Reports in Informatics no. 173, June 1999, pp. 27, ISSN 0333-3590.
Department of Informatics, University of Bergen, N-5020 BERGEN, Norway

Formal Software Engineering for Computational Modeling*

M. Haveraaen[†] H.A. Friis[‡] T.A. Johansen[§]
Universitetet i Bergen, N-5020 Bergen, Norway

June 23, 1999

Abstract

Software itself may be considered a formal structure and may be subject to mathematical analysis. This leads to a discipline of formal software engineering[a], where an understanding of what software components are and how they may interact is used to engineer both the components themselves and their composition. A strategy is using the concepts that are suited for organising the problem domain itself to organise the software as well. Here we sketch this approach for the development of computational modeling software, in particular a family of related programs for simulation of elastic wave propagation in earth materials, and present some data on the technique's effectiveness.

[a]Note that formal software engineering is not the same as the use of formal methods in software engineering.

1 Introduction

Formal methods and reuse of software components has for a long time been acknowledged as important for the improvement of software quality and a reduction in software cost. Here we want to raise the issue of investigating the formal aspects of software to achieve significant improvements in software quality and development productivity. Hopefully this may form the start of a discipline of *formal software engineering*. Formal software engineering is not the same as the use of formal methods in software engineering. Rather it is a focus on the formal, mathematical side of software artifacts, and using this insight in the engineering of software products. Following the tradition of mathematics, applying such an insight does not require the use of formal methods. Informal arguments suffice, as long as they are rooted in a precise semantical understanding. The approach to formal software engineering used in this paper is based on the algebraic "toolbox" [EGW98]. This ranges from universal algebra, suited to investigate the language of the problem domain, to category theory, suited to discuss design principles for software architecture.

*This research was supported in part by the European Union under ESPRIT Project 21871 (SAGA — Scientific Computing and Algebraic Abstractions), The Research Council of Norway (NFR), and by a computing resources grant from NFR's Programme for Supercomputing.

[†]Institutt for Informatikk; Magne.Haveraaen@ii.uib.no

[‡]Formerly at Institutt for Informatikk, presently at Rogalandsforskning; P.O.Box 2503, Ullandhaug, N-4004 Stavanger, Norway; HelmerAndre.Friis@rf.no

[§]Institutt for den faste jords fysikk; TorArne.Johansen@ifjf.uib.no

When designing computer programs for some problem domain, one is at least faced with three problems: (1) what are the concepts that have to be used for the construction of the software in this domain, (2) what is a good programming notation for these concepts, and (3) how can they be implemented as software code. This has spawned work in areas like *domain specific embedded languages* (DSEL) [Hud96] and *software architectures* [BST99]. Domain specific embedded languages provide the concepts from the problem domain with syntax in a general purpose programming language setting ensuring full algorithmic and data structure declaration capabilities. Software architecture is the organisation of software into packages and modules and how to combine these in order to build specific software. Ideally software should be decomposed in such a way that different considerations may be confined to separate modules. Also, modules that provide alternative implementations of a concept should be easily interchangeable. This requires an extensive analysis of both the problem domain and of the software structuring methodologies available. Doing this well gives a very flexible software structure, easy to tailor for specific needs, as well as adapt to changing requirements – a characteristic of mature software design [Rac95].

Domain specific languages are abundant in all our application development tasks. Very often the DSL is "invisible", in the sense that it was embedded already in the design of the programming language we will be using, and thus we often do not relate consciously towards what DSL we are using. In the 1960'es and 1970'es building domain concepts into a programming language was the common way of providing DSLs. With the advent of advanced modularisation concepts such as abstract data types and object orientation, DSLs may now be provided as separate libraries. Thus a programming language may be adapted to any domain by including the appropriate software libraries for that domain. Often a domain may have several distinct, but coexisting, domain specific languages. Then it becomes very important to be aware of this, and be able to make conscious decisions about which one to use. In the worst case, it may be beneficial to develop an alternative DSL – a DSL that may have better properties for expressing simple solutions to complex problems.

Our idea for a lifecycle model for developing and implementing a domain specific language (DSL) is based on the following[1]:

1. *Establish an appropriate DSL* by defining important properties of the problem domain concepts using algebraic specifications. This identifies sorts and function symbols to express application data and algorithms. It also nails down the semantics of the DSL.

2. *Validate the suggested DSL* by checking its

    - *usefulness* by formulating problems and designing solutions treating the DSL as a DSEL. (This is mostly a test of the concepts themselves, but may also be part of an application development process, the functional requirements and high level design phases of a typical application development process, that may be continued independently of the DSL lifecycle process.)
    - *implementability* by designing data structures for the sorts and algorithms for the function symbols. The design should be verified by showing the relationship between the code and the specified concepts. (Note that this only will provide

[1] As a proper lifecycle process the model needs to be amended with additional feedback and control loops in the same way as the basic application development lifecycle model, the waterfall model [Roy70], has been amended, [Boe76] giving just a few variations.

one of many possible designs, each with different accuracy and resource usage characteristics, but that one such design is sufficient to show implementability of the concepts.)

3. *Decide the architecture of the software library* that provides the domain specific language concepts. This requires grouping the sorts and functions for software components, studying their interplay and trying to parameterize the components as much as possible. The aim is to reduce the number of components and the complexity[2] of each by focusing on reusability. This generally increases the versatility of the components, and simplifies the library itself. It may also greatly reduce the work needed to implement the full library. Categorical reasoning is useful for this purpose.

4. *Design and implement a prototype of the software library* by coding the data structures and algorithms in a suitable programming language.

5. *Maintain the library* by adding variants of the components, as well as correcting (unavoidable) errors. Variants may be demanded by application program development in order to meet efficiency requirements.

The first of these steps identifies the concepts and finds a notation for these. Once the specifications are in place, it is possible to experiment with the notation and concepts to design programs that will use the library for solutions of actual problems. This allows early checking of the usefulness of the concepts, and an adjustment of the concepts and notation at this stage is fairly cheap. We also have to check that the concepts are implementable before they are meaningful. Thus we have both outward (usability) and inward (implementability) requirements on a DSL in the validation of its specification. When specifying the domain concepts we develop a DSL, while we treat it as a DSEL when designing solutions.

We firmly believe that the development of a DSL and its implementation as a software library is a separate activity from application program development, and that they have different software lifecycle models[3]. While an application program is developed to solve a specific problem for one costumer, a DSL is to serve many application developers. Thus finding the right domain concepts is crucial, and devising a flexible library architecture is important, for the success of both the library development and the application development. Of course one will never in practice proceed along the sketched steps of the lifecycle models in linear order. A more reflected understanding of a lifecycle model will introduce all kinds of feedback loops, quality assurance steps, detailed guidelines on how to approach standard problems, etc. Such issues are investigated by the software process community, and is captured in models like [PCCW93]. The basic steps we have identified are still central in these models.

This paper is devoted to a case study developed according to the ideas above. The case study is taken from the area of computational modeling, in the typical form of a physical phenomenon described by a partial differential equation (PDE). Computational modeling of real world phenomena is becoming an important research tool in the sciences. Currently this is hampered by the time and effort needed to develop good computational models, and the

[2] *Software complexity* is a measure of the difficulty and cost of writing the software. It does not relate to the resource usage at runtime which is the subject of traditional complexity theory.

