Pergamon

Computers Math. Applic. Vol. 29, No. 12, pp. 81–102, 1995
Copyright©1995 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0898-1221/95 $9.50 + 0.00

0898-1221(95)00061-5

Applying Abstraction and Formal Specification in Numerical Software Design

H. H. TEN CATE
Faculty of Technical Mathematics and Informatics, Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands

(Received January 1994; accepted February 1994)

Abstract—Numerical software development tends to struggle with an increasing complexity. This is, on the one hand, due to the integration of numerical models, and on the other hand, due to change of hardware. Parallel computers seem to fulfill the need for more and more computer resources, but they are more complex to program.

The article shows how abstraction is used to combat complexity. It motivates that separating a specification, "what," its realisation, "how," and its implementation, "when, where," is of vital importance in software development. The main point is that development steps and levels of abstraction are identified, such that the obtained software has a clear and natural structure.

Development steps can be cast into a formal, i.e., mathematical framework, which leads to rigourous software development. This way of development leads to accurate and unambiguous recording of development steps, which simplifies maintenance, extension and porting of software. Portability is especially important in the field of parallel computing where no universal parallel computer model exists.

Keywords—Software development, Finite difference method, Formal specification, Formal verification, Code generation.

1. INTRODUCTION

Most present-day numerical software packages have been written in Fortran. The structure of such a language makes it very difficult to modify the code. For instance, the introduction of a new type of element in a finite element package, or the addition of a new matrix solving technique can involve the adaption of large portions of code. Another difficulty is the port of Fortran77 code to a message passing parallel computer. In general, this requires adaption or even rewriting of data-structures.

Another problem with large present-day scientific and technical programming packages is that of the correctness of the software. In large packages, it is inevitable that errors will be present. Furthermore, experience teaches that the removal of errors does not exclude the possibility of the introduction of further mistakes in the program.

Aimed at easing these problems, the ATES-project had been initiated as part of the first ESPRIT Software Technology programme. Within the area of scientific application programming, this project aimed at the integration of three advanced techniques into an integrated software development environment [1–3]. This paper is focused on two of them: "complete abstraction of data-types and operators" and "formal specification and proof." All the way through the project, efficiency was considered to be a vital constraint.

In software development, it is important to abstract (in the first instance) from the target machine on which the software is running. A separation has to be made between a specification -**what is it**-, its realisation -**how is it done**-, and its implementation -**when and where is**

it done-[4–6]. The last aspect, about "when and where," is especially important for parallel computing. For instance, **when** must processors synchronise or **when** do they communicate, and in a distributed-memory system the question "**where** is data located?" arises.

When developing software, a language is needed in which the development is expressed. For ease of maintenance and extension of the software, recording of development steps and design decisions is important. Very often, a natural language and/or a graphical language is used. However, a natural language has its limitations as specification language [7]. Natural language and most graphical languages have no clear semantics. What is required is a more formal language for expressing specifications, for expressing exactly and without ambiguity **what** software must do. In a way, formal specifications of software can be compared to a "blue-print" used by architects or engineers. The main advantages of a blue-print are that it is a *model* of the final product expressing the "*what*" rather than the "*how*" and that it is readable for the designer as well as the implementer.

In numerical mathematics, problems are defined in a mathematical, thus formal, language. In this paper, it will be demonstrated how to rewrite or transform the mathematical formulation into software. Software development is a creative process and very often it is carried out *ad hoc*. Nevertheless, there exist some fundamental steps, which can be cast into a formal framework (theory on algebras). A consequence is that, for some development steps, automatic development support is possible. Furthermore, the formal framework opens the possibility to prove correctness of the software. Another consequence of the suggested formal framework is an automatic structuring in software development. This results in a simplification of maintenance, extension and portability of software. Another implication of this structuring is that a separation between aspects concerning numerical mathematics and aspects concerning software engineering will become clear.

Abstraction forms the basis of this paper. The paper first discusses different forms of abstraction, two basic abstraction mechanisms in software development and examples of the use of abstraction in practice. As a startup for formal software development, first three elementary software development steps are discussed. This is illustrated by software development to solve a matrix equation. Next, the software development to solve a matrix equation is worked out in a formal method. Thereupon, the methodology is applied to a 1D heat transfer problem. Finally, the paper ends with some conclusions.

2. ABSTRACTION IN SOFTWARE

As many engineers have observed when tackling a complex problem, it is very useful to abstract only those details from observations of nature, which are essential in the development of theories and models. The nonessential details are ignored. Abstraction is an established support to combat complexity. The current section will show what abstraction means for software development.

2.1. Different Forms of Abstraction

Abstraction can be obtained in three different ways [8]:

- discarding, i.e., omitting irrelevant detail;
- hiding, i.e., not revealing, or not determining, detail, knowledge of which would be unnecessary and immaterial at a given stage;
- generalising, i.e., capturing the common "essence" of a collection of similar or related objects by a single description.

In fact, the problem of discarding is a modelling problem. Software is a model of a real world environment. In this model, some existing characteristics of the real world may be, for some reason, neglected. Discarding is an analysis technique, rather then a specification and design technique.

In the preliminary stages of specification and design, it is important not to be overwhelmed by too much detail or to concentrate unnecessarily on aspects of software that can, and should, be left until later. Information hiding separates the "what" and the "how." At first, it is important to realise "what is the problem"; afterwards, the realisation can be worked out, "how can the problem be solved." In general, it is possible to find several realisations for one specification, some of them will be more efficient than others.

The process of generalisation simplifies software specification, design, and maintenance. It does this by reducing to a single description what would otherwise be an abundance of separate descriptions. Obviously, the descriptions must possess a sufficient degree of commonality for the generalisation to be possible. If, for example, a part of the common description must be modified, this can be done once and for all for the general description instead of modifying every specific description. Generalisation is a famous technique in mathematics. For example, a theory on partial differential equations deals with unknowns rather than with temperatures or velocities.

2.2. Two Basic Abstraction Mechanisms: Procedural Abstraction and Data Abstraction

Two important abstraction mechanisms in programming are abstraction of data (*data abstractions*) and abstraction of operations which can be performed on the data (*procedural abstractions*) [6]. Both mechanisms already exist, to some extent, in computer languages. Procedural abstractions are called procedures or functions in Pascal, subroutines or functions in Fortran, and functions in C. Data abstractions are represented by data-types. However, data abstractions can be more general than the data-types in the previously mentioned languages.

The role of the two abstraction mechanisms in software development will be explained by means of an example. The example concerns solving a matrix equation. In the first instance, it is assumed that the coefficients of the matrix, the unknown and right-hand side are real numbers. Later this will be generalised.

INFORMATION HIDING BY A PROCEDURAL ABSTRACTION. Information hiding stands for separating a specification and a realisation. When solving a matrix equation, the first point is to know that $\underline{x}$ satisfies $A\underline{x} = \underline{b}$. The choice of the precise algorithm comes next, but is not of secondary importance. The two questions *what* and *how* can be answered separately:
what: $\underline{x}$ satisfies $A\underline{x} = \underline{b}$, possibly up to some small error;
how: use Gauss elimination or use Gauss-Seidel.

GENERALISATION USING A PROCEDURAL ABSTRACTION. Generalisation suggests to search for commonalities in a software development. For example, a procedural abstraction has not been written for *one* matrix A, *one* vector $\underline{x}$ and *one* vector $\underline{b}$. It has been written for *all* matrices and vectors which have real coefficients!

A further generalisation would be to write the procedural abstraction for all types of matrices.[1] The algorithm to solve the matrix is in fact a general algorithm. The algorithm is similar for complex matrices, real matrices, integer matrices and even matrices with polynomials as coefficients. The only requirement to perform the algorithm is that the operations in the algorithm can be performed on the matrix elements!

When solving a matrix equation, the matrix plays the main role. Information hiding and generalisation will be discussed for the data abstraction "matrix."

INFORMATION HIDING BY A DATA ABSTRACTION. Again a separation between specification and realisation is strived for. As long as efficiency is not interesting, there is no need to know how a data abstraction has been realised. The issue is "what does it mean," or better, "what operations are defined on the data abstraction." For example, the knowledge that a two-dimensional array

[1]Some programming languages offer the sketched generalisation with help of the inheritance principle.

is implemented in Fortran row-wise or column-wise only plays a role when striving for optimal performance.

For the specification of a matrix, it satisfies to know that addition, scalar multiplication, multiplication, selecting an element and other operations defined in each book on linear algebra exists! In fact, Berzins, Liskov and Guttag [6,9] concluded that, in principle, operations on an object define the characteristic of that object. For that, they introduced an "abstract data-type."

GENERALISATION USING A DATA ABSTRACTION. Observing the definition of a matrix, there is no real difference between a matrix with integer components or a matrix with real components. The only point is that the data-type of the components satisfies some properties necessary for the definition of the matrix. Consequently, a lot of work would be avoided if a matrix could be defined in terms of components of a generic type.[2]

2.3. Abstraction in Practice

Some of the different forms of abstraction and some of the two mechanisms presented in the previous section are more or less supported by present-day languages and/or systems. A very brief overview will be given.

