

If the domain D is infinite (see Section 1.2 *Power and dimension of the input*) than some (or even all) of the equivalence classes can be infinite too. If the number of classes is finite, getting one representative from each class forms a finite set of exemplar inputs. Still, there can be infinite number of equivalence classes. In order to restrict this number, additional restrictions should be imposed on the domain of valid input data.

The partition of the domain D into equivalence classes can be done manually basing on the characteristics of the subject domain and the problem itself or automatically through the inner properties of the exemplar method.

In programming, such exemplar method that predicts exemplar inputs and outputs is called an *oracle*[5].

[5] For a sketchy description of the oracle, see, for example, Wikipedia (https://en.wikipedia.org/wiki/Test_oracle or https://en.wikipedia.org/wiki/Black-box_testing).

3.3.3 *Check cases*

Having an exemplar method, one can trace all partial cases it processes. Since all inputs from an equivalence class are interchangeable, the representatives can be selected randomly.

Now that we have a set of exemplar inputs and an exemplar method, we easily produce the corresponding set of exemplar *outcomes*, each of these can consist of more than one "correct" *output* (see Section 2.3.1 *Correct or incorrect?*)

Let a *check case* be a pair of some exemplar input and the corresponding exemplar outcome. We refer to a pack of check cases as a *check set*. Here we follow the analogy with *test cases* and *test sets* in programming[6].

Mathematically, a check case is a set of points representing a section of the domain of the functional $\boldsymbol{M}$. In common words, a check case is a sub-problem of the initial problem where all variables in the $\boldsymbol{D}$, $\boldsymbol{C}$ and $\boldsymbol{R}$ parts have concrete values.

3.3.4 *Checking and judging*

Having an exemplar input and the corresponding exemplar output, we apply the method under checking to the exemplar inputs, get outputs, and compare each acquired output with the corresponding exemplar output or the exemplar outcome.

If the problem admits only finite number of correct answers, the comparison can be easily performed by verifying the coincidence (see Sections 1.3 *Power of the output* and 2.3.3 *Check cases*). In this case, the exemplar outcome consists of one or several exemplar outputs. Let us also note that a poly-dimensional output brings almost no difference into the result-checking procedure.

The case of an infinite $|\boldsymbol{R}|$ is more difficult. We cannot practically list all possible outputs; therefore, the "comparison" should mean performing a special checking formula, which depends on the type of the valid outputs. For example, we can ascertain that "a real $\boldsymbol{Z}$ belongs to the $\boldsymbol{[0..100]}$ interval" by checking that both $\boldsymbol{Z \geq 0}$ and $\boldsymbol{Z \leq 100}$ are true. A poly-dimensional output can demand more complex formulas. For example, the result "a point on a plain with coordinates $\boldsymbol{(x, y)}$ belongs to the circumference with the center in $\boldsymbol{(0, 0)}$ and radius $\boldsymbol{A}$" can be checked with the help of the $\boldsymbol{x^2 + y^2 = A^2}$ formula[7].

If the author of the problem would rather avoid such difficulties, the problem's statement should be revised and the type of the output changed.

If the acquired output coincides with an example output (when $|\boldsymbol{R}|$ if finite) or meets the differently stated conditions (when $|\boldsymbol{R}|$ if infinite), the *check case result* is correct. Otherwise, it is incorrect.

After all check cases are processed; a judgment about the correctness of the whole method can be formed. And there are two ways for this.

[6] Mostly, experts in programming (see, for example, Myers, 1979, 2011, Singh, 2012, or Spillner, 2014) use the term *test suite* to name a pack of test cases. Still, *ISO/IEC/IEEE 24765:2010 International Standard – Systems and software engineering – Vocabulary* does not mention this word at all. Instead, it uses the *test set* (3.3091). So, we use the *test set* as the synonym of the *test suite*, too.

[7] More accurately, this formula should look like the pair of inequations $\boldsymbol{A^2 - e \leq x^2 + y^2 \leq A^2 + e}$, where $\boldsymbol{e}$ is an admissible (and strictly specified in the problem's description) error.

The first way is the dichotomy "*all check case results are correct*" vs. *"at least one check case result is incorrect"*. The method is considered correct if and only if all its check case results are correct.

The second way is a gradation based upon the number of correct check case results. The metric for this gradation can be determined in various ways. For example, in an equipollent metric, each check case gives 1 point or *100/n* percent of the result. On the other hand, in a weighted metric, check cases make different contributions to the result. On this way, a method can be *more correct* or *less correct* than the other method, according to their metric values.

4. Problem complexes

Now let us look at a problem as the subject for the automated checking.

The *problem complex* should include:

- ***Description*** of the problem,
- ***Specifications*** of a valid input and output,
- A ***check set***, which is a pack of exemplar *input-output* pairs,
- An ***exemplar*** solution ***method***.

The first and the second parts are "exterior". Contestants may see them. The third and the fourth parts, on the contrary, are for inner use of checkers and judges only.

The last two items are discussed in Section 2, now let us consider the remained two.

