



An Implementation of Integer Presburger Arithmetic in a Math Education Tool

Antti Valmari¹ and Mika Hirvensalo²

¹ University of Jyväskylä, Faculty of Information Technology
PO Box 35, FI-40014 University of Jyväskylä, FINLAND

Antti.Valmari@jyu.fi

² University of Turku, Department of Mathematics and Statistics
Vesilinnantie 5, FI-20500, Turku, FINLAND
mikhirve@utu.fi

Abstract

An automata-theoretic implementation of a first-order theory of integers is discussed. It has addition, absolute values, minimum and maximum, and some bitwise operations, but lacks other multiplication than multiplication by a constant. The implementation is used in a tool that gives first- and second-year university students feedback on their answers to mathematics and computer science problems. The focus of this paper is on features of the tool supported by the implementation, and design and programming details that make it run sufficiently fast.

1 Introduction

This paper discusses an automata-theoretic implementation of integer Presburger arithmetic in a tool for giving students feedback on their answers to certain kinds of mathematics and computer science problems. As pointed out in [9], one of the greatest challenges in mathematics education tools is to accept reasonings or proofs as input and give meaningful feedback on errors in them. In [13], we reported a tool that checks reasonings that solve systems of linear (in)equations that may also contain the absolute value operator. That tool works on real numbers. The reasonings are represented using $\Rightarrow$ and $\Leftrightarrow$ not as logical connectives, but as symbols for comparing formulas. Our use of $\Rightarrow$ and $\Leftrightarrow$, and problems in other uses of them found in the literature, are extensively discussed in [16].

On integers, but not on real numbers, $x < y$ is equivalent to $x + 1 \leq y$. As a consequence, the above-mentioned tool proved not good for checking answers to questions like “what values can the integer variable i have on this line of this program”. Therefore, we expanded our tool with the ability to deal with integer Presburger arithmetic [14].

The worst-case complexity of checking satisfiability of Presburger arithmetic formulas is high [3, 6, 10]. Fortunately, like with many other problems of high complexity, implementations can be made that run reasonably fast much of the time in practice. Perhaps the most important approaches rely on integer linear programming or finite automata algorithms. Their basic ideas

have been described in [2, 5]. The automata-theoretic approach makes it easy to extend the logic with a predicate for powers of 2 and function symbols for bit manipulation operations. This extended theory (where the base may be any integer $k \geq 2$) is called Büchi arithmetic. In this paper it is exploited for checking reasonings on uninterpreted unary predicates, such as $\neg(P(x) \vee \forall x : Q(x) \vee R(x)) \Leftrightarrow \neg P(x) \wedge \neg \forall x : Q(x) \vee R(x) \Leftrightarrow \neg P(x) \wedge \exists x : \neg Q(x) \vee R(x)$.

The discussions on the automata-theoretic approach in [2, 5] only deal with non-negative integers, while in the present paper also negative integers are included. The paper [1] goes even further, allowing the representation of many real numbers using Büchi automata. Furthermore, in [2, 5], the label of each transition of each automaton is a tuple of bits. Instead, [1] uses what they call “serial encoding”. It simplifies implementation and facilitates some optimization. The present paper uses a variant of it called “single-bit”. For reasons explained in Section 4.1, in our case, the sign bit is first and then the numeric bits starting with the least significant. Furthermore, negative integers n are represented by making the sign bit be 1 and the subsequent bits represent $-1 - n$. These conventions avoid some complications that are present in [1].

Another related, much more ambitious tool than ours is Mona [11]. It implements an automata-theoretic decision procedure for the second-order logic WS1S, whose first-order variables represent natural numbers and second-order variables represent finite sets of natural numbers. Arithmetic is restricted to what can be implemented given the successor function and comparisons. This is sufficient for Presburger arithmetic but not for multiplication of natural number variables. Mona also implements a decision procedure for a similar logic W2S2 that features two successors. Mona began as a slow first version, but its performance has improved dramatically over the years by adopting various design and implementation ideas.

Instead of the tuple, serial and single-bit approaches discussed above, Mona uses binary decision diagrams (BDD) to represent the transitions of the automata. The paper [11] reports this as essential for Mona’s good performance. We will argue in Section 4.1, in agreement with [1], that the BDD and serial approaches are more similar than their names suggest.

Because of the nature of our application, our implementation need not survive very long formulas nor integers that are far from 0. Even so, we tried to make it efficient. The running time depends so wildly on the nature of the input that it is very difficult to give a comprehensive picture. Figure 6 in Section 5 will give some idea on this dependence and on the effect of some optimizations.

The tool that the new program is part of, is briefly introduced in Section 2.1. Section 2.2 outlines Presburger arithmetic and how it can be implemented using finite automata. Section 3 describes our implementation from the users’ point of view. Design-level ideas that our implementation relies on are discussed in Section 4, and Section 5 moves on to algorithm and programming issues. Most of the examples in this paper and many others can be tried anonymously at https://users.jyu.fi/~ava/number_logic.html. More detailed discussion on some examples can be found in [14].

2 Background

2.1 The MathCheck Tool

Our tool, which we call MathCheck, is a program that runs on a web server and is interacted with via web pages. Each web page contains one or more tasks for the student. Figure 1 shows an example where the student has answered the second question incorrectly. The tool has replied that when $x = \frac{10}{3}$, the model answer written by the teacher yields true, while the student’s answer yields false.

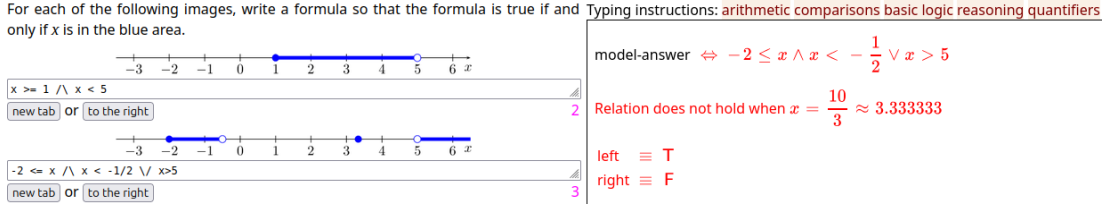


Figure 1: Two line segment problems

In itself, the tool does not know the identity of the student and does not keep track of points or record any other information. However, it can be used through learning environments that take care of such tasks. The tool only receives a sequence of commands, expressions, formulas and/or reasonings from the web page, checks the expressions, etc., as commanded, and sends back a reply. The answer by the student is part of the sequence. The rest has been written by the teacher, usually in such a way that the browser does not show it (e.g., “style=display:none”).

In the case of Figure 1, the teacher wrote `real_logic ok_text The second is right!` and `f_nodes 21 hide_expr -2 ≤ x < -1/2 ∨ x = 3 1/3 ∨ x > 5 ⇔`. It means that the tool has to check the answer in the real number logic mode of the tool, and reply “The second is right!” if the answer passes all checks. If the answer is longer than 21 symbols (not counting parentheses) but otherwise correct, the tool must reply, for instance, “The complexity of the final expression is 22, while it must be at most 21”. Because of `hide_expr`, the reply contains “model-answer” instead of the teacher’s formula. The symbols “ $\leq$ ” and “ $\Leftrightarrow$ ” were written using Unicode characters, but could have been written as `<=` and `<=>`.

What the teacher has written, is only lightly hidden from the student. In many browsers, (CTRL)-U shows the source code of the web page. Interested readers of this paper are welcome to apply this to the example web page of this paper! On the other hand, during an examination, students must be prevented from seeing model answers. A natural solution is to use the tool through a learning environment that isolates the student’s browser from the problem source code, and also keeps track of the identity of the student and points earned.