One of the first languages for computers was assembly language. Today these are replaced by languages of a higher level, like Fortran or C, or by languages with a better support of data abstraction: object oriented languages. Functional and logic programming languages (so called nonprocedural languages) are more abstract in the sense that they are based on the idea that in principle a program need only declare the *relationship* between its input and its output [10,11], rather than specifying for example the *order of execution* to obtain the output.

In the above mentioned languages, there is no clear separation between the specification of a procedural abstraction and its realisation. The body of a Fortran subroutine is its specification but also its realisation. A clear separation has been made in the ATES-system [1–4]. Specification languages [12] also support the mentioned separation. This will be explained in the current paper.

Abstract languages which are often used in civil engineering are simulation languages. In general, those languages allow to work with vectors, mathematical functions. In fact, they support a higher form of data abstraction. Furthermore, they offer a lot of tool support and predefined functions dedicated for a specific area of application. In other words, they offer a high level of procedural abstraction, but only for the certain area of application.

Software libraries like the NAG-library support procedural abstraction. They offer a lot of methods to be used. However, in general, they do not support a high level of data abstraction. For example, when using the Fortran NAG-library, data has been formulated in terms of arrays. Furthermore, a user must declare a lot of work arrays.

Finally, in the book "Templates for the Solution of Linear Systems" [13], templates are introduced to overcome the discrepancy in users which want to use software as a black box and users which want to be able to tune data structures for a particular application. By introducing templates for algorithms and abstracting from data structure, the authors want to support both data abstraction and procedural abstraction.

3. ELEMENTARY STEPS IN SOFTWARE DEVELOPMENT

Based on the two basic abstraction mechanisms discussed in the previous section, three kinds of development steps can be defined in software development.

- Realise a procedural abstraction, e.g., decide to solve a tridiagonal matrix equation with help of Gauss elimination.
- Realise a data abstraction, e.g., decide to represent a tridiagonal matrix as a three dimensional array. This is also called *Data refinement* or *Data reification* [14].

[2]Object oriented programming languages offer this possibility (inheritance).

- Realise a procedural abstraction and a data abstraction at the same time, e.g., decide to represent a tridiagonal matrix as a $3 \times m$ dimensional array, and implement Gauss elimination for this array.

The identification of the three development steps in software stimulates a structuring of the software. Each development step decreases the "amount of abstraction." A development step "transforms" a formulation on one *level of abstraction* into one on a level of lower abstraction.

Development steps can easily be identified when asking the questions *what*, *how*, and *when* and *where*. The last two questions are in fact a form of *how*, but then focussed on the computer. In other words, a separation has been made between *specification*, *realisation*, and *implementation*. Below, the different development steps will be illustrated by an example. It concerns solving a matrix equation.

```
what: find the unknown such that $A\underline{x} = \underline{b}$

how: use Gauss elimination:
     n = Order(A)
     for k = 1 to n-1 do
         for i = k+1 to n do $a_{ik} := a_{ik}/a_{kk}$
         endo
         for j = k+1 to n do
             for i = k+1 to n do $a_{ij} := a_{ij} - a_{ik} * a_{kj}$
             endo
         endo
         for i = k+1 to n do $b_i := b_i - a_{ik} * b_k$
         endo
     endo
     EliminateUpperTriangularPartOf($A, \underline{b}$)
     $\underline{x} := \underline{b}/diagonal(A)$

where: insert this procedural abstraction in-line in the object code to be run on one
     processor.
```

Figure 1. Specification, realisation and implementation of solving a matrix equation.

Development Step: Realise a procedural abstraction

Based on Section 2.2, the answers on the questions *what* and *how* are simple (see Figure 1). The answer to the first question is as "abstract" as possible. The answer to the second question is concrete, as Gauss elimination has been chosen. The algorithm is not completely worked out. Elimination of the upper triangular part of the matrix and the computation of the unknown are presented as procedural abstractions. The first procedural abstraction modifies both A and $\underline{b}$. The second does not. Note that the algorithm does not set matrix elements to zero. This is not necessary for the computation of $\underline{x}$. The question *where* is especially concerned with performance. It can be handled by a compiler for example. Asking *when* is not relevant with respect to the choice in *where* in Figure 1. The chosen realisation and implementation are one of many! For Gauss elimination, there are a lot of different ways to walk through the matrix elements [15]. Given a certain (type of) computer for each algorithm, the rate of performance can be investigated, and the best algorithm can be selected.

Development Step: Data reification

When solving the tridiagonal matrix equation $A\underline{x} = \underline{b}$, it must be clarified what the data means. The meaning of A is explained in Figure 2. For the moment, it is assumed that the definition of a tridiagonal square matrix is known. In Section 4.1, a precise definition of a tridiagonal square matrix will be given. In the realisation of the matrix, advantage can be taken of the fixed number of nonzero elements. Only those need to be stored in the memory of the computer. For that reason, the array presented in Figure 2 has been chosen as the realisation data-type. The relation between a matrix and its representation has also been (informally) described.

what: $A \stackrel{d}{=}$ tridiagonal square matrix of reals

how: $ar \stackrel{d}{=}$ array[1..3,1..order(A)] of reals, where
a_{ii-1} is stored as $ar[1,i]$, $i \in \{2..n\}$, a_{ii} is stored as $ar[2,i]$, $i \in \{1..n\}$ and a_{ii+1} is stored as $ar[3,i]$, $i \in \{1..n-1\}$

where: *ar* is stored in one piece of memory (contiguously)

or *where:* *ar*[j,.] is stored in the memory of processor j, with $j \in \{1,2,3\}$

Figure 2. Specification, realisation and implementation of a matrix.

The chosen realisation still leaves some freedom for the implementation! In Figure 2, two possibilities are given. They describe a mapping of the data in memory. It can be observed that a relation similar to the one between a matrix and its realisation exists between the realisation and implementation. In fact, the questions *what*, *how*, and *where* can be replaced by *what*, *how*, and *what* (replacing *how*), *how* (replacing *where*).

what: find the unknown in the three diagonal matrix equation $A\underline{x} = \underline{b}$

how: use Gauss elimination, implement the matrix in the previously defined array *ar* (see Figure 2) and implement $\underline{b}$ as an array *br* with the same dimension as $\underline{b}$:

```
n = second_dim(ar)
for k = 1 to n-1 do
    $ar[1,k+1] := ar[1,k+1]/ar[2,k]$
    $ar[2,k+1] := ar[2,k+1] + ar[1,k+1]/ar[3,k]$
    $br[k+1] := br[k+1] - ar[1,k+1] * br[k+1]$
endo
EliminateUpperTriangularPartOf(ar, br)
xr := br/diagonal(ar)
```

where: **data:** *ar*[*j*, .] on processor j, *br* on processor 4.

operations: are carried out at the location of the result variable.

or *where:* **data:** *ar*[1, .] on processor 1, *ar*[2, *i*] on processor 3, *ar*[3, *i*] on processor 2, *br* on processor 4.

operations: the division in the second statement of the loop is carried out on processor 3 and the others at the location of the result variable.

Figure 3. Specification, realisation and implementation of a matrix and of solving a matrix equation.

Development steps in case of data reification and realisation of a procedural abstraction

The algorithm to solve a matrix equation as presented in Figure 1 does not take into account that a tridiagonal matrix contains a lot of zeros. In Figure 3, the algorithm has been rewritten in a form which only performs operations on nonzero elements. In the algorithm, the matrix *A* is represented by the array *ar*, and the vector *b* is represented by the array *br*. The procedural abstractions which are used in the algorithm of Figure 1 are also replaced by their realisations in terms of the array *ar* and *br*. Note that the improvement in efficiency is on the cost of the generality of the algorithm. In Section 4.1.3, it will be shown that the algorithm in Figure 3 can be generated automatically.

Supposing that the software must be implemented on a ring of four processors, each having its own memory, the realisation still leaves open where data must be stored and where operations will be carried out. In the implementation, two possibilities are presented. The first one does not efficiently eliminate the lower triangular part of the matrix. Processor 3, which contains the third row of *ar*, does not have to compute anything. The second implementation reduces the amount of communication. It will be more efficient if communication needs more time than computation.

After a thorough analysis of the amount of parallelism in the algorithm and comparing this with other algorithms to solve a matrix equation, one should conclude that the chosen algorithm is not the best for the chosen architecture. Further discussion goes beyond this paper. The example shows the necessity to separate a specification, *what*, its realisation, *how*, and its implementation, *where*, and *when*.

