Chapter 5

Implementation

MUSAC started as a small project. But it quickly grew when it turned out to be feasible. Already in the very first days of its implementation it became obvious that a lot of programming would be needed in order to satisfy at least some of all wishes. We decided that the entire program must be divided into several functional parts. One big part would be the *user interface* together with the *database* storing almost all required data and finally there would be one big functional unit called the *calculator module* that performs the calculations based on the descriptions entered via the user interface. Luckily we found very competent partners for implementing the user interface. I would like to express my special thanks to the people at *creasoft ag*[1] in Liechtenstein for their cooperation at this point.

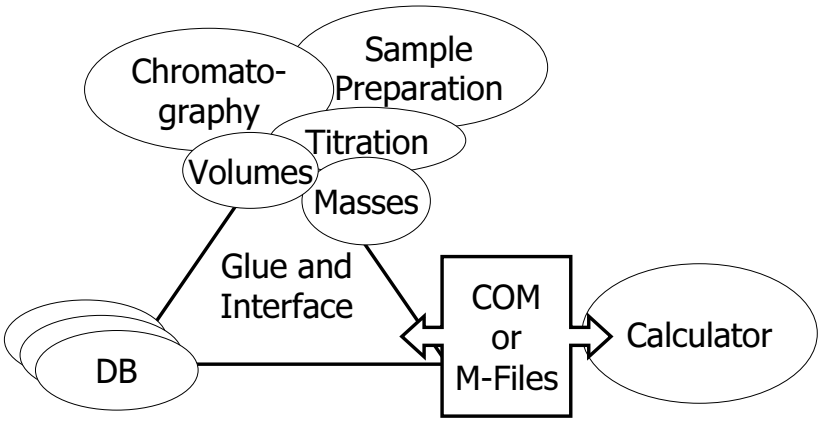
5.1 General Architecture

At the beginning it was not clear at all whether the user interface or the calculator should control the access to the database. The calculator could very well look up any default value in the database if needed. On the other hand, if the user interface should suggest a viable value to the user it must look it up somehow in the database as well. Later it turned out that the description of the measurement procedure as sent to the calculator can elegantly

[1] `http://www.creasoft.li`

function as a *protocol* of the executed measurement. With this additional requirement in mind the answer to the question "who should control access to the database" is now naturally clear: Imagine the protocol of a measurement done a year ago was reused and sent to the calculator anew. The calculator would now look up some missing values in its current database exactly as it did a year ago. First of all, it would be a funny kind of "protocol" not containing all needed values, and second the calculator, based on the updated database, might look up missing values totally different now than at earlier times, since the state of the database will have changed during that last year. Therefore the user interface should be in duty of maintaining the data repository. The calculator on the other hand can and indeed does rely on the fact that all needed data is supplied by the user interface.

Figure 10 Schematic Architecture of MUSAC



Thus the final architecture can be depicted as in Figure 10. The central glue code combines various modules. There are modules that know about volumetric measurements, about titration, sample preparation an so forth. We call them *instrument modules*. All these instrument modules are governed by the main program, the central glue code. It is obvious that various information must flow in both directions to and from the instrument modules, to and from the data base, and finally to and from the calculator module. The instrument modules together with the glue code form the *user interface*

as experienced on the screen of your computer. The responsibility of the instrument modules is to render the relevant information and/or questions into the entire user interface. The responsibility of the glue code is on one hand to channel informations to and from the separate modules and on the other hand to provide many basic functionalities on which the other modules can rely.

The communication between an instrument module's user interface and the database is for example entirely done under the regime of the glue code: one of its data object is encapsulating the database. Upon request by the user interface it issues the relevant SQL-statements to the database, prepares the received data and returns the answer to the instrument module. In a similar way the different user interface components communicate with the calculator module via the central glue code: In principle the glue code is generating one big M-file describing an entire measurement procedure in tight collaboration with the instrument modules. Imagine this as a collage of M-fragments that are stored, maintained and collected by the instrument modules. It turned out that sending one big M-file was a very — in fact too much of an — inflexible way of communicating, but nevertheless this was a good starting point. The calculator module is now wrapped within Microsoft's COM technology which provides very flexible mechanisms for communication between separate modules in both directions. So that now the big M-file is not explicitly generated by the glue code but rather it is constructed by "interactively" sending M-fragments to the calculator module. Errors and problems that may arise due to such a fragment can give rise to more detailed questions or informations to the user via the glue code and the instrument module.

5.2 COM Interface

The Calculator Module can be viewed as follows: It maintains an internal structure representing the measurement description. The internal structure can be manipulated and evaluated in various ways using the provided methods. This is all done by calling methods through the module's interface. The interface for the Calculator COM Module is as follows:

```
interface ICalculator : IDispatch {
   HRESULT ErrorCode([out, retval] LONG *pVal);
   HRESULT StatusMsg([out, retval] BSTR *pbstrMsg);
```

```
    HRESULT AddRule([in] BSTR bstrRule);
    HRESULT AddRules([in]VARIANT vRules);
    HRESULT Clear();
    HRESULT RemoveSymbol([in] BSTR sym);
    HRESULT ReadFromFile([in] BSTR FileName);
    HRESULT WriteToFile([in] BSTR FileName);
    HRESULT GetSymbols([in, out] VARIANT *vRet);
    HRESULT GetTree([in] BSTR SymbolName,
                    [in, out] BSTR *Tree);
    HRESULT CalcUreal([in] BSTR SymbolName,
                      [in, out] VARIANT* pUreal );
    HRESULT CalcMonteCarlo([in] BSTR SymbolName,
                           [in, out] VARIANT *res);
    HRESULT CalcISO([in] BSTR SymbolName,
                    [in, out] VARIANT *pUreal);
    HRESULT UnitCheck([in] BSTR lhs, [in] BSTR rhs,
                      [in, optional] BSTR unit);
    HRESULT Check([in, optional] BSTR SymbolName);
    HRESULT Validate([in] BSTR bstrFileIn,
                     [in, optional] BSTR bstrFileOut);
    HRESULT MusDir([out, retval] BSTR *pVal);
    HRESULT MusDir([in] BSTR newVal);
    HRESULT SelfTest([out, retval] BOOL *pVal);
    HRESULT SelfTest([in] BOOL newVal);
    HRESULT DirDelim([out, retval] BSTR *pVal);
    HRESULT DirDelim([in] BSTR newVal);
    HRESULT LastUnit([out, retval] BSTR *pVal);
    HRESULT FmtWithOp([out, retval] BOOL *pVal);
    HRESULT FmtWithOp([in] BOOL newVal);
    HRESULT FmtEng([out, retval] LONG *pVal);
    HRESULT FmtEng([in] LONG newVal);
    HRESULT LibVersion([out, retval] BSTR *pbstrVersion);
};
```

It is not the intention of this text to explain each and every line of this interface definition. However, let us still take a closer look at some random details in order to get the glimpse of an idea of how the calculator module may be driven by an other program adhering to the COM standards.

The interface of the calculator module `ICalculator` inherits from the interface `IDispatch` which offers the basic functionality to the COM framework in order to take part in the COM framework at all. The data types

HRESULT, BSTR, VARIANT and LONG are defined in the COM framework. HRESULT is used to indicate success or failure and type of failure of a method call. BSTR is the COM type name for character strings, and the type VARIANT is the infamous type that can stand for almost anything. User data is passed via formal parameters in the usual way. If it wasn't for those tags in square brackets, the IDL (interface description language) resembles very much an ordinary C++ class definition. The square brackets enforce the notion of *how* an argument or a method should be used.

File access There are few methods offered through which the calculator module accesses files. There is one (ReadFromFile) that reads in an M-file, thereby replacing the internal structure by a new one representing the contents of the input file. On the other hand there is a method to write the current internal structure into a file (WriteToFile). Finally there is the method Validate which reads a file, executes it, and writes all the result along with some diagnostic output to the output file. It is a relict of the entry point into the calculator from early development stages. Now, it serves the purpose of checking the entire calculator module. Developers, after having added a new feature, simply compare the output file with an expected output. If they match then the newly added feature didn't break any existing functionality. This technique, taken to its extreme, is called *Extreme Programming*. It is promoted by several authorities e.g. [8], and well worth a closer look by the way. In the same spirit the method SelfTest toggles whether the module should perform some self test in certain situations. Anyhow, these file accessing methods provide some intelligence in handling files, file suffixes and default directories not really worth being discussed any further at this place.