The real number logic mode checks reasonings on linear formulas using Fourier–Motzkin elimination. The most important other modes of the tool are the integer Presburger mode discussed in [14] and this paper; an arithmetic mode for checking equality and inequality chains on real numbers and integers by incomplete testing; a mode for checking solutions (including how they are found) to equations on real numbers based on teacher-given roots; a mode for checking claims on arrays by incomplete testing; and a mode dealing with context-free grammars. The (in)equality chain and equation modes can deal with logarithms and trigonometric functions.

The tool accepts Unicode versions of many symbols. Therefore, expressions, formulas and reasonings can often be copied from PDF files or web pages and pasted to the answer boxes, and they work with little or no fixing.

In a pedagogical experiment, studying using an early version of the tool improved learning results ($p = 0.02$) while studying using Wolfram Alpha (which students used a lot at that time) did not [8, 17]. Then there was a re-organization of the university. Valmari was moved from the Department of Mathematics to a different Faculty, and told that developing pedagogical tools does not belong to the strategy of that Faculty. The University of Jyväskylä gave Valmari the chance to continue the development of the tool and use it in his courses. Also one or two mathematics teachers use it. However, it has not been possible to conduct pedagogical experiments there. Until Covid19, students provided feedback that they like the tool and

believe it is useful. Since Covid19, it has been next to impossible to get any feedback.

2.2 Presburger Arithmetic and Automata-Theoretic Implementation

Presburger arithmetic may be defined over natural numbers or integers. In both cases it means a logical language, in which claims on numbers can be expressed as logical formulas that, in a minimalistic version, may contain $\neg$, $\wedge$, $\forall$, $=$, $\leq$, 0 , 1 and $+$. Practice-oriented versions may contain many other function and predicate symbols that can be constructed from the minimalistic set. Perhaps the most important are integer constants in decimal (such as “2026”) and multiplication of a variable by an integer constant. Our symbols are discussed in Section 3.

Presburger arithmetic has important applications, because there are algorithms that input an arbitrary Presburger arithmetic formula and provide an assignment to its free variables that makes it true, or tell that such assignments do not exist. The computational complexity of this problem — and thus all algorithms that fully solve it — is high. Even so, some of the algorithms can be made to run reasonably fast in many practical cases.

It is perhaps worth pointing out that although the representation of an integer constant in the form $1+1+\dots+1$ is exponentially longer than in the familiar decimal system, this difference is not important for the computational complexity of the above-mentioned problem [10]. This is because any formula of the form $\varphi(c)$, where c is an integer constant greater than 1, is logically equivalent to $\exists x_k : \varphi(x_k) \wedge \exists x_{k-1} : x_k = x_{k-1} + x_{k-1} + b_0 \wedge \dots \wedge \exists x_1 : x_2 = x_1 + x_1 + b_{k-2} \wedge x_1 = b_{k-1}$, where each b_i is 0 or 1 according to the binary representation $\sum_{i=0}^{k-1} b_i 2^i$ of c . As a consequence, c can be represented succinctly even in the absence of other integer literals than 0 and 1.

An *alphabet* is a finite non-empty set of symbols and a *string* is a finite sequence of symbols picked from the alphabet. For instance, the decimal system represents non-negative integers such as “2026” as strings over the alphabet $\{0, 1, 2, \dots, 9\}$. Any finite, and many but not all countably infinite, sets of strings can be represented by a *deterministic finite automaton (DFA)*. It is a tuple $(Q, \Sigma, \delta, \hat{q}, F)$, where Q is a finite set, Σ is an alphabet, δ is a function from $Q \times \Sigma$ to Q , $\hat{q} \in Q$ and $F \subseteq Q$. The elements of Q , δ and F are called *states*, *transitions* and *final states*, and $\hat{q}$ is the *initial state*. Given a DFA and a string $a_1 a_2 \dots a_n$ over the same alphabet, one may start at $q_0 = \hat{q}$, move to $q_1 = \delta(q_0, a_1)$, and continue like this until reaching $q_n = \delta(q_{n-1}, a_n)$. If $q_n \in F$ then the DFA *accepts* and otherwise *rejects* the string.

When δ is clear from the context, $\delta(q, a) = q'$ may also be written as $q \xrightarrow{a} q'$.

Also tuples (or vectors) of integers can be encoded as strings. Here implementations of Presburger arithmetic differ significantly. Let V be the size of the tuple. As described in detail in Section 4.1, we use an encoding where (after an initial segment which we ignore in this section) a_{i+1} is the $((i \div V) + 1)$:th bit of the $((i \bmod V) + 1)$:th integer. A formula with V free variables can be represented as the set of V -tuples that satisfy it, which, in turn, is represented as a DFA that accepts precisely the strings that encode the tuples.

The formula $a + b = c$ can be represented as $\{(a, b, c) \mid a + b = c\}$. With a suitable encoding, this set of triples can be represented as a DFA. This is because addition can be computed from least significant to most significant digits so that in each step, only the current digits and the carry are needed. Multiplication needs several scans over the digits of one of the numbers, denying any obvious representation as a DFA. As a consequence of Gödel’s famous first incompleteness theorem, there are also no non-obvious representations.

For each DFA, there is a unique (up to the names of states) smallest DFA accepting the same set of strings. Computing it is called *minimization*. As discussed in Section 5.3, minimization is so fast that it makes sense to perform it always when it is not certain in advance that the

DFA already is minimal.

Negation may be represented by replacing F with $Q \setminus F$.

Conjunction and disjunction may be computed as products of DFAs. Because of the use of the single-bit approach discussed in Section 4.1, the state of a minimized DFA does not necessarily determine the variable whose turn is next. Therefore, the index of this variable must be kept track of in the state of the product.

More formally, let $\mathbb{V} = \{0, 1, \dots, V-1\}$ be the set of the indices of the free variables in the formula, and for $v \in \mathbb{V}$, let $v' = (v+1) \bmod V$. The product of $(Q_1, \Sigma, \delta_1, \hat{q}_1, F_1)$ and $(Q_2, \Sigma, \delta_2, \hat{q}_2, F_2)$ is the DFA $(Q, \Sigma, \delta, \hat{q}, F)$, where $\hat{q} = (0, \hat{q}_1, \hat{q}_2)$, $F = (\mathbb{V} \times F_1 \times F_2) \cap Q$ for conjunction and $F = ((\mathbb{V} \times F_1 \times Q_2) \cup (\mathbb{V} \times Q_1 \times F_2)) \cap Q$ for disjunction, and Q and δ are the smallest sets such that $\hat{q} \in Q$ and for every $(v, q_1, q_2) \in Q$, also $\delta((v, q_1, q_2), a) = (v', \delta(q_1, a), \delta(q_2, a)) \in Q$. In Section 5.4 we slightly optimize this and adapt it to the situation where the original DFAs do not necessarily have the same alphabet.

When there is at most one free variable, the computation of $\exists$ is simple. In the remaining case, the basic idea comes from the well-know method of making a DFA from a nondeterministic finite automaton (NFA): keep track of the set of states that the NFA could be in after reading a string. Similarly to the computation of the product, also the variable whose turn is next must be kept track of. Whenever the quantified variable would be getting the next turn, the NFA skips it by taking an arbitrary transition without reading anything from the input.

To describe this more formally, let $v' = (v+1) \bmod V$. Let w be the index of the quantified variable. If $(Q, \Sigma, \delta, \hat{q}, F)$ represents φ , then $\exists x_w : \varphi$ is represented by $(Q', \Sigma, \delta', \hat{q}', F')$ as follows. The elements of Q' are pairs (v, A) , where $w \neq v \in \mathbb{V}$ and $A \subseteq Q$. If $w \neq 0$, then $\hat{q}' = (0, \{\hat{q}\})$ and else $\hat{q}' = (1, A')$, where $A' = \{q' \mid \exists b \in \Sigma : q' = \delta(\hat{q}, b)\}$. Let $(v, A) \in Q'$. If $v' \neq w$, then $\delta'((v, A), a) = (v', A') \in Q'$, where $A' = \{q' \mid \exists q \in A : q' = \delta(q, a)\}$. If $v' = w$, then $\delta'((v, A), a) = (w', A') \in Q'$, where $A' = \{q' \mid \exists q \in A : \exists b \in \Sigma : q' = \delta(\delta(q, a), b)\}$. Finally, $F' = \{(v, A) \in Q' \mid A \cap F \neq \emptyset\}$.

