Improvement of a module solving a set of discrete equations in a medicament prescription system

Lila Klatter, 4013751
Supervisor: Rob Bisseling

January 25, 2016

Contents

1 Introduction

In this thesis we will optimize a module solving a set of discrete equations as presented in an open source medicament prescription system. We will further call this module *solver*. Below we will first give a definition of the relevant terms and then give a general description of how the original solver works. For an overview of the terms discussed in this section, see figure 1. In the next chapter we will describe the specifications of the system and thereafter the enhancements we made. We will present our results by investigating a sample case which we use to test a prototype of the current version and a prototype of our optimized version of the solver. We conclude with our final recommendations for making the solver more efficient.

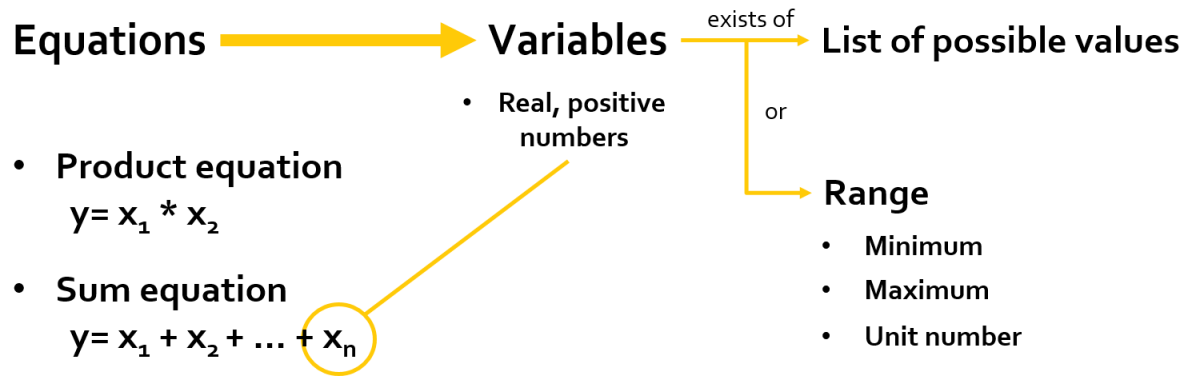

Figure 1: The relevant terms for the solver module

Equations

We distinguish two kinds of equations: a *sum equation* and a *product equation*. Both are defined by a dependent left-hand side variable, say y, and a list of arguments, say $[x_1, x_2, ..., x_{n-1}, x_n]$. If this were a sum equation, the equation is $y = x_1 + x_2 + ... + x_{n-1} + x_n$. We may assume all product equations consist of two arguments, so $n = 2$ for product equations.

Variables

Every variable has a possible range of values it can assume. These values are real numbers larger than zero. Our knowledge about the values of some arguments in an equation can restrict the possible values of others. The program applies these restrictions by changing the minimum and maximum of the possible values. For example, if in an equation $y = x_1 * x_2$ we know the maximum value of x_1 is 10 and the maximum value of x_2 is 20, this implies the value of y can never become larger than $10 * 20 = 200$. So the program will set the

maximum value of y to 200. It is important that this change is always a restriction, so the variables may never take on incompatible values. If the maximum of y was already set to, say, 190, the maximum of y is not changed, for then it would be able to take on values that are not compatible with its previously set maximum. In the next section we will provide all implications certain minima and maxima may have on other variables.

We represent the possible values of a variable by either a finite list of real positive numbers or a *Range*. A Range is a range of numbers limited by either a minimum, a maximum, a basic unit or a combination of these. The basic unit is a number that divides all possible values of the variable. For example, for a variable $x_n \in \mathbb{N}$ the basic unit is 1. When all three are known, the Range becomes a list of numbers. For example, a Range of minimum 2, maximum 7 and a basic unit of 1.5 is the list [3, 4.5, 6]. In section 2.1 we will provide a deeper analysis of these and other restrictions. As you can see, a list takes on discrete values and a Range is continuous when it is not yet transformed to a list.

Solver

The solver takes a list of equations and after a restriction (e.g. a maximum is changed) is added to one of the variables in these equations, it checks what the implications are for all other variables. This is done by checking for each equation whether the known restrictions provide a new restriction. An equation is checked by rewriting it several times so that every variable becomes a left-hand side variable. For example, $y = x_1 * x_2$ is rewritten as $x_1 = y/x_2$ and $x_2 = y/x_1$. Next, new restrictions are found by applying the rules as presented in section 2.2. If a new restriction is set, the process is repeated, again checking the implications for each equation. Once all equations are checked and no changes occur anymore, the user (or the program) may provide new input, only choosing from the possible values. After every input the solver systematically reduces the possible solution space for all variables to guarantee that possible values are always valid. An equation is considered solved when all variables have single values. By providing new input every equation can eventually be solved.

In the medicament prescription system GenPres where this solver module originated from, the solver is applied to assist the user in choosing proper drug prescriptions. For example, dependent on the dose a doctor may want to give a patient, the system provides him/her with a list of options to choose from regarding portion, frequency, etc. The aim of this system is to improve the quality and efficiency of electronic prescribing.[1]

2 Specifications

In this section we present specifications of the solver.