[3] An application development lifecycle model is based on the steps of requirements specification where we utilise a DSL, design using a DSEL, validating design with respect to requirements, coding and maintenance [Roy70].

time and cost needed to port such models onto a high performance computer. We ascribe much of these costs to the use of a less than optimal DSL for conventional numerical software, namely that of indexed array structures, as exemplified by Fortran [For66]. A conventional solver for a PDE embodies the discretisation method, coordinate system, as well as the actual solver algorithm. Typically such software only handles the limited set of problems it was developed for, and can not easily be adapted to related phenomena or account for change of numerical discretisation methods. In many ways such a solver becomes a legacy code for the group that developed it: both being their main tool for success, but also limiting what problems they may tackle.

Instead we want to develop an alternative DSL, namely that of *coordinate free numerics*, for this domain. This DSL captures more of the abstract concepts of the underlying mathematics, and which therefore may blend more naturally with modern notions of software structure. This work started with the algebraic specification of concepts from partial differential equations in [HMMK92]. Here we continue through the remaining steps of the the development process and we sketch the algebraic tools we have chosen to aid us through this. We also measure the effectiveness of this approach to software development by implementing a collection of application programs for the acoustic imaging of small and large scale geological objects, problems important in oil exploration. This problem is challenging, in that the same set of fundamental PDEs has to be formulated for various kinds of experimental setups, e.g., in cartesian or cylindrical (borehole) coordinates, and with various kinds of geophysical models and boundary conditions.

The choice of computational modeling as our domain was motivated in part by its general importance, but also since this is an area to a large extent neglected by the software methodology community. Showing the relevance of software engineering methodology to this area may then both open it up for more research from the software community, and may also benefit the practitioners in the computational modeling domain. The lack of familiarity with this area for most software engineers has made us include material describing more of the domain background than normal for case studies. We feel this may be needed, in order for the reader to appreciate the complexity of this kind of software and to gain an insight in why the proposed methodology is beneficial for an area which has been selfsufficient since the early days of computing.

This paper is organised as follows: Section two sketches the use of algebraic techniques for investigating domain specific concepts and defining the coordinate free numerics DSL. Then we sketch the specific problems we will use as examples, both in order to validate the DSL, and in order to indicate the potential of this approach. In section four we discuss the software architecture for the DSL and describe the Sophus software library which implements the DSL. Section five presents how the sample problems were solved, how a version of the DSL library adapted for the problems at hand were developed, and also presents new variations of the problem and solutions to these. Then, in section six, we discuss the experience gained. Finally we summarise our findings in section seven.

2 Developing the domain specific language

There is a long tradition in looking at software as a formal entity. This spans from work in programming language semantics [Flo67], via axiomatic formalisms [Hoa72] and systematic development techniques [Dij76] to development by program refinement [Bac81] and proof tools [MW80]. One crucial observation is that a program text can be moved between computers (and compilers), but that the results computed by the program text may depend on the computer[4]. This leads to an acknowledgment of the distinction between syntax and semantics. Such a distinction also exists in the mathematical discipline of universal algebra, which originated with [Whi98] and had matured by the 1960's, [Coh65, Grä68]. In universal algebra the syntactic entities are called a *signature* and the semantics a *model* for that signature. It took till the late 1970's till universal algebra, in the form of many-sorted universal algebra, was employed in computer science [GTWW75, GH78]. By that time category theory, [Mac71], was also making an influence in how to structure mathematical concepts. Now universal algebra and category theory is used together in the form of algebraic development methodologies [EGW98].

Thus algebraic tools seem to have the ability to bridge the gap between formalising the problem domain and the software domain, at least for any domain suitable for formal analysis. This is a property needed of tools that are to be useful for formal software engineering.

2.1 Domain investigation tools: algebraic specifications

A *signature* Σ declares a set of *sort names* $s, s_1, \ldots, s_n$, and function symbols $f : s_{i_1}, \ldots, s_{i_m} \to s_{i_{m+1}}$, where $s_{i_1}, s_{i_2}, \ldots, s_{i_m}$ for $m \geq 0$ are the argument sorts and $s_{i_{m+1}}$ is the result sort (including the case $m = 0$ for a constant). Obviously we may think of a sort as a type name or a class name in programming language terms. A function symbol corresponds to a side-effect free function or typed method. A procedure or method that changes its environment (has side-effects) can be decomposed into one or more side-effect free functions and explicit assignments to program variables. So we may treat a signature as idealised declarations in a program.

A *model* A for a signature Σ defines for each sort s of Σ a mathematical set $A(s)$, called the carrier, and for each function symbol $f : s_{i_1}, \ldots, s_{i_m} \to s_{i_{m+1}}$, a mathematical function $A(f) : A(s_{i_1}) \times \cdots \times A(s_{i_m}) \to A(s_{i_{m+1}})$. In a programming context we may let $A(s)$ be a data structure, perhaps coupled with a data invariant, which defines the set of values that may be stored in the data structure, or the subset thereof that satisfies the data invariant. Likewise, $A(f)$ may denote an algorithm, which defines a computable function from its argument data values to its result data values.

A specification restricts the class of allowable models for a signature. Algebraic specifications only focuses on the properties that we want satisfied, rather than devising specific constructions of models. Thus it is a rather abstract approach, but permits both mathematical models and programming language oriented models. The CASL specification language [Mos97] is an attempt to bring together and standardise various approaches to algebraic specifications. Given a signature Σ and a specification, we may ask whether an implementation satisfies the specification. Already in [Mor73] this problem has been addressed in a clear way, and used as a development technique. This is currently known as *programming by contract*, [Mey91], and is supported by object-oriented programming languages.

2.2 Domain concepts for partial differential equations

The computational modeling domain is that of mathematics, typically the field of partial differential equations (PDEs).

[4]This problem was much more evident a few years ago, as it is gradually reduced through standardisation.

If we start investigating the problem domain concepts using algebraic methods, we will of course rediscover the basic structures of algebra[5]: monoid, group, ring, field, vector space, linear mappings (matrices), tensors (which generalise rings, fields, vectors, matrices and (multi)linear mappings), etc. [HMMK92] contains specifications of many of these concepts in a start at analysing this problem domain using algebraic software methodologies. As an example, a ring R has binary operations $+$ (addition), $-$ (subtraction) and $*$ (multiplication), and constants 0 (zero) and 1 (one). These form the ring signature,

$$\begin{aligned} + &: R, R \to R, \\ - &: R, R \to R, \\ * &: R, R \to R, \\ 0 &: \to R, \\ 1 &: \to R. \end{aligned}$$

This is a slight deviation from the presentation in mathematics, where 0 and 1 are perceived as elements of the carrier, rather than symbols of the interface, and the additive inverse is treated the same way. An algebraic specification of ring could be the following, where a, b, c range over all ring elements R.

$$(a+b)+c = a+(b+c), \tag{1}$$
$$a+b = b+a, \tag{2}$$
$$(a*b)*c = a*(b*c), \tag{3}$$
$$(a+b)-b = a, \tag{4}$$
$$0+a = a, \tag{5}$$
$$1*a = a, \tag{6}$$
$$a*1 = a, \tag{7}$$
$$(a+b)*c = (a*c)+(b*c), \tag{8}$$
$$a*(b+c) = (a*b)+(a*c). \tag{9}$$

Here we see that any model for a ring must obey the laws that addition is associative (1) and commutative (2), subtraction is the inverse of addition (4), and multiplication is associative (3) and distributes over addition in the familiar way (8)–(9). Further, the neutral element with respect to addition is denoted by 0 (5) and with respect to multiplication by 1 (6)–(7). The neutrality and inverse axioms are strong enough to uniquely identify the constants, making it possible to ignore their presence, right up until the moment they again are needed. This simplifies the specification of, e.g., linear functions $L : R \to R$ to

$$L(a+b) = L(a)+L(b), \tag{10}$$
$$L(a*b) = a*L(b), \tag{11}$$

because equations such as $L(0) = 0$ and $L(-b) = -L(b)$ can easily be derived. Reducing the text of specifications make the important points clearer.