4. A FORMAL DEVELOPMENT METHOD WORKED OUT: MODEL BASED SPECIFICATION

In a model based specification language, a data abstraction is specified, or better "modelled," in terms of some predefined data abstractions, such as sets and functions [11,14]. If the specified data abstraction has not been efficiently implemented on the computer for which the software must be developed, a realisation (or representation) of the data abstraction must be found. The found realisation is also a data abstraction but at a lower level of abstraction. This last data abstraction will be referred to as the realisation data abstraction. It is specified by a model.

For the specification of a procedural abstraction, an axiomatic specification [16] has been used in the paper. This means that the operands of the procedural abstraction must exactly be described (i.e., the operands which are input, which are output and their data-type). Furthermore, the interface with the "outer world" must be defined. In a postcondition, it must be specified "what is done" by the procedural abstraction. A precondition describes under which condition the postcondition is valid. If the precondition is not satisfied for some input variables, the procedural abstraction returns something undefined.

An axiomatic specification is descriptive. The postcondition does not have to define an explicit relation between the input operands and output operands. An example is the postcondition for solving a matrix equation (see Figure 5). In that case, a realisation for the procedural abstraction must be formulated. The realisation has to define how "what is done" can really be solved. The realisation is a so-called operational specification. The difference between an operational specification and a subroutine in Fortran is that the operational specification focuses on explaining how the problem is solved. It does not have to take account for efficiency whereas the subroutine should.

After a specification and a realisation has been defined, the realisation should be verified against the specification. Sequential software is defined to be correct if and only if a realisation satisfies its specification and the computation is finite. In this section, a formal specification and a realisation of a procedural abstraction and of a data abstraction will be illustrated by an example: solving a matrix equation. Proofs of correctness will only concern partial correctness. The proof of finiteness can be carried using the the same techniques as necessary for partial correctness.

4.1. Solving a Matrix Equation as an Illustration

A matrix equation $Ax = b$ consists of a matrix A and two vectors x and b. Assuming that functions and sets are known in the specification language, matrices and vectors can be defined (see Figure 4). The matrices are tridiagonal, which is expressed by the property *tridiag*(A). This property and other properties like addition, scalar multiplication are not worked out in the figure. But, in fact, a small theory[3] should be introduced around a set modelling a data-type. This is not as strange as it may seem. For example, in mathematics for vectors, a theory on vector spaces exists.

```
The matrix: $A \in \mathit{TridiagMatrix}$ where
    $\mathit{TridiagMatrix} == \{A \in \mathit{SquareMatrix} | \mathit{tridiag}(A)\}$
    $\mathit{SquareMatrix} == \{\mathit{Index\_set} \times \mathit{Index\_set} \to \mathbb{R}\}$
    $\mathit{Index\_set} \subset \mathbb{N}$.

    for $M \in \mathit{TridiagMatrix}$
    $\mathit{order}(M) = \mathit{cardinality}(\mathit{Index\_set}(M))$
    $\mathit{tridiag}(M) = \forall i, j \in \mathit{Index\_set} : (i \neq j - 1 \wedge i \neq j \wedge i \neq j + 1) \Rightarrow m_{ij} = 0$
    etc.

The vectors: $\underline{x}$ and $\underline{b} \in \mathit{Vector}$ where
    $\mathit{Vector} == \{\mathit{Index\_set} \to \mathbb{R}\}$.

    for $\underline{v} \in \mathit{Vector}$
    $\mathit{dim}(\underline{v}) = \mathit{cardinality}(\mathit{domain}(\underline{v}))$
    etc.
```

Figure 4. Specification of a Matrix and a Vector as functions.

```
what: input data: $A \in \mathit{TridiagMatrix}$, $\underline{b} \in \mathit{Vector}$; output data: $\underline{x} \in \mathit{Vector}$
    postcondition: $A\underline{x} = \underline{b}$; precondition $\mathit{invertable}(A)$

how: use Gauss elimination:
    $n = \mathit{Order}(A)$
    for k = 1 to n-1 do
        for i = k+1 to n do $a_{ik} := a_{ik}/a_{kk}$
        endo
        for j = k+1 to n do
            for i = k+1 to n do $a_{ij} := a_{ij} - a_{ik} * a_{kj}$
            endo
        endo
        for i = k+1 to n do $b_i := b_i - a_{ik} * b_k$
        endo
    endo
    $\mathit{EliminateUpperTriangularPartOf}(A, \underline{b})$
    $\underline{x} := \underline{b}/\mathit{diagonal}(A)$
```

Figure 5. Formal specification and realisation of solving a matrix equation.

[3]Such a theory also exists in traditional software. It is implicitly coded in the software.

4.1.1. Formal specification and realisation of a procedural abstraction

So far, the data playing a role in solving a matrix equation is specified. Next, the procedural abstraction to solve the matrix equation can be specified. It's specification must be as abstract as possible. For that reason, the postcondition in Figure 5 does not represent an algorithm. An algorithm is presented in the realisation. The procedural abstraction can only be applied on matrices which are invertible, which is specified by the precondition. Naturally, the property *invertable*(A) should be defined in the same way as *tridiag*(M) in Figure 4.

After the specification (what) and realisation (how) are chosen, it must be proven that the realisation satisfies its specification. This is called a proof obligation. There are several techniques to do this. The use of a technique explained by Jones [14] comes, for this case, close to a proof as it appears in linear algebra. One must prove that $\mathit{diagonal}(A')\,x = b'$(the eliminated system) $\Leftrightarrow$ $Ax = b$ under the assumption that A is invertible. In this proof obligation, A and b are the input *values* and A' and b' are the *values* of the matrix and vector after applying the algorithm. An essential detail is that the presented algorithm is optimised: matrix elements which should be zero and which do not influence the solution are not set to zero! For that reason the proof obligation is formulated in terms of $\mathit{diagonal}(A')$, a matrix containing only the diagonal of A' and further zeros.

Another technique is based on the classical method of Floyd [17] and Hoare [18] and Dijkstra's weakest preconditions [19]. In general, it is used for the verification of existing software. Generating Weakest Preconditions for each statement in a program results in a verification condition. This condition must be proved to be true. Verification of numerical software according to this technique can be found in [4].

what: $\mathit{TridiagMatrix} == \{A \in \mathit{SquareMatrix} | \mathit{tridiag}(A)\}$
$\mathit{SquareMatrix} == \{\mathit{Index\_set} \times \mathit{Index\_set} \rightarrow \mathbb{R}\}$

how: the realisation of *TridiagMatrix* has been chosen as an array [1..3,1..order (A)] of reals. A model of this array is:
$\mathit{Repr} == \{\{1, 2, 3\} \times \mathit{Index\_set} \rightarrow \mathbb{R}\}$

Figure 6. Formal specification and realisation of a matrix.

4.1.2. Data reification

Matrices and vectors (in Figure 4) are specified with the help of sets and functions. If the programming language does not implement sets and/or functions efficiently, a software developer must do that. For that, a realisation for the data-type must be chosen. A possible realisation of a matrix and a vector is an array. In the example, the matrix is tridiagonal, so that it contains a lot of zeros, which must always be zero. To avoid the storage of the zeros, the realisation in Figure 6 can be chosen. In that figure, first the specification model is reprinted, next the realisation model is presented.

For the current data reification, the proof obligation requires to show that there exists a function, the *abstraction function*, which has the realisation model as domain and the specification model as range. For this function, some properties must hold. In this paper, the work of Jones [14] is followed. He uses the term retrieve function instead of abstraction function.

The first property is that every element in the abstract model must have a "realisation." This means that the abstraction function must be surjective. Jones denotes this property as *adequacy*. In practice, the realisation model, the domain of the abstraction function, can be chosen such that the function is also *total*, every element in the realisation model has an image.

The abstraction function (if it is total) is, in fact, a homomorphism as known in the theory on algebras. It is used to represent operations (read procedural abstractions) on elements in the specification model by 'similar' operations on elements in the realisation model (see the following section).

For the current situation, an abstraction function is presented in Figure 7. It's domain is the realisation model *Repr* (which is a set) and it's range is *TridiagMatrix*. Suppose that $Ra \in Repr$ then $ABS(Ra) \in TridiagMatrix$, and consequently, $ABS(Ra)$ is a tridiagonal matrix. To emphasize that Ra is an array, the elements of Ra have been denoted by $Ra[i,j]$, which should be read as the function Ra applied to (i, j). The presented abstraction function is surjective, but not injective. This is due to the fact that the array element $Ra[1,1]$ will never be mapped on any matrix element. So two arrays with different values in $Ra[1,1]$ and the same value in the other elements will be mapped on the same array. Obviously, the function is total.

```
ABS : $Repr \rightarrow TridiagMatrix$
ABS(Ra) : $Index\_set \times Index\_set \rightarrow \mathbb{R}$
  ABS(Ra)(i,j) =
    if $j = i - 1$ then $Ra[1,i]$
    elseif $j = i$ then $Ra[2,i]$
    elseif $j = i + 1$ then $Ra[3,i]$
    else 0)
```

Figure 7. An abstraction function for the realisation of a matrix.