2.1 Calculation rules for adding information to a variable

We will present a synopsis of how we deal with adding more information to a variable (a minimum, maximum, basic unit or list) to provide a deeper understanding of the specifications of the solver module. We will present the rules we implemented in the prototype program, in particular pointing out which values overrule and which values are overridden. In general we may state that restrictions, these being a minimum, maximum or basic unit, overrule any given list. Within these restrictions, the minimum and maximum values change along with the value of a list. When a minimum or maximum is added we must always check that either the minimum or maximum is not set yet or that it's an actual restriction, so the new minimum is higher than the set minimum and vice versa with a maximum. For the implementation, see the code in appendix D, lines 35-148.

When setting a maximum, there are essentially three important cases: either the minimum and basic unit are known and there is no list set, only the basic unit is known and there is not list set, or the variable already has a list set. In the first case a list is created by starting at the first number above the minimum divisible by the basic unit. Then elements are added to the list by constantly increasing each element by the basic unit, until it is just below the maximum. Finally, the new minimum and maximum are added to the variable. The second case is similar, only now the minimum is automatically set at the value of the basic unit. Because all values must be larger than zero and divisible by the unit number, the minimum value must be the unit number. The list and new maximum are subsequently set. In the last case the new maximum overrules the existing list, so the elements above the new maximum are removed and the highest element below the maximum is set as the new maximum.

When setting a minimum, only the last of the above cases is needed, for when a maximum and basic unit are known, the second case is applied and a minimum is already automatically set. So, for a minimum the only thing we need to do is check for a set list what the implications of the new minimum are. The list is filtered so it only contains values above the minimum and the minimum is set as the lowest element in the resulting list.

Next we will consider the basic unit. Again we distinguish three cases: either the minimum and maximum are set and the list not, the maximum is set and the list not, or the list is set. The first two cases coincide with the first two of the maximum, so we will not cover these any further. In the last case the list is filtered, so that only elements of which the basic unit is a divisor remain. If necessary, the minimum and maximum are altered. The list may not become empty, so the basic unit must always divide at least one element of the list. We will come back to this in section 5.

Finally, when a list is set, we do not distinguish cases, but do check the minimum and

maximum values. Let's without loss of generality consider the checks for the minimum value. If it is not set, the minimum of the list becomes the new minimum. If it is set, however, we must check whether this minimum is lower than the minimum of the list. If this is the case, no additional checks are necessary and we set the minimum of the list as the new minimum. If this is not the case, we must filter the list to match the set minimum. The new minimum is again the lowest element that meets the restriction.

2.2 Restrictions imposed by arguments of equations

As stated earlier, known properties of arguments of an equation may pose restrictions on the possible values of other arguments in the equation. See appendix C, lines 1-78, for how the restrictions are implemented in the code of our prototype.

We will present table 1 containing all formulas for the restrictions on simple sum and product equations with two variables. We also present division and subtraction equations. These are useful when equations are rewritten to other left-hand side variables. We only discuss restrictions posed by known minima and maxima. Nothing can additionally be said about restrictions posed by the basic unit of an argument until all possible values in the arguments are known. There is, however, a restriction for the basic unit when a list is set for a variable. This will be discussed in section 5. For all formulas, we should keep in mind that the restriction should always be stricter than the known value. This is ensured by taking the maximum of the known value and the new value in case of a minimum and the minimum of the known value and the new value in case of a maximum. This is of course not necessary if the value is not assigned yet. For readability purposes we have chosen not to include these rules in our table. We arranged our findings on known attributes of the arguments in an equation. If a combination of these attributes is known, a combination of the formulas apply.

Most formulas speak for themselves, but we'd like to make some additional remarks about the first two rows. These rows treat the single known values. All formulas here are without loss of generality only written for x_1. As you can see, y appears as a known value only in these rows. This is because in the other rows x_1 and x_2 take the place of y in the rewritten formulas. If only one variable is known, however, we must also check the known values of y. The known values mentioned in the table are the only ones that provide new information, so the other ones are omitted. It is important to mention that the restriction in field $[1, 2]$ of the table is not the sharpest possible. If $x_1 = max(y)$ is set, there are no more possible values for x_2. This problem is discussed in section 5.

We have chosen to present the sum formula in the table as a simple formula with two arguments. The restrictions for a sum $y = x_1 + x_2 + ... + x_n$ with n variables are similar, so we shall describe them here based on table 1. First, let's consider field $[2, 2]$ in the table. The restriction becomes as follows:

$$min(y) = \sum_{i:\, min(x_i) \text{ is set}} min(x_i)$$

The above restriction is also a more generalized version of the restriction in field $[3, 2]$. For the restriction in field $[5, 2]$ we get:

$$max(y) = \sum_{i=1}^{n} max(x_i), \quad max(x_i) \text{ is set}$$

Finally for field $[4, 4]$.

$$min(x_i) = min(y) - \sum_{j \in \{1,2,..,i-1,i+1,..,n\}} max(x_j), \quad min(y) \text{ and } max(x_j) \text{ are set}$$

The formula in field $[6, 4]$ is found by replacing min by max and vice versa, only it is slightly different. This formula can applied for any known values $min(x_j)$, not all minimum values have to be known. For more details, see section 5. In these last two formulas it is important that $min(y)$ is only set if it is a positive number and if the maximum is larger than the minimum.

$$max(x_i) = max(y) - \sum_{j:\, min(x_j) \text{ is set},\, j \neq i} min(x_j), \quad max(y) \text{ is set}$$

