Automatic Feedback on Logic Problems about Real Numbers and Integers

Antti Valmari[1,*]

[1]University of Jyväskylä, PO Box 35, FI-40014 University of Jyväskylä, Finland

Abstract

We discuss examples of automatically providing feedback on students' answers to problems that are about real numbers or integers, but where also logical operators such as "and", "not" and "there is" are needed. If the answer is mathematically incorrect, a counter-example is provided (unless the teacher has switched this off). The teacher may tell our tool to check that the final answer satisfies certain requirements, such as is in a simplified form. If such a requirement is violated in an otherwise correct answer, the tool tells that. For instance, it accepts $2 \leq x < 7$ but rejects $2 \leq x < 5 \vee 4 \leq x < 7$. Although our tool can only deal with "almost linear" expressions, we illustrate that versatile problems at different levels of difficulty are possible. Reasonings that follow paths chosen by students (instead of teachers) can be checked to the extent needed in solving systems of (in)equations.

Keywords

automatic feedback, predicate logic, Presburger arithmetic, deterministic finite automata

1. Introduction

This paper discusses our efforts in advanced automatic feedback on problems that apply logic to real numbers or integers at the school and early university level. By "automatic" we refer to feedback generated by the tool and given to the student immediately. So we do not mean, for instance, feedback provided later by a human teacher based on answers and statistics collected by the tool. Automatic feedback helps students not to get stuck. As a consequence, they can make more progress at home before the next meeting with a human teacher. In a self-study course, automatic feedback may be the only feedback students get.

An automatic assessment tool can check whether the answer is correct. This piece of information can be given to the student as a simple form of automatic feedback. If the answer is incorrect, the tool may give a link to a human-written fixed explanation of how to solve the problem correctly. Here "fixed" means that the explanation does not depend on what was the particular error that the student made (although it may depend on, for instance, how many times the student has tried). Automatic feedback of this kind was used in [1]. It discusses teachers' experiences with a mathematics education tool with various functionality (assessment, feedback, adaptation to what the student already knows, and so on).

However, in this paper we are interested in automatic feedback that in the case of a wrong answer, helps the student diagnose the problem *with that particular* answer. Furthermore, providing the feedback must not require extensive amounts of human work in advance.

Perhaps the best known tool for checking mathematical answers at the university level is STACK[1], that is, System for Teaching and Assessment using a Computer algebra Kernel [2]. It accepts answers in the form of expressions on real numbers. Because often the correct answer can be expressed in different but mathematically equivalent ways, STACK uses Maxima for comparing the student's answer to the model answer written by the teacher. STACK has recently [3] been linked with GeoGebra, which is a well-known interactive pedagogical tool focusing on probability and statistics, geometry, algebra

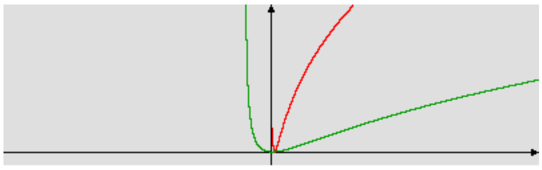
NWISEd 2025: Workshop on Co-Creating New Ways of Information Systems Education, September 10–11, 2025, Maribor, Slovenia
*Corresponding author.
ava@jyu.fi (A. Valmari)
https://users.jyu.fi/~ava/ (A. Valmari)
0000-0002-5022-1624 (A. Valmari)

[1]https://stack-assessment.org/

$$\log^2(a+b) = (\log(a+b))^2 = (\log a + b)^2$$

Relation does not hold when $a = 0$ and $b = 1$

left $=$ 0
right $=$ undefined



$-10 \leq a \leq 10$ and $-0.2 \leq \uparrow \leq 2.2$

Figure 1: An example of feedback by MathCheck to an arithmetic error (layout changed to save space)

and calculus. Another tool with emphasis on assessment of mathematics was discussed in [4]. It used Mathematica as the computational kernel.

STACK gives answer-specific feedback by checking a wrong answer against a set of likely wrong answers. In the words of [2]: "Feedback is provided to students using "potential response trees", written by the teacher for each question." Experienced teachers can often anticipate some such answers. When the problem is in use, commonly occurring wrong answers can be identified and added to the set. This approach makes very high-quality feedback possible. Unfortunately, it is laborious for the problem designer [3]: "the feedback feature of STACK needs a differentiated analysis of the mathematical concepts and procedures addressed in a problem, and of the variety of solving strategies that learners could follow." Furthermore, if the answer is outside this set, answer-specific feedback will not be provided [2]: "It is probably not worthwhile adding nodes to test for strange individual responses."

A different, pedagogically less ambitious but also much less laborious method of providing feedback is to provide a counter-example. Figure 1 shows the feedback provided by the arithmetic mode of our MathCheck tool [5] to `log^2(a+b) = (log(a+b))^2 = (log a+b)^2` . It tells that $\log^2(a+b) = (\log(a+b))^2$ is correct but $(\log(a+b))^2 = (\log a + b)^2$ is not, gives $a = 0$ and $b = 1$ as an example of a value combination of variables where the latter two yield different results, and shows a graph of both as a function of a when $b = 1$. The graphs are not always informative, but the student can always use at least the numeric counter-example as a starting point of diagnosing the error.

Surprisingly, although [6] is based on wide expertise and discusses feedback a lot, it does not mention providing counter-examples at all. Our guess is that it is not easy enough to make the computer algebra systems that check the answers to provide counter-examples. The arithmetic mode of MathCheck is not based on a computer algebra system. Instead, it tests each $=$, $\leq$ and so on by assigning a great number of combinations of values of variables to both sides and computing what they yield. In principle it may fail to see that two expressions are different, but in practice this is extremely rare (except in special cases, but then MathCheck warns that it was not able to be careful). It uses precise rational numbers as long as it can, and then switches to a representation consisting of two approximations, between which the precise value is. The latter representation is also vulnerable in principle, but not much in practice.

However, the topic of this paper is not the arithmetic mode but three predicate logic modes of MathCheck. In them, MathCheck applies methods that are unerring in principle. Nevertheless, the possibility of programming errors in MathCheck cannot of course be ruled out. The domains of discourse of the modes are real numbers, natural numbers and integers, restricted to expressions that are linear or "almost linear". MathCheck also has logic modes for propositional logic, modular arithmetic and arrays in the sense of programming languages, but they are not discussed in this paper. A major reason for our emphasis on logic and discrete mathematics is that they are more important to software engineering and computer science than calculus. For instance, [7, p. 186] has no hours of calculus in its most central core hours, while it has 29 hours of discrete mathematics including logic.