4.1 *Problem's description*

The structure of descriptions of programming problems has been studied by Andreyeva (2002, in Russian). Here we repeat some of those results.

The full description of a problem must contain, explicitly or implicitly, the following parts:

- ***Introduction***. A more or less detailed characterisation of the subject domain.

  If this part is omitted, the problem is already formalized (such problems are often called *dry*).

  It is important that, with the help of different ***Introduction*** parts, the same base (formalized) problem can produce several seemingly not similar problems. Their descriptions can have nothing in common at all; still, they are the same problem and their solutions can be checked by the same check set.

- ***Definitions***, agreements, terms, if necessary.

  This part can be omitted if the problem only uses commonly known notions. Still, putting anything *by default* can cause difficult-to-locate errors and misunderstandings.

- ***Statement***. A formalized presentation of the problem, its conditions and restrictions.

If this part is omitted, the problem needs formalization (as if it just has emerged from some informal subject field). Nonetheless, the ***Task*** part always can serve as a clue for formalization.

- ***Task***. Requirements whose fulfilment means that the problem is solved.

This part cannot be omitted.

- ***Formats for the input and output data***. Definition of the way to write down the input variables (if any) and the results.

This part is important not only for programming problems but also in tests and quizzes as well.

- ***Example*** of a correctly written down solution and result.

This part is not obligatory; still, it is strongly recommended that it is provided.

If any of these parts is omitted or feeble then understanding and solving of the problem, checking of the acquired results can become much more difficult. Mistakes, inaccuracies, ambiguities, inconsistencies, conflicts between parts will necessarily lead to an erroneous solution of the problem (Andreyeva, 2002).

4.2 *Specifications*

All variables must be listed in the ***Input Specifications*** subsection of a problem complex (see Section 3. Problem complexes); for all of them, the type and the bounds must be specified. Still, variables of complicated types may have an undefined length (see Section 1.2.4 *Indefinite length of the input*). This is mostly stated in the ***Formats for the input and output data*** section (see Section 3.1 *Problem's description*).

Input and output data specifications, restrictions and clauses are specified in the problem's description written in a natural language. A textual analysis can automatically extract the preliminary specification list, which should be revised manually (Andreyeva, 2018a).

5. Automation

Systems for the automated checking of solutions imply some restrictions on the types and wording of the problems. This also demands a higher discipline from authors of all parts of a problem. The special systems for the automated preparation of program complexes can reduce the number of possible errors, ambiguities, and inconsistencies.

5.1 *Preparation of problem complexes*

The process of preparing a problem complex is iterative: creating or changing each part (see Section 3. Problem complexes) can impel changes in any other parts.

Stage 1. According to the original idea of the problem, the author of the problem's description

(a) defines restrictions ***R*** on all variables in use;

(b) makes a preliminary decomposition D^* of the input data domain D into equivalence classes showing all possible partial cases[8];

(c) sets specifications S for the input and output data.

Stage 2. From specifications S and decomposition D^*, a check set CS is prepared. This can be done manually or automatically with the help of a test-preparing system (see Section 4.2. Automated systems for preparation of problem complexes).

Stage 3. An exemplar solution ES is written (manually) and is debugged with the help of the check set CS.

In order to reduce the number of possible errors, it is recommended that problem complexes are created collegially. If two authors A_1 and A_2 write two different exemplar solutions ES_1 and ES_2 and use two check sets CS_1 and CS_2 for debugging, both of them fulfil stages 1 to 3, and then Stage 4 arises.

Stage 4. Two check sets are compared and combined. Both solutions ES_1 and ES_2 must be tested on the united check set $CS = CS_1 \cup CS_2$. If no cross-errors were detected, it is necessary to ascertain that this united check set agrees with the final decomposition D and meets the final specifications S. Most likely, the united check set CS will be superfluous; and, therefore, some surplus check cases should be excluded.

The 4th stage can also be useful for the individual preparation of a problem complex. The author's initial check set and the automatically generated check set can be treated as CS_1 and CS_2.

5.2 *Automated systems for preparation of problem complexes*

Automated systems for preparation of problem complexes (ASPPCs) are described by Andreyeva (2018b, in Russian). The important part of an ASPPC, the automated system for the generation of test sets (ASGTS) was also studied by Andreyeva (2016). Here we translate some of those results.

ASPPCs make preparation of problem complexes easier and more accurate and eliminate the amount of possible errors, especially if the problem's description, specifications, the check set, and the exemplar solution are created collegially.

At Stage 1 (see Section 4.1 *Preparation of problem complexes*), an ASSPPC should:

- Extract a preliminary set of specifications S_o from the description of the problem (see Sections 3.1 *Problem's description* and 3.2 *Specifications*) by means of the textual analysis,
- Check the consistency of specifications,
- Extract possible information about boundaries, exceptional points and so on from the problem's description and specifications S_o,
- Construct a preliminary partition P of the valid data domain D into equivalence classes (basing on the specifications S_o and additional information provided by the author(s) of the problem),
- Compare, join and intersect partitions P_1 and P_2,

[8] On the very first stage, it is impossible to use the exemplar solution since it is not created yet.