Known values	$x_1 \cdot x_2 = y$	$x_1 + x_2 = y$	$x_1 / x_2 = y$	$x_1 - x_2 = y$
$max(y)$		$max(x_1) = max(y)$		
$min(x_1)$		$min(y) = min(x_1)$		
$min(x_1)$ and $min(x_2)$	$min(y) = min(x_1) \cdot min(x_2)$	$min(y) = min(x_1) + min(x_2)$		
$min(x_1)$ and $max(x_2)$			$min(y) = \frac{min(x_1)}{max(x_2)}$	$min(y) = min(x_1) - max(x_2)$
$max(x_1)$ and $max(x_2)$	$max(y) = max(x_1) \cdot max(x_2)$	$max(y) = max(x_1) + max(x_2)$		
$max(x_1)$ and $min(x_2)$			$max(y) = \frac{max(x_1)}{min(x_2)}$	$max(y) = max(x_1) - min(x_2)$

Table 1: All restriction formulas for minima and maxima

3 Optimization

In this section we will present all enhancements we made to the original solver. First, we will explain our changes in a general manner and then we will describe the workings of our new program more thoroughly and illustrate it with an example.

3.1 General structure

The original solver program has a few weaknesses, causing it to become slow in certain cases. During the iterative process of checking each equation after a change in the variables, no distinction is made between what equations to check and in what order to check them. Simply all are checked. The program also immediately calculates the resulting list once all arguments in an equation are lists. This may result in creating very long lists and then having to calculate a product of two long lists. The calculation time increases with an order of magnitude of $n * m$ with lists of length n and m. Our main focus for improvement was on these points.

We started off by adding a tool for the representation of equations. These were first only presented in a list. Now, the equations are also stored as a matrix, showing which variable appears in which equation. For every system of I equations in which a total of J variables appear, we will create an $I \times J$ matrix A, see figure 2.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,J} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I,1} & a_{I,2} & \cdots & a_{I,J} \end{pmatrix}$$

Figure 2: The general form of the matrix A with $a_{ij} \in \{0, 1\}$

The value of a_{ij} is 1 when variable j appears in equation i and 0 if it doesn't. This representation gives us the possibility of accessing all relevant equations, given a variable, and vice versa. We also still keep a list of equations in which is preserved what kind of equation it is and what the left and right hand variables are, as described in the first section. Not all variables appear in all equations, so the matrix contains many zeros. It is therefore a sparse matrix. Because the matrix sizes are small, the matrix is stored as a dense array, instead of in a sparse format.

Next, we implemented a method carefully choosing the order of evaluation of the equations. When a variable changes, we will only check the equations in which the variable appears, for it can only induce changes in these equations. By checking these equations other variables may change. The equations in which these changed variables appear are added to a queue by an insertion sort, sorted by the number of variables. An insertion sort is viable because the queue always has a relatively small size. If a certain equation already appears in the queue, it is not added, for every equation only had to be checked once.

Finally, an improvement is to postpone the calculation of a product or a sum as long as possible. Once the arguments of an equation are lists of values, the possible values of the left-hand side variable are the sum or product of all combinations of these lists. Our observation is that calculating a product or sum does not provide any more information about the minimum and maximum values of a variable, only about the intermediate values. The intermediate values are only needed as an output to the user. By postponing the calculation, lists will often be more restricted, so of a smaller size. This reduces the calculation time. So, instead of calculating the product or sum as soon as it is possible, we postpone the calculation until new input is required from the user. For further recommendations on this subject, see section 5.

3.2 Algorithms

Here we will describe the workings of the program in a more detailed manner. We will present two algorithms, one for the original solver and one for the new solver.

Algorithm 1 The original Solver

```
1: procedure OriginalSolver(tochecklist, checkedlist)
2:     if tochecklist is empty then return checkedlist
3:     else
4:         i = first equation of tochecklist
5:         If required, change the minimum and maximum values of variables in i
6:         if no variables change then
7:             return OriginalSolver(tochecklist - i, checkedlist + i)
8:         else
9:             return OriginalSolver(tochecklist + checkedlist, empty list)
```

This algorithm is initiated with as *tochecklist* the list of all equations and an empty list as *checkedlist*. In the equations in *tochecklist* a variable is restricted, so the algorithm will find all restrictions that follow from this restriction. Line 5 refers to restricting minima and maxima as explained in section 2.2. Notice that in line 9 the algorithm starts all over again, again with as input all equations and an empty list.

Algorithm 2 The new Solver

```
1: procedure NewSolver(queue, checkedlist)
2:     if queue is empty then return checkedlist
3:     else
4:         i = first equation of queue
5:         If required, change the minimum and maximum values of variables in i
6:         if any variables change then
7:             for all changed variables x do
8:                 for all equations y in which x appears do
9:                     if y is not in queue then add y to queue sorted by number of variables
10:        return NewSolver(queue - i, checkedlist + i)
```

The new algorithm differs from the previous one in initialization and in how it deals with changing variables. Notice that *tochecklist* is now called *queue*, for its function is slightly different. Upon initialization *checkedlist* is still an empty list, but in the queue are only the equations in which the changed variable appears. Then every time new variables change, the equations in which this variable appears are added to the queue. An equation is only added when it is not yet in the queue. It is inserted by an insertion sort, so that an equation is added after other equations in the queue, but still sorted by number of variables. The current equation is then always removed from the queue and added to *checkedlist*. Please note there may be no double equations in *checkedlist*, so any duplicate added will override a previous value. This is not included in the algorithm for readability purposes.

In the code in appendices A and B the solver methods have no return value. This is because all changes are done directly in global variables. Again for readability purpose we presented our above algorithms with a return value.