For the realisation of a vector as an array, a trivial abstraction function can be presented. It maps every array element with index i on a vector element with the same index i. This abstraction function is an isomorphism (a bijective homomorphism). The role of the abstraction function in software development becomes clear in the following section.

4.1.3. Realisation of procedural abstraction and data reification

Previously, Gauss elimination has been formulated for a general matrix. No advantage has been taken of the fact that the matrix is tridiagonal. In Figure 8, a realisation was chosen, which does take the advantage and which has been formulated in terms of the elements of the array which implement the matrix.

The proof obligation is to show that the realisation satisfies its specification and that an abstraction function with the properties as defined in the previous section exists. The required

```
what: input data: $A \in TridiagMatrix$, $\underline{b} \in Vector$; output data: $\underline{x} \in Vector$
      postcondition: $A\underline{x} = \underline{b}$; precondition $invertable(A)$.
      Model: $TridiagMatrix == \{A \in SquareMatrix \,|\, tridiag(A)\}$
      $SquareMatrix == \{Index\_set \times Index\_set \rightarrow \mathbb{R}\}$

how:  use Gauss elimination and implement the matrix as the array $ar \in Repr$:
      Model: $Repr == \{\{1,2,3\} \times Index\_set \rightarrow \mathbb{R}\}$
      $n = second\_dim(ar)$
      for k = 1 to n-1 do
          $ar[1,k+1] := ar[1,k+1]/ar[2,k]$
          $ar[2,k+1] := ar[2,k+1] + ar[1,k+1]/ar[3,k]$
          $br[k+1] := br[k+1] - ar[1,k+1] * br[k+1]$
      endo
      $EliminateUpperTriangularPartOf(ar, br)$
      $xr := br / diagonal(ar)$
```

Figure 8. Specification, realisation and implementation of a matrix equation.

abstraction function is already formulated in Figure 7. The proof that the realisation satisfies its specification consists of two steps. First, show that the algorithm in Figure 8 implies the abstract algorithm in Figure 5. Second, prove that the abstract algorithm implies the specification. This last proof obligation has already been discussed in Section 4. The essential characteristics of the first proof are presented below. The part of the algorithm which eliminates the lower triangular of the matrix is considered:

```
for k = 1 to n-1 do
 for i = k+1 to n do $a_{ik} := a_{ik}/a_{kk}$
 endo
 for j = k+1 to n do
     for i = k+1 to n do $a_{ij} := a_{ij} - a_{ik} * a_{kj}$
     endo
 endo
 for i = k+1 to n do $b_i := b_i - a_{ik} * b_k$
 endo
endo
```

Matrix A has been realised by the array *ar* (see Figure 8). Consequently, matrix A can be replaced by $ABS(\text{ar})$ in the abstract algorithm using the homomorphism property of ABS. Working this out means that the matrix elements a_{ij} are replaced by the matrix elements $ABS(\text{ar})_{ij}$:

```
for k = 1 to n-1 do
 for i = k+1 to n do $ABS(ar)_{ik} := ABS(ar)_{ik}/ABS(ar)_{kk}$
 endo
 for j = k+1 to n do
     for i = k+1 to n do $ABS(ar)_{ij} := ABS(ar)_{ij} -$
                                   $ABS(ar)_{ik} * ABS(ar)_{kj}$
     endo
 endo
endo
```

After evaluation of the abstraction function (see Figure 7), a lot of matrix elements will be replaced by zero. The nonzero elements are the elements with indices for which the absolute value of their difference is less than two. These elements are extracted from the loops. The first iterate in the first loop over i is extracted from the loop. After application of the abstraction function ABS, the matrix elements in the rest of this loop ($a_{ik} = 0$ for $i = k + 2$ to n) will reduce to zero. In the same way, the first iterate in the double loop is extracted, such that this loop is rewritten in one statement and two loops. After application of the abstraction function, the statements in the two loops will be rewritten in "identities." This is due to $a_{ik} = 0$ for $i = k + 2$ to n and to $a_{kj} = 0$ for $j = k + 2$ to n. The rewritten algorithm is

```
for k = 1 to n−1 do
   $ABS(ar)_{k+1\,k} := ABS(ar)_{k+1\,k}/ABS(ar)_{kk}$
   for i = k+2 to n do $ABS(ar)_{ik} := ABS(ar)_{ik}/ABS(ar)_{kk}$
   endo
   $ABS(ar)_{k+1\,k+1} := ABS(ar)_{k+1\,k+1} - ABS(ar)_{k+1\,k} * ABS(ar)_{k\,k+1}$
   $j := k + 1$; for i = k+2 to n do $ABS(ar)_{ij} := ABS(ar)_{ij} -$
                                          $ABS(ar)_{ik} * ABS(ar)_{kj}$
           endo
```

```
    for j = k+2 to n do
        for i = k+1 to n do  $\mathrm{ABS(ar)}_{ij}$ := $\mathrm{ABS(ar)}_{ij}$ –
                                     $\mathrm{ABS(ar)}_{ik} * \mathrm{ABS(ar)}_{kj}$
        endo
    endo
endo
```

Applying the abstraction function and evaluation of the if statements result in a lot of identities. After removing those, we obtain Gauss elimination for a matrix implement as the defined array *ar*

```
for k = 1 to n–1 do
  $\mathrm{ar}[1,k+1] := \mathrm{ar}[1,k+1]/\mathrm{ar}[2,k]$
  $\mathrm{ar}[2,k+1] := \mathrm{ar}[2,k+1] - \mathrm{ar}[1,k+1] * \mathrm{ar}[3,k]$
endo
```

The aim of the previous rewriting was to prove that the algorithm satisfied its specification. But a side effect is that the above rewriting generates software. Hence, with help of a tool, this software can be generated.

5. SOFTWARE DEVELOPMENT FOR A 1D HEAT TRANSFER PROBLEM

In this section, a rigourous software development for a 1D heat transfer problem will be presented. The problem is modelled by a partial differential equation prescribed on the interior of the domain and boundary conditions prescribed on the boundary of the domain. The boundary conditions are chosen to be Dirichlet boundary conditions. The problem is

$$-k\frac{\partial^2 T}{\partial x^2} = S, \qquad \text{if } x \in (0,1), \tag{1}$$

$$T(x) = T0, \qquad \text{if } x = 0, \tag{2}$$

$$T(x) = T1, \qquad \text{if } x = 1, \tag{3}$$

where S is the volumetric heat source ($Jm^{-3}s^{-1}$), k is the thermal conductivity ($Jm^{-1}s^{-1}\,{}^0\mathrm{K}^{-1}$), T is the temperature (^{0}K). Without the loss of generality, the domain is chosen to be the interval $(0,1)$ with boundaries at $x = 0$ and $x = 1$. The equations are derived from a three-dimensional problem, supposing that k is a constant and that S is independent of T.

The development has already started with an explanation of the problem. Very helpful is a context diagram which gives an informal overview of the system modelling the 1D heat transfer problem. It shows all the main inputs and outputs of the system. In this diagram, just the name of the system and of the data-items are given. They are not defined. Hence, it is a very abstract presentation of the system. The next step is to define the data-items (see the subsequent paragraph). Finally, the semantics of the system is defined by preconditions and postconditions.

The data-items are defined in abstract mathematical terms (e.g., sets and functions). Because the formal specification technique that will be used is model oriented [8], for each data-item a set to which it belongs must be defined. The data-type is specified via its model. Notwithstanding that a model has been defined for a data-type, the data-type must still be implemented in a programming language. A possible implementation is by means of an abstract data-type [4,6].

After a context diagram is presented, the system can be split up in sub-operations such that the context diagram becomes a data-flow diagram. For a more elaborate reading on diagrams in relation with formal methods, see [12].

5.1. Physical and Continuous Mathematical Model

For the physical 1D heat transfer problem, a context diagram can be drawn (see Figure 9). The diagram defines the main input and output of the system. It consists of one operation which is the computation of the temperature field on a bar. A user has to supply a source, some boundary conditions and a thermal conductivity, such that a temperature field can be computed. This temperature field is offered to the user. The bar has a fixed geometry, so the user is not supposed to supply it.