Continuing the analysis process for the realm of PDEs we note that a basic assumption is that every spatial point in the physical world, such as a 3-dimensional section of the earth,

[5] It was the investigation and generalisation of these concepts that lead to the discovery of universal algebra [Whi98].

can be represented by an element of a set $\mathcal{M}$ called a *manifold*. The physical properties are then ascribed to each point in the form of a *value field*, akin to a function from the manifold to some value domain. A value field has the same algebraic properties as the value domain it self. If the values at each point are reals, such as those for pressure or density, they are said to form a scalar field. A scalar field has ring properties. If they are vectors, such as those for particle displacement, they are said to be vector fields. A vector field likewise has vector properties, with the corresponding scalar being the scalar field. The values may also be matrices, linear or multi-linear mappings, or some other form of data, such as tensors, which generalise scalars, vectors, matrices and multilinear mappings. Tensor fields correspondingly are tensors over vector and scalar fields. If the manifold has sufficient structure, at least a notion of proximity and direction, we may define integration (internal and surface integrals) and differentiation operators on the value fields, such as Lie derivatives, gradients, and divergence, provided the value fields are smooth enough. A time dependent partial differential equation provides a relationship between spatial derivatives of tensor fields representing physical quantities and their time derivatives. Given constraints in the form of the values of the tensor fields at a specific instance in time together with boundary conditions, the aim of a PDE solver is to show how the physical system will evolve over time.

In [HMMK92] we sketched the algebraic specification of many of these concepts. Although they are standard mathematical concepts, they are rarely presented strictly as a signature with axioms. Doing a proper algebraic specification required work which became more involved as we approached the more advanced concepts, but no really hard problems occurred. However, a noticeable effect was that we were moving from the normal indexed-based presentation of the concepts, and in the direction of coordinate free mathematics, as we pursued identifying the concepts involved. This may be because our analysis tool, algebraic specifications, favours the use of high-level, abstract concepts. A coordinate free formulation is valid independent of choice of coordinate system, i.e., it will be valid whether we use cartesian coordinates, cylindrical coordinates, or some other curvilinear coordinate system. This readily gives a much greater flexibility in choosing implementation strategies: some coordinate systems simplify the implementation of the operators and boundary conditions by introducing symmetries and vanishing terms (terms that will be 0 and thus may be eliminated from the computation), while other coordinate systems may improve numerical accuracy or reduce the amount of computation needed. From this it appears that coordinate free mathematics is a much more versatile framework then the indexed-based one. An introduction to coordinate free mathematical physics can be found in [Sch80].

2.3 Domain notation versus programming notation

Thinking about how to blend this DSL into a programming language we made another interesting observation. Mathematics and the specifications' notation favour use of functional and operator style expressions. An object-oriented programming language like C++ [Str97] favours implementations using imperative style, where argument variables are modified or mutated. As an example, consider an infix binary operator like +. C++ allows the declaration of an operator

```
template<class T> T + (const T & a, const T & b) const;
```

which is used in infix notation as `a+b` which returns the sum as a value, but the operator preferred by the OO style is

```
template<class T> void += (T & a, const T & b);
```

which is used infix as `a+=b`, but where the left argument is modified to contain the sum of the two variables. C++ itself does not have any intrinsic semantical restrictions on these operators, so there is no formal relationship between two such userdefined operations. In essence, neither operator may have anything to do with summation at all.

Since we are working with mathematical concepts it is natural to allow the user to write expressions involving +. The template class parameter `T` may amount to several megabytes, so an implementation using `+=` will definitely be more efficient. We solved this by imposing requirements on the use of symbols such that there would be a clear semantical connection between functions/operators and their mutating counterparts, not just for the built-in operators in C++. A few rules expressing this connection is given in the following table.

`a = a + c;`	$\longleftrightarrow$	`a += c;`
`a = b + c;`	$\longleftrightarrow$	`a = b; a += c;`
`a = a * c;`	$\longleftrightarrow$	`a *= c;`
`a = b * c;`	$\longleftrightarrow$	`a = b; a *= c;`
`a = b * c + a;`	$\longleftrightarrow$	`{t = a; a = b * c; a += t;}`

These may be used to (1) generate an operator declaration for `+`, `*`, etc., whenever a declaration for `+=`, `*=`, etc. is seen, and (2) rewrite expressions using `+`, `*`, etc., to an equivalent code segment using only `=`, `+=`, `*=`, etc. This is described in more detail in [DHH98], which also compares the run-time efficiency of the two styles.

3 Validating the domain specific language

We need to check the usefulness of the DSL by making sure that we both can use it to express (and solve) problems in the domain and that its concepts are implementable. Only then will the DSL be valuable as a tool for developing programs. But implementability does not necessarily require exact compliance with the specification, in which case numerical programs would not exist since it is impossible to represent exact real arithmetic on a known computer architecture. Rather, we need a pragmatic notion of compliance, a notion that will accept programs that deliver useful results.

3.1 Usability: Elastic Wave Simulation Problems

The problems we will use to illustrate requirements of computational modeling are taken from the oil industry. Here the application of elastic wave modeling for the interpretation of various acoustic data from potential hydrocarbon reservoirs, is important. Repeatedly solving the corresponding PDEs for different data sets requires the use of high performance computers.

The recovery of the geological subsurface structure, i.e. the origin and the geometrical picture of the geological layers, is of vital importance in developing prospects of hydrocarbon reservoirs. Also, in order to obtain optimum production strategies of existing oil and gas fields, detailed acoustic illumination of the target zones is required. The target zones are generally quite heterogeneous where the geological and reservoir properties vary within few meters. Thus, any modeling tool has to be capable of handling heterogeneous models in 2 and 3 dimensions with quite good resolution.

The seismic method is an active remote sensing technique, where the acoustic wave field, generated from a man made shock wave, is recorded on so called geophones at different spatial positions in the earth. The most common experimental setups are: marine seismics (source and geophones (hydrophones) are at the sea surface), well-to-well or surface-to-well seismics where the source is in a borehole or at the sea surface and the geophones are within a borehole, and ocean bottom seismics where the geophones are sited at the seabed. In all these experiments, the shock waves are of relatively low frequency, e.g. $5 - 100$Hz. This energy illuminates geological structures. The picture arises from reflection and conversion of pressure (P) and shear waves (S) at layer interfaces, where the physical properties of the adjacent rocks are discontinuous. By combining the reflection data with estimates of the P or S wave velocity, a geometrical picture of the interfaces is produced. Normally a whole series of recordings are taken with small variations in the position of the source and the geophones.

For all these studies, the assistance of an acoustic wave modeling tool is of vital importance for the confidence of the information retained from the data. The aim is that acoustic wave simulations on the *estimated model* coincide with the acoustic recordings of the real world. This requires the acoustic simulation software to be able to place virtual geophones at any position within the simulated area. The simulation itself extends from sealevel and several thousands of meters downwards into the earth. A simulation which is repeated with small variations in the positioning of the virtual geophones and virtual source.