The sets A may be of various sizes, and there may be many of them. Therefore, this operation is the most challenging to implement. In Section 5.5 we tackle this problem.

The formula $\forall x : \varphi$ is represented via $\neg \exists x : \neg \varphi$.

3 The Integer Presburger Language and Its Use

3.1 An Approximate Grammar

In the subsections of this section we introduce the language via which the teachers and students use the tool. The grammar in Figure 2 has been designed to give, in succinct form, sufficient information for using the language. Ordinary parentheses $()$ are terminal symbols and bold green parentheses **$()$** belong to the representation of the grammar. Optional parts are inside bold green square brackets **$[]$** . Alternatives are separated by **$|$** . Repetition at least zero times is expressed by **$*$** and at least once by **$^+$** .

Succinctness was obtained at the cost of losing some precision. For instance, a $\langle nat_number \rangle$ can sometimes also be given as a constant expression such as $3 \cdot 2 + 1$, although Figure 2 does not say so. As another example, here and there additional parentheses are required for clarity. Therefore, $\neg x < 3$ does not work, but $\neg(x < 3)$ does. The differences between the grammar and the reality should not be a big problem, because the tool gives error messages as appropriate.

We will discuss the non-obvious details of the grammar in subsections 3.2 to 3.5, except $\langle print_part \rangle$. The latter prints in a textual form the deterministic finite automaton (DFA) constructed by the tool from an expression or formula.

```

<reasoning> ::= (<definition>* <assumption>* <reasoning_steps>)|<print_part>
<definition> ::= define <pred_name> ( (<variable> (<variable>)* ) ≡ <formula> );
<pred_name> ::= A | B | ... | Z
<variable> ::= a | b | ... | z
<formula> ::= F | T | <comparison> | ( <formula> )
            | <pred_name> | <pred_name> ( <expr> (<expr>)* ) | pow2 ( <expr> )
            | ¬ <formula> | <formula> (↔ | → | ∨ | ∧) <formula>
            | (∀ | ∃) <variable> [; <formula>] : <formula>
<comparison> ::= <expr> ((< | ≤ | = | ≠ | ≥ | >) <expr>)+
<expr> ::= <nat_number> | <variable> | ( <expr> ) | | <expr> |
            | (min | max | bnot | lpow2 | hpow2) ( <expr> )
            | (+ | −) <expr> | <expr> (+ | − | band | bor | bxor) <expr>
            | <nat_number> <expr> | <nat_number> · <expr> | <expr> · <nat_number>
            | <expr> (div | mod) <nat_number>
<assumption> ::= assume <formula> ;
<reasoning_steps> ::= <formula>
            | <formula> ([<subreasoning> original] (↔ | ⇒ | ⇐ | ≡) <formula>)+
<subreasoning> ::= subproof <reasoning> subend
<print_part> ::= printDFA (<expr> | <formula>)

```

precedence: highest · (div mod) (band bor bxor) (+ −) ¬ ∧ ∨ → ↔ (∀ ∃) lowest
→ associates to right, everything else to left

Figure 2: Succinct, not fully precise grammar

3.2 Lexical Level

The non-ASCII symbols in Figure 2 can be typed as follows:

symbol	≡	↔	⇒	⇐	∀	∃	↔	→	∨	∧	¬	F	T	≤	≠	≥	·
type	===	<=>	==>	<==	AA	EE	<->	-->	\	/\	!	FF	TT	<=	!=	>=	*

Lowercase letters denote variables that hold integers. Uppercase letters may be used as propositions. As will be discussed in Section 3.4, they may also be used as names of user-defined formulas. For brevity, the grammar refers to both as $\langle \text{pred_name} \rangle$.

A $\langle \text{nat_number} \rangle$ is a non-empty sequence of 0, 1, ..., 9. Because most educational applications do not need integers that are far from zero, and because such integers may make the running time too big, a limit of one million has been set. If it is exceeded, the tool reports “Too big or too small integer:” and shows the integer.

3.3 States, Assumptions and Reasonings

By *state* we mean an assignment of integer values to the free variables, and truth values to the propositions, of the formula or set of formulas in question. This use of “state” is similar to the famous book “The Science of Programming” by Gries [4, p. 66]. For brevity, from now on we will mostly not mention propositions, although much of what follows also applies to them.

An $\langle \text{assumption} \rangle$ restricts the context of a formula or reasoning. A formula is considered true and a reasoning is considered correct if and only if it does not have any counter-example that satisfies all assumptions within whose scope the formula or reasoning is. For instance, the

tool says that $|n| = -n$ has $n = 1$ as a counter-example, but accepts **assume** $n \leq 0$; $|n| = -n$ as correct. Because of technical limitations, assumptions cannot contain quantifiers.

A state is *possible* if and only if it satisfies all assumptions in effect. The symbols $\Leftrightarrow$, $\Rightarrow$, $\Leftarrow$ and $\equiv$ compare formulas. They reside between, not in, formulas; and their use is correct or incorrect, not true or false. The reasoning step $\varphi \Rightarrow \psi$ is correct if and only if every possible state that makes φ true also makes ψ true. So $\varphi \Rightarrow \psi$ is correct under the assumption χ if and only if $\varphi \wedge \chi \Rightarrow \psi$ is correct under no assumptions. The step $\varphi \Leftrightarrow \psi$ is correct if and only if every possible state that makes one of φ and ψ true, also makes the other true. Of course, $\varphi \Leftarrow \psi$ means the same as $\psi \Rightarrow \varphi$.

In the context of this paper, but not everywhere in our tool, $\varphi \equiv \psi$ means the same as $\varphi \Leftrightarrow \psi$. Because indexing an array out of bounds is undefined, division by 0 is undefined and so on, some modes of the tool use the three-valued logic introduced in [15, 16]. In them, $\varphi \equiv \psi$ means that φ and ψ yield the same truth value, while $\varphi \Leftrightarrow \psi$ allows one to be false and the other be undefined. So, without any assumptions, we have, for instance, $\frac{|x|-1}{x+1} = 0 \Leftrightarrow x = 1$. It is unambiguous about the fact that -1 is not a root of $\frac{|x|-1}{x+1} = 0$, while "if $x \neq -1$ then $\frac{|x|-1}{x+1} = 0 \Leftrightarrow x = 1$ " is not. This is important from pedagogical point of view.

However, the language in Figure 2 does not contain anything that could produce an undefined result. So in this paper we need not worry about the third truth value. In its absence, but not in its presence, $\varphi \Rightarrow \psi$ is correct under the assumption χ if and only if $\forall v_1 : \dots \forall v_n : \varphi \wedge \chi \rightarrow \psi$ is true, where $v_1, \dots, v_n$ are the free variables in question.

As was told in Section 2.1, the tool receives a sequence containing both the student's answer and commands written by the teacher. In the integer Presburger mode, the student's answer is usually a formula or a reasoning of the kind in Figure 2. The tool either accepts it as correct or gives a counter-example. The teacher may tell the tool to not accept certain operators in final answers (we will soon explain "final"). So the teacher may ask the student to write, without using $|$, an expression that yields the same value as $|n|$. A possible good answer is $\max(n, -n)$.