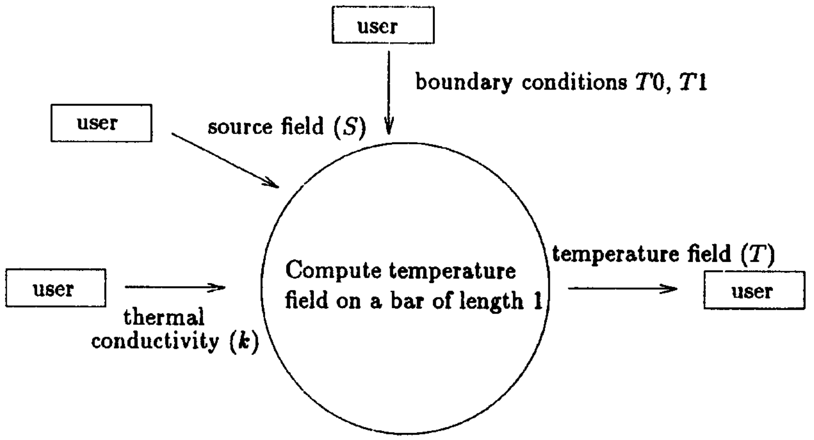
user
boundary conditions $T0$, $T1$
user
source field (S)
Compute temperature field on a bar of length 1
temperature field (T)
user
user
thermal conductivity (k)

Figure 9. Context diagram for the physical 1D heat transfer problem.

At this point, the data can be described in an abstract way independent of any implementation. For each data item, a set to which it belongs should be introduced (see Table 1). The temperature field, for example, is a function from a set of coordinate vectors to a set of values. The coordinate vectors are an element of $[0, 1]$. The values are measured in degrees Kelvin, such that the set of temperature values is $\mathbb{R}/\mathbb{R}^-$. In other words, the field is an element of $\{T : [0,1] \to \mathbb{R}/\mathbb{R}^-\}$.

Table 1. Models for the datatypes to be used for variables in 1D heat transfer.

T	temperature (^{0}K)	$\{T \mid T : [0,1] \to \mathbb{R}/\mathbb{R}^-\}$
$T0$	temperature (^{0}K)	$\mathbb{R}/\mathbb{R}^-$
$T1$	temperature (^{0}K)	$\mathbb{R}/\mathbb{R}^-$
k	thermal conductivity ($J m^{-1} s^{-1}\, {}^0\mathrm{K}^{-1}$)	$\mathbb{R}^+$
S	volumetric heat source ($J m^{-3} s^{-1}$)	$\{S \mid S : [0,1] \to \mathbb{R}\}$

Note that modelling, for example, a temperature field by the above mentioned set is not complete. Operations such as addition, multiplication and others are normally defined on the field. So these operations must also be defined on elements in the above mentioned set. In fact, a small theory should be introduced around a set modelling a data-type. This has already been discussed in Section 4.1.

A necessary extension to the context diagram are a precondition and postcondition defining what the operation does.

POSTCONDITION. The computed temperature satisfies the partial differential equation[4] (1) and the boundary conditions (2) and (3).

PRECONDITION. The input values must be such that the computed temperature is nonnegative. This requirement lays a restriction on the possible source fields. For an arbitrary source field which domain and range are $[0, 1]$ and $\mathbb{R}$, respectively, the partial differential equation and boundary conditions do not guarantee a nonnegative temperature field.

[4]For arbitrary temperature fields, it can only be required that the partial differential equation and the boundary conditions hold in some discrete points and up to some error due to the fact that the problem is modelled by a discrete model (discretization error). For linear temperature fields, the discretization error is zero.

The power of a context diagram in relation with an abstract definition of data and a precondition and postcondition is that it exactly shows *what* happens. It abstracts from *how* the problem is solved, *when* the problem is solved and *where* the problem is solved. It also abstracts from how data is implemented on a computer and how data is offered to the operation. Data can be offered via a file, some GUI or something else. In this paper, it will be left open. A possible elaboration can be found in [4].

It is important to abstract from the type of computer. It gives the possibility to choose a realisation and implementation which is best for a typical computer, i.e., best in the sense of efficiency. Especially in the field of parallel computation, this freedom is important because there does not, and in the near future will not, exist one parallel computer model.

5.2. Discrete Mathematical Model

To solve problem (1)–(3) numerically, it is discretized using a finite difference method. The interval $[0, 1]$ is divided into $n+1$ equidistant nodes. Consequently, an interval length is $h = 1/n$.

The temperature will be discretized in the nodes. Instead of the usual habit to denote a temperature value by T_i, where i is a number assigned to a node, here a temperature value will be denoted by $T_{\underline{x}_i}$ $(i \in \{1, \dots, n+1\})$. This notation emphasizes that a node has some coordinates and that the temperature value is computed in these coordinates. The notation is independent of the distribution of the nodes over the interval.

For the second order derivative of T, a central difference formula is used, namely, $\frac{\partial^2 T}{\partial x^2} = T_{\underline{x}_\ell} - 2T_{\underline{x}} + T_{\underline{x}_r}$. The stencil contains three nodes, $\underline{x}_\ell$, $\underline{x}$, $\underline{x}_r$, where $\underline{x}_\ell$ and $\underline{x}_r$ are the left and right neighbour of $\underline{x}$, respectively. The source is simply discretized by computing its value in a node, $S_{\underline{x}} = S(\underline{x})$. Hence, the discretized form of the differential equation becomes as is formulated in equations (4)–(6) below.

For the discrete model also, a context diagram can be given (see Figure 10). This context diagram differs from the previous one by the fact that the user has to supply the number of intervals in which the bar is divided. The form of the data differs from that of the data in the physical model. The source and the temperature fields are discrete functions. The models for the datatypes are presented in Table 2. The extra input, the number of intervals, must always be positive, and thus it is a positive integer.

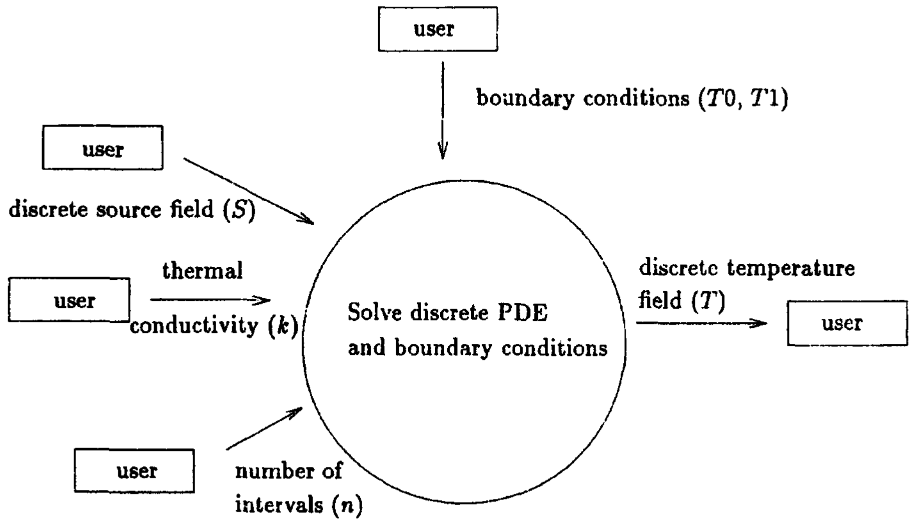
user
boundary conditions (T0, T1)
user
discrete source field (S)
user
thermal conductivity (k)
Solve discrete PDE and boundary conditions
discrete temperature field (T)
user
user
number of intervals (n)

Figure 10. Context diagram for the discrete mathematical model of the 1D heat transfer problem.

The precondition and postcondition which belong to the context diagram differ from those presented for the physical model.

Table 2. Models for the datatypes to be used for *discrete* variables in 1D heat transfer.

n	number of intervals	$\mathbb{N}^+$
T	temperature (^{0}K)	$\{T \mid T : \{\underline{x}_i \mid i \in \{1,\ldots,n+1\}\} \to \mathbb{R}/\mathbb{R}^-\}$
$T0$	temperature (^{0}K)	$\mathbb{R}/\mathbb{R}^-$
$T1$	temperature (^{0}K)	$\mathbb{R}/\mathbb{R}^-$
k	thermal conductivity ($J\,m^{-1}\,s^{-1}\,{}^0\mathrm{K}^{-1}$)	$\mathbb{R}^+$
S	volumetric heat source ($J\,m^{-3}\,s^{-1}$)	$\{S \mid S : \{\underline{x}_i \mid i \in \{2,\ldots,n\}\} \to \mathbb{R}\}$

POSTCONDITION. The original postcondition is replaced by the "discrete postcondition": the discrete partial differential equation and boundary conditions

$$T_{\underline{x}_\ell} - 2T_{\underline{x}} + T_{\underline{x}_r} = -\frac{S_{\underline{x}}}{2kn^2}, \quad \text{for all } \underline{x} \in Domain(T)/\{\underline{0},\underline{1}\} \text{ with} \tag{4}$$

$$T_{\underline{0}} = T0 \tag{5}$$

$$T_{\underline{1}} = T1. \tag{6}$$

The rest of the continuous postcondition remain unchanged. The interval $[0,1]$ is divided in equidistant nodes, thus the postcondition is extended with: $|\underline{x} - \underline{x}_\ell| = 1/n$ and $|\underline{x} - \underline{x}_r| = 1/n$ for all $\underline{x} \in Domain(T)/\{\underline{0},\underline{1}\}$.