M-Code Manipulation Routines for manipulation of the internal data structure are providing some features, such as adding one or more M-statements (AddRule[s]), removing an M-symbol (RemoveSymbol) or clearing (Clear) the entire internal state.

Querying the internal state There are some methods to check and query certain things about the current internal state of the calculator module. E.g. the method UnitCheck compares up to three symbols and determines whether they evaluate to the same measurement unit. This can be handy for

the driving user interface to check users input for measurement unit consistency. The method `Check` performs two tests. First it examines whether the symbol in question together with all its dependencies are defined at all, and second it checks whether all operations needed for an evaluation are legal and well defined too. There are few more functions like `GetSymbols` returning a list of all known i.e. defined symbols, or the method `GetTree` returns depending on the input parameters practically the entire internal state of the calculator module. And finally there is the method `WriteToFile` which writes the internal state of the calculator module into a file. Later, this file can be read in to put the calculator module into the exact same state as it was at the time when `WriteToFile` was issued. This can be handy for the user interface to construct a kind of a protocol about the current session.

M-Code evaluation There are (currently) three methods implemented to evaluate a symbol. The methods `CalcUreal`, `CalcIso` and `CalcMonteCarlo` evaluate the given symbol in a manner described in Section 2.2.4, 2.2.3 and Section 2.2.5 respectively. The first applies the simple rules on page 27. It is maintained in order to be able to compare the results of the other two methods with this oversimplified method. However, it is not always necessary to use the more complicated methods as described in Section 2.2.4.

All three methods accept one symbol to be evaluated at a time. The returned result is a pair of numbers in the case of `CalcUreal` and `CalcIso` representing mean and variance of the result, and it is a bunch of numbers representing mean, variance, minimum, maximum and histogram data in the case of `CalcMonteCarlo`.

Properties Finally there are a variety of properties for the calculator module. They are giving the user the flexibility to set default behaviors such as directory to read from and write to on one hand. On the other hand they are the means to retrieve additional information about the calculator module such as the last result's measurement unit or the last error that might have caused an infeasible result.

5.3 Several Modules

5.3.1 SI-Units

In the MUSAC-project numbers are often typed with a *unit* such as m, °C, mol/l etc. Defining a set of operations and procedures that handle all cases arising in this context is not a trivial task. There are 7 different SI units, i.e. there are 7 fundamentally different entities that can be measured and form the basis of the SI-System for Units [16]. Additionally, there are many accepted units which are combinations of those basic units. Furthermore the SI-System allows to prefix any unit with an engineering prefix encoding some power of 10. Table 6 on page 73 lists a number of SI accepted units and prefixes. Handling this variety automatically rises among others the following questions: How can the unit type be encoded elegantly and efficiently? How should the prefix be encoded? A prefix for 1 l which is equal to $1\,\mathsf{dm}^3$ must be raised to the correct power implicitly. And last but not least there might be different possibilities to print out a given unit with respect to the accepted units.

In the following section we are going to discuss some of these considerations to some details.

Representation of a Unit In the MUSAC-system a measurable quantity is expressed by a number and a symbol within square brackets representing the quantity type, i.e. the *unit of the quantity*. The system of SI of units is based on 7 base units [16]. All measurable quantities can be expressed using combinations of those 7 base units. For convenience there are a number of combined units allowed by the SI-System. Thus, any unit can be represented as a vector of powers of base units. Let's call it an *SI-vector* $\mathbf{u}$. It's a 7-dimensional vector and consists only of integer entries since there are no reasonable units with fractional powers of SI base units: $\mathbf{u} \in \mathbb{Z}^7$. The 7 base vectors are each encoding the power of one base SI unit: $\boldsymbol{u}_1 = (1,0,0,0,0,0,0)^\top \mathrel{\hat{=}} 1\,\mathsf{s}$, $\boldsymbol{u}_2 \mathrel{\hat{=}} 1\,\mathsf{m}$, $\boldsymbol{u}_3 \mathrel{\hat{=}} 1\,\mathsf{g}$, $\boldsymbol{u}_4 \mathrel{\hat{=}} 1\,\mathsf{s}$, $\boldsymbol{u}_5 \mathrel{\hat{=}} 1\,\mathsf{A}$, $\boldsymbol{u}_6 \mathrel{\hat{=}} 1\,\mathsf{mol}$ and $\boldsymbol{u}_7 \mathrel{\hat{=}} 1\,\mathsf{cd}$. With this definition it is clear that a multiplication or a division of two units is represented as the addition or the subtraction of the two SI-vectors respectively, for example

$$\begin{aligned} 1\,\mathsf{Hz} * 1\,\mathsf{m} &\mathrel{\hat{=}} -\boldsymbol{u}_1 + \boldsymbol{u}_2 = \boldsymbol{u} \\ &= (-1,1,0,0,0,0,0)^\top. \end{aligned} \tag{5.1}$$

Prefixes are transformed into a power of 10 according to Table 6 and stored in a separate variable $e \in \mathbb{Z}$, and finally the mantissa of the unit is stored in a third variable $m \in \mathbb{R}^d$, in the context of measurement uncertainty the mantissa would also encode the standard deviation ($d = 2$) maybe even higher order moments ($d > 2$). In order to avoid unnecessary complications we use $d = 1$ in this section. Any unit number is thus represented as a *Unit-Triplet* $\mathcal{U} = \langle \boldsymbol{u}, e, m \rangle$.

Algorithm 8 Construction of a Unit-Triplet

function $\langle m, e, \boldsymbol{v} \rangle = \mathsf{Constructor}(m, s)$
— m: mantissa, s: prefix-unit-string —
$\langle p, u \rangle \leftarrow \mathsf{split}(s)$ — p: *prefix power*, u: *unprefixed unit string* —
$\langle n, e, \boldsymbol{v} \rangle \leftarrow \mathsf{lookup}(u)$ — *consult list of known units* —
$e \leftarrow e + p$ — *multiply prefix . . .*
$m \leftarrow n * m$ *. . . and mantissa* —

Consider the construction of $\mathcal{U}$ for 0.45 ml. The unit string `ml` is split into a factor $p = -3$ for the prefix "m" and a string $u =$"l" which is looked up in the table of known units yielding $n = 1$, $e = -3$ and $\boldsymbol{v} = (3, 0, 0, 0, 0, 0, 0)^\top$. The factor p must now be multiplied by e, because e and p are powers they're added up, yielding $e \leftarrow e + p = -6$. The quantity 0.45 ml is now represented by the triplet $\langle (3, 0, 0, 0, 0, 0, 0)^\top, -6, 0.45 \rangle$. It could be normalized such that the mantissa m is within certain bounds, e.g. $1 \leq m < 10$. But it turns out that this is not needed. If and only if the mantissa and the exponent must be shifted in any direction, it will be done then *only*. This is another incarnation of the *principle of lazy evaluation*. It is considered good and efficient practice. Furthermore, it is proven to be correct, in the sense that no unnecessary work is done and yet no needed work is forgotten [32]. Algorithm 8 constructs a Unit-Triplet $\mathcal{U} = \langle m, e, \boldsymbol{v} \rangle$ given a mantissa m and a unit string s. The multiplication and division of two Unit-Triplets is pretty easy as can be seen in Algorithm 9.

Similar to the multiplication and division the addition and subtraction of two Unit-Triplets $\mathcal{X} = \langle m_x, e_x, \boldsymbol{v}_x \rangle$ and $\mathcal{Y} = \langle m_y, e_y, \boldsymbol{v}_y \rangle$ are straightforward. First, we have to check whether the unit vectors match $\boldsymbol{v}_x = \boldsymbol{v}_y$. Second, the two mantissae must be normalized such that $e_x = e_y$. Then the mantissae can be added or subtracted respectively. Algorithm 10 shows the exact procedure.