3.3 Example

Below we will present a simple example consisting of a few equations and a new input value. We will give the execution of the new program after one alteration in the variables. In contrast to the code in the appendices, here we will begin counting the equation and variable numbers at 1 instead of 0. Take the following equations and their corresponding matrix:

1. $x_1 = x_5 \cdot x_6$
2. $x_4 = x_1 \cdot x_5$
3. $x_1 = x_2 + x_3$
4. $x_5 = x_6 + x_7 + x_8$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Say the following values are already set:

- $min(x_1) = 2$
- $min(x_2) = 2$

The second value was found by executing the program after the first value, so we're certain these values don't contradict each other. Now take a new input: $max(x_6) = 20$. We'll show step by step what the queue is, what equation is being checked and what new information is deduced from which equation.

First, the solver will be executed with as input the equations in which x_6 appears, so we look in the matrix in column 6. See figure 3. Equations 1 and 4 are added to the queue:

$$queue = [eq_1, eq_4]$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & \boxed{1} & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & \boxed{0} & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & \boxed{0} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \boxed{1} & 1 & 1 \end{pmatrix}$$

Figure 3: Column 6

We take the first equation of the queue, eq_1. This equation is rewritten as $x_1 = x_5 \cdot x_6$, $x_5 = x_1/x_6$ and $x_6 = x_1/x_5$. Because we know $min(x_1)$ and $max(x_6)$, the second formula provides more information:

$$min(x_5) = \frac{2}{20} = 0.1$$

The known values now are:

- $min(x_1) = 2$
- $min(x_2) = 2$
- $min(x_5) = 0.1$
- $max(x_6) = 20$

Looking in column 5 of the matrix gives us the new queue:

$$queue = [eq_1, eq_2, eq_4]$$

Notice the new equations are inserted before the longer equation 4. Equation 1 provides no new information, so it is removed from the queue and we move on to the next element: equation 2. Because we know $min(x_1)$ and $min(x_5)$ this formula provides:

$$min(x_4) = 2 \cdot 0.1 = 0.2$$

Adding this to the known values and looking in column 4 of the matrix provides:

- $min(x_1) = 2$
- $min(x_2) = 2$
- $min(x_4) = 0.2$
- $min(x_5) = 0.1$
- $max(x_6) = 20$

$$queue = [eq_2, eq_4]$$

After checking every rewritten formula of equation 2 and afterwards equation 4, our queue is empty, so no other changes may occur. The above values are our final values and the solver is finished. New input may be provided by the user or the program. Here we conclude our example.

4 Results

We had no practical problem to test our program on, so we provided a synthetic test input. We first derived a system of equations from the equations used in GenPres, so we got a coherent system of equations. We then provided our own test input. We tested our prototype of the original solver (for the code, see appendix B) and the new solver (for the code, see appendix A).

In our test case no sum or product of lists has to be calculated. Postponing this calculation was an enhancement we presented in section 3.1. We have implemented this concept in our prototype, but we found this did not improve the speed of the program significantly. For more information, see section 5.

Below, we present the input and the results.

4.1 Sample Cases

Equations:

1. $x_1 = x_2 \cdot x_3$
2. $x_4 = x_1 \cdot x_5$
3. $x_6 = x_7 \cdot x_8$
4. $x_1 = x_7 \cdot x_9$
5. $x_4 = x_7 \cdot x_{10}$
6. $x_2 = x_7 \cdot x_{11}$
7. $x_9 = x_{12} + x_{13} + x_{14}$
8. $x_6 = x_{15} + x_{16} + x_{17} + x_{18}$

Input values:

- $max(x_1) = 30$
- $min(x_2) = 5$
- $min(x_4) = 4$
- $max(x_7) = 20$
- $max(x_{10}) = 35$
- $max(x_9) = 1$
- $max(x_{11}) = 10$

- $unit(x_7) = 0.2$
- $max(x_9) = 1$
- $max(x_6) = 30$
- $unit(x_4) = 10$
- $max(x_{12}) = 0.3$
- $max(x_{13}) = 0.1$
- $min(x_{13}) = 0.1$
- $max(x_{16}) = 7$
- $max(x_{17}) = 3$
- $max(x_{18}) = 5$
- $min(x_{12}) = 0.2$
- $min(x_{15}) = 2$
- $min(x_{14}) = 0.2$
- $min(x_{17}) = 3$
- $min(x_{18}) = 1$
- $min(x_{16}) = 0.4$

4.2 Original solver program

We find that especially equations 7 and 8 leads to speed loss in the original solver. These equations contain variables that only appear there. Changes in these variables generally do no lead to changes in many other equations. Because for every new restriction the original solver checks each equation, many unnecessary equations are checked. These kinds of sum equations are common in the GenPres program.

The original solver program checked 325 equations during this input and this cost a total of 2688 milliseconds.

4.3 New solver program

The queue worked well on this input. By checking only required equations, significantly fewer equations had to be checked.

The new solver program checked 93 equations and this cost a total of 842 milliseconds.