As an example of more complicated use, consider the problem of writing a not too long formula that describes the possible states of a volleyball set. The student may first write $0 \leq p \wedge 0 \leq q \wedge (p \leq 25 \wedge q \leq 25 \vee p - 2 \leq q \leq p + 2)$. The tool replies that it is mathematically correct but 25 symbols long, while at most 18 is allowed. Keeping the already written formula in sight may be helpful when thinking about shorter ways of saying the same thing. So the student leaves it there, writes $\equiv$ after it and then $\min(p, q) \geq 0 \wedge (p \leq 25 \geq q \vee |p - q| \leq 2)$. In this way the student may answer with a reasoning chain. The last formula in the chain is the final answer.

The tool allows writing subproofs (that is, sub-reasonings) inside reasonings. They are useful, for instance, when solving pairs of equations in the real logic mode of the tool. Furthermore, $\Rightarrow$ is handy there. In the main proof, **original** refers to the first formula in the reasoning chain. In a subproof it may also be used as the first formula in a reasoning chain (although Figure 2 does not say so). Then it refers to the same formula as **original** refers to in the next level up in the nesting of the main proof and subproofs. However, we have not found much use for subproofs in the integer Presburger mode. So we will not discuss them further in this paper.

3.4 User-Defined Subformulas

A *definition* introduces a subformula. In the "sum of cubes" problem on the web page mentioned above, the teacher has hiddenly written

define $C(n, m) \equiv n = m = 0 \vee n = m = 1 \vee n = 2 \wedge m = 8 \vee \dots$

$$\vee n = 15 \wedge m = 3375 \vee n = 16 \wedge m = 4096;$$

This makes $C(n, m)$ true if and only if $0 \leq n \leq 16$ and $m = n^3$. The student is then asked to write a formula saying that x is the sum of the cubes of n and m . (The student is told that the tool does not know multiplication of variables, and trickery has been used to make $C(n, m)$ work at the extent needed by the problem.) Two possible answers are $\exists c : \exists d : C(n, c) \wedge C(m, d) \wedge x = c + d$ and $\exists c : C(n, c) \wedge C(m, x - c)$. This formula is given the name $S(n, m, x)$.

The student is told that there is a non-negative integer x that can be represented as the sum of two cubes in two different ways. The student is then asked to write a formula, using $S(n, m, x)$, that claims the opposite, with x being the integer and $n^3 + m^3$ and $k^3 + h^3$ being the two ways. If the student writes, for instance, $\neg(S(m, n, x) \wedge S(k, h, x) \wedge k \neq m \neq h)$, then the tool replies that it has the counter-example $m = 10$, $n = 9$, $x = 1729$, $k = 1$ and $h = 12$. In this way the student is guided to fool the tool to reveal the taxicab number of Hardy and Ramanujan. Finally the student is told to find another taxicab number using the tool.

Not all variables of the formula need appear in the parameter list. Given **define** $A(x) \equiv x > y$, $\neg A(y)$ expands to $\neg(y > y)$ and is always true, while $\neg A(z + y + x)$ expands to $\neg(z + y + x > y)$ and has many counter-examples. If asked whether $\neg A(z + y + x)$ holds, the tool provides the counter-example $y = 0$, $z = 0$ and $x = 1$.

The tool currently does not accept a *definition* without a parameter list. However, the same effect can be obtained by **assume** $\langle pred_name \rangle \leftrightarrow \langle formula \rangle$.

3.5 Formulas and Integer Expressions

The syntax for quantifiers varies in the literature: $\forall x(\varphi)$, $(\forall x)(\varphi)$ and so on. The book [4] uses $(\mathbf{A}i : \psi : \varphi)$, where ψ is often something like $0 \leq i < n$ and i is used for indexing an array. **A** and **E** were used instead of $\forall$ and $\exists$ because of limitations of the phototypesetter [4, p. ix].

Our tool has an “array claim” mode that can be used similarly. Its syntax is not dictated by a phototypesetter! It supports checking answers to problems like “write a formula saying that the element at index i is different from all other elements”, to which students are expected to answer something like $0 \leq i < n \wedge \forall j : 0 \leq j < n \wedge j \neq i : A[i] \neq A[j]$. We use “;” instead of “:” as the first separator, to make it easier to see when reading from left to right whether the formula has or lacks the ψ part. To reduce the number of (and) needed, we gave $\forall$ and $\exists$ the lowest precedence.

The notations $\forall x ; \psi : \varphi$ and $\exists x ; \psi : \varphi$ mean the same as $\forall x : \neg\psi \vee \varphi$ and $\exists x : \psi \wedge \varphi$, respectively. For instance, $\forall n ; n < 0 : |n| = -n$ is true. In the integer Presburger mode, ψ may be just any formula. This much freedom is perhaps not the best choice from the pedagogical perspective, but because we do not know what would be better, we chose what was simultaneously liberal enough and easy to implement.

There is no agreement in the literature whether $\rightarrow$ should be left- or right-associative. Ours is right-associative. It makes $\varphi \rightarrow \psi \rightarrow \xi \rightarrow \chi$ equal to $\varphi \wedge \psi \wedge \xi \rightarrow \chi$, while $((\varphi \rightarrow \psi) \rightarrow \xi) \rightarrow \chi$ equals the less elegant $\varphi \wedge \neg\psi \vee \xi \rightarrow \chi$. We gave $\vee$ lower precedence than $\wedge$. This is similar to many programming languages, and often reduces the number of parentheses.

The predicate $\text{pow2}(x)$ will be discussed soon. The remaining aspects of $\langle formula \rangle$ are more or less what one would expect. Also $\langle expr \rangle$ is mostly familiar. There are some unusual operators and functions, which we will now discuss.

The infix binary operators **band**, **bor** and **bxor** are bitwise and, or and xor. Furthermore, **bnot**(x) is bitwise not. To avoid confusion, its argument must always be surrounded by parentheses. Their behaviour on negative numbers has been made to agree with two’s complement representation, although the tool uses internally a different representation. So $\text{bnot}(x) = -1 - x$.

```

define  $T(x, y) \equiv$ 
    pow2( $x$ )  $\wedge$   $x$  band  $m = x \wedge y = x \vee$ 
     $\neg(\text{pow2}(x) \wedge x \text{ band } m = x) \wedge y = \text{lpow2}(m)$ ;
define  $P(x) \equiv \exists y : T(x, y) \wedge$ 
    ( $y$  band 2 = 2  $\vee$   $y$  band 8 = 8  $\vee$   $y$  band 32 = 32  $\vee$   $y$  band 128 = 128);
define  $Q(x) \equiv \exists y : T(x, y) \wedge$ 
    ( $y$  band 4 = 4  $\vee$   $y$  band 8 = 8  $\vee$   $y$  band 64 = 64  $\vee$   $y$  band 128 = 128);
define  $R(x) \equiv \exists y : T(x, y) \wedge$ 
    ( $y$  band 16 = 16  $\vee$   $y$  band 32 = 32  $\vee$   $y$  band 64 = 64  $\vee$   $y$  band 128 = 128);
assume  $1 \leq m \leq 255$ ;
 $\exists x ; P(x) : Q(x) \wedge R(x) \equiv (\exists x ; P(x) : Q(x)) \wedge \exists x ; P(x) : R(x)$ 

```

Figure 3: Testing unary predicate law candidates

This makes De Morgan's laws hold, that is, $\text{bnot}(x \text{ band } y) = \text{bnot}(x) \text{ bor } \text{bnot}(y)$ and $\text{bnot}(x \text{ bor } y) = \text{bnot}(x) \text{ band } \text{bnot}(y)$. We also have $\text{bnot}(x) \text{ bxor } y = \text{bnot}(x \text{ bxor } y)$, $x \text{ band } \text{bnot}(x) = 0$ and $x \text{ bor } \text{bnot}(x) = x \text{ bxor } \text{bnot}(x) = -1$. Of course, $\text{bnot}(\text{bnot}(x)) = x$.