Algorithm 9 Multiplication of Two Unit-Triplets

function $\langle m_z, e_z, \boldsymbol{v}_z \rangle = \mathsf{Multiply}\big(\langle m_x, e_x, \boldsymbol{v}_x \rangle , \langle m_y, e_y, \boldsymbol{v}_y \rangle \big)$
$\quad — \mathcal{Z} \leftarrow \mathcal{X} * \mathcal{Y} —$
$\quad m_z \leftarrow m_x * m_y$
$\quad e_z \leftarrow e_x + e_y$
$\quad \boldsymbol{v}_z \leftarrow \boldsymbol{v}_x + \boldsymbol{v}_y$

Algorithm 10 Addition of Two Unit-Triplets

function $\langle m_z, e_z, \boldsymbol{v}_z \rangle = \mathsf{Add}\big(\langle m_x, e_x, \boldsymbol{v}_x \rangle , \langle m_y, e_y, \boldsymbol{v}_y \rangle \big)$
$\quad — \mathcal{Z} \leftarrow \mathcal{X} + \mathcal{Y} —$
$\quad$ **if** $\big(\boldsymbol{v}_x \neq \boldsymbol{v}_y\big)$ **then** throw(“Unit mismatch error”) **end if**
$\quad \boldsymbol{v}_z \leftarrow \boldsymbol{v}_x$
$\quad$ **if** $e_y \leq e_x$ **then**
$\quad\quad m \leftarrow m_x * 10^{e_x - e_y}$ *— normalize to lower exponent —*
$\quad\quad e_z \leftarrow e_y$
$\quad\quad m_z \leftarrow m + m_y$
$\quad$ **end if**
$\quad$ **if** $e_y > e_x$ **then**
$\quad\quad m \leftarrow m_y * 10^{e_y - e_x}$ *— normalize to lower exponent —*
$\quad\quad e_z \leftarrow e_x$
$\quad\quad m_z \leftarrow m + m_x$
$\quad$ **end if**

The reader might have noticed, that already the simple example in Equation (5.1) shows the problem of ambiguity for printing out a unit: should $\boldsymbol{u}$ be printed as 1 Hz m rather than 1 m/s? What about a unit like 0.3 m^2 should this be 3 hl/m or as 0.3 kl/m? And 1 g m/A s^3 could be printed as 1 N/C or 1 V/m.

The rules for "optimal print out" are arbitrary. But the following set of rules leads to very satisfying results.

1. Use as few separate units raised to the correct powers as possible, and

2. distribute the prefixes over the used units such that the resulting mantissa m is within $1 \leq m < 10^g$, using prefixes encoding powers $p \geq |g|$, where $g \in \{0, 1, 2, 3\}$ if possible.

The parameter g governs the use of the somewhat exotic prefixes like d for "dezi" (10^{-1}) or even da for "deka" (10^1). With $g = 2$ these previous prefixes are banned, but prefix h for 10^2 would still be allowed. Setting $g = 0$ inhibits use of prefixes in the output altogether. The first rule for nice printing is ambiguous. Both representations 1 N/C and 1 V/m use 2 units for the same quantity. In this case we prefer the latter, because V and m are more commonly used. This decision is again arbitrary, but remember that in Algorithm 8 we are maintaining a list of known units already for the construction of a Unit-Triplet. We will use the order of appearance in that list for preferring V/m over N/C. In this case m is the first one appearing in the list of known units and therefore its combination is preferred.

In the COM interface (see page 95) the property `FmtEng` controls the output format of units by means of the aforementioned parameter g. It is a number between $[-1 \ldots 3]$ representing basically the allowed unit prefixes. If `FmtEng` $= -1$ then no prefixes are used in the output of units and only the seven base SI units are used. If `FmtEng` $= 0$, other accepted SI units are used, `FmtEng` $= 1$ allows to use any prefix trying to shift the number into the range of $1.0 \ldots 9.9$ using , with `FmtEng` $= 2$ the intended range is $1.0 \ldots 99.9$ with the prefixes for 2 and 3 powers allowed and finally with `FmtEng` $= 3$ the range is $1.0 \ldots 999.9$ using the prefixes for powers of 3 only. It is of course not always possible to shift the mantissa by the allowed prefixes into the requested range. The order of fall back is `FmtEng` $= 1 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow -1$.

The task of printing out a Unit-Triplet $\langle m, e, \boldsymbol{v} \rangle$ is divided into two steps.

Algorithm 11 Decomposing an SI-Vector by Greedily Reducing the 1-norm

function $\langle \Lambda, \boldsymbol{Q} \rangle = \mathsf{Decompose}(\boldsymbol{v})$
  — Λ, $\boldsymbol{Q}$: Lists of λ and $\boldsymbol{q}$ —
  $\Lambda \leftarrow \emptyset$ — *List of optimal λ_i* —
  $\boldsymbol{Q} \leftarrow \emptyset$ — *List of optimal $\boldsymbol{q}_i$* —
  while $\|\boldsymbol{v}\|_1 > 0$ **do**
    $\boldsymbol{q}_{\mathrm{opt}} \leftarrow \boldsymbol{0}$ — *optimal $\boldsymbol{q}$ so far* —
    $\lambda_{\mathrm{opt}} \leftarrow 0$ — *optimal λ for $\boldsymbol{q}_{\mathrm{opt}}$* —
    $\omega_{\mathrm{opt}} \leftarrow \infty$ — *optimal $\|\boldsymbol{v} - \lambda_{\mathrm{opt}}\boldsymbol{q}_{\mathrm{opt}}\|_1$* —
    forall $\boldsymbol{q} \in$ List of known units **do**
      $(\lambda_{\min}, \lambda_{\max}) = \mathsf{FindRange}(\boldsymbol{v}, \boldsymbol{q})$ — *see Algorithm 12* —
      $\omega \leftarrow \infty$
      forall $\lambda_{\min} \le l \le \lambda_{\max}$ **do** — *Find optimal λ* —
        if $\|\boldsymbol{v} - l\boldsymbol{q}\|_1 < \omega$ **do**
          $\omega \leftarrow \|\boldsymbol{v} - l\boldsymbol{q}\|_1$
          $\lambda \leftarrow l$
        end if
      end forall
      if $\omega < \omega_{\mathrm{opt}}$ **then** — *Update optimal values* —
        $\omega_{\mathrm{opt}} \leftarrow \omega$
        $\lambda_{\mathrm{opt}} \leftarrow \lambda$
        $\boldsymbol{q}_{\mathrm{opt}} \leftarrow \boldsymbol{q}$
      end if
    end forall
    $\boldsymbol{v} \leftarrow \boldsymbol{v} - \lambda_{opt}\boldsymbol{q}_{\mathrm{opt}}$ — *Greedily reduce $\boldsymbol{v}$ by the optimal λ and $\boldsymbol{q}$* —
    $\Lambda \leftarrow \mathsf{Append}(\Lambda, \lambda_{\mathrm{opt}})$
    $\boldsymbol{Q} \leftarrow \mathsf{Append}(\boldsymbol{Q}, \boldsymbol{q}_{\mathrm{opt}})$
  end while

The first step is to find a solution for the following problem:

$$\min_{\lambda} \sum_{i=0}^{n} |\mathrm{sgn}(\lambda_i)| \tag{5.2}$$

subject to the conditions

$$\boldsymbol{v} = \sum_{i=1}^{n} \lambda_i \boldsymbol{q}_i \tag{5.3}$$

$$\boldsymbol{q}_i \in \text{List of known Units}$$

$$\lambda_i \in \mathbb{Z}. \tag{5.4}$$

Once the λ_i and $\boldsymbol{q}_i$ are found using Algorithm 11 or Algorithm 13, there remains to determine the prefixes e_i according to the rule 2 on page 102. This is done by Algorithm 15. Both algorithms return their solution in two lists Λ and $\boldsymbol{Q}$. Algorithm 11 finds the first solution that establishes

$$\left\| \boldsymbol{v} - \sum_{i=1}^{n} \lambda_i \boldsymbol{q}_i \right\|_1 = 0. \tag{5.5}$$

As stated above this solution is dependent on the order of the $\boldsymbol{q}_i$ stored in the list of known units. Algorithm 11 consists basically of two loops which are repeated until $\boldsymbol{v}$ is completely reduced according to Equation (5.5). The first and outer loop iterates over all known units $\boldsymbol{q}$, the second and inner loop finds the optimal λ for each $\boldsymbol{q}$. Optimal in the sense that for a given $\boldsymbol{q}$ the expression $\|\boldsymbol{v} - \lambda \boldsymbol{q}\|_1$ is minimized.