In conclusion, the new program checked $\frac{325}{93} \approx 3.5$ times less equations and it took $\frac{2688}{842} \approx 3.2$ times less time. Please note the calculation time is dependent on many factors, like computing power, the way the prototype was written, etc.

```
The final values are:
Variable 1 has minimum 0.3 and maximum 20
Variable 2 has minimum 5 and maximum 200
Variable 3 has minimum 0.0015 and maximum 4
Variable 4 has minimum 10 and maximum 700
Variable 5 has minimum 0.5 and maximum 2333.33333333333
Variable 6 has minimum 4.6 and maximum 30
Variable 7 has minimum 0.6 and maximum 20.2
Variable 8 has minimum 0.227722772277228 and maximum 50
Variable 9 has minimum 0.5 and maximum 1
Variable 10 has minimum 0.495049504950496 and maximum 35
Variable 11 has minimum 0.25 and maximum 10
Variable 12 has minimum 0.2 and maximum 0.3
Variable 13 has minimum 0.1 and maximum 0.1
Variable 14 has minimum 0.2 and maximum 0.6
Variable 15 has minimum 0.2 and maximum 15
Variable 16 has minimum 0.4 and maximum 7
Variable 17 has minimum 3 and maximum 3
Variable 18 has minimum 1 and maximum 5
325 equations were checked by the original solver and 93 equations by the new solver
The original solver took 2688 milliseconds and the new solver 842 milliseconds.
```

Figure 4: The final results

5 Recommendations

We'd like to conclude with a few recommendations for further improvement and future work.

As mentioned in section 2.1, if a list of a variable is set, the possible values of its basic unit must be bound to divisors of at least one element of that list. This is to assure that the list will not become empty if a basic unit is chosen. If the basic unit divides no element of the list, it becomes empty. This means there are no more possible values for the variable. We have not implemented this in our prototype yet, so this will need to be further explored.

As mentioned in section 2.2, if in a sum equation the left-hand side variable y has a set maximum, this has implications for the right-hand side variables. Their sum may never exceed this maximum, so their maximum must be restricted. We can't just set their maximum to $max(y)$, for if this maximum is chosen, all other values would have to be zero. Note that values always had to be positive and non-zero. One thing we can do is set their maximum to $max(y)$ minus the sum of all known minima of the right-hand side variables. Then at least these variables can have a possible value, must this maximum be chosen. Still, there's the problem of the variables that have no minimum set yet. A possible solution would be to initialize every value with a certain minimum above zero. This may depend on the variable. For example, in practice this minimum may be the measurement accuracy of a certain measurement system.

In section 3.1 we suggested to improve the solver program by postponing the calculation of a product or sum. We implemented this concept in our prototype as followed: in the original solver we check if a sum or product can be calculated at the beginning of the solver method (see appendix B, lines 8-15) and in the new solver we do this at the end of the solver method (see appendix A, lines 59-66). However, this offered no significant improvement in the speed of the prototype program. This is because we call the solver method after each modification. While running the solver method once, lists may not be restricted significantly. For example, we call the solver method after changing a unit number. In this case, no large changes may occur at all. Changing the unit number may induce small changes, though, so we do have to call the solver method. In general, the calculation is only needed as output to the user. By postponing the calculation as long as possible, longer than we did in our prototype, we believe an additional great amount of calculation time can be saved.

References

[1] Maat B, Au YS, Bollen CW, van Vught AJ, Egberts ACG, Rademaker CMA.
Clinical pharmaceutical interventions induced by electronic medication prescriptions in a childrens hospital: nature, frequency and determinants
PW Wetenschappelijk Platform. 2011;5:a1122.

Appendices

As appendices we will include the code written in C# to test the new program. In some places in the code a global variable from the class Program is used. Program.eqs is our list of equations and Program.matrix is the matrix derived from these equations. Please note we use the double type to represent values, but rounding can be risky and cause errors. To avoid rounding errors, the original code, which is written in F#, uses the type *BigRational*.

A Code of prototype new solver

```
    public static void NewSolve(List<Equation> tocheckRef)
    {
        List<Variable> changedvars = new List<Variable>();
        List<Equation> tocheck = new List<Equation>();
        tocheck.AddRange(tocheckRef);
        Equation current = tocheck.FirstOrDefault();
//If the queue is empty, done
        if (current == null) return;
        else if (current.GetType().Equals(typeof(ProductEquation)))
        {
            changedvars = checkProductMinMax(ref current);

            tocheck.Remove(current);
//If the equation doesn't change, go on and check the rest of the queue
            if (changedvars.Count() == 0) { NewSolve(tocheck); }
//If it does change, add all equations in which the changed variable appears to the queue
//in a sorted way and without duplicates, and check the new queue
            else
            {
                foreach (Variable changedvar in changedvars)
                {
                    List<Equation> changedeqs = Function.getEquations(changedvar, Program.eqs);
                    foreach (Equation changedeq in changedeqs)
                    {
                        if (!tocheck.Contains(changedeq))
                        {
                            Function.InsertionSort(changedeq, tocheck);
                        }
                    }
                }
                NewSolve(tocheck);
            }
        }
        else if (current.GetType().Equals(typeof(SumEquation)))
        {
            changedvars = checkSumMinMax(ref current);

            tocheck.Remove(current);
//If the equation doesn't change, go on and check the rest of the queue
            if (changedvars.Count() == 0) { NewSolve(tocheck); }
//If it does change, add all equations in which the changed variable appears to the queue
//in a sorted way and without duplicates, and check the new queue
            else
            {
                foreach (Variable changedvar in changedvars)
                {
                    List<Equation> changedeqs = Function.getEquations(changedvar, Program.eqs);
                    foreach (Equation changedeq in changedeqs)
                    {
                        if (!tocheck.Contains(changedeq))
                        {
                            Function.InsertionSort(changedeq, tocheck);
                        }
                    }
                }
                NewSolve(tocheck);
            }
        }
//Check for each equation if the product or sum can be calculated
        foreach (Equation eq in Program.eqs)
        {
            if (eq.GetType().Equals(typeof(ProductEquation)))
                eq.calcproduct();
            if (eq.GetType().Equals(typeof(SumEquation)))
                eq.calcadd();
```

```
66          }
        }
```