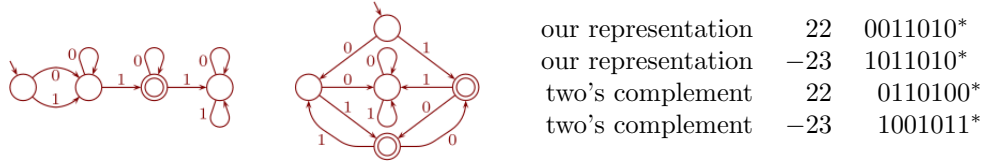
The predicate $\text{pow2}(x)$ returns true if and only if either x or $\text{bnot}(x)$ is a power of 2. That is, in two's complement representation, x has precisely one 1-bit or precisely one 0-bit (and is not 0). As a consequence, $\text{pow2}(x)$ is equal to $\text{pow2}(\text{bnot}(x))$. This is in harmony with the fact that also band, bor, bxor and bnot can deal with negative integers. If all negative integers should result in false, then one can use $x > 0 \wedge \text{pow2}(x)$.

If $x > 0$ then $\text{lpow2}(x)$ and $\text{hpow2}(x)$ return the power of 2 that matches the least significant and most significant 1-bit in x . So the tool accepts **assume** $x > 0$; $\text{lpow2}(x) \leq x < 2 \text{ hpow2}(x) \wedge \text{pow2}(\text{lpow2}(x)) \wedge \text{pow2}(\text{hpow2}(x))$. We have $\text{lpow2}(0) = \text{hpow2}(0) = 0$, $\text{lpow2}(\text{bnot}(x)) = \text{bnot}(\text{lpow2}(x))$ and $\text{hpow2}(\text{bnot}(x)) = \text{bnot}(\text{hpow2}(x))$.

Figure 3 contains an example of the use of bitwise operations. The last line contains a (perhaps student-written) candidate for a law in predicate logic, and the previous lines (written by the teacher and hidden) make the tool test the correctness of the candidate. The candidate must contain no other non-logical symbols than propositions and the unary predicates P , Q and R . In an interpretation, there either is or is not a value of x that makes $P(x)$, $Q(x)$ and $R(x)$ false; and either is or is not a value of x that makes $P(x)$ and $Q(x)$ true and $R(x)$ false; and so on. There are eight different possible truth value combinations of $P(x)$, $Q(x)$ and $R(x)$, each of which is or is not obtained by some value of x , except that obtaining none is not allowed. This is because in predicate logic, domains of discourse cannot be empty.

This creates 255 possibilities. They are scanned by m . On the second line, y goes through those powers of 2 whose corresponding bit in m is 1. The third line makes those values of x that do not qualify as y to be replaced by a power of 2 that is certainly in m . Lines 4 to 9 give $P(x)$, $Q(x)$ and $R(x)$ the truth values dictated by the index of the chosen 1-bit of m .

The tool says that the given candidate fails when $m = 40$, and then the left hand side yields false and right hand side yields true. We have $40 = 8 + 32$. So the interpretation contains two states: one where $P(x)$ and $Q(x)$ are true and $R(x)$ is false, and one where $P(x)$ and $R(x)$ are true and $Q(x)$ is false. Indeed, in such an interpretation, $Q(x) \wedge R(x)$ is never true, so the left hand side is false; but there is a state making $\exists x ; P(x) : Q(x)$ true and a (different) state making $\exists x ; P(x) : R(x)$ true, so the right hand side is true. Using $\Leftarrow$ instead of $\equiv$ on the last line elicits the same counter-example, but $\Rightarrow$ is accepted. Changing each $\wedge$ to $\vee$ or each $\exists$ to $\forall$ is also accepted, but making both changes is not.

Figure 4: The DFA for $\text{pow2}(x)$ and $\text{bnot}(x) = y$, and an example of both representations

4 Design Ideas

4.1 Representation of Formulas as DFAs

As has already been mentioned, lowercase letters a to z represent integer variables and uppercase letters A to Z represent propositions or user-defined predicates. However, the names of user-defined predicates are not present at the DFA level, so there A to Z only represent propositions. For dealing with arithmetic expressions, the program uses four additional variables $v_{\{}$, $v_{|}$, $v_{\}}$ and $v_{\sim}$ named by the ASCII characters $\{$, $|$, $\}$ and $\sim$ that come after z in ASCII order. This will be discussed in Section 4.2.

A formula is represented by a list of its propositions and free variables together with a DFA. We call this list *variable list*. A formula may be equivalent to another formula with a different variable list. For instance, $P \wedge (x < y \vee x \geq y)$ is equivalent to P . The variable lists are always in ASCII order, that is, first the uppercase letters in alphabetic order, then the lowercase letters in alphabetic order, and finally the four special variables in the order used above.

In our representation, every finite bit string may be interpreted as an integer. The empty string represents 0. If the first bit $b_{\pm}$ exists, then $b_{\pm}b_0b_1 \cdots b_n$ represents $\sum_{i=0}^n b_i 2^i$ if $b_{\pm} = 0$; and $-1 - \sum_{i=0}^n b_i 2^i$ if $b_{\pm} = 1$. Extending a bit string by 0 does not change the value that it represents, no matter what the sign bit is. This is different from the most widely used representation of negative integers, that is, two's complement (or more generally r 's complement, where r is any integer at least 2; [1] uses it). There the extension bit that does not change the value is 0 for non-negative and 1 (more generally, $r - 1$) for negative integers. As we will soon see, having it always 0 may both increase and decrease the sizes of DFAs for atomic formulas. More importantly, it simplifies the implementation of the DFA operations discussed below.

The bit operations discussed in Section 3.5 have been implemented so that they behave according to two's complement, although the internal representation is what was described above. For instance, as is shown in Figure 4 left, the DFA for $\text{pow2}(x)$ ignores the sign bit and then insists on precisely one 1-bit. It thus accepts precisely the n of the form 2^k or $-1 - 2^k$ where k is a natural number. The latter are those non-zero integers whose two's complement representation has precisely one 0-bit.

The tuple consisting of the truth values of propositions $P_1, \dots, P_k$ and values of integer variables $v_1, \dots, v_V$ is represented by any bit string that first has a bit for each proposition, then V bits representing the sign bits of $v_1, \dots, v_V$, then V bits representing the b_0 -bits of $v_1, \dots, v_V$, and so on, at least up to and including the last 1-bit. Also these bit strings have the property that extending them with 0 or removing a 0 at the end does not modify the represented values. Indeed, the empty string represents $\neg P_1 \wedge \dots \wedge \neg P_k \wedge v_1 = \dots = v_V = 0$. Therefore, the DFAs are made to obey the rule that the tail state of a 0-transition is a final state if and only if also the head state is.

Figure 4 middle shows the DFA for $\text{bnot}(x) = y$. The sign bits of x and y must be different, and from then on each bit of y must be the same as the corresponding bit of x . This has the

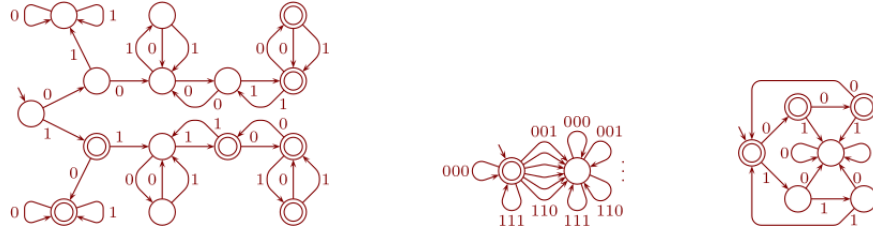


Figure 5: A DFA for $x < y$; and a DFA for $a = b = c$ using the tuple approach and single-bit approach

same effect as two's complement has when each bit of y is different from the corresponding bit of x . This is illustrated in Figure 4 right by showing how both representations map 22 to its complement -23 . In the figure, “0*” and “1*” denote repeating 0 or 1 zero or more times.