By linear expressions we mean those that can be easily reduced to the form $c_0 + c_1x_1 + \ldots + c_nx_n$, where the c_i are constants and the x_i are variables. By "almost linear" we refer to the fact that claims on other kinds of expressions can in some cases be reduced to Boolean combinations of claims on linear expressions. For instance, $3|x| \geq |x-3| + 5$ can be reduced to $x < 0 \land -3x \geq -(x-3) + 5 \lor 0 \leq x < 3 \land 3x \geq -(x-3) + 5 \lor x \geq 3 \land 3x \geq (x-3) + 5$, and $\frac{x+1}{x-1} = 2$ can be reduced to $x \neq 1 \land x + 1 = 2(x-1)$. The restriction to almost linear expressions is because they can be dealt with much more easily than more general expressions. Even so, versatile automatically assessable problems can be designed using them, as this paper tries to demonstrate.

Section 2 tells a little more about MathCheck in general. Sections 3 and 4 illustrate problems that can be dealt with the real logic and integer logic modes, respectively, with comments on the natural number logic mode. The implementation of the integer logic mode exploits concepts from basic courses on theoretical computer science and advanced courses on data structures and algorithms. It can thus serve as an example in teaching such topics. This is discussed in Section 5. A couple of suggestions for future improvements are made in Section 6.

2. About MathCheck in General

When developing MathCheck, our emphasis has been on supporting homework, and less in assessment in the sense of giving or not giving points. We want MathCheck to be useful both for developing routine in solving simple problems and practicing bigger problems that consist of many steps. We want MathCheck not only check and give feedback on the final answer, but also help finding it, by making it easy to write intermediate results and test them before continuing. The first examples in Section 3 illustrate what me mean by this. There the student first has to solve a number of simple formula writing problems, and then a much bigger problem where formula-writing skills must be applied many times and the pieces have to be put together in a not entirely trivial way.

The first version of MathCheck only had the arithmetic mode briefly illustrated in Figure 1. It lacked the graph drawing functionality exemplified by the right hand side of the figure. Terhi Kaarakka conducted an experiment [8, 5], where it was compared to Wolfram Alpha. While the latter is not a pedagogical tool, at that time students used it a lot when doing homework. The participants ($n = 106$) did a set of 10 problems using either MathCheck or Wolfram Alpha. Then they participated in a small examination whose maximum was 16 points. Those who had used MathCheck at least one hour ($n = 30$) got 9.7 points on the average, while users of Wolfram Alpha ($n = 31$) got 8.2. Those who were told to use MathCheck but used it less than 1 hour ($n = 26$) got 7.1, and the similar number for Wolfram Alpha ($n = 19$) was 7.2. The difference between the two MathCheck user groups was significant with $p = 0.02$. The paper [8] discusses also other early experiments.

The equation chain $\frac{x^2-7x+10}{x-5} = \frac{(x-2)(x-5)}{x-5} = x - 2$ is not correct, because when $x = 5$, the last expression yields 3 but the other two expressions are undefined. With $\frac{\sqrt{x}}{x-5}$ we have to assume that x is neither negative nor 5. This phenomenon forced us to implement a mechanism for stating assumptions containing some logic.

Computer science is full of undefined expressions, such as the value of a non-existing array entry (for instance, $A[-1]$ when the legal indices of A start from 0), and the result of a function subroutine when it fails to terminate. Intuition suggests, for instance, that neither $\frac{1}{0} < 0$ nor $\frac{1}{0} \geq 0$ should be true. But if we declare both of them false, then we lose the handy principle that $a \geq b$ means the same as $\neg(a < b)$. And what should be the truth value of $\frac{1}{0} = \frac{1}{0}$? This has been a big problem for the developers of formal software specification methods for decades, and many approaches have been suggested. Perhaps the most widely used is *underspecification* [9], where every expression always has a value, but nothing is told about the value when it is undefined in the mainstream mathematics sense. While this approach has unintuitive consequences (for instance, $x = 0$ is a root of $\frac{1+x}{x} + x = \frac{1}{2x}$), it can be used in practice.

We faced this problem when implementing the array claim and real logic modes of MathCheck. Unhappy with the oddities of earlier approaches, we developed our own based on three-valued logic. Then we learnt (to our great pleasure) that in the survey [10], the majority of participants considered the third truth value “undefined” more natural than other alternatives. To convince ourselves that our approach is on healthy basis, we proved that it is complete in the sense of Gödel’s famous completeness theorem. We succeeded in publishing this result in a high-quality journal on mathematical logic [11].

To develop the representation of reasonings used by the logic modes of MathCheck, we had to consider the difference between implication and equivalence as logical operators and as notation for expressing reasoning [12]. Because of cases like $\sin 2x = 2 \sin x \cos x$, developing the syntax for arithmetic expressions was a surprisingly big problem. We are aware of no other program than MathCheck that interprets $\sin 2x = 2\sin x \cos x$ correctly in the absence of additional parentheses.

Technically, MathCheck problem pages are ordinary web pages and the MathCheck program just inputs whatever those pages send to it and then sends something as a response. The MathCheck program does not store any information whatsoever. In addition to being able to deal with expressions, formulas and reasonings, the program also recognizes various commands that the web pages send to it. By writing those commands, teachers tell MathCheck what it should do. For instance, if the teacher wrote `f_polynomial;` `n(2n-1)(2n+1)=`, then MathCheck accepts $n(4n^2-1)$ as an intermediate but not as the final result, and accepts $4n^3-n$ as the final result.

This design has proven very flexible and easy to extend. Its perhaps biggest drawback is that each submit takes place in isolation from earlier submits. As a consequence, some problem pages ask the user to copy an answer from one answer box to another, so that it can be used as a component in the next stage of a multi-stage problem.

In 2017–2018 we wrote a problem authoring tool for MathCheck, but it quickly went out of date. All recent MathCheck problem pages have been hand-written. Many web browsers show the source code of the web page, if the user presses ⟨CTRL⟩-U. Using this feature and the link mentioned below, you may read the source code of all problems discussed in this paper. Feel free to do so, if you want! If MathCheck ever gains more problem designers, writing a new authoring tool becomes necessary.

Although the present author does not use MathCheck as an examination tool, two different mechanisms for that purpose have been implemented. First, MathCheck can be and has been used through a learning environment that keeps track of the identity of the students, sends their answers to MathCheck, and gets the numbers of points from the feedback by MathCheck. Second, MathCheck can also be compiled in an examination mode that does not check the semantics of the answer, but does check the syntax and that those format requirements are satisfied that the teacher has chosen. In the simplest version of this kind of use, the student can then email the answer to the teacher. In these uses, the student either cannot see the source code, or the correct answers are not there. (Because of the author's own experiences and those reported in [13], he believes that feedback should be given without points. Points belong only to the examination at the end.)