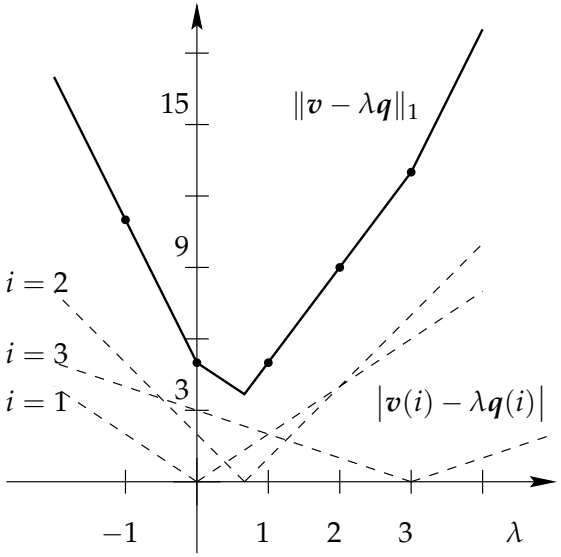
Algorithm 12 returns bounds for λ_{opt} which reduces $\|\boldsymbol{v} - \lambda \boldsymbol{q}\|_1$ maximally. That optimal λ lies within distinct bounds: for a given pair of $\boldsymbol{q}$ and $\boldsymbol{v}$ the element wise contribution to $\|\boldsymbol{v} - \lambda \boldsymbol{q}\|_1$ is 0 at $\lambda = \boldsymbol{v}(i)/\boldsymbol{q}(i)$. The contribution rises to both sides of that λ for each element i. Thus the optimal $\lambda \in \mathbb{Z}$ for $\|\boldsymbol{v} - \lambda \boldsymbol{q}\|_1$ lies between the minimum and the maximum of these element wise quotients, see Figure 11.

Put in other words, Algorithm 11 finds a solution for Equation (5.3) by greedily reducing that stated 1-norm. This solution is also fulfilling Condition (5.4). But the procedure will not necessarily find the optimum according to Condition (5.2). Consider the following example: The list of known units (which for this example are in $\mathbb{Z}^3$) is as follows:

Algorithm 12 Finding the Range for Optimal λ

function $\langle \lambda_{\min}, \lambda_{\max} \rangle = \mathsf{FindRange}(\boldsymbol{v}, \boldsymbol{q})$
— returns boundaries for λ with $\omega = \boldsymbol{v} - \lambda \boldsymbol{q} = \min$ —
$\lambda_{\min} \leftarrow \infty$
$\lambda_{\max} \leftarrow -\infty$
forall $1 \leq i \leq 7$ **do** — *Find bounds for optimal* λ —
if $\boldsymbol{q}(i) \neq 0$ **then**
$\lambda_{\min} \leftarrow \min\left(\boldsymbol{v}(i)/\boldsymbol{q}(i), \lambda_{\min}\right)$
$\lambda_{\max} \leftarrow \max\left(\boldsymbol{v}(i)/\boldsymbol{q}(i), \lambda_{\max}\right)$
end if
end forall
$\lambda_{\max} \leftarrow \lceil \lambda_{\max} \rceil$ — *Round up . . .*
$\lambda_{\min} \leftarrow \lfloor \lambda_{\min} \rfloor$ *. . . and down* —

Figure 11 Elementwise Contributions to $\left\| (0,2,3)^\top - \lambda (2,3,1)^\top \right\|_1$



Algorithm 13 Decomposing an SI-vector by Reducing the 1-norm and Searching the Best Next

function $\langle \Lambda, \boldsymbol{Q} \rangle = \mathsf{Decompose}(\boldsymbol{v})$
— Λ, $\boldsymbol{Q}$: Lists of λ and $\boldsymbol{q}$ solving (5.3) subject to (5.2) – (5.4)) —
$\Lambda \leftarrow \emptyset$ — *List of λ_i* —
$\boldsymbol{Q} \leftarrow \emptyset$ — *List of $\boldsymbol{q}_i$* —
$\mathcal{H} \leftarrow \emptyset$ — *Heap, ordering by Algorithm 14* —
while $\|v\|_1 > 0$ **do**
forall $\boldsymbol{q} \in$ List of known units **do**
$(\lambda_{\min}, \lambda_{\max}) = \mathsf{FindRange}(\boldsymbol{v}, \boldsymbol{q})$ — *see Algorithm 12* —
forall $\lambda_{\min} \le \lambda \le \lambda_{\max}$ **do**
$\mathsf{Push}\big(\mathcal{H}, \mathsf{Append}(\Lambda, \lambda), \mathsf{Append}(\boldsymbol{Q}, \boldsymbol{q}), \boldsymbol{v} - \lambda \boldsymbol{q}\big)$
— *push each sequence and remaining v onto heap and sort* —
end forall
end forall
$(\Lambda, \boldsymbol{Q}, \boldsymbol{v}) \leftarrow \mathsf{Pop}(\mathcal{H})$ — *fetch best triplet from heap* —
end while

$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{c}$	$\boldsymbol{d}$	$\boldsymbol{e}$
$\begin{pmatrix}1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\1\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\3\end{pmatrix}$

The vector $\boldsymbol{v} = (2,1,3)^\top$ will be reduced by Algorithm 11 via the following steps:

step number:		1		2		3	
$\boldsymbol{v}$:	$\begin{pmatrix}2\\1\\3\end{pmatrix}$	$\rightarrow$	$\begin{pmatrix}1\\1\\0\end{pmatrix}$	$\rightarrow$	$\begin{pmatrix}0\\1\\0\end{pmatrix}$	$\rightarrow$	$\begin{pmatrix}0\\0\\0\end{pmatrix}$
reduced by:		$\boldsymbol{q}_1 = \boldsymbol{e}$ $\lambda_1 = 1$		$\boldsymbol{q}_2 = \boldsymbol{a}$ $\lambda_2 = 1$		$\boldsymbol{q}_3 = \boldsymbol{b}$ $\lambda_3 = 1$	
1-norm ω_{opt}:	6		2		1		0

In other words, $\boldsymbol{q} = \boldsymbol{e} + \boldsymbol{a} + \boldsymbol{b}$, after all it is reducing the remaining 1-norm

stored in ω_{opt}. But consider the following reduction:

step number:		1		2	
$\boldsymbol{v}$:	$\begin{pmatrix}2\\1\\3\end{pmatrix}$	$\rightarrow$	$\begin{pmatrix}0\\0\\3\end{pmatrix}$	$\rightarrow$	$\begin{pmatrix}0\\0\\0\end{pmatrix}$
reduced by:		$\boldsymbol{q}_1 = \boldsymbol{d}$ $\lambda_1 = 1$		$\boldsymbol{q}_2 = \boldsymbol{c}$ $\lambda_2 = 3$	
1-norm ω_{opt}:	6		3		0

The second reduction is better in the sense of Condition (5.2)! But because the first iteration only reduces to $\omega_{\text{opt}} = 3$ instead of $\omega_{\text{opt}} \leq 2$ the second reduction will not be found by Algorithm 11. We alter Algorithm 11 in two ways in order to find the second reduction before the first one. The resulting Algorithm 13 is used in the MUSAC-program.

One modification is to keep track of *all* partial reductions considered so far instead of keeping only the best solution and its 1-norm ω_{opt}. To do so, we introduce a heap $\mathcal{H}$ storing pairs of reduction sequences $(\Lambda, \boldsymbol{Q})$. This heap is ordering the reduction sequences $(\Lambda, \boldsymbol{Q})$ by their length in first priority and by the remaining 1-norm in second priority. In fact this is done by calculating a niceness factor for each reduction sequence by

$$f_{\text{nice}}(\Lambda, \boldsymbol{Q}) = \mathsf{Length}(\Lambda) + \left\| \boldsymbol{v} - \sum_{i=1}^{n} \lambda_i \boldsymbol{q}_i \right\|_1 . \tag{5.6}$$

The advantage of this approach is that we can add a penalty for unwanted reductions: e.g. 1 Hz should be preferred over 1 1/s. Honestly, there are even more subtle ambiguities. To disambiguate between them we use the slightly more complicated Algorithm 14 for calculating the niceness of a $(\Lambda, \boldsymbol{Q})$ reduction. It is a symmetric expansion of Equation (5.6). Rather than just taking the length of the partial reduction in Λ, it calculates the expected length of the final reduction by adding the number of non zero elements of the remaining vector. Similarly not only the 1-norm of the remaining vector $\boldsymbol{v}$ is assessed but also the 1-norm of the partial reduction is taken into account. The function $\mathsf{Nz}(\cdot)$ returns the number of non zero elements and the expression $\mathsf{Nz}(\Lambda < 0)$ counts entries in Λ with $\lambda_i < 0$.