The mathematical equation describing the elastic wave simulation problem is taken to be the elastic wave equation, in our formalism given by

$$\begin{aligned} \rho\frac{\partial^2 \vec{u}}{\partial t^2} &= \nabla\cdot\sigma + \vec{f}(t), \\ \sigma &= \Lambda(e), \\ e &= \mathcal{L}_{\vec{u}}(g). \end{aligned} \tag{12}$$

A fundamental assumption here is that the materials are fully elastic. The components of the equation are

- $\vec{u}$, the particle displacement (a vector field),
- σ, the stress of the material (a matrix field),
- e, the strain of the material (a matrix field),
- ρ, the density of the material (a scalar field),
- $\vec{f}(t)$, an external force, in our case the generation of the seismic pulse, a point force (spatially) that very quickly tends to zero in time (a vector field).
- $\frac{\partial^2 \vec{u}}{\partial t^2}$, the second order time derivative of the particle displacement $\vec{u}$ (a vector field),
- $\nabla\cdot$, the divergence (yielding a vector field when applied to a matrix field),
- Λ, the stiffness tensor field relating the stress and strain matrix fields (a multilinear mapping from a matrix field to a matrix field),
- $\mathcal{L}_{\vec{u}}$, the Lie derivative along vector $\vec{u}$ of a matrix field (yielding a matrix field), and
- g, the geometric (metric) tensor of the problem (a matrix field).

The equation (12) is in coordinate free form, i.e., all the entities and operators belong to the coordinate free DSL. The scalar field density ρ and the stiffness tensor field Λ are given data that vary within the physical domain, in accordance with the varying geophysical properties of the rocks. The elastic wave equation is a standard equation from mathematical physics, and may be found in any textbook on the subject. Its application to seismics is discussed in much detail in [MH83, AR80].

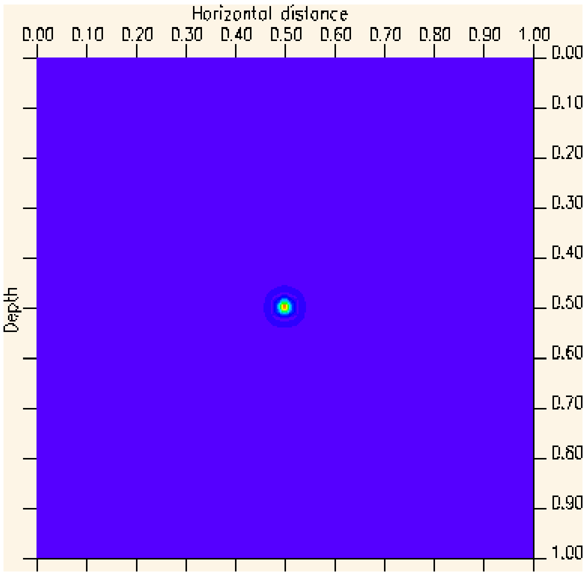

Figure 1: Isotropic case, seismic waves 25ms after the pulse started.

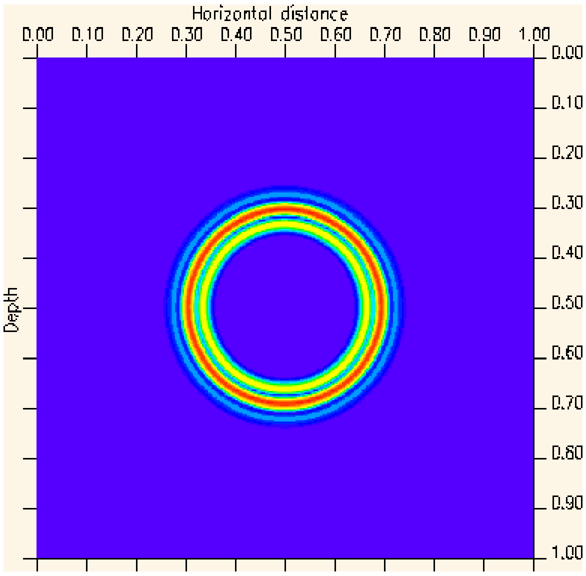

Figure 2: Isotropic case, seismic waves 100ms after the pulse started.

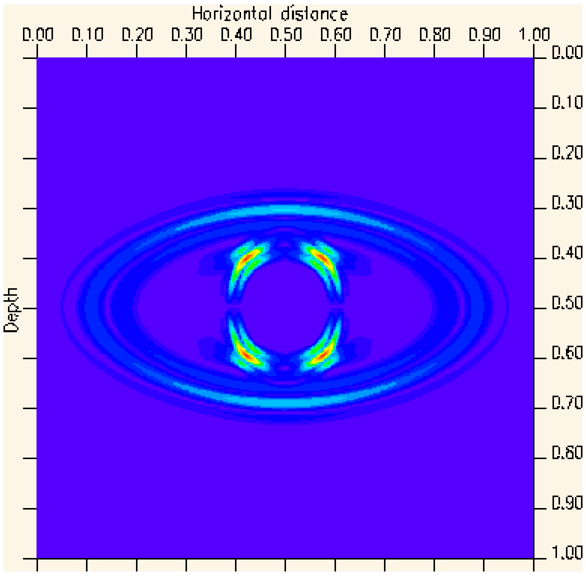

Figure 3: Transverse isotropic case with vertical axis of symmetry, seismic waves 100ms after the pulse started.

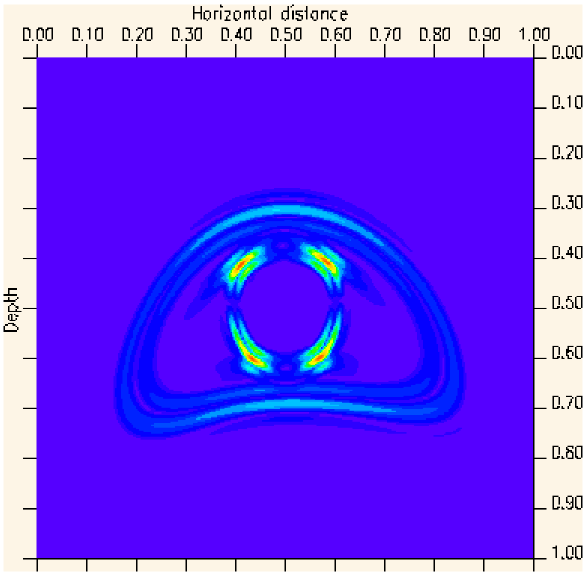

Figure 4: Transverse isotropic case with the axis of symmetry curved in the rock, seismic waves 100ms after the pulse started.

Elastic wave simulation is a very compute-intensive task, where one simulation easily may take several hours. Important factors here are the spatial resolution and frequency of the source. A larger accuracy requires a larger data set and more simulation steps, thus implying increasing the computation time. Another factor is the complexity of the physical properties of the geological models. In the simplest case, the stiffness tensor at each spatial position is made up from two independent elastic parameters only. Such a model is denoted as elastic isotropic inferring that the P and S wave speeds are independent of the wave propagation direction at this point. The isotropy introduces symmetries in the stiffness tensor Λ, so that the amount of computation can be greatly reduced.

An earth model of more general elastic properties, denoted as anisotropic, implies that the wave velocities do generally depend on the propagation direction at this point. In this case the stiffness tensor holds 21 independent elements. However, most earth materials have a rotational symmetry in the stiffness properties. In general, this happens to be an axis perpendicular to the internal layering of the material, or often perpendicular to the layer surface. In this case, the seismic velocities also have a rotational symmetry. These materials are denoted as transverse isotropic (ti), and hold only 5 independent elements in the stiffness tensor.