Depending on the problem, a correct solution may be explicit in the source code of the problem web page. At one point we added an obfuscation mechanism to prevent students from reading answers from the source code, but soon stopped using it. As was explained above, our philosophy is to use MathCheck only for learning, and not for distributing points. So keeping correct answers in great secrecy is not important to us. The examination version of MathCheck can be used such that no solution is present in the source code.

A weak point of our research is that no other well-organized pedagogical experiments have been made than those reported in [8]. As part of turbulent re-organization of the university where the author then worked, he was ordered to move from Mathematics department to Information Technology. The latter made it clear that developing educational tools has no place in its strategy.

The University of Jyväskylä offered more freedom of teaching and research, so he moved there in 2018. There he has used MathCheck in his own courses. Until Covid-19 MathCheck got some positive and very little negative feedback from students. Since then, it has been next to impossible to get feedback of any kind. Recently the teacher of "Introductory course to mathematics in engineering" started using MathCheck. Perhaps one quarter of the problems in the course are MathCheck problems. He likes especially problems of the kind in Figures 2 and 3, because they provide a natural environment for practicing set theory and logic. Unfortunately, neither of us has had sufficient time and skills to gather systematic experience, let alone design and organize pedagogical experiments.

The paper [2] discusses "updating STACK potential response tree based on students' concerns in learning induction proofs". As an example, it uses some tasks related to proving $\sum_{k=1}^{n}(2k-1)^2 = \frac{n(2n-1)(2n+1)}{3}$. A MathCheck problem page version of the same problem, with a comparison to [2], is at https://users.jyu.fi/~ava/induction_problem_AS.html. By reading [2] and reading and solving the problems on the web page, it is possible to get some idea of the differences between the two tools. However, one must take into account that while our web page covers the proof as a whole, the goal of

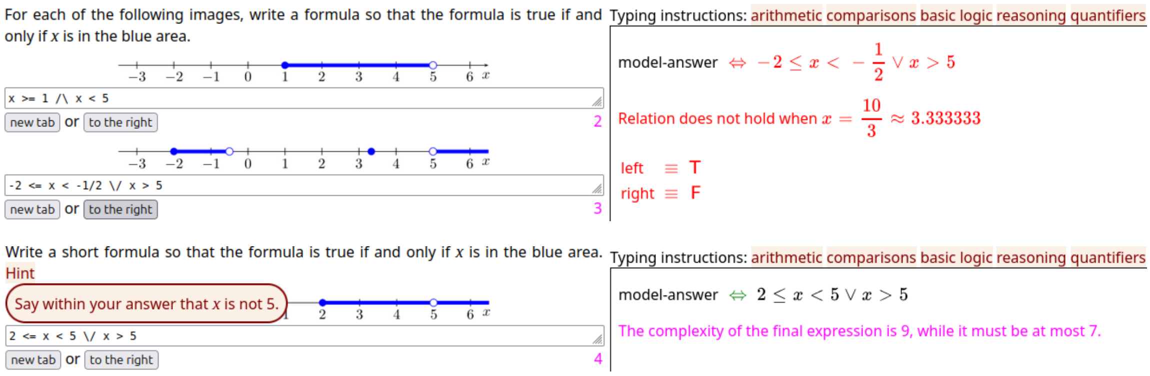


Figure 2: Formula writing problems on x-axis

the example in [2] is different. It is to illustrate how feedback by STACK can be improved based on students' answers.

3. "Almost Linear" Real Number Logic Problems

In this and the next section we discuss examples of logic problems. All of them are available at https://users.jyu.fi/~ava/number_logic.html (as long as the university lets them be there). No registration is required. The tool does not collect any information whatsoever about the user. Feel free to try! You may also try wrong answers, to see the feedback.

Figure 2 shows problems where the student should write formulas that correspond to points, line segments and half-lines on the x-axis. The student has solved the first problem correctly.

In the second problem, the student failed to take into account the isolated point. So MathCheck gave a counter-example to the answer. In this case, there is only one possible counter-example to choose from. By presenting it, MathCheck reveals the precise location of the isolated point. It may be considered a disadvantage. On the other hand, it is also helpful for those students who fail to read the precise location from the picture, but who are smart enough to try an answer where nothing represents that point. To an answer that contains the isolated point at a wrong location, the current version of MathCheck gives that location as a counter-example, thus not revealing the correct location. For instance, to $-2 \leq x < -\frac{1}{2} \vee x = 3\frac{1}{2} \vee x > 5$ it gives $x = \frac{7}{2} \approx 3.5$.

In general, instead of trying to prevent the students from misusing MathCheck, we encourage to think out of the box and intentionally try also incorrect answers to gain more understanding of the problem and answer. On the other hand, we remind the students that ultimately they will have to be able to solve problems without MathCheck. In the end of the courses by the author, there is a paper and pencil exam where neither MathCheck nor a calculator nor other tools are available.

In the third problem, MathCheck has declared the answer $2 \leq x < 5 \vee x > 5$ as mathematically correct but too long. The pedagogical goal is to make the student think about other possibilities of expressing the same claim. To help the student, there is a hint that becomes visible when the mouse pointer is moved on top of the brown word "Hint". The intended answer is $x \geq 2 \wedge x \neq 5$.

Figure 3 shows a problem where a formula must be written that represents a green-and-white picture of a cottage on x–y-coordinates. To make it easier to solve, auxiliary answer boxes are given where the student may design and check the wall, window, chimney and roof separately. The wall, roof and chimney are allowed to overlap. In particular, MathCheck checks the chimney by checking that $3 \leq x \leq \frac{7}{2} \wedge 6 - \frac{x}{2} < y \leq 5 \Rightarrow \text{chimney} \Rightarrow 3 \leq x \leq \frac{7}{2} \wedge \frac{5}{2} \leq y \leq 5$.