B Code of prototype original solver

```
        public static void OriginalSolve(List<Equation> eqtocheckref, List<Equation> eqchecked)
2       {
            List<Variable> changedvars = new List<Variable>();
            List<Equation> eqtocheck = new List<Equation>();
            eqtocheck.AddRange(eqtocheckref);
            Equation current = eqtocheck.FirstOrDefault();
7
    //Check for each equation if the product or sum can be calculated
            foreach (Equation eq in Program.eqs)
            {
                if (eq.GetType().Equals(typeof(ProductEquation)))
12                  eq.calcproduct();
                if (eq.GetType().Equals(typeof(SumEquation)))
                    eq.calcadd();
            }

17  //If there are no more equations to check, done
            if (current == null) return;
            else if (current.GetType().Equals(typeof(ProductEquation)))
            {
                changedvars = checkProductMinMax(ref current);
22
    //If the equation doesn't change, go on and check the rest of the equations
                if (changedvars.Count() == 0) { eqtocheck.Remove(current);
                    eqchecked.Add(current); OriginalSolve(eqtocheck, eqchecked); }
    //If it does change, check all equations all over again
27              else OriginalSolve(eqchecked.Concat(eqtocheck).ToList(), new List<Equation>());
            }
            else if (current.GetType().Equals(typeof(SumEquation)))
            {
                changedvars = checkSumMinMax(ref current);
32
    //If the equation doesn't change, go on and check the rest of the equations
                if (changedvars.Count() == 0) { eqtocheck.Remove(current);
                    eqchecked.Add(current); OriginalSolve(eqtocheck, eqchecked); }
    //If it does change, check all equations all over again
37              else OriginalSolve(eqchecked.Concat(eqtocheck).ToList(), new List<Equation>());
            }
        }
```