At Stage 2, an ASSPPC should:

- Create exemplar inputs basing on the partition $\boldsymbol{P}$,
- Ascertain that the author's check set $\boldsymbol{CS_o}$ covers the partition $\boldsymbol{P}$,
- Verify that the check set $\boldsymbol{CS_o}$ meets specifications $\boldsymbol{S_o}$ and restrictions $\boldsymbol{R}$.

If the necessity to change the initial specification set $\boldsymbol{S_o}$ is detected, the process of creating an exemplar check set should be started anew, now basing on the renewed specification set $\boldsymbol{S_o'}$.

At Stage 3, with the exemplar outputs generated with the help of the exemplar solution (method), an ASSPPC should:

- Check if the exemplar outputs meet the specifications $\boldsymbol{S}$ (which is the final version of the specification set),

At Stage 4:

- Define the equivalent check cases,
- Propose variants of reducing the joint check set.

6. Conclusion

Our aim is to automate processes of preparing the problem complexes in any subject field, in order to make the automated checking easier and its use wider.

We have considered the notions *checking* and *correctness* and have ascertained that not only programming problems but problems from other subject fields too can be checked automatically.

We have studied processes that constitute the preparation of a contest or a quiz and the checking of their results and have shown which of these processes can be automated.

We discussed the necessary parts of a full and consistent description of a problem and have proposed ways to reduce the number of possible author errors, ambiguities, and inconsistencies.

We have shown that automated systems make the preparation of problem complexes easier and more accurate.

The future aims of our work are (a) to design means for the coverage analysis of the partitions created automatically from exemplar solutions, (b) to develop the mathematical apparatus for operations with partitions of different types, (c) to create means of partition analysis in order to ascertain that all important equivalence classes inspired by the current subject field are considered, and (d) to develop means that can suggest additional partition variants basing on the analysis of the type, the power and the dimensions of the input data.

Acknowledgements

The work was supported by the Russian Foundation for Basic Research grant RFBR № 18-07-01048.

References

Anderson, R. B. (1979). *Proving programs correct*. John Wiley & Sons, Inc. (Chapter 5).

Andreyeva, T. (2002). Structure and classification of contest problems' texts (Структура и классификация текстов олимпиадных задач). *Компьютерные Инструменты в Образовании,* 3-4, 50-59 (in Russian). http://ipo.spb.ru/journal/index.php?article/223/ or http://ipo.spb.ru/journal/index.php?magazines/2002/34/e/.

Andreyeva, T. (2016). Automated generation of test sets. In: *Science in the Modern Information Society IX: Proceedings of the conference* (pp. 110-112). North Charleston, USA, 1-2 August 2016.

Andreyeva, T. (2018a). Automated preparation of problem complexes. *Материалы Международной Научно-практической Конференции «Наука Сегодня: Теоретические и Практические Аспекты», г. Вологда, 27 декабря 2017 г.: в 2 частях. Часть 1*, 25-26 (in English). Retrieved from http://volconf.ru/files/archive/01_27.12.2017.pdf.

Andreyeva, T. A. (2018b). Automated preparation of problem complexes for programming contests (Автоматизированная подготовка задачных комплектов для олимпиад по программированию). *Наука. Информатизация. Технологии. Образование: материалы XI международной научно-практической конференции. – Екатеринбург, 26 февраля-2 марта 2018 г.* Екатеринбург, РГППУ, 10-23 (in Russian). Retrieved from http://nito.rsvpu.ru/files/nito2018/nito2018.pdf.

Ashby, W. R. (1956). *Introduction to cybernetics*. Chapman & Hall.

Beizer, B. (1995). *Black-box testing: Techniques for functional testing of software and systems*. New York, NY, USA: John Wiley & Sons, Inc.

Eysenck, H. J. (1962). *Know your own I. Q*. Penguin Books.

Floyd, R. W. (1967). Assigning meanings to programs. In: J. T. Schwartz (Ed.), *Mathematical Aspects of Computer Science. Proceedings of Symposium on Applied Mathematics. 19. American Mathematical Society* (pp. 19-32). ISBN 0821867288

Hoare, C. A. R. *(*1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*(10), 576-580. https://doi.org/10.1145/363235.363259

ISO/IEC/IEEE 24765:2010 International Standard. *Systems and software engineering – Vocabulary*. IEEE. https://doi.org/10.1109/IEEESTD.2010.5733835

Myers, G. J. (1979). *The art of software testing*. New York: John Wiley & Sons.

Myers, G. J., Badgett, T., & Sandler C. (2011). *The art of software testing* (3rd ed.). New York: John Wiley & Sons.

Ponrod, C. (2014). *The study of black-box testing technique for collateral management system*. Mahidol University Press.

Singh, Y. (2012). *Software testing* (Chapter 1.3.4.). Cambridge University Press.

Spillner, A., Linz, T., & Schaefer, H. (2014). *Software testing fundamentals: A study guide for the certified tester exam* (4th ed.). Rocky Nook Inc.