In the case of `pow2`, two's complement would have required two branches in the DFA, one for positive (precisely one 1-bit) and one for negative (precisely one 0-bit) integers. As Figure 4 left shows, in our case the two branches were merged. On the other hand, it is often the other way round. An example is in Figure 5 left, where our approach requires two branches.

The same transition may correspond to different propositions and variables, depending on how many transitions have been traversed before it. As an extreme example, $(P \vee \neg P) \wedge x = x \wedge (Q \vee \neg Q) \wedge y = y$ results in the variable list P, Q, x, y and the DFA $0 \rightarrow 1$, where each transition corresponds to every proposition and variable.

When computing a DFA for $\exists x : \varphi$, it is necessary to ensure that if a string and its proper prefix represent the same tuple, and the longer string is accepted, then also the shorter string is. In two's complement representation this is challenging, because the extension bit whose addition to the end does not change the value of an integer, depends on the sign of the integer. The paper [1] briefly discusses how solving this problem was improved from “ $O(2^n)$ ” to “experimental results have shown it to be of moderate practical cost”. In the single-bit approach, the problem is solved by a linear-time graph search along reversed 0-transitions, starting from the final states and marking all visited states final.

In the single-bit approach, the length of the representation of a tuple need not be a multiple of the number of variables, because any 0-bits at the end could be removed. Regarding minimization, this gives the single-bit approach an advantage over [1], where “to encode a vector of real numbers, we represent each of its components by words of identical integer-part length”. This is apparent in the DFA pictures in this paper as transitions from a state to itself.

In [2, 5, 10], labels of transitions of the DFAs are tuples of bits, one bit for each considered variable. In our case, the alphabet is just $\{0, 1\}$. Assume that there are V variables. What is one state and 2^V transitions in the tuple approach, is at most $2^V - 1$ states and $2^{V+1} - 2$ transitions in our case. On the other hand, as is illustrated in Figure 5 middle and right, sometimes our representation has fewer transitions than the tuple approach. In these respects [1] is similar to our paper, except that it may use also other bases than 2.

When computing the conjunction, disjunction, etc. of formulas, the corresponding DFAs may have different variable lists. It is complicated to make the variable lists the same by adding extra transitions. On the other hand, it is easy to do as follows. Let $b \in \{0, 1\}$ and let $q_1 \xrightarrow{b} q'_1$ and $q_2 \xrightarrow{b} q'_2$ be transitions of the DFAs. Keep track of the variable in question in the state of the product. Create either $(v, q_1, q_2) \xrightarrow{b} (v', q'_1, q_2)$, $(v, q_1, q_2) \xrightarrow{b} (v', q_1, q'_2)$ or $(v, q_1, q_2) \xrightarrow{b} (v', q'_1, q'_2)$ depending on whether v is in the variable list of only the first, only the

second, or both DFAs. Here v' is the variable whose turn is next after v .

Each state having precisely two outgoing transitions brings also low-level performance advantage. In the tuple approach, the number of outgoing transitions depends on the variable list. So the amount of memory needed by a state varies. Although the compiler and run-time environment may take care of memory management, it consumes time and may result in inefficient use of memory.

Despite its name, the “shared multi-terminal BDD” representation in [11] seems almost the same as the serial and single-bit approaches. States are represented explicitly, and the SMBDD represents the transition relation. The terminal nodes of the SMBDD are states (instead of “false” and “true”). The BDD nodes of the SMBDD approach become ordinary DFA states in the single-bit and serial approaches. What is referred to by “reduced” or “canonicity” in the context of (ordered) BDDs, is obtained by DFA minimization. The serial and single-bit approaches lack the elimination of nodes whose 0-child and 1-child are the same. This does not affect canonicity. It only replaces each SMBDD arc that skips ℓ levels by $\ell + 1$ arcs and ℓ states. This increases memory consumption but also reduces it, by removing the need to store the variable identity in the nodes.

Also [1] remarks “in the context of minimized deterministic automata, the serial encoding is essentially equivalent to using the simultaneous scheme, while representing the transitions from a state to another by a BDD (an r -ary Decision Diagram for bases r other than 2)”.

4.2 Dealing with Expressions

Expressions are represented by DFAs for formulas of the form $\langle expr \rangle = v_\sim$. Arithmetic operations are represented by DFAs whose variable lists only contain $v_\sim$ and one or more of $v_\{$, $v_\}$ and $v_\}$.

For instance, a DFA representing Euclidean division by a positive integer constant c is constructed as follows. First a DFA for $cv_\{ = v_\}$ is constructed by calling a corresponding subroutine with c as an argument. Then it is intersected with the DFA that represents $v_\{ + v_\} = v_\sim$. The latter DFA has been encoded as a constant in the program. The result represents $cv_\{ = v_\} \wedge v_\{ + v_\} = v_\sim$. Next variable elimination (Section 5.5) is applied to $v_\}$, resulting in a DFA representing $\exists v_\} : cv_\{ = v_\} \wedge v_\{ + v_\} = v_\sim$, which equals $cv_\{ + v_\} = v_\sim$. Given c , another subroutine constructs a DFA for $0 \leq v_\} < c$. Intersecting these two yields a DFA for $cv_\{ + v_\} = v_\sim \wedge 0 \leq v_\} < c$.

As another example, a DFA for $\langle expr1 \rangle + \langle expr2 \rangle = v_\sim$ is computed by first computing a DFA for $\langle expr1 \rangle = v_\sim$, changing the name of its last variable to $v_\}$, intersecting it with the above-mentioned DFA for $v_\{ + v_\} = v_\sim$, and applying variable elimination to $v_\}$. The result represents $\langle expr1 \rangle + v_\} = v_\sim$. Then a DFA for $\langle expr2 \rangle = v_\sim$ is computed and the name of its last variable is changed to $v_\}$. Finally these two DFAs are intersected and the variable $v_\}$ is eliminated.

4.3 Generation of Counter-Examples

When more than one counter-example is available, the problem of selecting a pedagogically good one is not trivial. It seems clear that one should favour integers with small absolute values over those with big absolute values, to make it easier to use the counter-example for trying to see what is wrong with the answer. Even so, it is not necessarily a good idea to always produce a counter-example that minimizes the maximum absolute value of the numbers in it. It risks obscuring in the student’s mind the fact that any example violating a formula or reasoning step is a valid counter-example. Getting “ $n = 112$ ” instead of “ $n = 100$ ” as a counter-example to

$n < 100$ may be confusing. However, then it is a cognitive conflict in the sense of [18, Sect. 2], and thus may help to unlearn the misconception that counter-examples must be unique.

To check a formula or reasoning step, our tool computes a DFA representation of the set of its counter-examples. If this set is not empty, the tool chooses a counter-example by breadth-first search, trying 0-transitions before 1-transitions. Thus the selected counter-example has as short a bit string representation as possible. This gives the tool the tendency to roughly, although not precisely, minimize the biggest absolute value of the numbers in the counter-example. For instance, to $\neg(x = 1 \wedge y = 100 \vee x = -40 \wedge y = 30)$ it gives the counter-example $x = -40 \wedge y = 30$.

Among counter-examples whose shortest bit string representations are of the same length, the tool favours those whose first bit is 0, then those whose second bit is 0, and so on. This makes the tool favour, as a secondary criterion, non-negative integers over negative ones. Among positive integers, it also makes the tool prefer those that are divisible by the biggest possible power of two. The shortest representations of numbers from 100 to 127 use 8 bits, and 128 needs 9 bits. Among 100 to 127, 112 is divisible by 2^4 and the others are not. So the tool provides $n = 112$ as the counter-example to $n < 100$.