The second modification for Algorithm 11 stems from the observation that each λ between $\lambda_{\min}$ and $\lambda_{\max}$ must be considered and each respective

Algorithm 14 Calculation of Niceness of a Partial Unit Reduction

function $s = \mathsf{Niceness}(\Lambda, \boldsymbol{Q}, \boldsymbol{v})$
— s: niceness factor for reduction $(\Lambda, \boldsymbol{Q})$, $\boldsymbol{v}$: remaining unit vector —
$\alpha \leftarrow 1/2$ — *weight for expected 1-norm* —
$\beta \leftarrow 2$ — *weight for expected reduction length* —
$s \leftarrow \beta\big(\mathsf{Length}(\Lambda) + \mathsf{Nz}(\boldsymbol{v})\big)$ — *prefer short reduction* —
$s \leftarrow s + \alpha\big(\|\Lambda\|_1 + \|\boldsymbol{v}\|_1\big)$ — *prefer small exponents* —
if $\mathsf{Nz}(\Lambda < 0) \neq 0$ **and** $\mathsf{Nz}(\Lambda > 0) = 0$ **do**
$s \leftarrow s + 1/2(\alpha + \beta)$ — *add a penalty if reduction contains only negative powers* —
end if

reduction must be pushed onto the heap $\mathcal{H}$. The feasible λ lay still in the same range because any reduction step will also reduce the 1-norm beside the niceness factor. Additionally Condition (5.4) forces to sample the 1-norm at discrete points and it is well possible that there are two different λ yielding the same "minimal" 1-norm as can be seen in Figure 11, the 1-norm is equal to 5 at $\lambda = 0$ and $\lambda = 1$. Finally, Algorithm 13, a modification of Algorithm 11, is now returning the correct pair $(\Lambda, \boldsymbol{Q})$ by applying a "best next" search strategy.

Given a reduction $(\Lambda, \boldsymbol{Q})$ for a Unit-Triplet $\mathcal{U}$ Algorithm 15 returns a string u containing the prettily printed representation of $\mathcal{U}$ and a scale factor s by which the mantissa m_u of $\mathcal{U}$ must be multiplied to fit the representation in u. This splitting is done because it adds more flexibility. After calling $\mathsf{PrePrint}(\cdots, \mathcal{U})$ one can now decide to print only the unit string or one can decide to prepend the string with the mantissa number. In the latter case one must simply scale m_u. As insinuated on page 100 the type of a unit m_u is a perfect candidate for a *generic type*. In C++ they're called *templates*. And the MUSAC-program makes heavy use of this technique. That's another reason why the final scaling of the unit mantissa is left to the programmer in knowledge of the actual type of m and the needed work for this multiplication.

Algorithm 15 takes as its arguments a reduction $(\Lambda, \boldsymbol{Q})$ for v_u, a parameter g, and the Unit-Triplet $\mathcal{U}$ to be printed out prettily according to the rules on page 102. Parameter g controls the range for the printed mantissa and via Algorithm 16 the use of allowed prefixes. The function Apply executes the

Algorithm 15 Distributing Prefixes

function $\langle u, s\rangle = \mathsf{PrePrint}(\Lambda, \boldsymbol{Q}, g, \mathcal{U})$
— u: unit string with prefixes, s: scale factor for u, $\mathcal{U} = \langle m_u, e_u, \boldsymbol{v}_u\rangle$ —
$\langle m, e, \mathbf{0}\rangle \leftarrow \mathsf{Apply}(\Lambda, \boldsymbol{Q}, \mathcal{U})$ — *execute the reduction on* $\mathcal{U}$ —
$s \leftarrow {}^{m}/_{m_u}$ — *initialize scale factor* —
repeat
if $g \neq 0$ **do**
$x \leftarrow \left\lfloor {}^{\log(m)}/_{\log(10^g)} \right\rfloor$ — *exponent* x *shifts mantissa into range* $1 \leq m < 10^g$ —
else
$x \leftarrow 0$
end if
— *total exponent to be encoded by prefixes* —
$\pi \leftarrow e + x$
$(\pi, u) \leftarrow \mathsf{MakeUnit}(\Lambda, \boldsymbol{Q}, \pi, g)$ — *see Algorithm 16* —
if $\pi \neq 0$ **and** $g < 3$ **and** $g \neq 0$ **do** — *if exponent* π *was not entirely distributed then try another* g —
$g \leftarrow g + 1$
end if
until $g = 3$ **or** $g = 0$ **or** $\pi = 0$
$s \leftarrow s10^{\pi - x - e}$ — *update scale factor by remaining exponent* —

reduction on $\mathcal{U}$. This leads to a "unitless" Unit-Triplet, i.e. the resulting unit vector must be $\mathbf{0}$, otherwise the reduction $(\Lambda, \boldsymbol{Q})$ was illegal. The resulting mantissa and the exponent however yield the numbers that can be printed nicely.

Algorithm 16 Pretty Printing a Unit Vector

function $\langle \pi, u \rangle = \mathsf{MakeUnit}(\Lambda, \boldsymbol{Q}, \pi, g)$
— π: (remaining) exponent to encode by prefixes
u: unit string with prefixes —
forall $(\lambda_i, \boldsymbol{q}_i) \in (\Lambda, \boldsymbol{Q})$ **do**
if $\pi \neq 0$ **then**
$(p, s) = \mathsf{ExpToPrefix}\big(\lfloor \pi/\lambda_i \rfloor, g\big)$ — *find prefix for exponent* π/λ_i —
$\pi \leftarrow \pi - \lambda_i p$ — *update remaining exponent* —
$u = \mathsf{Append}(u, s)$ — *print prefix for i-th step* —
end if
$u = \mathsf{Append}(u, \boldsymbol{q}_i)$ — *pretty print unit* $\boldsymbol{q}_i$ —
$u = \mathsf{Append}(u, \lambda_i)$ — *pretty print exponent* λ_i —
end forall

Consider the expression $1\,\mathsf{mm} \times 1\,\mathsf{mm} = 1\,\mathsf{mm}^2$. The prefix m stands for a factor 10^{-3} on the left hand side but on the right hand side it encodes a factor 10^{-6} because of the squared unit. Algorithm 16 solves this final hitch with prefixes. It iterates over the reduction $(\Lambda, \boldsymbol{Q})$ and looks for a prefix among the allowed ones for each separate power λ_i. If it returns with $\pi \neq 0$ then the exponent π could not be encoded entirely using prefixes in the string u and the remaining exponent must be handled be the callee of Algorithm 16 itself. Algorithm 15 for example does so by trying other restrictions on the allowed prefixes and different mantissa ranges. Any number in the MUSAC-system is built on this generic mechanism.

As an example of how this units module works, let's look at the gas equation

$$pV = nRT, \tag{5.7}$$

with the pressure $[p] = \mathsf{bar} = \mathsf{g/ms^2}$, the volume $[V] = \mathsf{l} = \mathsf{m}^3$, the number of mol $[n] = \mathsf{mol}$ and the temperature $[T] = {}^\circ\mathsf{C}$. Equation (5.7) expresses the relation between the above quantities of an ideal gas. R is the so called *gas*

constant which can be looked up in any schoolbook. In Example 19 we want to find the gas constant. Further we assume that we remember that 1 mol of ideal gas at *normal condition* (i.e. 0 °C and a pressure of 1.013 bar) expands to a volume of 22.4 l.

Example 19 Input:

```
t = 0 [°Cabs];
n = 1 [mol];
p = <1.013 :r 0.0005> [bar];
v = <22.4 :r 0.05> [l];

r = p*v/(n * t);

iso(r);
```

Example 19: Note, that we don't remember the exact numbers for the volume and the pressure at normal conditions. Therefore, we allow for a rectangular uncertainty of 0.5 of the least significant digit. The MUSAC-program is returning the result of the function call to `iso` as follows: `<8.30693 : 0.010964> [J/(K*mol)]`. The resulting unit of $[R] = \mathrm{m^3\,g/m\,s^2\,mol\,K} = \mathrm{g\,m^2/s^2\,mol\,K} = \mathrm{J/mol\,K}$. The unit $\mathrm{J} = \mathrm{g\,m^2/s^2}$ is successfully split off from the strange unit of R.