Writing a formula for the roof involves finding equations of the sloping lines. When combining the parts, the formula for the window must be used differently from the others. So the student must get many details right. The feedback by MathCheck helps in this. For instance, if the student writes a formula that corresponds to wall $\vee$ window $\vee$ chimney $\vee$ roof, MathCheck tells that when $y = 2$ and $x = 2$, the model answer yields false but the student's answer yields true. The point $(2, 2)$ is inside the

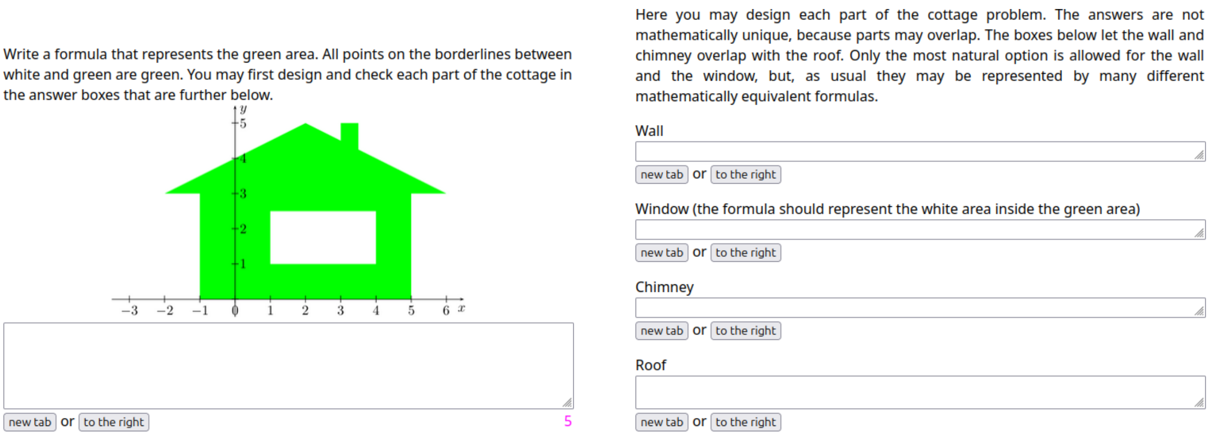


Figure 3: A formula writing problem about x-y–coordinates (layout changed to save space)

window, so the counter-example reveals that the window should have been negated. If the student then just negates the window without changing how it is connected to the rest of the formula, MathCheck gives a point outside the green area as a counter-example. In this way the student can fix the answer step by step helped by the feedback.

The teacher can write the message that tells that the reply is correct separately for each problem. For this problem group, the following messages were chosen: Yes, that is a correct wall!, You got the window right!, That is an okay chimney! You can use it as part of the final answer., You got the most difficult part of the cottage right, the roof! and The cottage is now correct. Congratulations!. Also messages for incorrect answers can be written problem by problem. Colours and fonts can be changed with usual World Wide Web commands (HTML and CSS).

The next problem is much easier. It starts by asking to write a formula that says that x is one of a and b. Then the student should write a formula saying that x is at most as big as a and b. The last task is to write, without using "min" or "max", a formula that says that x is the minimum of a and b. The first two questions encourage the student to write $(x = a \lor x = b) \land x \leq a \land x \leq b$. There are brown hints of the kind in Figure 2 that suggest combining the previous answers and adding parentheses around the $\lor$-subformula. Then the problem encourages more ambitious students to find a shorter formula. Via radio buttons, the student can choose between two maximum lengths. The answer mentioned above consists of 15 tokens (that is, basic elements; parentheses are not counted), but the shorter one is allowed to contain at most 11.

The logic modes of MathCheck actually check reasonings and not formulas. The checking is semantic. Thus no knowledge of any formal proof system is required. The operators $\Leftrightarrow$, $\Rightarrow$, $\Leftarrow$ and $\equiv$ can be used to relate formulas to each other, analogously to how $=$, $\geq$ and so on can be used to relate numbers to each other. (Material equivalence and implication, that is, equivalence and implication within formulas instead of between formulas, are expressed as $\leftrightarrow$ and $\rightarrow$.) The difference between $\Leftrightarrow$ and $\equiv$ is that the former treats the undefined truth value as equivalent to false. This is necessary to express roots of (in)equations. For instance, $\frac{x+1}{x-1} = 2 \Leftrightarrow x = 3$ is valid, but the same with $\equiv$ in the place of $\Leftrightarrow$ is invalid, because its left hand side is undefined but right hand side false when $x = 1$.

Problems where the student must write a formula are typically implemented as reasonings where a model formula and $\Leftrightarrow$ or $\equiv$ have been pre-programmed by the teacher together with a command which makes MathCheck write "model-answer" in the place of the teacher's formula. This implies that the student may develop the answer step by step in the answer box. For instance, MathCheck accepts $(x = a \lor x = b) \land x \leq a \land x \leq b \Leftrightarrow x = a \leq b \lor x = b \leq a$ as a correct answer to the above problem with the strict length requirement.

In some cases the model answer may be left visible. For instance, while the above problem uses the real logic mode, it could have used the integer logic mode. The latter has but the former currently lacks a `min` operator. However, the teacher may ban its use. Then, in the integer logic mode, `x = a <= b`

Here is a C++ implementation of Selection Sort. Let n = `A.size()`.

```
1  void SelectionSort( array_type & A ){
2    for( unsigned i = 0; i+1 < A.size(); ++i ){
3      unsigned p = i;
4      for( unsigned j = i+1; j < A.size(); ++j ){
5        if( A[j].x < A[p].x ){ p = j; }
6      }
7      elem_type tmp = A[i]; A[i] = A[p]; A[p] = tmp;
8    }
9  }
```

What is known on the value of *i* in the beginning of line 3?

`0 <= i < n-1  <=>  0 <= i <= n-2`

new tab or to the right

What is known on the values of *i*, *j* and *p* in the beginning of line 5?

new tab or to the right

What is known on the values of *i* and *p* in the beginning of line 7?

`0 <= i <= p < n`

new tab or to the right

Figure 4: A problem about the indices of an array

`\/ x = b <= a` elicits the feedback $x = \min(a,b) \Leftrightarrow x = a \leq b \lor x = b \leq a$ That is right!, but `x = min(a,b)` elicits $x = \min(a,b) \Leftrightarrow x = \min(a,b)$ The final expression must not contain min.

A pair of equations can be solved in many different ways: a variable may be solved from one equation and the solution assigned to the other, or the same with the other variable, or the equations may be multiplied by suitable numbers and then added in such a way that one variable disappears. To make it possible for the student to choose the solution path, MathCheck allows writing subproofs within a reasoning. The example web page of this paper contains a commented solution to $x+7y+3v+5u = 16 \land 8x + 4y + 6v + 2u = -16 \land 2x + 6y + 4v + 8u = 16 \land 5x + 3y + 7v + u = -16$. The problem is from Pólya's famous book [14]. This story has been told in [15], so we do not go into the details here.