The Figures 1–4 illustrate some examples of elastic waves propagating outward from a point source (producing a shock wave) located within various types of materials, taken at different times after the shock was initiated. The propagation medium is here defined 2-dimensional of 1km by 1km size and uniform (i.e. no alteration in the stiffness tensor within the grid). The shock wave is described by a Ricker pulse of 30Hz center-frequency. For the isotropic case, we consider an explosion in water where no S wave exists. In case of a transverse isotropic (ti) material, we clearly see the effect of the directional variation in wave speeds (in particular for P waves), and a separation between the faster moving P waves and the slower moving S wave. The final example shows a wave field occurring within a strongly anisotropic medium. Here the wavefronts are strongly deformed from the circular shape associated to isotropic media, or the more ellipse-alike shape associated to transverse isotropic media.

As the computation time increases by a factor of more than two from the isotropic to the general transverse isotropic case, there is a clear demand for program versions being as specific as possible for the different requirements. Also, there is a need for each program version to run on single workstations (e.g., a SUN UltraSPARC), in parallel on a network of such workstations, and on a supercomputer (e.g., a SGI Cray Origin 2000) with multiple processors.

3.2 Implementability

There are a host of numerical methods that provide implementation strategies for scalar fields, but neither provides an exact representation of a scalar field. A discretisation method will only provide a more or less inaccurate representation of a scalar field, but an approximation that given some assumptions is close enough to provide useful results. The more wellknown approaches are finite difference methods, finite element methods, finite volume methods, and spectral methods. The discretisation methods vary in implementation complexity. More importantly, a method may also put severe constraints on the software complexity of the solver. In numerics, the choice of which discretisation method to use depends to a large extent on properties of the PDE to be solved. Which method we should choose as our prototype is to a large extent dependent on what applications are to be developed. Generally, finite difference methods are among the simplest to implement, with finite element methods being among the most difficult with respect to implementation and use.

The elastic wave equation is a good equation to start with from this point of view, as it works well with the finite difference discretisations. This means that we initially avoid unnecessary complications both when formulating the solver and when implementing the scalar fields – nice properties of a prototype implementation.

4 Software Architecture and DSL Structure

4.1 Structuring Concepts

A collection of related mathematical structures, such as the data structures of a programming language, typically form a category, [Gog91]. A *category* $\mathbf{C}$ is a collection of objects $A, B, \ldots$, and morphisms $f : X \to Y$, with an associated associative composition rule $\circ$ on morphisms and a neutral morphism (with respect to $\circ$) for each object. We will use the category **Prog** as our prime example. The objects of **Prog** are data structures, and the morphisms are all side-effect free algorithms from a data structure to a data structure. The identity morphism and composition rule for morphisms should be obvious. Functions of more than one argument are defined from special data structure objects called *product objects* in the category. The category **Set** is a standard example from mathematics of a category. It has sets as objects and total functions between sets as morphisms.

Categories are related by *functors*, functions between categories. A functor $F : \mathbf{C} \to \mathbf{D}$, from category $\mathbf{C}$ to category $\mathbf{D}$, maps objects to objects and morphisms to morphisms such that identities and compositions are preserved. It is not to hard to construct a functor from **Prog** to **Set** that relates the data structures and algorithms (for one specific computer) with the sets of values and mathematical functions being computed.

Functors are in many ways like C++ template classes, [Str97], or Ada generic packages, [Bar83]. These mechanisms will take a data type as argument and define a new data type based on it. We may for instance define a generic `list` package with a type parameter, such that whenever we instantiate the package with a data structure D, we get a data structure `list of` D. The idealised functor version of data type constructors have some additional properties. A list data constructing functor $L : \mathbf{Prog} \to \mathbf{Prog}$ takes a data structure D and returns a `list of` D data structure $L(D)$. But in addition to defining the list data type, it will take any function $f : D \to E$ and define an iterated function $L(f) : L(D) \to L(E)$. When $L(f)$ is given a list of D as argument it will perform f on every element of that list, returning a list of E of the results. Likewise we may treat array data structure constructors as functors. For every index type I we have an array constructing functor $A_I : \mathbf{Prog} \to \mathbf{Prog}$ which takes an element object E and defines `array [`I`] of` E, the array structure with elements of type E. But we also get the iterated functions, so given for instance a binary operation $+ : E \times E \to E$ we have $A_I(+) : A_I(E) \times A_I(E) \to A_I(E)$ which adds, componentwise, i.e., for each index $i \in I$, the elements of the two argument arrays, yielding a new array with the summed values. This is very convenient, and may only be simulated by explicit programming of these functions in current programming languages[6]. A nice observation is that this functor can be used to generate the value fields for a manifold $\mathcal{M}$ by simply applying

[6]Unfortunately, this is not fully sufficient, as the generic package mechanisms do not allow enough genericity power to let us do this once and for all. We will omit the technical proofs of these deficiencies.

$A_{\mathcal{M}}$ to the appropriate value domain, such as the reals, vectors or matrices. We can also use the constructions to define finite vector spaces by the expression $A_{\{1,\ldots,n\}}(\mathcal{R})$, for appropriate natural numbers n.

A good modularisation of software is achieved if we minimise the number of distinct functors, and the software complexity of each, needed to build the entities for our application software. Then how we combine the functors to achieve these entities will be a blue-print for the software architecture. Good choices here have large potential for greatly reducing the software development effort. Both by directly reducing our coding effort, and, more importantly, by identifying reusable components for other applications in related problem domains. Carefully structuring the modules reduces our coding effort and reusable software components be defined. This is work at the software architecture level. In the algebraic specification language CASL, for instance, this kind of software architecture can be explicitly defined, [BST99].

4.2 Structuring the Problem Domain

Now we need to analyse the problem domain in order to find the concepts that we can use in structuring our software. Our approach to formal software engineering is to use algebraic techniques, specifically category theory, for this.

When developing the DSL, Section 2.2, we pointed out that a scalar field has ring properties. A vector field is a value field with vectors, such that the vector fields form a vector space with the scalar field as ring. properties. As a consequence, n-dimensional vector fields over a manifold $\mathcal{M}$ may be constructed by either of the two approaches:

1. applying the value domain construction to vectors, $A_{\mathcal{M}}(A_{\{1,\ldots,n\}}(\mathcal{R}))$, or
2. applying the vector construction to scalar fields, $A_{\{1,\ldots,n\}}(A_{\mathcal{M}}(\mathcal{R}))$,

and similarly for tensor fields. There does not seem to be any immediate reason to prefer one over the other, and conventional numerical software uses the first construction. However, a closer scrutiny of the problem domain reveals that a tensor field contains operations like integration and derivation which are not definable from the tensor abstraction, but requires access to the value field properties, i.e., to the discretisation.

- Thus applying the value domain construction to vectors requires us to rebuild the discretisation for every level of construction, i.e., one for scalar fields, another for tensor fields etc. This was observed in the tensor oriented implementation reported in [VHBR93].
- But the integration and differentiation operators may be expressed from operations like integration and partial derivatives on the scalar fields.

This reveals that apparently equivalent constructions from a data structure and algorithmic complexity viewpoint, may have dramatically different software complexity. Based on these observations it is clear that a more fruitful approach is to use the second construction above as starting point. Instead of building many different constructors for the value domains (vectors, matrices, linear mappings, etc.), we also note that it suffices to build a tensor constructor, which, given certain assumptions, encompass all these. Tensors also give us the building blocks needed to define coordinate free operators. The implementation may then be reduced to build a constructor for scalar fields and a constructor for tensor fields.