Example 19 Output:

```
t = 273.16 [K];
n = 1 [mol];
p = <1.013 :r 0.0005>[bar];
v = <22.4 :r 0.05>[l];
r = (p * v) / (n * t);
<8.30693 : 0.010964>[J/(K*mol)];
```

Note the use of the temperature: In this situation the absolute temperature of 0 °C is meant which is equal to the well known 273.1 K. As explained in Section 4.1.3 on page 72 we need to indicate this fact by `°Cabs`.

In Example 20 we will use the looked up value for the gas constant in order to compute the pressure at normal condition with less uncertainty and using this normal pressure to compute the volume difference induced by a temperature difference.

Example 20 Input:

```
t0 = 0 [°Cabs];
v0 = <22.4 :r 0.05> [l];

dt = 5 [°C];

r = < 8.3144126 :r 5e-9> [J/mol/K]; # looked up
n = 1 [mol];

p = n*r*t0/v0; # normal pressure
dv = n*r*dt/p;

iso(dv,p);
```

Example 20: Having looked up the gas constant to be 8.3144126 $\mathrm{J/mol\,K}$ we now can calculate the normal pressure `p` with less uncertainty, and using this result we can compute the volume difference `dv` induced by a temperature difference of 5 °C, getting `p = <1.01391 : 0.00130666> [bar]` and `dv = <410.016 : 0.528399> [ml]`.

Example 20 Output:

```
t0 = 273.16 [K];
v0 = <22.4 :r 0.05>[l];
dt = 5 [K];
r = <8.31441 :r 5e-009>[J/(K*mol)];
n = 1 [mol];
p = ((n * r) * t0) / v0;
dv = ((n * r) * dt) / p;
(<410.016 : 0.528399>[ml], <1.01391 : 0.00130666>[bar]);
```

Note again the use of the temperature in this case: not the absolute temperature is meant but only the temperature difference of 5 °C is important. In this situation the quantities 5 K and 5 °C are exactly the same. Therefore the unit `°C` is used instead of `°Cabs`.

5.3.2 Parser

The *parser* reads in an M-script. The particular implementation in MUSAC was generated using the *Purdue Compiler Construction Tool Set* (PCCTS) which was originally written by Terence Parr[2]. The version used for the MUSAC-project is the C++-version which is maintained and enhanced by Tom Moog since several years now[3]. I am grateful for their work.

The task of the parser is to generate a data structure based on the M-text. This data structure is generally called a *abstract syntax tree* (AST). A simple example of an AST is shown in Figure 7 on page 58. Observe that this is a true tree, i.e. any two different paths from the root to a leaf lead to two different leafs.

The AST consists of various types of nodes such as $\boxed{b}$, $\boxed{\boxed{\times}}$ or $\boxed{\boxed{=}}$ each representing a syntactic construct. The PCCTS tool set allows to describe a grammar in a fashion much like the EBNF notation. In fact the content of Section 3.2.5 is a slightly edited version of the actual grammar definition used for the MUSAC-system. The things that are not shown in Section 3.2.5 are the actions that are performed during parsing. This means, that whenever the parser read in a token, it could perform some actions, not directly related to the parsing process. These are in most cases feasibility checks or the actual construction of the syntax tree. As an example of these rules lets look at the rule `u_factor` which is was already used to explain the EBNF on page 63. Here is now the whole truth behind that simple looking grammar rule:

```
u_factor > [T_Unit ret] :
  u_expr > [$ret]
  { HAT i:NUMBER
    << double exp = atof(i->getText());
       double trunc = floor(exp);
       if (exp != trunc) {
         ostringstream os;
         os << "Only integer exponents allowed for units! ";
         os << "Will assume " << trunc << " instead of ";
         os << exp << "!";
         Warning(os.str());
       }
       $ret ^= int(trunc);
```

[2] http://www.antlr.org/

[3] http://www.polhode.com/pccts.html

```
    >>
  };
```

When the rule `u_factor` is processed, then the first thing done is calling the rule `u_expr` which returns its result into the variable `$ret`. Then the parser tries to recognize the token `HAT` and the token `NUMBER`. The text which matched the token `NUMBER` is stored in the token variable `i`. If both of these tokens are successfully read in, then the action within `<<` and `>>` is performed. In this case we are checking whether the exponent given to the unit expression is integer. If it is not then there is a warning issued and the truncated exponent is assumed in the following. Then the exponentiation of the unit stored in `$ret` is executed in place by the operator `^=`. If the token `HAT` was successfully read in, but the token `NUMBER` failed for some reason, then the parser issues a syntax error message, saying the expected token `NUMBER` was not successfully read in. If the token `HAT` was not present at all, then the rule `u_factor` is finished without an error, since the exponentiation is optional due to the curly braces { and }. The returned value of this grammar rule is finally the content of the variable `$ret`. This is not yet an AST-node rather this result is used in a higher level rule constructing an actual AST-node.

This was just a simple grammar rule, and it is not feasible to explain each of the grammar rules in this level of details here. However, this simple rule and its explanation helps certainly to understand the following rule which is one of the more complex constructs in the M-language: the definition of an `ureal`.

```
ureal : <<0>>
   LEAF_START
      ( { PLUS | MINUS } NUMBER <<1>> { unit <<2>>}
        | NAME <<3>>)
      Distribution <<4>>
      ( NUMBER <<5>> { ( unit <<6>> | PERCENT <<7>> ) }
        | NAME <<8>> )
   LEAF_END
  <<9>> ;
```

The actions executed during the parsing of this grammar rule are numbered and explained in the following list. The PCCTS system implicitly checks the grammatical correctness and would issue a syntax error if the syntax was

violated. Therefore the actions need not to be concerned about the entire class of possible syntax errors.

0. Perform all initializations of several temporary variables.

1. If a `MINUS` was present then the `NUMBER` is multiplied by a factor -1.

2. If an optional `unit` was present the second optional unit (c.f. item 6) is reinitialized with that same unit just read in. Note that for processing the rule `unit` the rule `u_factor` described above is processed amongst others.

3. If a `NAME` token was seen (instead of the `NUMBER` branch) then an AST-node is constructed and a flag is set to remember that fact.

4. The mnemonic of the requested distribution is stored, for later us by the construction of the final AST-node representing the result of this `ureal`.

5. A `NUMBER` (for the uncertainty of this quantity) was read, so check that it is positive.

6. There was an non-empty input for the second optional `unit`, check that it matches the type of the first `unit`. Note that it is therefore considered an error, if only the second unit is given although the syntax would allow this, since the first `unit` is initialized to 1 in step 0.

7. If the token `PERCENT` is read, then the number stored in Action 5 is multiplied by $1/100$.

8. As in Action 3 the `NAME` is stored for later use.

9. The final action now constructs an AST-node representing the `ureal`. First the two units the one for the mean and the uncertainty of the quantity are multiplied by their respective mantissae, then `NUMBERS` read in and stored. Then there are four cases to be handled if none, exactly one or both parts of the `ureal` were specified by a `NUMBER` or by a `NAME`.

This was an example of a rule constructing a *leaf* of the abstract syntax tree.

The construction of an interior AST-node having several children is generally quite simple:

1. Read in and construct the children,
2. construct the correct type of interior node, and
3. attach the children to the new interior node.

5.3.3 Visitor Pattern

In the previous section we showed how the parser constructs an abstract syntax tree (AST) which represents the M-description. Once the parsing is successfully finished the system operates exclusively on the resulting AST.

Imagine an evaluator and a pretty printer who receive an an AST of an M-description. Both will have to traverse the data structure and perform several particular operations during the traversal. The former will e.g. have to check whether variables are properly initialized and set before they can be used in the evaluation, the latter will perform peculiar operations to print out the AST to a file, to the screen or wherever might be requested. All of these operations must treat AST-nodes differently depending on whether they represent an expression, a variable or any other syntactic construct being represented by its AST-node. Therefore both the evaluator and the pretty printer define their own methods for each type of node being treated. They do not have much in common with each other except that both are traversing the same data structure and both are treating each type of the data structure. The well known *visitor pattern* [23] proposes a class hierarchy called *Visitor*. It combines in an abstract base class the commonness of all visitors:

1. the code executing the traversal through the AST and
2. the interface for every concrete visitor to derive from which comprises at least a method for each AST-node type.