The example web page also asks to solve $3|x| \geq |x-3| + 5$. The following solution illustrates three different ways of using subproofs. If `<=> x <= -4 \/ x >= 2` is removed, then MathCheck replies The final answer is not fully explicit: $x \geq 3 \land x \geq 1$, to tell the student to simplify the final answer further.

```
subproof x < 0 /\ -3x >= -x+3 + 5 <=> x < 0 /\ x <= -4 <=> x <= -4 subend
subproof assume 0 <= x < 3; 3x >= -x+3 + 5 <=> x >= 2 <=> 2 <= x < 3 subend
subproof 3x >= x-3 + 5 <=> x >= 1 subend
original <=> x <= -4 \/ 2 <= x < 3 \/ x >= 3 /\ x >= 1 <=> x <= -4 \/ x >= 2
```

4. "Almost Linear" Integer Logic Problems

While real number logic proved very useful, two reasons emerged to also implement integer logic.

First, in programming there are situations where numbers are definitely integers. Loop variables of `for`-loops, often used as indices to arrays, are an example. In the case of integers, $i < n$ means the same as $i + 1 \leq n$, but with real numbers it does not. Therefore, use of real number logic in such applications would unduly reject perfectly correct answers.

Figure 4 shows an example of such a problem. Both $0 \leq i < n-1$ and $0 \leq i \leq n-2$ are correct for the first question, and there is no reason to teach the students to favour one over the other. When in the integer logic mode, MathCheck accepts both as correct. Because MathCheck accepts not only final answers but also reasonings that lead to them, it even accepts the answer shown in the figure that presents both options. However, it rejects $0 \leq i < n-1 \Rightarrow 0 \leq i \leq n-2$, because it does not claim that $0 \leq i \leq n-2$ is a correct answer; it only claims that it is implied by the correct answer.

While first-order logic on integers or natural numbers with 0, 1, + and $\cdot$ is undecidable in the sense of computability theory, it becomes decidable if $\cdot$ is dropped. The result is known as *Presburger arithmetic* [16]. We implemented Presburger arithmetic for natural numbers in 2020, with applications of the above kind in mind. Unfortunately, because of the absence of negative integers, almost all teaching applications that we thought about seemed misleading. With natural numbers, $n > 0$ is equivalent to $n \neq 0$. Of course, we did not want MathCheck to encourage students to think of them as equivalent, because they are not equivalent in mainstream mathematics. Also subtraction of a bigger number from a smaller number could not work like in mainstream mathematics. So we did not implement subtraction at all.

It took until 2024 before we had the chance to implement Presburger arithmetic for integers. For this

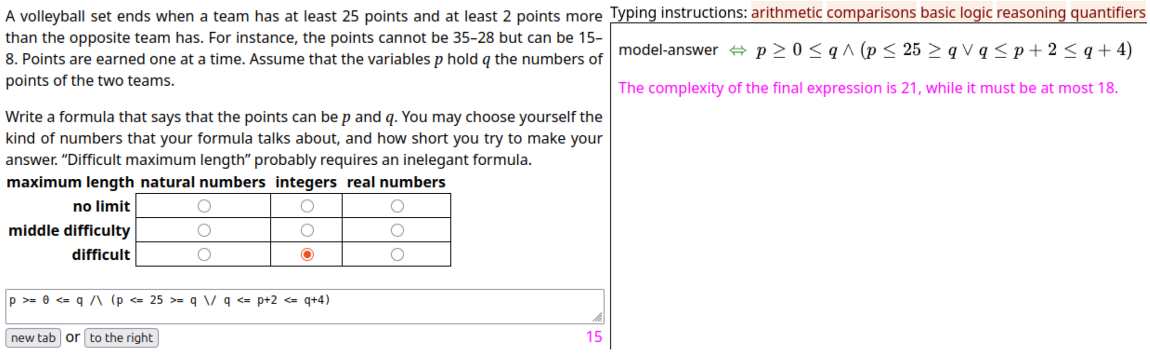


Figure 5: A volleyball points problem

discussion it is important to be aware that making either kind of Presburger arithmetic work fast is challenging [16]. Because of earlier experience with the natural number implementation, we were able to design some details better. With integers, it was reasonable to implement subtraction and absolute value. While writing this paper, we implemented “min” and “max”. We also found a reasonably simple and efficient way to implement subroutine-like subformulas that have a name and parameter list, and can be used as components of bigger formulas. For these reasons, the integer logic mode offers features that the natural number logic and real number logic modes do not offer, although they could.

Consider Figure 4 again. In https://en.wikipedia.org/wiki/Selection_sort#Implementations (accessed 2025-05-26), instead of the condition on line 2 there is (in essence) `i < A.size()-1`. It is correct there, because there `i` is of type `int`. However, `A.size()` is of some unsigned type in C++. This makes it natural (albeit not obligatory) to use an unsigned type for `i`. (Otherwise, depending on its settings, the compiler may give a warning of a type mismatch.) Then the algorithm fails with empty `A`, because `A.size()-1` becomes $0 - 1$, which yields a very big positive integer in the case of unsigned types. This causes the loop to continue past the end of the array. The difference between signed and unsigned types is thus important. As a consequence, it would be beneficial to bring the existence of the difference forward in the teaching of mathematics and logic for software students.

Figure 5 shows an example that serves this purpose. The student has to choose among natural numbers, integers and real numbers. This is hoped to increase awareness that they are different mathematical systems with different operations and properties (or, in computer jargon, different data types). The student can also choose a length limit so that more ambitious students are given a challenge while not frustrating less ambitious students with too difficult a problem. Of course, the student is free to try more than one option.

A straightforward lengthy answer using natural numbers is $p \leq 25 \wedge q \leq 25 \vee p \leq q + 2 \wedge q \leq p + 2$. To make it shorter, it may be tempting to modify its latter part to $q - 2 \leq p \leq q + 2$. However, MathCheck rejects it, because subtraction is not available in the natural number logic mode, because in the absence of negative numbers it is impossible to subtract a bigger number from a smaller one. On the other hand, $p \leq 25 \wedge q \leq 25 \vee q \leq p + 2 \leq q + 4$ works and meets the middle difficulty limit. To also meet the most difficult limit, $p \leq 25 \wedge q \leq 25$ can be replaced by $p \leq 25 \geq q$.

The formula $p \leq 25 \geq q \vee q \leq p + 2 \leq q + 4$ fails with integers, because it does not rule out negative points. MathCheck gives $p = -1$ and $q = 0$ as a counter-example. So we try $p \geq 0 \leq q \wedge (p \leq 25 \geq q \vee q \leq p + 2 \leq q + 4)$. It is mathematically correct and meets the middle limit. The most difficult limit is met by $p \geq 0 \leq q \wedge (p \leq 25 \geq q \vee |p - q| \leq 2)$. These answers work also with real numbers. Currently $0 \leq \min(p, q) \wedge (\max(p, q) \leq 25 \vee |p - q| \leq 2)$ works only with integers.