PRECONDITION. The precondition is the same as for the physical model, extended with one requirement: the number of intervals is a positive integer.

A more general form of equations (4)–(6) is equation (7). L_d^e stands for the discretized partial differential operator *and* the operators on the boundary. That is why the superscript e is added in the notation

$$L_d^e\mathbf{T} = \mathbf{f}_d^e. \tag{7}$$

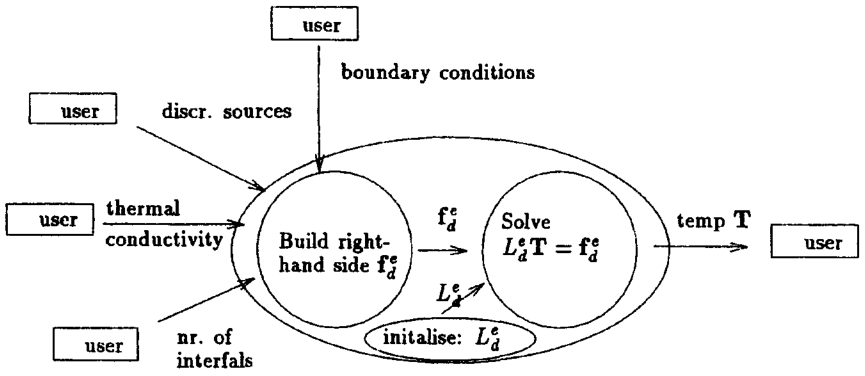

Figure 11. Data flow diagram for the discrete mathematical 1D heat transfer model; second level.

Table 3. Models for the datatypes for f_d^e and L_d^e.

f_d^e	Field	$\{T \mid T : \{\underline{x}_i \mid i \in \{1,\ldots,n+1\}\} \to \mathbb{R}/\mathbb{R}^-\}$
L_d^e		$\{Field \to Field\}$

The procedural abstraction solving the general form (7) can be integrated in the context diagram. This results in Figure 11. In this figure, two extra bubbles are introduced. For each bubble, the data, at least the new data, must be defined. In this case, the only new data is f_d^e and L_d^e (see equation (7)). Both are defined in Table 3. Next, for the new bubbles, preconditions and postconditions must be introduced. For "**Build right-hand side**" this is:

POSTCONDITION. $f^e_{d\underline{0}} = T0$, $f^e_{d\underline{x}} = -S_{\underline{x}}/2kn^2$, where $\underline{x} \in Domain(f^e_d)\{\underline{0},\underline{1}\}$ and $f^e_{d\underline{1}} = T1$.

PRECONDITION. $k \neq 0$, $n \neq 0$.

For **initialise:** L^e_d there are no preconditions. The postcondition is the following.

POSTCONDITION.

$$\begin{aligned}
&L^e_d\mathbf{T}(\underline{x}) = T_{\underline{x}_\ell} - 2T_{\underline{x}} + T_{\underline{x}_r} \text{ for all } \underline{x} \in Domain(T)/\{\underline{0},\underline{1}\},\\
&L^e_d\mathbf{T}(\underline{0}) = T_{\underline{0}},\\
&L^e_d\mathbf{T}(\underline{1}) = T_{\underline{1}}.
\end{aligned}$$

The preconditions and postconditions for "**Solve** $L^e_d\mathbf{T} = \mathbf{f}^e_d$" are presented below.

POSTCONDITION. $\mathbf{T}$ must satisfy $L^e_d\mathbf{T} = \mathbf{f}^e_d$ (which also requires that the domain of T is equal to the domain of b).

PRECONDITION. A solution exists if the right-hand side field is in the image of the discrete differential operator: $\mathbf{f}^e_d \in Image(L^e_d)$.

Table 4. Model for the datatype to be used for *discrete* variable T in 1D heat transfer.

T	Field	$\{T \mid T : \{\underline{x}_i \mid i \in \{1,\ldots,n+1\}\} \to \mathbb{R}/\mathbb{R}^-\}$

Table 5. Model for the realisation of the datatype to be used for *discrete* variable T in 1D heat transfer.

T	*RealisedField*	*Values* $\times$ *Coordinates*
	Values	$\{\{1,\ldots,n+1\} \to \{T_i \mid i \in \{1,\ldots,n+1\}\}\}$
	Coordinates	$\{\{1,\ldots,n+1\} \to \{\underline{x}_i \mid i \in \{1,\ldots,n+1\}\}\}$

```
$AbsF$: $RealisedField \to Field$
$AbsF : Rf \mapsto AbsF(Rf)$
where
    $AbsF(Rf) : \{\underline{x}_i \mid i \in \{1..n+1\}\} \to \mathbb{R}/\mathbb{R}^-$
    $AbsF(Rf) : \underline{x} \mapsto Values(Rf)(i\_of(\underline{x}, Rf))$
    where
        $i\_of : Coordinates \times RealisedField \to \{1..n+1\}$
        $i\_of : (\underline{x}, Rf) \mapsto i$
            where $\underline{x}(i) = Coordinates(Rf)(i)$
```

Figure 12. An abstraction function for the realisation of a matrix.

5.3. A Matrix Equation

Equations (4)–(6) can be represented by a matrix equation. Consequently, instead of solving equations (4)–(6), the matrix equation, for which a lot of theory has been developed, can be solved. The matrix equation is obtained after chosing an ordering of all the nodes. After ordering, the values $T_{\underline{x}}$ are written as a vector. Similarly, the form of the matrix and right-hand side are fixed. Below, it will be shown how to express the development step (i.e., representing equations (4)–(6) by a matrix equation) in software development, but two ordering examples are presented.

The most natural ordering of the coordinates follows the ordering of the real numbers (see equation (8)). However, different orderings are allowed. Different ordering result in a different form of the matrix. Assuming that the order of the matrix is odd, renumbering the nodes by starting with the odd numbered nodes (odd number according to the old ordering) and finishing with the even numbered nodes results in a different matrix form (9). The effect of different orderings on the efficiency of certain matrix equation solvers is very well known [20], especially for parallel computing [21].

$$\begin{bmatrix} 1 & 0 & 0 & 0 & & & 0 & \\ 1 & -2 & 1 & 0 & & & & \\ & \vdots & & & & & \vdots & \\ & & & & 0 & 1 & -2 & 1 \\ & 0 & & & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_{\underline{x}_1} \\ T_{\underline{x}_2} \\ \vdots \\ T_{\underline{x}_n} \\ T_{\underline{x}_{n+1}} \end{bmatrix} = \begin{bmatrix} T0 \\ -\frac{S_{\underline{x}_2}}{2kn^2} \\ \vdots \\ -\frac{S_{\underline{x}_n}}{2kn^2} \\ T1 \end{bmatrix} \tag{8}$$

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -2 & 0 & & 0 & 0 & 1 & 1 & & 0 & 0 & 0 \\ & \vdots & & & & \vdots & & & & & \vdots & \\ 0 & 0 & 0 & & -2 & 0 & 0 & 0 & & 1 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & -2 & 0 & 0 & \cdots & 0 & 1 & 1 \\ 1 & 1 & 0 & & 0 & 0 & -2 & 0 & & 0 & 0 & 0 \\ 0 & 1 & 1 & & 0 & 0 & 0 & -2 & & 0 & 0 & 0 \\ & \vdots & & & & \vdots & & & & & \vdots & \\ 0 & 0 & 0 & \cdots & 1 & 1 & 0 & 0 & \cdots & 0 & -2 & 0 \\ 0 & 0 & 0 & & 0 & 0 & 0 & 0 & & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_{\underline{x}_1} \\ T_{\underline{x}_3} \\ \vdots \\ T_{\underline{x}_{n-2}} \\ T_{\underline{x}_n} \\ T_{\underline{x}_2} \\ T_{\underline{x}_4} \\ \vdots \\ T_{\underline{x}_{n-1}} \\ T_{\underline{x}_{n+1}} \end{bmatrix} = \begin{bmatrix} T0 \\ -\frac{S_{\underline{x}_3}}{2kn^2} \\ \vdots \\ -\frac{S_{\underline{x}_{n-2}}}{2kn^2} \\ -\frac{S_{\underline{x}_n}}{2kn^2} \\ -\frac{S_{\underline{x}_2}}{2kn^2} \\ -\frac{S_{\underline{x}_4}}{2kn^2} \\ \vdots \\ -\frac{S_{\underline{x}_{n-1}}}{2kn^2} \\ T1 \end{bmatrix} \tag{9}$$

Representing the discrete differential operator, the unknown field, and the field at the right-hand side of equation (7) by a matrix, a vector, and a vector, is a form of data reification. To be sure that the realisation is correct, a realisation model and an abstraction function must be found.

The realisation model can easily be found. The field, for example, is represented by a vector and, not to forget, a vector of coordinates. For those who think in terms of arrays and records, the objective is to realise `Field` as a record of two components. The first is an array of temperature values and the second is an array of coordinate-vectors. The array indices denote the number of a node in the already chosen ordering. Based on this idea, a realisation model can be formulated.