A concrete visitor "visits" or "traverses" an AST. A traversal consists of the following steps:

1. Instantiate a concrete visitor object (e.g. `CPrettyPrinter`).

2. The root element (e.g. a `CSequence`) of the AST accepts the visitor object and calls the method of the visitor treating this node thereby passing itself as an argument (e.g. `TreatSequence(this)`).
3. The visitor object is performing the treatment of the node passed in as an argument, e.g. the sequence is printed. In this particular example the *printing of a sequence* is done of course by visiting each element of the sequence by this same pretty printer object.

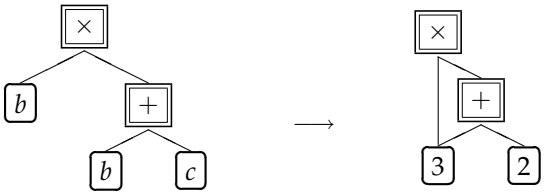
The visitor pattern has of course its advantages and disadvantages. If for any reason the set of AST-nodes types is changed then a lot of code changes are needed in each and every concrete visitor. Because the M-language is (now) quite stable this disadvantage is not very sever (anymore) and even if the M-languages changed this does not necessarily mean a change in the set of AST-node types. The advantage of the visitor pattern outweighing this by far is the fact that it is very easy to add a new visitor performing a new set of operations. In fact the MUSAC-system consists of many such concrete visitor classes, each performing its set of operations on an AST. In the course of developing the system there was a vivid evolution of many such visitors. Some disappeared entirely some even reappeared. For a more general and thorough description and analysis of the visitor pattern the reader is referred to the [23].

The following sections will show the layout of three different visitors each evaluating an AST in some particular way.

Visitor "Eval"

The parser generates *true trees* i.e. no two different paths starting at the root node lead to the same leaf. The evaluator *Eval* processes an AST in such a way that each leaf representing a variable (is of type `CAstName`) is replaced by its content. This leads to a structure *not* being a true tree anymore rather it becomes a so-called *directed acyclic graph* (DAG) where different paths starting at the root node can lead to the same leaf. Additionally the resulting DAG does not contain any nodes of types `CAstName` at leaf positions. Figure 12 displays the situation in a tiny example. The evaluator transforms a tree into a DAG. The M-function `eval` is in fact calling an "Eval" evaluator. In other words the function call `eval(a)` is doing nothing more than combining all information about `a` into a DAG. Frankly, it is doing one more thing after having constructed the DAG. It calls the "Pretty Printer" visitor to print out the DAG into the output channel. It is important to keep in

Figure 12 Syntax Tree for $b \times (b + c)$ is evaluated with $b = 3$ and $c = 2$. The result is a DAG.



mind that many built in functions such as `mc`, `iso`, `diff`, `tree` and others (c.f. Section 4.3.1) are calling "Eval" in their initialization phase. Then they are performing their particular task by calling one or several visitors such as the Monte Carlo Evaluator etc. and finally they print out the result.

All subsequently described visitors are therefore traversing a DAG, the result of "Eval". There is one particularly obvious thing to note about DAG-traversal: the visitor could revisit a leaf of even an entire subtree several times. But to traverse a subtree several times doesn't change its result. Therefore it is well worth to check whether the node and its subtree being processed has already been visited before. If so the result of that earlier traversal can be reused and needn't be recalculated. This optimization implies that each DAG traversing visitor must maintain a cache of known results in order to be able to exploit the DAG structure. The complexity of the cache depends heavily on the type of results that should be cached. Also the need to identify every single AST-node uniquely leads to additional tedious complications. All of the following visitors do implement this optimization in a way that suits their needs and we won't neither describe nor mention it any further.

Visitor "Monte Carlo Simulator"

The Monte Carlo Simulator is in principle very simple as insinuated in Section 2.2.5. The visitor processes every single node of the AST. Either the visited node is a leaf of the DAG-structure or an inner node. In the first case the Monte Carlo Simulator generates a random sample representing the leaf, i.e. a random histogram as depicted in Figure 1 on page 14. At each

inner node it executes the operation (addition, subtraction, ...) with the arguments' samples and thus produces a sample for that visited inner node. Finally, after having processed an entire DAG-structure there is a sample for each node, especially there is a sample for the root node which is the one of interest in almost all cases.

However, there was a decision to be made: The Monte Carlo Simulator generates n "true values" for each node. In principle this can be done in two ways:

- The Monte Carlo Simulator traverses the DAG-structure n-times, each time generating 1 "true value" for each sample, or
- the Monte Carlo Simulator traverses the DAG-structure exactly once. When visiting a node it generates n "true value" for that visited node.

It turns out that the second possibility is better for two reasons: First, the overhead for traversing an AST is payed only once instead of n-times and second, executing n times the same relatively simple operation in a simple loop is by far quicker, than initiating that one simple operation n times.

While the above decision is easily cleared there are at least two other still open questions concerning "proper Monte Carlo Simulations". One is the problem of *explicit covariances* and the other is the problem of simulation of *calibration*. In the following paragraphs we will explain the problem. The MUSAC-system currently provides no solution for the former and just on arbitrary solution for the latter problem. But the discussion amongst metrologists is currently vivid and not closing in towards a generally agreed solution.

Explicit Covariances The language M allows to require for two quantities to have a given distribution and simultaneously that they have a given correlation. This is asked for much, it is not easily possible to construct two samples with given distribution *and* given covariance, not to talk about n samples of prescribed distribution and prescribed covariance matrix. The problem to be solved is: Find $f(x, y)$ the bivariate probability density func-

tion, such that

$$\begin{aligned} \mathrm{pdf}_x(x) &= \int_{-\infty}^{\infty} f(x,y)\,dy \\ \mathrm{pdf}_y(y) &= \int_{-\infty}^{\infty} f(x,y)\,dx \\ c &= \iint_{-\infty}^{\infty} (x-\bar{x})(y-\bar{y}) f(x,y)\,dx\,dy \\ 1 &= \iint_{-\infty}^{\infty} f(x,y)\,dx\,dy, \\ f(x,y) &\geq 0, \quad \forall (x,y) \end{aligned} \tag{5.8}$$

where $\mathrm{pdf}_x(x)$ is the distribution for x and $\bar{x} = \int x\,\mathrm{pdf}_x(x)\,dx$, the mean of x and similarly for y. It seems not impossible that such distributions exist and can be found. After all we can make an ansatz for $f(x,y)$ with infinitely many degrees of freedom which are enough for fulfilling the Requirements (5.8). Indeed for some particular cases a pdf can be found. However, we failed to find a general algorithm to construct the pdf in general cases as they may arise in an arbitrary M-code.

If, however, the requirement on the distribution is relaxed then it is easy to construct correlated samples subject to the given covariance using Cholesky factorization or Eigenvalue decomposition of the given covariance matrix. Each method is yielding differently shaped histograms and the resulting distributions are strongly depending on the precise execution of sample construction. Even the order of appearance in the M-code would lead to different samples generated. The MUSAC-system ignores therefore the explicit covariance entry in such situations and issues a warning that explicit covariances are currently ignored in the Monte Carlo Simulation.

Calibration When simulating a calibration process the situation faced is as follows: the x-values and the y-values are both simulated by a sample of n "true values". In the spirit of Monte Carlo Simulation there are n function fits performed yielding samples for each parameter in the fitted function. The question arises as to which method for the function fitting should be applied for those n fittings? After all the simulated "true values" are not lying on a function curve. In each function fitting those "true variables"

must be shifted. A *classical least squares* fit shifts them in y-direction only in such a way that the total sum of squared shift is minimized. Why only shift in one direction? Why not apply an *orthogonal distance regression* (ODR) method which shifts the "true points" orthogonal to the fitted function, thus minimizing the total sum of orthogonal distance between function and "true point". Or one could even go one step further and argue, since we know the distribution of each sample, we allow the function fitting process to shift the "true values" weighted with the standard deviation of its sample, i.e. we should use a *weighted ODR* method. One thing is clear, each of these three methods will finally lead to different samples for the fitted parameters. But it is not clear which method is more correctly simulating the true physics best, and it is probably even depending on the particular case whether one or the other method is the preferable one.