We emphasize to the students that the goal is not to learn to write $p \leq 25 \geq q$ instead of $p \leq 25 \wedge q \leq 25$. We recommend them to in general favour easily understandable formulas over shorter but trickier ones. We tell them that the purpose of asking to write short but tricky formulas is to develop their problem solving and logical thinking skills, by encouraging them to try and find alternative ways of formulating the same claim.

Our second reason for implementing Presburger arithmetic is the following. When implementing linear real number logic, we hoped also that it could be used for teaching quantifiers, that is, the symbols "$\forall$" and "$\exists$" that mean "for every" and "there is". But it failed. Any linear real number formula with quantifiers is equivalent to some linear real number formula without. (Actually, our implementation of quantifiers in the real logic mode is based on this fact.) For instance, "there is a positive number that is less than x" is equivalent to "$x > 0$". As a consequence, almost any problem we considered had an unintended mathematically equivalent but pedagogically meaningless answer. Quantifiers are used in a meaningful way in the definition of limits. Unfortunately, designing problems about limits would have mostly required that the logic can handle non-linear expressions.

The situation is better with Presburger arithmetic. That n is divisible by 7 can be expressed as $\exists k : n = 7k$. Also $n \bmod 7 = 0$ says so, but the teacher can rule it out by telling MathCheck to not allow "div" and "mod" in the final answer. In the web page containing the MathCheck examples of this paper, the model answer is $n \bmod 7 = 0$ and it is not hidden in the feedback. Therefore, if the student answers `EE k: n = 7k`, then MathCheck replies $n \bmod 7 = 0 \Leftrightarrow \exists k : n = 7k$ Correct! Your answer is excellent!. But if the answer is `n mod 7 = 0`, then MathCheck replies $n \bmod 7 = 0 \Leftrightarrow n \bmod 7 = 0$ The final expression must not contain mod .

In our experience, many students have problems with understanding the notion of free variables. (We do not know why.) To $n = 7k$ MathCheck gives the counter-example $n = 0$ and $k = -1$. The presence of k in it reveals that the error is that k has not been quantified. While this feedback is not straight to the point, at least it brings the issue forward.

A much more challenging problem asks to misuse MathCheck to find the smallest natural number that can be expressed as a sum of two cubes of natural numbers in two different ways. It is known as the second taxicab number. The mathematician Hardy had ridden in a taxi-cab with that number, to which his friend Ramanujan replied that it has the above-mentioned property. The student is told that to the extent needed in the problem, $C(x, y)$ yields true if and only if $y = x^3 \geq 0$; and other expressions may be written in the places of x and y. (Because of limitations of Presburger arithmetic, MathCheck cannot deal with cubes in general in this mode. Therefore, using the above-mentioned subformula mechanism, $C(n, m)$ has been defined as $n = m = 0 \vee n = m = 1 \vee n = 2 \wedge m = 8 \vee \ldots \vee n = 16 \wedge m = 4096$.)

The first task is to write a formula that says that x is the sum of the cubes of n and m. A straightforward answer is $\exists c : \exists d : C(n, c) \wedge C(m, d) \wedge x = c + d$, but also $\exists c : C(n, c) \wedge C(m, x - c)$ works.

The student is then told to copy the previous answer to the next box. That makes it possible to refer to it in answers to the next problem as $S(n, m, x)$. That problem asks to write a formula that says that x cannot be represented both as $n^3 + m^3$ and $k^3 + h^3$, where these two are different. The idea is that because it is not true for the x that we are after, MathCheck will give a counter-example that shows x, n, m, k and h. Therefore, by writing the right formula and clicking the button the student can fool MathCheck to reveal some solution to the taxicab number problem. A wrong formula either yields wrong numbers or no numbers at all. There are answer boxes where the student may test the numbers.

In general, a counter-example that MathCheck yields is not necessarily the smallest. The students are told that in this problem it is. However, to teach to test counter-examples in this respect, they are asked to find also the next natural number that can be represented as a sum of two non-negative cubes in two different ways, and test the corresponding numbers in the final answer boxes. A hint is available saying "Change the previous formula so that the original counter-example no longer is a counter-example."

If x does not occur free in P, then $\exists x : P \wedge Q(x) \Leftrightarrow P \wedge \exists x : Q(x)$. This law can be proven by letting $P \equiv \mathsf{F}$, simplifying both sides, detecting that they yield the same result, and then doing the same with $P \equiv \mathsf{T}$. The next problem on our example web page asks the student to do these simplifications. When checking the answers, MathCheck uses a $Q(x)$ designed by the teacher so that it depends not only on x, but also on another variable q in such a way that the answer becomes tested with F, T, $x < 1$ and $x \geq 1$ in the place of $Q(x)$. Not all of this is told to the students, but they are told that some trickery has been used and the values of x and q in the feedback are useless to them. So this problem is not fully elegant. Nevertheless, it is a meaningful problem about an advanced topic, namely proving quantifier laws. This problem does not require integers, but does require the subformula mechanism that is currently not available in any other mode of MathCheck.

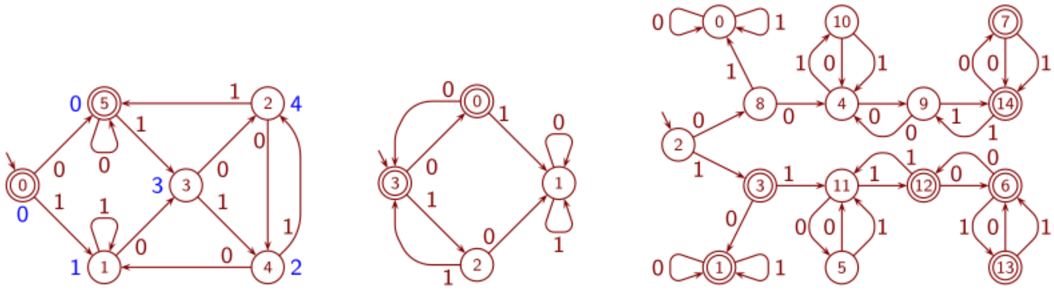


Figure 6: A DFA that represents the formula (left) $x \bmod 5 = 0$ (middle) $x = y$ (right) $x < y$ about integers

A related advanced skill is disproving alleged quantifier laws by providing counter-examples. So we designed a problem that asks students to write $P(x)$ and $Q(x)$ such that $\forall x : (P(x) \lor Q(x)) \Leftrightarrow (\forall x : P(x)) \lor (\forall x : Q(x))$ does not hold. One way of finding a correct answer is as follows. If $\forall x : P(x)$ is true, then it is simple to see that also $\forall x : (P(x) \lor Q(x))$ is true, and similarly if $\forall x : Q(x)$ is true. Thus $P(x)$ and $Q(x)$ must be chosen so that neither is true for all integers, but for any integer, at least one of them is true. There are numerous possibilities: $x > 2$ and $x < 5$, $x = 0$ and $x \neq 0$ and so on.