Introducing an ordering for all the $T_{\underline{x}}$ is no more than introducing an index for a node $\underline{x}$ and its value $T_{\underline{x}}$. On the one hand, the index points to a temperature value, which will be denoted by T_i, and on the other hand, the index points to the coordinate-vector, which will be denoted by $\underline{x}_i$. In terms of functions and sets, the set of *indexed temperature field values* is a set of functions from its indices to its values: $\{\{1,\dots,n+1\} \to \{T_i \mid i \in \{1,\dots,n+1\}\}\}$. And the set of *indexed coordinate-vectors* is a set of functions from indices to coordinate-vectors: $\{\{1,\dots,n+1\} \to \{\underline{x}_i \mid i \in \{1,\dots,n+1\}\}\}$. A realisation model for temperature field is the Cartesian product of these two sets which are called *Values* and *Coordinates*, respectively, (see Table 5).

Now that a realisation of the abstract data-type is chosen, the question arises whether it is a correct realisation. As stipulated in Section 4.1.2, an abstraction function must be found. One such abstraction function is presented in Figure 12. The first line in the figure defines the domain, *RealisedField* (see Table 5), and the range *Field* (see Table 4). The next line contains the function definition. *AbsF*(Rf) is also a function, in fact, it is a field. *AbsF*(Rf) maps a coordinate-vector on a field-value. The field-value is found by computing the *index* of the coordinate-vector $\underline{x}$ by *i_of*, and subsequently, computing *Values*(Rf)(*index*). *Values*(Rf) is the first element in the tuple Rf. As shown in Table 5, it is a mapping from a set of integers to $\mathbb{R}$.

The computation of the index of a coordinate-vector is handled by the function *i_of*. Its domain and range are defined in Figure 12. The function definition is implicit. It only specifies that an index i is returned, which is such that $\underline{x}(i) = Coordinates(Rf)(i)$. *Coordinates*$(Rf)$ is the second element in the tuple Rf, a mapping defined in Table 5.

Proof of adequacy, i.e., every element in the abstract model has a "realisation," is obvious. The function is total, every element in the realisation model has an image.

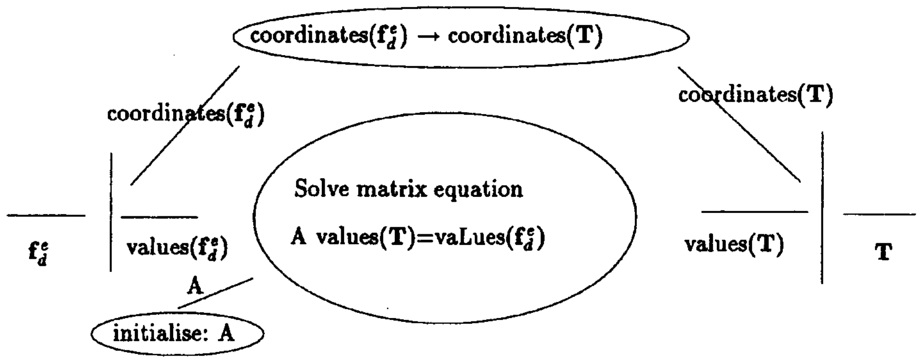

Figure 13. Data flow diagram for solving the set of equations in the discrete mathematical 1D heat transfer model; part of third level.

In Figure 13, a data-flow diagram is presented which replaces the bubble "Solve $L^e_d\mathbf{T} = \mathbf{f}^e_d$" in Figure 11. To stipulate the fact that this data-flow diagram is a level lower in *data-abstraction*, two vertical lines are introduced in Figure 13. These lines can be seen as filters, filtering the representation out of the abstract data-types. Furthermore, the initialisation of L^e_d has been replaced by the initialisation of A. Accordingly, the formal method for the realisation of L^e_d, also a realisation model and abstraction function, must be defined; however, for this problem the realisation of the initialisation of L^e_d by the initialisation of A is trivial.

Figure 13 is a realisation of the procedural abstraction "Solve $L^e_d\mathbf{T} = \mathbf{f}^e_d$." An algorithm can be derived from the figure:

```
initialise : $A$||
$\underline{x} := values(\mathbf{T})$||          ||$Coordinates(\mathbf{T}) := Coordinates(\mathbf{f}^e_d)$
$\underline{b} := values(\mathbf{f}^e_d)$;
Solve $\underline{x}$ given $A\underline{x} = \underline{b}$
```

Here "||" stands for parallel execution and ";" stands for sequential execution. Note that the order of execution can be derived from the dataflow in Figure 13. The first statement in the algorithm is a call for the above explained procedural abstraction which initialises the matrix. The other assignments are calls to a procedural abstraction which copies a vector. The statement "Solve $\underline{x}$ given $A\underline{x} = \underline{b}$" calls a procedural abstraction which is defined in Section 4.1.

Finally, the discussion of the algorithm can end with the following remarks concerning efficiency. First, the initialisation of A does not have to be implemented as a procedural abstraction, but can be directly inserted in the object-code! Second, in case the right-hand side ($\mathbf{f}^e_d$) has already been computed for some problem and stored in a file and in case the unknown $\mathbf{T}$ must be stored in a file, the procedural abstraction to copy the coordinates of ($\mathbf{f}^e_d$) and $\mathbf{T}$ can be implemented as a copy of parts of files. But of course, such optimisations can only be performed after development of procedural abstractions and data abstractions has been completed.

6. CONCLUSIONS

Applying abstraction in software according to the approach in the paper improves the quality of the software. All development steps are identified and motivated! Two basic development steps are: realising a procedural abstraction and realising a data abstraction. An example of the first was the choice of solving the equations by Gauss elimination. An example of the second is representing a discrete temperature field by a "record of two arrays." It has been concluded that a development step transforms a model of the problem into a model at a lower level of abstraction.

The suggested approach for developing software imposes a clear structuring of the software. On one hand there is a functional decomposition of procedural abstractions keeping the data representation the same; on the other hand, there is reification of data abstractions keeping the semantics of the procedural abstractions the same. This paper also shows that the obtained structure in the software enables a simple extension of new mathematical techniques. It is immediately clear where a new matrix equation solver has to be inserted into the software.

For the benefit of this flexibility, it is essential that a specification should only concern "what." If the postcondition of a solver of linear equations would express that the unknown is found by Gauss elimination, then the replacement of the current realisation by another direct method would require a proof of equivalence of both methods. This proof is more difficult than the proof that the unknown satisfies the linear equations. Furthermore, the "Gauss-postcondition" is more complex. First, one has to realise that the postcondition concerns Gauss elimination, whereafter it is obvious that a system of linear equations is solved. An even more abstract postcondition is required when also considering iterative matrix equation solvers. The postcondition should specify that the unknown is found upto some error.

Developing software according to the formal framework into which software development can be cast implies an accurate recording of *all* development steps. This is especially important in the field of parallel computing. At the moment, no universal parallel computer model exists in this field. However, software should still easily be ported to other (maybe future) computer models. A port that must result in efficient software requires, in general, the implementation of other techniques and data structures. Consequently, it is a necessity that the new techniques and methods can easily be integrated into the software, and that existing techniques and methods can be rewritten in terms of the new data structures.

A possible adaptation for parallel computation is the replacement of the present solver of linear equations by a parallel solver. For example, as described in Parchol [22], a solver for symmetric banded matrix systems that is especially suited for distributed parallel systems can be used. Such a solver demands a certain data distribution of the matrix, the unknown and the right-hand side. In the current problem, the right-hand side is computed from sources, etc. Hence, this data should also be distributed in such a way that the amount of communication is reduced. Note that the distributed data-types for matrix and vector are realisation data-types for the corresponding abstract data-types. The way of distribution can be described by means of an abstraction function.

In Figure 14, development steps in software development for a 1D heat transfer problem are presented. A box in the figure represents a context diagram or a dataflow diagram. For instance, the first box represents the context diagram in Figure 9. For each arrow in the figure, the question whether the representation is exact or not is answered. At the arrows where a representation is not exact, numerical mathematics must validate the representation.