The functor $S_{\mathcal{M}} : \mathbf{Prog} \to \mathbf{Prog}$ for the construction of scalar fields may be implemented by amending the construction $A_{\mathcal{M}}$ such that it also includes the definition of integration and partial differential operators. The tensor constructor $T_{\{1,\ldots,n\}} : \mathbf{Prog} \to \mathbf{Prog}$ amends $A_{\{1,\ldots,n\}}$ with the integration, general derivation operators and other tensor operators, assuming that the template parameter has an appropriate interface. The tensor field construction for a manifold $\mathcal{M}$ then becomes $T_{\{1,\ldots,n\}}(S_{\mathcal{M}}(\mathcal{R}))$. We should expect to reuse the functors $A_{\_}$ when building both scalar fields and tensors.

4.3 Software Architecture for PDE Problems

Our analysis of the PDE domain has given us the components we need for the software architecture. It has also provided us with a problem domain specific language and specifications of the types and operations that we need. But we also have to make sure that this really can work as a framework for implementing numerical methods. In this analysis we need to consider the issue of developing both sequential and parallel versions of the software.

The different components involved in a numerical solution of a PDE can be factored in three layers:

1. The numerical discretisation methods which makes it possible to represent the value fields $S_{\mathcal{M}}$ for the infinite set $\mathcal{M}$ by a finite approximation. The discretisation will need to provide the ring operations and the partial differentiation operations. $S_{\mathcal{M}}$ will need to contain a large set of values, often up to a million or more, in order to provide a good approximation. To represent the data values on $\mathcal{M}$ it is convenient to use the functor A_I, for some suitable index set I, as data structure.

2. The tensor construction $T_{\{1,\ldots,n\}}$ is where coordinate systems are handled and the more advanced differentiation operations are implemented. The construction should work with any scalar field, i.e., with any discretisation of a scalar field as well. Conceptually $T_{\{1,\ldots,n\}}$ is an extension of $A_{\{1,\ldots,m\}}$, for some $m \leq n$, and using the latter as the data structure seems natural.

3. The uppermost numerical layer is the solver algorithm itself. Here the time discretisation is decided, and the iterator that will generate the transient behaviour (such as for a seismic simulator where we are interested to know how the seismic wave propagates) or the steady state solution (for instance if we want to find a steady state flow pattern) is implemented. These algorithms are often normalised, i.e., the numbers they work with are scaled to be around 1.0, where the numerical resolution of the machine is best. The numbers are then scaled back for input/output purposes and as needed by the solver algorithm.

Using the array constructor to implement both the numerical discretisation and the tensor construction allows for a reuse of the array module. But more importantly it allows a separation of concerns when implementing these modules: the array constructor may focus on the data layout pattern, while the numerical modules may focus on the numerical aspects, using the array construction for the storage aspects. The architecture also implies that we only need to relate to, and thus implement, the discretisation method when we implement the scalar field, and that the vector and tensor field implementations are independent of this choice. If we need to change discretisation method, this will be localised to one module, and not being spread out all over the code, which is the normal case with traditional numerical software.

This also provides a route to parallelisation. We will, at the scalar field level at least, have a large collection of data values that may be distributed in a dataparallel fashion [Bou96]. Actually, it suffices to provide a parallel implementation of the array constructor to get a parallel version of the whole program. See [Hav98] for a discussion of this.

4.4 The Sophus library

The software architecture developed in the previous section is implemented by the Sophus software library. It provides the abstract mathematical concepts from PDE theory as programming entities. Its concepts are based on the notions of manifold, scalar field and tensor field, while the implementations are based on the conventional numerical algorithms and discretisations. Sophus is structured around the following concepts:

- Basic n-dimensional mesh structures $M_n : \mathbf{Prog} \to \mathbf{Prog}$ taking a ring R as argument. The meshes are like array constructors $A_{\{1,\dots,k_1\}\times\dots\times\{1,\dots,k_n\}}$ with element type R, and includes the general iterated operations. Specifically, operations like $+$, $-$ and $*$ are iterated over all elements (much like Fortran-90 array operators), and operations to add, subtract and multiply all elements of the mesh by a scalar. There are also operations for shifting meshes in one or more dimensions. Operations like multidimensional matrix multiplication and equation solvers may easily be implemented for the meshes. Sparse meshes, i.e., meshes where most of the elements are 0 or have some other fixed value, may also be provided. Parallel and sequential implementations of mesh structures can be used interchangeably, allowing easy porting between computer architectures of any program built on top of the mesh abstraction.

- Manifolds $\mathcal{M}$. These define sets with a notion of distance and direction which represent the physical space where the problem to be solved takes place.

- Scalar fields $S_{\mathcal{M}}$. They describe the measurable quantities of the physical problem to be solved. As the basic layer of "continuous mathematics" in the library, they provide the partial derivation and integration operations. Also, two scalar fields on the same manifold may be pointwise added, subtracted and multiplied. The different discretisation methods, such as finite difference, finite element and finite volume methods, provide different designs for the implementation of scalar fields. Scalar fields are typically implemented using the basic mesh structures for data.

- Tensors $T_{\{1,\dots,n\}}$. These provide coordinate free mathematics based on the knowledge of the coordinate system, whether it is cartesian, axisymmetric or general curvilinear. The tensor module provides the general differentiation and integration operations, based on the partial derivatives and integrals of the scalar fields. Tensors also provide operations such as componentwise addition, subtraction and multiplication, as well as tensor product, composition and application. The implementation uses the basic mesh structures, with scalar fields as the ring parameter.

Using Sophus, the solvers are formulated on top of the coordinate-free layer, forming an abstract, high level program for the solution of the problem. Solvers for time independent PDE's may likewise be formulated at this level.

5 Implementations: DSL library and The Elastic Wave Simulators

In Section 4 we have investigated the structure of the DSL and decided what basic components it consists of and how the DSL can be build from these components. We also studied, in Section 3.2, the implementability of the DSL, but postponed the decision about which discretisation method to use till we had an application that would require the use of a specific method. The usefulness of the DSL was validated in Section 3.1 where we formulated a collection of elastic wave simulators.

Here we will start the design of the elastic wave simulators, and use this to decide which variant of the library modules to implement. The piecewise implementation of the library this implies should be seen as a normal way of maintaining the library, making it more complete and versatile as time goes. But such an incremental building will only work out if the library architecture is well though out, i.e., mature, so that a reorganisation of the library will not be required as the library is gradually built.

The elastic wave simulators we are using as our example provide some additional complexity for the solver design. This is due to the boundary conditions of the problem. Since we must do our simulations in a finite computational domain, the seismic waves should ideally leave the domain when reaching the boundaries. This can not be achieved without special numerical treatment. The upper boundary (for instance the sea/air border) must also be handled specially in order to obtain the proper physical behavior. These problems are solved by implementing a boundary-aware scalar field on top of the plain scalar field discretisation. This boundary aware scalar field is then used when instantiating the tensors.

This leaves us with the following main software modules for the seismic simulators:

- `Mesh`: domain information, index set, and the basic n-dimensional meshes in sequential and parallel version.
- `Tn`: domain information, manifold elements, and n-dimensional toroidal scalar field (no boundaries).
- `Bn`: domain information, manifold elements, and n-dimensional scalar field with boundaries.
- `Tensor`: tensors with differentiation operators.
- `Seismod`: the solver algorithm in coordinate free form.

The following subsection describes the solver programs for our original problem. Then we describe variations of the seismic problem which exercise the flexibility of the software architecture developed above. All the variations we develop are captured in Figure 5, which identifies 32 specific versions of elastic wave simulators.