C Code of helper functions

```
    //Checks what minimum and maximum values change and changes these for a product equation
        public static List<Variable> checkProductMinMax(ref Equation eq)
        {
            if (eq.GetType().Equals(typeof(SumEquation)))
5           { throw new System.ArgumentException("Not a product equation"); }
            List<Variable> varlist = new List<Variable>();
            double xmin = eq.Y.Min;
            double xmax = eq.Y.Max;
            double min1 = eq.Xs.ElementAt(0).Min;
10          double max1 = eq.Xs.ElementAt(0).Max;
            double min2 = eq.Xs.ElementAt(1).Min;
            double max2 = eq.Xs.ElementAt(1).Max;

            //The checks for the equation x=x1*x2
15          if (min1 != -1 && min2 != -1 && (xmin == -1 || min1 * min2 > xmin))
            { eq.Y.SetMin(min1 * min2); varlist.Add(eq.Y); }
            if (max1 != -1 && max2 != -1 && (xmax == -1 || max1 * max2 < xmax))
            { eq.Y.SetMax(max1 * max2); varlist.Add(eq.Y); }
            //The checks for the equation x1=x/x2
20          if (xmin != -1 && max2 != -1 && (min1 == -1 || xmin / max2 > min1))
            { eq.Xs.ElementAt(0).SetMin(xmin / max2); varlist.Add(eq.Xs.ElementAt(0)); }
            if (xmax != -1 && min2 != -1 && (max1 == -1 || xmax / min2 < max1))
            { eq.Xs.ElementAt(0).SetMax(xmax / min2); varlist.Add(eq.Xs.ElementAt(0)); }
            //The checks for the equation x2=x/x1
25          if (xmin != -1 && max1 != -1 && (min2 == -1 || xmin / max1 > min2))
            { eq.Xs.ElementAt(1).SetMin(xmin / max1); varlist.Add(eq.Xs.ElementAt(1)); }
            if (xmax != -1 && min1 != -1 && (max2 == -1 || xmax / min1 < max2))
            { eq.Xs.ElementAt(1).SetMax(xmax / min1); varlist.Add(eq.Xs.ElementAt(1)); }
            return varlist.Distinct().ToList();
30      }

    //Checks what minimum and maximum values change and changes these for a sum equation
        public static List<Variable> checkSumMinMax(ref Equation eq)
```

```
    {
        if (eq.GetType().Equals(typeof(ProductEquation)))
        { throw new System.ArgumentException("Not a sum equation"); }
        List<Variable> varlist = new List<Variable>();
        List<double> mins = new List<double>();
        List<double> maxs = new List<double>();
        List<double> knownmins;
        List<double> knownmaxs;
        double xmin = eq.Y.Min;
        double xmax = eq.Y.Max;
        for (int i = 0; i < eq.Xs.Count(); i++)
        {
            mins.Add(eq.Xs.ElementAt(i).Min);
            maxs.Add(eq.Xs.ElementAt(i).Max); //The lists of all values
        }
        knownmins = mins.Where(x => x != -1).ToList();
        knownmaxs = maxs.Where(x => x != -1).ToList(); //The lists of known values
//If no values are known, no variables will change
        if (knownmins.Count() == 0 && knownmaxs.Count() == 0) return varlist;
//The checks for the equation x=x1+x2 if at least on min value is known
        if (knownmins.Count() != 0 && (xmin == -1 || knownmins.Sum() > xmin))
        { eq.Y.SetMin(knownmins.Sum()); varlist.Add(eq.Y); }
//The checks for the equation x=x1+x2 if all max values are known
        if (knownmaxs.Count() == eq.Xs.Count() && (xmax == -1 || knownmaxs.Sum() < xmax))
        { eq.Y.SetMax(knownmaxs.Sum()); varlist.Add(eq.Y); }
//The checks for the equation xi = x - (x1 + x2 +...+ xi-1 + xi+1 +...+ xn)
        for (int i = 0; i < eq.Xs.Count(); i++)
        {
            List<double> minsWithouti = new List<double>();
            List<double> maxsWithouti = new List<double>();
            minsWithouti.AddRange(mins);
            maxsWithouti.AddRange(maxs);
            minsWithouti.Remove(mins[i]);
            maxsWithouti.Remove(maxs[i]);
            knownmins = maxsWithouti.Where(x => x != -1).ToList();
            knownmaxs = maxsWithouti.Where(x => x != -1).ToList();
            if (xmin != -1 && knownmaxs.Count() == eq.Xs.Count() - 1 && xmin - knownmaxs.Sum() > 0
                && (mins[i] == -1 || xmin - knownmaxs.Sum() > mins[i]) && (maxs[i] == -1 || xmin -
    knownmaxs.Sum() < maxs[i]))
            { eq.Xs.ElementAt(i).SetMin(xmin - knownmaxs.Sum()); varlist.Add(eq.Xs.ElementAt(i)); }
            if (xmax != -1 && knownmins.Count() == eq.Xs.Count() - 1 && xmax - knownmins.Sum() > 0
                && (maxs[i] == -1 || xmax - knownmins.Sum() < maxs[i]) && (mins[i] == -1 || xmax -
    knownmins.Sum() > mins[i]))
            { eq.Xs.ElementAt(i).SetMax(xmax - knownmins.Sum()); varlist.Add(eq.Xs.ElementAt(i)); }
        }
        return varlist.Distinct().ToList();
    }

//Creates a matrix from a list of equations that shows in which equation which variable appears
    public static int[,] ToMatrix(List<Variable> vars, List<Equation> eqs)
    {
        int numbereq = eqs.Count();
        int numbervars = vars.Count;
        int[,] matrix = new int[numbereq, numbervars];
        for (int i = 0; i < numbereq; i++)
        {
            foreach (Equation eq in eqs)
            {
                foreach (Variable var in vars)
                {
                    if (eq.Contains(var.Num)) matrix[eq.Num, var.Num] = 1;
                    else matrix[eq.Num, var.Num] = 0;
                }
            }
        }
        return matrix;
    }

//Returns the equations in which the given variable appears
    public static List<Equation> getEquations(Variable var, List<Equation> eqs)
    {
        List<Equation> result = new List<Equation>();
        for (int i = 0; i < eqs.Count(); i++)
        { if (Program.matrix[i, var.Num] == 1) result.Add(eqs[i]); }
        return result;
    }

//Caculates all possible combinations of sums without duplicates given two variables
    public static List<double> Addlists(List<double> var1, List<double> var2)
    {
        List<double> result = new List<double>();
        foreach (double val1 in var1)
        {
            foreach (double val2 in var2)
            {
                if (!result.Contains(val1 + val2))
                    result.Add(val1 + val2);
            }
```

```
        }
        return result;
    }

//Caculates all possible combinations of products without duplicates given two variables
    public static List<double> Productlists(List<double> var1, List<double> var2)
    {
        List<double> result = new List<double>();
        foreach (double val1 in var1)
        {
            foreach (double val2 in var2)
            {
                if (!result.Contains(val1 * val2))
                    result.Add(val1 * val2);
            }
        }
        return result;
    }

//Finds the lowest number above arg that is divisible by mult
    public static double Round(double arg, double mult)
    {
        double i = mult;
        while (i < arg) i += mult;
        return i;
    }

//Inserts an equation sorted by the number of variables
    public static List<Equation> InsertionSort(Equation elem, List<Equation> queue)
    {
        int length = queue.Count();
        if (length == 0)
            queue.Add(elem);
        for (int i = 0; i < length; i++)
        {
            if (queue.ElementAt(i).Xs.Count() > elem.Xs.Count())
            { queue.Insert(i, elem); break; }
            else if (i == queue.Count() - 1)
                queue.Add(elem);
        }
        return queue;
    }
```