5. The Implementation as a Source of Computer Science Examples

The implementation of the integer Presburger arithmetic utilizes many basic and advanced ideas of computer science: binary representation of integers, the relationship between sets and the corresponding formulas, deterministic finite automata, and certain efficient advanced algoritms and data structures. It can thus serve as an example that illustrates many concepts. It is widely believed that real-life applications help students get motivated. Unfortunately, real-life applications of more theoretical topics are often hard to find. While the integer logic mode of MathCheck is perhaps not quite real life, for our students it is at least student life.

The implementation represents a formula as a list of variables together with a certain kind of edge-labelled directed graph called *deterministic finite automaton* or *DFA*. The list must be in strictly increasing order and it must contain at least the free variables of the formula.

Assuming that the list consist of just x, the DFA on the left hand side of Figure 6 represents the formula $x \bmod 5 = 0$. The circles are *states* and the arrows are *transitions*. One of the states is designated as the *initial state* with a short arrow that ends at it and starts at no state. Each state is the tail state of precisely two transitions, one labelled with 0 and the other with 1. The DFA reads a sequence of 0's and/or 1's by starting at the initial state and then, for each bit in the sequence in turn, choosing the next transition according to the next bit in the sequence. If the reading ends at a doubly circled state, then the DFA *accepts* the sequence, and otherwise *rejects* it. For instance, our example DFA accepts the empty sequence, 0 and 0101, but rejects 01. Doubly circled states are *final states*.

Each sequence of bits represents an integer. The integer is negative if and only if the first bit is 1. The empty sequence represents 0, and the sequence 1 represents -1. For $i > 1$, if the ith bit of the sequence is 1, then it adds (if the first bit is 0) or subtracts (if the first bit is 1) 2^{i-2} to the integer. The representation is thus otherwise standard, but the sign bit is at the beginning, the other bits are least significant first, there is no limit to the number of bits in the sequence, and the other than the first bits of a negative integer are opposite to what they are in two's complement. The absence of the limit is crucial because unlike typical computer integers, there is no maximal integer in mathematics. The reasons for the other three deviations will be explained below.

Our example DFA accepts precisely those bit strings which represent the integers $\ldots, -10, -5, 0, 5, 10, \ldots$. Those students who are strong in modular arithmetic and invariant proofs can verify this fact by checking that for each state with blue number k in the figure we have $(k2^i) \bmod 5 = |x| \bmod 5$, where i is the number of bits that have been read after the sign bit. (The brown numbers inside the states were chosen by MathCheck when it constructed the DFA. They have no special significance.) At the opposite end, some of our students have difficulties in distinguishing between an integer and a set of integers, and with the fact that a set may be infinite even if all of its elements are finite. It is hoped that seeing a rigorous finite representation of an infinite set of integers helps them.

Assume that the representation of x as a bit string is $x_\pm x_0 x_1 \cdots x_n$, and similarly with y and z. Then $x_\pm y_\pm z_\pm x_0 y_0 z_0 x_1 y_1 z_1 \cdots z_n$ represents the triple (x, y, z). To make this work also if the original bit strings are of different lengths, the DFAs representing MathCheck integer logic formulas have the special property that if the state on one end of any 0-transition is a final state, then also the state on the opposite end is a final state. As a consequence, any bit string extended with 0 represents the same value combination as the bit string without the extra 0. If two's complement were used for negative integers, then extending the bit string without changing the represented value combination would require 0 or 1 depending on the sign of the corresponding integer. It would be much more complicated.

Figure 6 (middle) and (right) show DFAs that represent $x = y$ and $x < y$. For instance, the self-loops of the state number 1 in the latter reflect the fact that if x is negative (the transition $2 \xrightarrow{1} 3$) but y is not ($3 \xrightarrow{0} 1$), then no matter what the remaining bits are, we have $x < y$.

A DFA for $x + y = z$ can be constructed that mimics how natural numbers are added at elementary schools. It adds the least significant bits first, then the second least significant and so on, and uses a so-called "carry" when the result at some position affects the next position. For this to work, it is essential that the bits come least significant first. The sign bit is at the front so that it is easy to tell which bit is the sign bit. The DFA for $x + y = z$ has 33 states. The link https://users.jyu.fi/~ava/number_logic.html#printDFA leads to a box where you can write an integer Presburger expression or formula, and ask MathCheck to construct a corresponding DFA and present it in list form.

There is no DFA for multiplication with this representation of triples of integers because, roughly speaking, the amount of information that has to be remembered grows without limit when moving from less significant to more significant bits. (In the case of addition the carry suffices, and it is just one bit of information.) One of the important facts in the theory of DFAs is called "pumping lemma". If it belongs to the curriculum, then here we have an excellent application illustrating it, because it makes the rough argument above precise. Changing the representation would not help, because of Gödel's famous incompleteness theorem. Namely, if both addition and multiplication were representable, then we would have a decision algorithm for integer arithmetic of the kind that the theorem rules out.

The implementation of "$\neg$" is trivial: just convert every final state to non-final and vice versa. The implementation of "$\wedge$", "$\vee$", "$\rightarrow$" and "$\leftrightarrow$" is based on a construction known as "product", with additional functionality to deal with the facts that the lists of the variables of the participating DFAs may differ from each other, and the same original transition may correspond to more than variable. Its implementation in MathCheck consists of about 150 lines of code. It performs a breadth-first search on the implicitly represented product DFA, constructing an explicit representation for those of its states and transitions that it encounters. It contains a hash table, memory-efficient packing of information using modular arithmetic, a generic implementation of binary Boolean operators (it actually implements all the 16 distinct binary Boolean functions), and some speed-up heuristics. So it can be used to illustrate non-trivial application of many programming and algorithm topics.

The product automaton may be much bigger than the smallest possible DFA for the same set of bit strings, even if the original DFAs are smallest possible. Fortunately, there is a fast algorithm for minimizing DFAs [17] with improvements [18]. Its implementation (together with backwards-propagation of final state status along 0-transitions) consists of about 180 lines.