5.1 Solvers for the original problem

The original elastic wave simulator problem, see Section 3.1, was described for a general 3-dimensional earth with any complexity and heterogeneity in the physical properties (density and stiffness) within this model. In full generality, this should infer a stiffness tensor Λ at each grid point to have $3^4 = 81$ components. However, the physical constraints implied by

Configuration	seismic	ultrasonic in borehole
`Mesh`	D S or P	D S or P
`Tn`	D S	D S
`Bn`	D S or U	D U
`Tensor`	SI or TI or TA	UI or UTI
`Seismod`	SE or PE	SE or PE

Figure 5: Some configurations for the seismic simulator. The codes used in the table are:

- D is domain information, such as shape and indices or elements of a manifold
- `Mesh` S is sequential implementation
- `Mesh` P is parallel implementation
- `Tn` S is staggered finite difference implementation
- `Bn` S is sea surface boundary handler, i.e., the scalar field extends all the way up to the sea surface,
- `Bn` U is underground boundary handler, i.e., the scalar field is surrounded by rock
- `Tensor` SI is standard isotropic
- `Tensor` TI is transverse isotropic with vertical axis of symmetry
- `Tensor` TA is transverse isotropic with arbitrary axis of symmetry (very close to full anisotropic)
- `Tensor` UI is ultrasonic for borehole with isotropy (cylindrical coordinates)
- `Tensor` UTI is ultrasonic for borehole with transverse isotropy (cylindrical coordinates)
- `Seismod` SE is standard elastic
- `Seismod` PE is poro-elastic

considering a linear and infinitesimal deformation theory, cause the stiffness tensor to contain at most 21 different components. The computational effort may, however, be simplified by using 2D models when the earth models vary only slowly perpendicular to the considered 2D plane. Implementing this, using cartesian coordinates and for the isotropic simplification, reduces the number of distinct components in any tensor involved in the computation to at most 3. Even if we assume the more general transverse isotropic case, we never need more than 6 distinct components. But this reduction of storage and computations comes at a price: all the algorithms in `Tensor` must be specially adapted to the combined choices. So we need three distinct `Tensor` implementations, one for each of isotropic, transverse isotropic (assuming vertical axis of symmetry), and anisotropic cases.

Using a sequential `Mesh` in the implementation of `Tn` gives a sequential implementation of the seismic simulators which will run on any machine with a decent compiler. We may then exchange this with a parallel `Mesh` to achieve a parallel version of the simulators. Implementing `Mesh` using MPI [SOHL^{+}96] gives us a code that can run on any machine configuration supporting MPI, such as a network of workstations or a multiprocessor.

The seismic simulator has been implemented for all the three rock model complexities both for sequential execution on SUN UltraSPARC and SGI Cray Origin 2000 machines. It also exists for parallel execution on a network of SUN's and on the Cray. All in all providing a total of 6 source versions for these different machine architectures: `Mesh S` or `P`, `Bn S`, `Tensor SI`, `TI` or `TA`, and `Seismod SE`. These are the applications called for in Section 3.1.

5.2 Modeling of ultrasonic measurements in boreholes

While the typical seismic experiments reveal earth models several kilometers wide and deep, acoustic measurements in boreholes is performed to retrieve details of the material properties in the close vicinity of the well. Here both high-frequency P, S and surface waves are studied in order to depict the zones to be carefully handled during the production and injection phase. The near borehole reservoir properties are important both for the estimate and production of the hydrocarbon fluids. In an analogous way to seismics, these properties are seeked for by ultrasonic acoustic experiments. In this, a source and an array of transducers are placed within the borehole. Here acoustic waves within the range of $5-20$kHz are considered. For these frequencies the acoustics waves illuminate only some few decimeters of the surrounding formation. In these experiments P and S waves and boundary waves occurring in the borehole wall are considered.

From a modeling viewpoint, the problem is simplified if we consider the earth model to be symmetric around the borehole, i.e., it is axisymmetric with the well bore itself as the axis of symmetry. This is very useful since then the 3D problem becomes a 2D problem using cylindrical coordinates, resulting in a corresponding reduction of CPU-time. This implies the need to create a new implementation of `Tensor`. Unfortunately, since the geological model complexity also influences the implementation of `Tensor`, we need to create different versions of it, if we are to cater for the various geological models (isotropy, transverse isotropy, anisotropy). However, two versions is enough, since we in the case of transverse isotropy, only need to consider an axis of symmetry parallel with the borehole.

Furthermore, we here also have to take into account that the boundary conditions in the borehole are different from those on the surface. So we need a new `Bn` with open boundaries in all directions.

Thus we need 3 modified implementations to cater for the requirements of the borehole problem (`Bn U` and `Tensor UI` and `UTI`). But the changes are not radical from the original versions, and more than 50% of the code of the modules we replace can be reused in the creation of these modules. A fact which in some cases may be made explicit by the use of inheritance. The changes do not affect any other modules, nor how they should be fitted together. We also get similar parallel and sequential versions without any further implementation effort, giving us, all in all, 4 new seismic simulators.

5.3 Modeling of seismic tomography experiments

The variants of modules we have defined allow us to configure yet another version of SeisMod, a system for seismic tomography simulation. This is when a shock pulse is initiated at the surface or within a borehole, and the geophones are placed in surrounding boreholes. The modeling is here to compute the purely transmitted wavefield. We already covered the case when the tomography extends to the surface in our original simulators. But if the target area is entirely subsurface, we only have to apply the underground boundary handler `Bn U`. This may be combined with any of the `Tensor` versions `SI`, `TI` or `TA`, and of course with a sequential or parallel mesh implementation, increasing the number of specific simulators from 6 seismic and 4 ultrasonic to 12 seismic and 4 ultrasonic.

5.4 Modeling of wave propagation in poro-elastic materials

The hydrocarbons we want to investigate are stored in porous rocks. So the elastic waves will propagate in two-phase materials, or poro-elastic materials. In particular, the use of a poro-elastic modeling may describe several wave phenomena observed in ultrasonic borehole data. Typical poro-elastic materials are high porosity rocks containing viscous fluids. In such materials the particle displacements, associated to an induced stress field, of the solid skeleton and the fluid may differ. This physical system is described by two coupled equations of motion, which, accordingly, have to be solved simultaneously. The solution gives the existence of two P waves and one S wave, [Bio55, Bio56a, Bio56b]. The two P waves are usually denoted the fast and the slow P wave, where the fast P wave propagates mainly within the skeleton, but modified by the presence of the fluid. The slow P wave, accordingly, propagates mainly along the fluid phase and is modified by the skeleton. The fast P wave carries the main P wave energy, while the slow wave is mainly of phenomenological character.