In this paper, the transition from the continuous formulation of the problem (the context diagram in Figure 9) to the discrete formulation (diagram in Figure 10) is not formally described. A better analysis of the development step identifies a model at an intermediate level of abstraction. This model is the partial differential equation and boundary conditions but then formulated for field functions with certain restrictions. The discrete mathematical model is an exact represen-

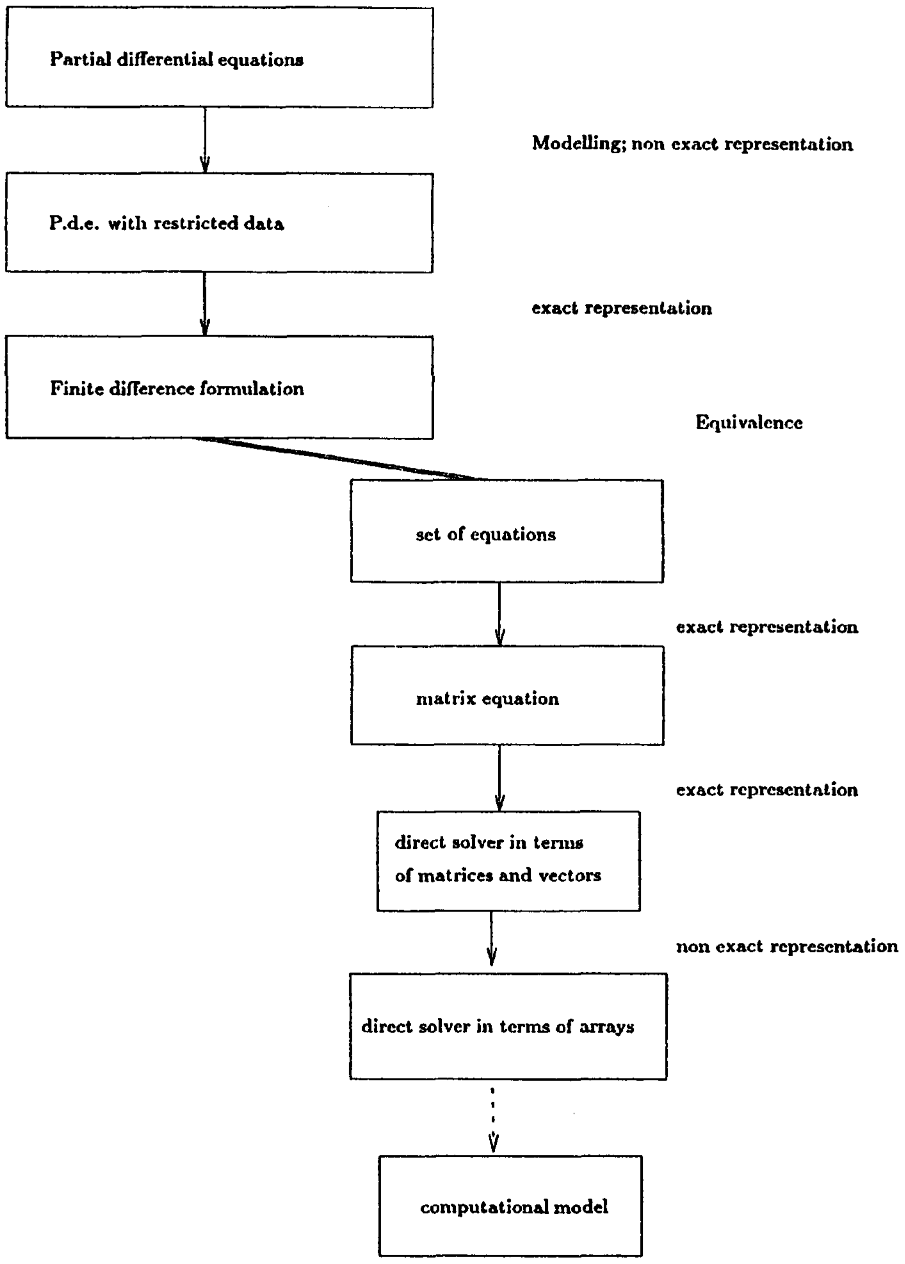
Partial differential equations
Modelling; non exact representation
P.d.e. with restricted data
exact representation
Finite difference formulation
Equivalence
set of equations
exact representation
matrix equation
exact representation
direct solver in terms of matrices and vectors
non exact representation
direct solver in terms of arrays
computational model

Figure 14. Development steps in software development for a 1D heat transfer problem.

tation of this model. The restriction on the field functions is such that the discretisation error is zero. It can also be remarked that the extra input variable (the number of grid points), which appears in the diagram in Figure 10, comes already at this point into the design.

The previous remark confirms the proposition that the mathematical solution process and the software development process are integrated in one process. The whole process can be formulated in terms of procedural abstractions and data abstractions. Consequently, software can be expressed in abstract mathematical terms, e.g., meshes, differential operators, bilinear forms, etc. The highest level in abstraction is the continuous problem (the context diagram in Figure 9) which is transformed into a discrete formulation (the diagram in Figure 10), the next level of abstraction. Subsequently, this is transformed into a matrix equation, again a level of abstraction.

It must be noted that the followed approach does not have disadvantages with respect to efficiency. Efficiency can be obtained by changing realisations which are specific for a particular computer. For example, on vector computers the vector length must fit with the required vector length, but on message passing computers, communication must be reduced. Consequently, when descending in abstraction, software will be focused more and more on a specific computer and thus it will be less general and portable (efficiency at the cost of generality). When and how the realisation decisions are made is the responsibility of the software developer. That is what makes the job interesting!

For a 2D heat transfer problem, for instance, it can be useful to reorder the nodes ($\underline{x}_i$) such that the linear equations have a better structure and that a more efficient solver can be used. It is clear that the reordering operation takes place just before and after the call for the solver and nowhere else.

In the ATES project, it became evident that tools are a necessity. Without tools, formal software development will never be practiced. It also became clear that automatic proof tools cannot be used. They must be interactive to allow a user to use his intelligence.

Finally, it must be remarked that, in general, **formal** software development goes in cooperation with **informal** software development as two interactive processes. It is not so easy to formalise all the new ideas. Possibly, the ATES-system or a similar system can serve as a sort of intermediate between the "formal specification world" and the "programming world."

REFERENCES

1. A. Puccetti, Editor, *The Programming and Proof System ATES*, Volume 1 of Research Reports ESPRIT, Springer-Verlag, Berlin, (1991).
2. H.H. ten Cate and B.H. Gilding, The finite element method implemented in the programming system ATES, In *Numerical Methods in Thermal Problems, Vol. VI, Part I, Proc. Sixth Int. Conf.*, (Edited by R.W. Lewis and K. Morgan), pp. 130–140, Pineridge Press, (1989).
3. A. Puccetti, The integrated software development and verification system ATES, In *VDM '91 Formal Software Development Methods*, Volume 1, (Edited by S. Prehn and W.J. Toetenel), pp. 629–645, Springer-Verlag, New York (1991).
4. H.H. ten Cate, Towards formal specification and proof of finite element software within the ATES development system, Ph.D. Thesis, University of Twente, the Netherlands (1991).
5. K.M. Chandy and J. Misra, *Parallel Program Design*, Addison-Wesley, Reading, MA, (1988).
6. B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, MIT Press, Cambridge, (1986).
7. I.D. Hill, Wouldn't it be nice to write programs in English—or would it?, *Comput. Bull. Notes* **16** (1972).
8. B. Ratcliff, *Software Engineering: Principles and Methods*, Blackwell, Oxford, (1987).
9. B. Liskov and V. Berzins, An appraisal of program specifications, In *Software Specification Techniques*, pp. 3–23, (1979).
10. L. Kronsjö and D. Shumseruddin, Editors, *Advances in Parallel Algorithms*, Advanced topics in Computer Science, Blackwell Scientific Publications, London, (1992).
11. I. Sommerville, *Software Engineering*, Addison-Wesley Publ. Co., Wokingham, (1989).
12. D. Andrews and D. Ince, *Practical Formal Methods with VDM*, The McGraw-Hill series in software engineering, McGraw-Hill, London, (1991).
13. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Publications, (1993).
14. C.B. Jones, *Systematic Software Development Using VDM*, Second edition, Prentice-Hall, Hemel Hempstead, UK, (1990).
15. M. Cosnard, M. Marrakchi and Y. Robert, Parallel Gaussian elimination on an MIMD computer, *Parallel Computing* **6**, 275–296 (1988).
16. D. Gries, *The Science of Programming*, Springer-Verlag, New York, (1983).
17. R.W. Floyd, Assigning meaning to programs, *Proc. Symp. in Applied Mathematics*, In *Mathematical Aspects in Computer Science*, (Edited by J. Schwartz), **19**, 19–32, New York, (1967).
18. C.A. Hoare, Proof of correctness of data representations, *Acta Inf.* **1**, 271–281 (1972).
19. E.W. Dijkstra, Guarded commands, nondeterminacy, and formal derivation of programs, In *Programming Methodology*, (Edited by D. Gries), 166–175, Springer-Verlag, New York, (1978).
20. A. George and J.W.H. Liu, The evolution of the minimum degree ordering algorithm, *SIAM Review* **31** (1989).

21. H.X. Lin, A methodology for the parallel direct solution of finite element systems, Ph.D. Thesis, Delft University of Technology, (1993).
22. H.X. Lin and M.R.T. Roest, Parallel Cholesky factorization of symmetric banded systems, In *Proc. ParCo '93 Conference*, Grenoble, France (to appear).