5 Implementation-level Issues

5.1 Remarks on the Programming Language Constructs Used

The implementation was written in C++. It has more or less everything that is needed when developing complicated high-performance programs. Many of its features are notoriously error-prone, but also other powerful programming languages suffer from some of the same problems. On the other hand, C++ contains constructs that can be used to protect against errors, such as an advanced type system. We do not claim that our implementation is free from errors, but we do claim that detecting errors during production runs has been rare and all known errors have been fixed.

Memory management is a major issue in programs that create big intermediate data structures, such as big DFAs. Some programming languages, but not C++, have a garbage collector that automatically detects memory blocks that are no longer used, and circulates them to be used again. In C++, objects can be made to automatically release the memory they have reserved, when they are destructed. The standard data structures provided by C++ use this extensively.

Our program exploits this by introducing each object in the topmost (in the call hierarchy) subroutine that mentions it. It is passed as a modifiable parameter to subroutines that put data in it and thus reserve memory in its name. When the topmost subroutine is exited, the memory is automatically released. This is simple, more efficient than garbage collectors, and meets all the needs of the program. As far as we know, this idea is not used much elsewhere. We speculate that this is because returning results via parameters is widely advised against. Instead, subroutines should return all their results via their names, similarly to how $\log x$ in mathematics returns its result. So we deviated from the usual convention, because it would have made memory management more difficult.

C++ features both signed and unsigned integer types. Many data structures in the C++ standard library use unsigned integers as sizes and indices. This is natural, because sizes cannot be negative, and indices start from 0 in C++. In our program, signed integers are used where negative integers may arise, and unsigned integers are used elsewhere.

A *C++ vector* is an extensible array. Its *size* is the number of elements in it, and its

i	3597	3598	3599	3600	3601	3602	3603
max size	6 971 886	3 483 018	62 641 777	430 468	6 974 778	3 484 464	10 828 665
minimized	200 250	99 436	37 586 755	11 978	200 451	99 537	6 498 909
final size	2 597 058	349 020	25 056 258	55 010	1 008 480	97 356	4 331 002
time	12.0	1.2	140.	0.12	5.1	0.70	16.9
no reuse	13.7	1.3	158.	0.15	5.8	0.77	20.1
no dom.	12.2	1.2	143.	0.13	5.2	0.73	17.4
no rej. st.	12.9	1.2	148.	0.13	5.4	0.73	17.8
max size	9 093 102	3 515 017	87 700 302	434 941	7 038 241	3 516 475	15 161 990
time	14.6	1.4	168.	0.16	6.1	0.83	21.2

Figure 6: $(t \bmod i) \operatorname{div} 60 = x$ with all, two or no optimizations

capacity tells the number of elements for which room has been reserved. In addition to reading and writing elements at the location indexed by an unsigned integer, elements may be inserted and removed, causing the size of the vector to grow or become smaller. The elements are stored in a dynamically reserved and released memory block. When the capacity is exceeded, a new, bigger memory block is reserved (usually twice the previous size), the elements are copied there, and the old memory block is released. During the lifetime of a vector, the memory block is never automatically replaced by a smaller one, but the programmer may explicitly make this happen. The contents of two vectors may be swapped in small constant time. The library implements it by swapping their size and capacity information and their pointers to their memory blocks.

Even if memory management is automated one way or another, repeated allocation and deallocation of memory may chop the free memory to blocks that are too small to serve the needs. Therefore, the potentially biggest data structures in our program use and reuse ten vectors that are never released, only cleaned up for the next use.

An example of the effect of this optimization is shown in Figure 6. There a DFA for $(t \bmod i) \operatorname{div} 60 = x$ is computed for $3597 \leq i \leq 3603$. The lines “final size” and “time” tell the number of states in the DFA and how many seconds its computation took. The size of the biggest DFA constructed during the computation and how small it became in minimization are shown on lines “max size” and “minimized”. The line “no reuse” tells the computation time, when the reuse optimization is switched off. The next two lines tell the time when one or another optimization discussed later is switched off. On the last two lines, all three optimizations have been switched off.

5.2 Representation of DFAs

Finished DFAs have a terse representation. The DFA state type has only three, small instance variables: unsigned integers pointing to the 0-child and 1-child, and a Boolean telling whether the state is final. This is only 12 bytes in total! The DFA type has a vector of characters for storing variable names, a vector of DFA states, and an unsigned integer specifying the initial state. Making the initial state always be state number 0 would not have been feasible, because DFA minimization cannot easily be made to respect it.

DFAs under construction have complicated representations. They are discussed in Sections 5.3 to 5.5.

5.3 DFA Minimization

The smallest possible DFA on the alphabet $\{0, 1\}$ that accepts the same language as a given DFA can be found in $O(n \log n)$ time, where n is the number of states [7]. Our implementation is similar to that in [12], except that the tricks are omitted that work on partial transition functions. The program runs DFA minimization after every operation whose result may be non-minimal.

5.4 Propositional Connectives

The implementation of $\neg$ is next to trivial: change every final state to non-final and vice versa.

We now discuss the implementation of $D_1 \wedge D_2$, $D_1 \vee D_2$, $D_1 \rightarrow D_2$ and $D_1 \leftrightarrow D_2$. Let us call the result D . If it has an empty variable list, or if D_1 or D_2 has only one state and does not affect the variable list of D , then a trivial algorithm is used. Otherwise the implementation is based on the product of DFAs, with some modifications because the variable lists of the components may differ and because of attempts to improve performance.

The standard DFA product computation makes the transition $(q_1, q_2) \xrightarrow{b} (q'_1, q'_2)$ from the transitions $q_1 \xrightarrow{b} q'_1$ and $q_2 \xrightarrow{b} q'_2$ of D_1 and D_2 . As was mentioned in Section 2.2, in our case, the states of the result also must keep track of the variable whose turn is next. Depending on the variable lists of D_1 and D_2 , our program makes either $(v, q_1, q_2) \xrightarrow{b} (v', q'_1, q'_2)$, $(v, q_1, q_2) \xrightarrow{b} (v', q_1, q'_2)$ or $(v, q_1, q_2) \xrightarrow{b} (v', q'_1, q'_2)$, where v is the current and v' is the next variable. The next variable is the next in ASCII order, starting again with the first after the last.

The product is computed in breadth-first order. To quickly see whether a newly constructed (v, q_1, q_2) has been constructed also before, the program stores the (v, q_1, q_2) in a hash table. Initially the hash table has 2^{16} hash lists. Each time the number of elements in it reaches the number of hash lists, the latter is multiplied by 16. As can be seen from Figure 6, the program survives DFAs with millions of states.

Until September 2026, the number of hash lists was always 2^{16} . Then we realized that more hash lists could extend the range of inputs to which the program replies within reasonable time. We tried some fixed numbers of hash lists greater than 2^{16} , but they made the program slower on small DFAs. So we modified the program to make the number of hash lists grow as needed.

In addition to D_1 and D_2 , the product subroutine gets an unsigned integer *opr* specifying the operation that is computed. Let f_i be 1 if the current state of D_i is final, and 0 otherwise. Whether or not the state of D is final is decided by looking at the $(2f_2 + f_1)$:th least significant bit of *opr*. So 8 means $\wedge$, 14 means $\vee$, 13 means $\rightarrow$ and 9 means $\leftrightarrow$.

By a *dominating state* of D_1 or D_2 we mean a state such that both of its outgoing transitions lead to itself, and its final state status is such that the final state status of D does not depend on the final state status of D_2 or D_1 . For instance, if the operation is $\vee$, q_1 is a final state of D_1 , $q_1 \xrightarrow{0} q_1$ and $q_1 \xrightarrow{1} q_1$, then q_1 is a dominating state of D_1 . In the case of $\wedge$, $\vee$ and $\rightarrow$, a minimized DFA can have at most one dominating state. With $\leftrightarrow$, there cannot be any.