D Code of variable and equation classes

```
public class Variable
{
    public double Min { get; set; }        //The minimum
    public double Max { get; set; }        //The maximum
    public double Unit { get; set; }       //The basic unit
    public List<double> List { get; set; }//The list of possible values
    public int Num { get; set; }           //The number indentifying the variable
    public Variable(double min, double max, double unit, int i)
    {
        Min = min;
        Max = max;
        Unit = unit;
        List = new List<double>();
        Num = i;
    }

    public Variable(List<double> list, int i)
    {
        Min = list.Min();
        Max = list.Max();
        Unit = -1;
        List = list;
        Num = i;
    }

    public Variable(int i)
    {
        Min = -1;
        Max = -1;
        Unit = -1;
        List = new List<double>();
        Num = i;
    }

    public void SetMax(double max)
    {
//First, check that the maximum is not set yet or that it's a stricter maximum than the previous one
        if (this.Max == -1 || max < this.Max)
        {
```

```
//If the basic unit and minimum are already known,
//add the list with as minimum the lowest multiple of the basic unit above the minimum
//and as maximum the highest multitude of the basic unit
            if (this.Max == -1 && this.Unit != -1 && this.Min != -1 && this.List.Count() == 0)
            {
                double i = Function.Round(this.Min, this.Unit);
                this.Min = i;
                while (i <= max)
                { this.List.Add(i); i += this.Unit; }
                this.Max = i;
            }
//If the basic unit is already known,
//add the list with as minimum the basic unit and as maximum the highest multitude of the basic unit
            else if (this.Max == -1 && this.Unit != -1)
            {
                double i = this.Unit;
                this.Min = i;
                while (i <= max)
                { this.List.Add(i); i += this.Unit; }
                this.Max = i;
            }
            //If the list is already set, adapt its values to the new maximum
            else if (this.List.Count() != 0)
            {
                this.List = this.List.Where(x => x <= max).ToList();
                this.Max = this.List.Max();
            }
            else this.Max = max;
        }
    }

    public void SetMin(double min)
    {
//First, check that the minimum is not set yet or that it's a stricter minimum than the previous one
        if (this.Min == -1 || min > this.Min)
        {
//If the list is already set, adapt its values to the new minimum
            if (this.List.Count() != 0)
            {
                this.List = this.List.Where(x => x >= min).ToList();
                this.Min = this.List.Min();
            }
            else this.Min = min;
        }
    }

    public void SetUnit(double unit)
    {
//If the maximum and minimum are already known,
//add the list with as minimum the lowest multiple of the basic unit above the minimum
//and as maximum the highest multitude of the basic unit
        if (this.Unit == -1 && this.Max != -1 && this.Min != -1 && this.List.Count() == 0)
        {
            double i = Function.Round(this.Min, unit);
            this.Min = i;
            this.Unit = unit;
            while (i <= this.Max)
            { this.List.Add(i); i += this.Unit; }
            this.Max = i;
        }
//If the maximum is already known,
//add the list with as minimum the basic unit and as maximum the highest multitude of the basic unit
        else if (this.Unit == -1 && this.Max != -1 && this.List.Count() == 0)
        {
            double i = this.Unit;
            this.Min = i;
            this.Unit = unit;
            while (i <= this.Max)
            { this.List.Add(i); i += this.Unit; }
            this.Max = i;
        }
//If the list is already set, adapt its values to the new basic unit
        else if (this.List.Count() != 0)
        {
            this.Unit = unit;
            this.List = this.List.Where(x => x % unit == 0).ToList();
            this.Min = this.List.Min();
            this.Max = this.List.Max();
        }
        else this.Unit = unit;
    }

    public void SetList(List<double> list)
    {
        double min = list.Min();
        double max = list.Max();
        this.List = list;
//For the minimum check if it is already set and if the new list needs to be restricted
```

```
            if (this.Min == -1) this.Min = min;
            else
            {
                if (this.Min <= min) this.Min = min;
                else
                {
                    this.List = this.List.Where(x => x >= this.Min).ToList();
                    this.Min = this.List.Min();
                }
            }
        //For the maximum check if it is already set and if the new list needs to be restricted
            if (this.Max == -1) this.Max = max;
            else
            {
                if (this.Max >= max) this.Max = max;
                else
                {
                    this.List = this.List.Where(x => x <= this.Max).ToList();
                    this.Max = this.List.Max();
                }
            }
        }
    }

    public class Equation
    {
        public Variable Y { get; set; }        //The left-hand side variable
        public List<Variable> Xs { get; set; }//The right-hand side variable
        public int Num { get; set; }           //The number identifying the equation

        public Equation(Variable y, Variable[] xs, int i)
        {
            Y = y;
            Xs = new List<Variable>();
            Xs.AddRange(xs);
            Num = i;
        }

        public bool Contains(int i) //Checks if a equation contains variable i
        {
            if (this.Y.Num == i) return true;
            foreach (Variable var in this.Xs)
                if (var.Num == i) return true;
            return false;
        }

    //If it is possible, calculate the sum of the right-hand side variables of this equation
        public void calcadd()
        {
            bool condition = true;
            List<double> result = new List<double>();
    //Only calculate when all variables of the equation have non-empty lists
            for (int i = 0; i < this.Xs.Count(); i++)
            {
                if (Xs.ElementAt(i).List.Count() == 0)
                    condition = false;
            }
            if (condition)
            {
    //Calculate all possible sums of the right-hand side variables
                List<List<double>> lists = this.Xs.Select(x => x.List).ToList();
                result = lists.Aggregate((x, y) => Function.Addlists(x, y));
    //Set this as the left hand variable
                this.Y.SetList(result);
            }
        }

    //If it is possible, calculate the product of the right-hand side variables of this equation
        public void calcproduct()
        {
            bool condition = true;
            List<double> result = new List<double>();
    //Only calculate when all variables of the equation have non-empty lists
            for (int i = 0; i < this.Xs.Count(); i++)
            {
                if (Xs.ElementAt(i).List.Count() == 0)
                    condition = false;
            }
            if (condition)
            {
    //Calculate all possible products of the right-hand side variables
                List<List<double>> lists = this.Xs.Select(x => x.List).ToList();
                result = lists.Aggregate((x, y) => Function.Productlists(x, y));
    //Set this as the left hand variable
                this.Y.SetList(result);
            }
        }
    }
```

```
public class ProductEquation : Equation
{
    public ProductEquation(Variable x, Variable[] xs, int i)
        : base(x, xs, i)
    {
    }
}

public class SumEquation : Equation
{
    public SumEquation(Variable x, Variable[] xs, int i)
        : base(x, xs, i)
    {
    }
}
```