The lack of a statement on this issue in the literature, and the fact, that the classical least squares fit is very stable and comparably quick for general problems was the reason to implement the very classical least squares fit to be used by the Monte Carlo Simulation in the current MUSAC-system. Results by this Monte Carlo Simulation are in close accordance with the results that the GUM method reports. This is a strong indication that differences due to using ODR or weighted ODR in the Monte Carlo Simulation can be expected to be very small. This is certainly true for general cases, but it may be worthwhile to study this issue in greater detail for special situations. We suggest this issue for interesting future work.

Visitor "Differentiator"

The differentiator is transforming a DAG-structure into another DAG-structure representing the derivative of the former. In the initialization phase the differentiator is set up such that one leaf x which — differentiated with respect to x — should become 1 all the other leafs become 0 again differentiated with respect to x. Then the differentiator is processing every node: first differentiate the descendants, then construct the derivative for the current node according to the well known differentiating rules: $(uv)' = u'v + uv'$, $\big(f(u)\big)' = u'f'(u)$ etc.

The function call `diff(b * ( b + c ), b)` initializes a differentiator who will differentiate visited nodes with respect to `b`. Using the differentiation rules this differentiator constructs a DAG representing `1 * (b +`

`c) + b * ( 1 + 0)` which gets simplified into `(b + c) + b` by applying obvious reductions like $1u = u$, $0u = 0$ or $0 + u = u$ etc. In fact the construction and the simplification of the result have merged into one step over time. In a first implementation of the differentiator there was indeed a visitor called "Simplifier". But it turned out on one hand that simplifications of these simple types are not worth their own visitor. On the other hand, more elaborate simplifications like already $(b + c) + b = 2b + c$ are in general very complicated. The problem of general simplifications is that common subexpressions in different subtrees must be located. In this example the system would have to detect that the subtree for `b` appears in both arguments of the second (i.e. the top) $\boxed{+}$–node of the DAG. In general such comparisons are very complicated and justify large and expensive symbolical mathematical tools. Apart from the overly complex task it is not even always clear which expression is simpler. Simpler in what sense? Therefore we implemented only the obvious and in some sense trivial simplifications such as operations with the neutral element.

Visitor "ISO-Evaluator"

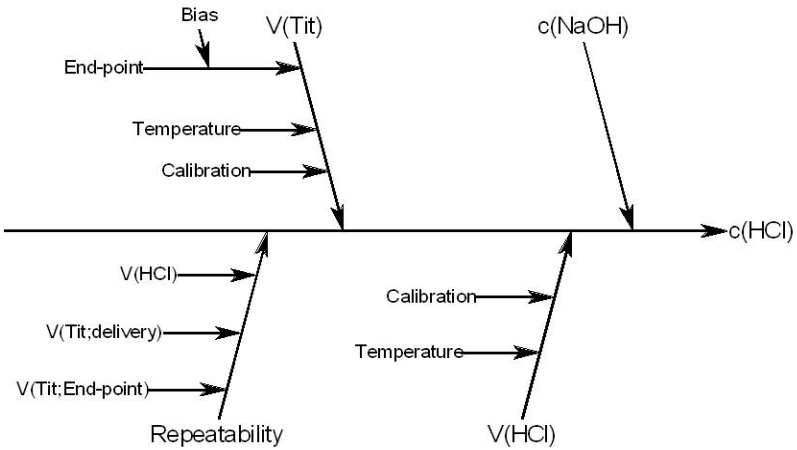
This visitor is performing the uncertainty evaluation exactly according to the GUM [2]. It is basically executing Algorithm 7 described in Section 4.3.1 on page 83. The visitor traverses the DAG representing an M-expression and performs two tasks at each node: First, compute the first order approximation of the mean. This is according to Approximation (2.13) as simple as executing the nodes' own operation. Second, the variance for each node is computed according to Approximation (2.16) which is tantamount to calling Algorithm 7.

5.3.4 Drawing Cause and Effect Diagrams

The graphical user interface of the MUSAC-system was designed and implemented by *creasoft ag*[4]. One particularly nice representation of a measurement procedure is the so-called *Cause and Effect Diagram* (CED). It is a diagram consisting of arrows that are arranged almost like fish bones. Each arrow symbolizes an influence on the "parent"-arrow. The entire CED thus visualizes all influences on the main-arrow in a hierarchical manner. Martin

[4] `http://www.creasoft.li`

Figure 13 An Example of a CED as Displayed in the MUSAC-system



Müller describes in his semester project [34] the implementation of a component to visualize such CED in a appealing way. Figure 13 shows an example of a CED. The interesting problem of the visualization is to ensure that all relevant information is displayed, such that no text tags are overlapping, and that different levels of influence are still observable. In particular he solved the following problems:

- What is a sensible layout for an arbitrary CED, such that lucidity of the diagram is maximal?
- How can the diagram interactively and intuitively be manipulated, like collapse and expand sub-arrows, add new arrows, etc.
- What are good trade-offs when the diagram grows too big to be displayed in a window? How to combine scrolling and zooming intuitively?

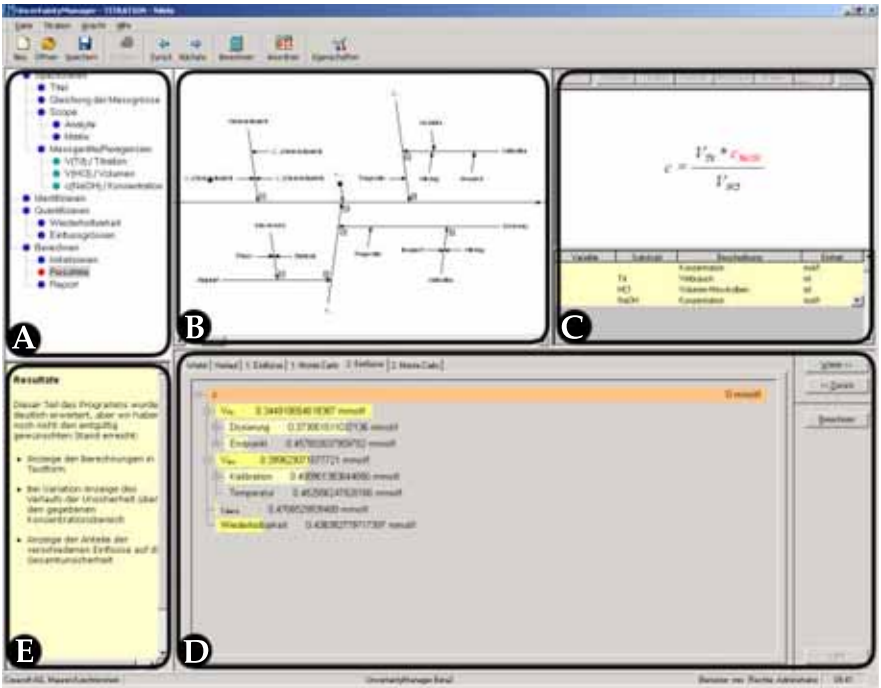
5.3.5 The program *Uncertainty Manager*

For a long time it was unclear how a measurement process could generally be visualized. And it was equally open how the user of the MUSAC-system

could be guided through the process of measurement uncertainty evaluation for a particular measurement process. Figure 14 shows a screen shot of the current MUSAC-system. It is the result of many discussions on drawings by Matthias Rösslein, the experience on user interfaces by *creasoft ag* and the programming efforts of Martin Müller and Marco Wolf.

Figure 14 shows a screen shot of the running program named *Uncertainty Manager*. The screen is subdivided into several areas. Area A is displaying the entire measurement evaluation process in a hierarchical manner. The current working step is always highlighted. Area E is showing online help, instructions and explanation texts. These texts are always referring to the currently active area, sometimes even to a particular active object. Area

Figure 14 Screen Shot of the Current β-version of MUSAC-tool Called *Uncertainty Manager*



A and E are always visible. They offer a clear and instructive view over the entire uncertainty evaluation process. Area B is visualizing the measurement procedure model by a *Cause and Effect Diagram* (CED), while Area C is showing the mathematical Model for the same measurement procedure. Area D is finally the most variable part. It is basically giving access to the data base storing a wide variety of different tools, substances, effects and experienced data from which the user can select to complete the model for his measurement procedure. Or, as in the case of Figure 14, Area D is displaying the result of an evaluation run. Here, we are seeing the uncertainty contribution of each influence arranged in the same hierarchical way as is suggested by the CED.