Reasonably efficient implementation of the quantifiers is a challenge. Results from computational complexity theory [19, 16] imply that there cannot be a solution that is never woefully slow. If an implementation of "$\exists$" is available, then "$\forall$" can be implemented as "$\neg\exists\neg$". The former can be implemented by using the so-called subset construction. Transitions that correspond to the quantified variable are treated as ε-transitions in the sense of nondeterministic finite automata. As a consequence, the state of the result DFA may correspond to more than one state of the input DFA. The result DFA is constructed similarly to the product DFA, but in this case the constructed states are subsets of the set of the original states. Therefore, a good data structure for dealing with subsets is needed. MathCheck uses first an ordered sequence, but switches to a bit set when that becomes equally memory-efficient. This data structure uses about 150 lines of code, and the rest of the implementation of "∃" 90 lines.

Altogether the integer Presburger module consists of about 1500 lines of code.

6. A Couple of Concluding Remarks

When encountering a syntax error, MathCheck typically lists at most 30 tokens that would be correct in that place. At the time of designing the parser of MathCheck, this was believed to be a good idea. But it eventually became a nuisance. Upon arrival of new modes with slightly different syntax, it became better to give error messages of the kind "div is not available in real logic mode".

While the possibility of issuing various format checking commands separately is flexible, it is also prone to the risk that the teacher forgets a command that should be there and writes another command that should be not. The technical rules of the World Wide Web dictate that each answer box has an internal name. This name is currently irrelevant for MathCheck, except that if it is "exam", then MathCheck rejects many commands, including those that are intended to be written only by teachers. Using more names, each with its own default setting of format checking commands, writing commonly occurring combinations of commands could be made easier for teachers.

Acknowledgments

The author thanks the reviewers for their efforts and constructive feedback, and Roope Vehkalahti for all the discussions we have had on teaching mathematics and logic, with or without MathCheck.

Declaration on Generative AI

The author has not employed any Generative AI tools beyond minor English language checking.

References

[1] T. Kumor, L. Uribe-Flórez, J. Trespalacios, D. Yang, Aleks in high school mathematics classrooms: Exploring teachers' perceptions and use of this tool, TechTrends 68 (2024) 506–519. doi:10.1007/s11528-024-00955-0.

[2] M. Alarfaj, C. J. Sangwin, Updating STACK potential response trees based on separated concerns, Int. J. Emerg. Technol. Learn. 17 (2022) 94–102. doi:10.3991/IJET.V17I23.35929.

[3] G. Pinkernell, J. M. Diego-Mantecón, Z. Lavicza, C. J. Sangwin, Authomath: Combining the strengths of STACK and geogebra for school and academic mathematics, Int. J. Emerg. Technol. Learn. 18 (2023) 201–204. doi:10.3991/IJET.V18I03.36535.

[4] P. Kajetanowicz, J. Wierzejewski, Application of computer algebra systems in automatic assessment of math skills, Teaching Mathematics and Computer Science 6 (2008) 395–408. doi:10.5485/TMCS.2008.0187.

[5] A. Valmari, J. Rantala, Arithmetic, logic, syntax and MathCheck, in: Proceedings of the 11th International Conference on Computer Supported Education, CSEDU 2019, Heraklion, Crete, Greece, May 2-4, 2019, Volume 2, SciTePress, 2019, pp. 292–299. doi:10.5220/0007708902920299.

[6] G. Kinnear, I. Jones, C. Sangwin, M. Alarfaj, B. Davies, S. Fearn, C. Foster, A. Heck, K. Henderson, T. Hunt, P. Iannone, I. Kontorovich, N. Larson, T. Lowe, J. C. Meyer, A. O'Shea, P. Rowlett, I. Sikurajapathi, T. Wong, A collaboratively-derived research agenda for e-assessment in undergraduate mathematics, International Journal of Research in Undergraduate Mathematics Education 10 (2024) 201–231. doi:10.1007/s40753-022-00189-6.

[7] The Joint Task Force on Computer Science Curricula: Association for Computing Machinery (ACM), IEEE-Computer Society (IEEE-CS), Association for the Advancement of Artificial Intelligence

(AAAI), Computer Science Curricula 2023, ACM Press, IEEE Computer Society Press and AAAI Press, 2024. doi:10.1145/3664191.

[8] T. Kaarakka, K. Helkala, A. Valmari, M. Joutsenlahti, Pedagogical experiments with MathCheck in university teaching, LUMAT: International Journal on Math, Science and Technology Education 7 (2019) 84–112. doi:10.31129/LUMAT.7.3.428.

[9] D. Gries, F. B. Schneider, Avoiding the undefined by underspecification, in: J. van Leeuwen (Ed.), Computer Science Today: Recent Trends and Developments, volume 1000 of *Lecture Notes in Computer Science*, Springer, 1995, pp. 366–373. doi:10.1007/BFB0015254.

[10] P. Chalin, Logical foundations of program assertions: What do practitioners want?, in: Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany, IEEE Computer Society, 2005, pp. 383–393. doi:10.1109/SEFM.2005.26.

[11] A. Valmari, L. Hella, A completeness proof for a regular predicate logic with undefined truth value, Notre Dame Journal of Formal Logic 64 (2023) 61–93. doi:10.1215/00294527-2022-0034.

[12] A. Valmari, Adapting formal logic for everyday mathematics, in: Proceedings of the 14th International Conference on Computer Supported Education, CSEDU 2022, Online Streaming, April 22-24, 2022, Volume 2, SCITEPRESS, 2022, pp. 515–524. doi:10.5220/0011063300003182.

[13] G. Gibbs, Using Assessment to Support Student Learning, Technical Report, Leeds Metropolitan University, 2010. URL: http://eprints.leedsbeckett.ac.uk/2835/1/100317_36641_Formative_Assessment3Blue_WEB.pdf.

[14] G. Pólya, How to Solve It, Princeton University Press, 1945.

[15] A. Valmari, Automated checking of flexible mathematical reasoning in the case of systems of (in)equations and the absolute value operator, in: Proceedings of the 13th International Conference on Computer Supported Education, CSEDU 2021, Online Streaming, April 23-25, 2021, Volume 2, SCITEPRESS, 2021, pp. 324–331. doi:10.5220/0010493103240331.

[16] C. Haase, A survival guide to Presburger arithmetic, ACM SIGLOG News 5 (2018) 67–82. doi:10.1145/3242953.3242964.

[17] J. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971), New York: Academic Press, 1971, pp. 189–196.

[18] A. Valmari, Fast brief practical DFA minimization, Information Processing Letters 112 (2012) 213–217. doi:10.1016/j.ipl.2011.12.004.

[19] M. J. Fischer, M. O. Rabin, Super-exponential complexity of Presburger arithmetic, in: R. M. Karp (Ed.), Complexity of computation. SIAM-AMS Proceedings, volume 7, American Mathematical Society, 1974, pp. 27–41.