Before the computation of the product, a dominating state of each component is found, if it exists. During the computation of the product, when one component is in a dominating state, the outgoing transitions of the product state are made to lead back to the state, and computation is not continued further from it. The effect of switching this off is shown on line “no dom.” and the lower “max size” line of Figure 6. This was the only optimization that affected the size of the biggest DFA.

In the serial encoding, when there is more than one free variable, this heuristic would have to deal with cycles of length greater than one. That would be more difficult to implement.

5.5 Quantifiers

The most challenging task is the implementation of $\exists$. If the quantified variable $v_{\exists}$ is not in the variable list, the result is the input DFA as such. If $v_{\exists}$ is the only element in the variable list, then a single-state DFA with no variables is made whose only state is final if and only if the input DFA has a final state.

From now on we discuss the remaining cases. The basic idea was explained in Section 2.2. The states of the resulting DFA are pairs (v, A) , where v is the variable whose turn is next, and A is a subset of the states of the original DFA. To quickly see whether a newly constructed (v, A) has been constructed also before, the program stores the (v, A) in a growing hash table similarly to Section 5.4.

A list set represents a set by listing its elements explicitly in an array, linked list or similar. A bit set has a bit for each element in the universe, telling whether or not the element is in the set. Our program represents a set of states as an ordered list of state numbers in an array as long as it consumes less memory than the bit set representation. Then it switches to the bit set representation. This is important, because which representation is better, depends on the input. Changing the program to only use the bit set representation made it run out of memory on other values of i in Figure 6 than 3600 and 3602. Changing it to only use the list set representation made it run 127 seconds on the input in Figure 3, while originally the same task took 0.01 seconds.

In the single-bit representation, many minimized DFAs contain a “reject-all” state that is not final and whose outgoing transitions lead to itself. The existence of reject-all states when there are more than one free variable depends on the extension bits being always 0. Although it is not said above, the reject-all state is not stored in A . The effect of switching this optimization off is shown on line “no rej. st.” in Figure 6. That it did not affect the maximum DFA size is because the biggest DFAs always came from the computation of $\wedge$, etc., instead of $\exists$.

6 Conclusion

Although Presburger arithmetic does not allow multiplication between variables, a decision algorithm for it facilitates providing automatic feedback on versatile problems in educational use. Automata-theoretic algorithms make it possible to also use bitwise operations which, in turn, allow testing predicate logic laws on unary predicates.

A fast enough implementation can be made using automata-theoretic algorithms and heuristic optimization tricks. The use of two different methods for storing sets of states, and automatic switching between them, during the computation of $\exists$ proved particularly important. The single-bit approach with a modified representation of negative numbers simplified the implementation. It made it feasible to use heuristics that would have been infeasible or at least much more difficult to implement with two’s complement representation.

Declaration on the use of Artificial Intelligence. Artificial intelligence has been used only for finding relevant literature and for improving the English language.

References

- [1] Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: An overview. In Peter J. Stuckey, editor, *Logic Programming, 18th International Conference, ICLP*

- 2002, Copenhagen, Denmark, July 29 - August 1, 2002, *Proceedings*, volume 2401 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002. https://doi.org/10.1007/3-540-45619-8_1.
- [2] Dmitry Chistikov. An introduction to the theory of linear integer arithmetic (invited paper). In Siddharth Barman and Slawomir Lasota, editors, *44th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2024, Gandhinagar, Gujarat, India, December 16-18, 2024*, volume 323 of *LIPIcs*, pages 1:1–1:36. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. <https://doi.org/10.4230/LIPIcs.FSTTCS.2024.1>.
- [3] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In Richard M. Karp, editor, *Complexity of computation. SIAM-AMS Proceedings*, volume 7, pages 27–41. American Mathematical Society, 1974.
- [4] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer, 1981. <https://doi.org/10.1007/978-1-4612-5983-1>.
- [5] Christoph Haase. A survival guide to Presburger arithmetic. *ACM SIGLOG News*, 5(3):67–82, 2018. <https://doi.org/10.1145/3242953.3242964>.
- [6] Christoph Haase, Shankara Narayanan Krishna, Khushraj Madnani, Om Swostik Mishra, and Georg Zetsche. An efficient quantifier elimination procedure for Presburger arithmetic. In Karl Bringmann, Martin Grohe, Gabriele Puppis, and Ola Svensson, editors, *51st International Colloquium on Automata, Languages, and Programming, ICALP 2024, Tallinn, Estonia, July 8-12, 2024*, volume 297 of *LIPIcs*, pages 142:1–142:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. <https://doi.org/10.4230/LIPIcs.ICALP.2024.142>.
- [7] John 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)*, pages 189–196. New York: Academic Press, 1971.
- [8] Terhi Kaarakka, Kirsi Helkala, Antti Valmari, and Marjukka Joutsenlahti. Pedagogical experiments with MathCheck in university teaching. *LUMAT: International Journal on Math, Science and Technology Education*, 7(3):84–112, 2019. <https://doi.org/10.31129/LUMAT.7.3.428>.
- [9] George Kinnear, Ian Jones, Chris Sangwin, Maryam Alarfaj, Ben Davies, Sam Fearn, Colin Foster, André Heck, Karen Henderson, Tim Hunt, Paola Iannone, Igor’ Kontorovich, Niclas Larson, Tim Lowe, John Christopher Meyer, Ann O’Shea, Peter Rowlett, Indunil Sikurajapathi, and Thomas Wong. A collaboratively-derived research agenda for e-assessment in undergraduate mathematics. *International Journal of Research in Undergraduate Mathematics Education*, 10(1):201–231, 2024. <https://doi.org/10.1007/s40753-022-00189-6>.
- [10] Felix Klaedtke. On the automata size for Presburger arithmetic. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 110–119. IEEE Computer Society, 2004. <https://doi.org/10.1109/LICS.2004.1319605>.
- [11] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. <https://doi.org/10.1142/S012905410200128X>.
- [12] Antti Valmari. Fast brief practical DFA minimization. *Information Processing Letters*, 112(6):213–217, 2012. <http://dx.doi.org/10.1016/j.ipl.2011.12.004>.
- [13] Antti 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*, pages 324–331. SCITEPRESS, 2021. <https://doi.org/10.5220/0010493103240331>.
- [14] Antti Valmari. Automatic feedback on logic problems about real numbers and integers. In Luka Pavlič and Stephan Schlögl, editors, *Proceedings of the Third Workshop on Co-Creating New Ways of Information Systems Education, Maribor, Slovenia, September 10-11, 2025*, volume Vol-4076 of *CEUR Workshop Proceedings*, pages 1.1–1.13. CEUR-WS.org, 2025. <https://ceur-ws.org/Vol-4076/paper1.pdf>.
- [15] Antti Valmari and Lauri Hella. A completeness proof for a regular predicate logic with undefined

- truth value. *Notre Dame Journal of Formal Logic*, 64(1):61–93, 2023. <https://doi.org/10.1215/00294527-2022-0034>.
- [16] Antti Valmari and Mika Hirvensalo. Towards three-valued practice-oriented formal reasoning. *Journal of Applied Logics*, accepted to. <https://www.collegepublications.co.uk/ifcolog/forthcoming/?00052>.
- [17] Antti Valmari and Johanna 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*, pages 292–299. SciTePress, 2019. <https://doi.org/10.5220/0007708902920299>.
- [18] R. Zazkis and E.J. Chernoff. What makes a counterexample exemplary? *Educational Studies in Mathematics*, 68:195–208, 2008. <https://doi.org/10.1007/s10649-007-9110-4>.