The modeling of poro-elastic materials is receiving increased attention with the demand for a more precise elastic wave simulation in reservoir zones. The coupled set of equations to be considered are

$$\begin{aligned}
\rho_{11}\frac{\partial^2\vec{u}}{\partial t^2} + \rho_{12}\frac{\partial^2\vec{U}}{\partial t^2} &= \nabla\cdot\sigma + \vec{f}(t) + b(\frac{\partial\vec{U}}{\partial t} - \frac{\partial\vec{u}}{\partial t}),\\
\rho_{12}\frac{\partial^2\vec{u}}{\partial t^2} + \rho_{22}\frac{\partial^2\vec{U}}{\partial t^2} &= \nabla s + \vec{f}(t) - b(\frac{\partial\vec{U}}{\partial t} - \frac{\partial\vec{u}}{\partial t}),\\
\sigma &= \Lambda(e) + Q(\epsilon),\\
s &= M(e) + R(\epsilon),\\
e &= \mathcal{L}_{\vec{u}}(g),\\
\epsilon &= \nabla\cdot\vec{U}.
\end{aligned} \tag{13}$$

Here the components that differ from the those in equation (12) are

- $\vec{u}$, the displacement of the solid phase (a vector field),
- $\vec{U}$, the displacement of the fluid phase (a vector field),
- σ, the stress of the solid (a matrix field),
- s, the stress of the fluid (a scalar field),
- e, the strain of the solid (a matrix field),
- ϵ, the strain of the fluid (a scalar field),
- ρ_{11}, ρ_{12} and ρ_{22} are density coefficients dependent on the frequency of the source pulse (may be assumed to be constant within a simulation – scalar fields),
- b, a friction coefficient dependent on the wave frequency, but assumed to be constant within each simulation (a scalar field),
- $\frac{\partial\vec{v}}{\partial t}$, the time derivative of the displacement $\vec{v}$ (a vector field),
- ∇, the gradient (yielding a vector field when applied to a scalar field),

- Λ, the stiffness tensor field relating the stress and the strain of the solid (a multilinear mapping from a matrix field to a matrix field),
- Q, the stiffness tensor field relating the stress of the solid and the strain of the fluid (a linear mapping from a scalar field to a matrix field),
- M, the stiffness tensor field relating the stress of the fluid and the strain of the solid (a multilinear mapping from a matrix field to a scalar field),
- R, the stiffness tensor field relating the stress and the strain of the fluid (a linear mapping from a scalar field to a scalar field),
- $\nabla\cdot$, the divergence (yielding a vector field when applied to a matrix field and a scalar field when applied to a vector field),

Even though these equations are very different from those of (12), we see that all the concepts involved are tensor fields and tensor field operations. The new equations initiate a new implementation of `Seismod`, the `PE` version. But *no other module* needs modification, so poro-elastic materials may now be embedded in geological models for either large-scale seismics or ultrasonic experiments in boreholes, doubling the number of elastic wave simulators from 16 to 32.

6 Experience report

6.1 Program development productivity

In the previous section we have shown that the Sophus PDE software architecture has made it easy to pin-point exactly which module needs to be modified when requirements are changed. Besides the obvious intuitive benefits from this, there are definite quantitative benefits as well. This can be seen when considering various cost estimating techniques, of which the COCOMO model, [Boe81], is the most well known. Essentially, a model like COCOMO correlates the cost of developing a piece of software with the size of the software. The advanced estimation techniques take into account the software complexity as well. Experience, as captured by the cost models, shows that cost grows more than linearly in program size, both for development and maintenance. Thus halving the code that needs to be modified in order to meet new requirements, more than halves the cost of the modification.

Figure 6 relates different modules in the Sophus library to the size of the code that implements them. The versions that can be generated are discussed in the previous section, and summed up in Figure 5. The total number of lines has been tallied under three sets of columns. The first tally represents the lines of code needed for the standard seismic simulator (cartesian coordinates, isotropic rock model, sequential), disregarding other support modules and configuration files. This tally will be used as the reference cost of developing a seismic simulator. The second tally gives the total number of lines needed to create the 5 additional simulator versions described by the original requirements specification in Section 3.1. The third tally gives the total number of lines for the 4 versions of ultrasonic, axisymmetric borehole acoustics. This also gives us 6 versions of underground tomography, giving a total of 16 versions of elastic wave simulators. The last tally gives the cost of the poro-elastic version Note that the size of the variants always gives a too high estimate of the cost for creating that version, since much of the code from the original version of the module can be reused.

Module	D	S/SI/SE	P	TI	TA	U/UI	UTI	PE
`Mesh`	2004	2040	2729					
`Tn`	1723	1568						
`Bn`	1488	1918				2079		
`Tensor`		975		947	1769	990	est. 1000	
`Seismod`		621						641
Total:	12337		5445			4069		641

Figure 6: The sizes of the modules in lines of code. The legend is given in Figure 5. Currently the `UTI` column is only estimated.

Investigating some specific example configurations, we see that creating a parallel version of `Mesh` only costs two thirds of that of implementing a full `Mesh`, since we do not need to reimplement the domain and index types (column D). The cost of the parallel code represents redeveloping less than 25% of the full program. Once the parallel module has been developed, it can be freely mixed with any version of the program, yielding a parallel version of that program for no additional cost.

Shifting from the isotropic version to one of the anisotropic version represents a redevelopment cost of 8%–15% of the software cost, as can be seen by the changes needed, which only affects `Tensor`. Achieving parallel versions of the anisotropic codes now comes for free. So the development of the 6 versions required in Section 3.1 adds less than 50% to the cost of developing the first version.

Creating the borehole version of the seismic simulator means we have to upgrade both the `Bn` and the `Tensor` modules. The former due to the change of boundary conditions, the latter due to the use of cylindrical coordinates to improve efficiency. These changes represent a total cost of less than 35% of the reference program code. In fact the modifications involve much less than 50% of the 4069 lines listed, bringing the cost of the borehole version down to about 15% of the development costs. We get a parallel borehole version for free. More interestingly, the revised `Bn` implementation may be combined with the other tensor classes to yield a underground tomography simulator. This adds 6 new seismic simulator versions for free, in addition to the 4 borehole versions we conscientiously developed.

This gives us a tally of 16 versions of the seismic simulator, counting sequential and parallel versions. The total cost of the 15 extra versions developed represent less than 80% of the development cost of the initial simulator. If we amortise the total cost of developing all the 16 versions, we end up with a development cost of each in the range of 10%–15% of the development of the initial program. Recognising that the number of versions will double when coding the poro-elastic version (the last column), we see that the average development cost of a version drops to less than 10% of the initial development costs. Assuming the existence of a fully developed library, we see that the marginal cost of writing a version of a program may easily become less than 10% of the development cost of a full program. The most notable case in our examples is when we create the 16 poro-elastic versions at about 5% of the cost of the first version, even though the change in the mathematical equations are large. This is a tremendous gain in software development productivity over traditional development. A clear demonstration of the benefits of software reuse, contingent on a flexible and robust software architecture, i.e., a mature software design.

6.2 Discussion

It can be argued that all of these examples are taken from the same small area within PDE. Thus we have only shown that these gains are achievable when many closely related versions of a program are to be developed. This may be the case, but work performed using the Sophus software architecture for solving problems from computational fluid dynamics, see [GHW98], supports the view that the software architecture is mature. In that case study, many new implementations were needed. Almost all were instances of the Sophus modules already identified here, and none required a modification of the library architecture. This consolidates the maturity of the software architecture developed using these abstract concepts. In general we see that a new discretisation method will obviously require a new scalar field implementation, new equations to be solved require new top level solvers, etc, but all within the same framework.

Comparing this with experience from other numerical libraries for solving PDEs, e.g. [BPR92] and [VHBR93], we note that Sophus seems to avoid many of the difficulties and added costs that appear when extending the domain of a library. Typical problems that have been identified by library designers include inappropriate abstractions and inappropriate architectures. In the problem domain we have presented here, an inadequate abstraction could be a coordinate system dependent definition of the operators. Then equations (12) and (13) would need to be changed according to what coordinate system was to be used (this is the normal case in current computational models). The choice of an inappropriate architecture shows when using the construction $A_{\mathcal{M}}(A_{\{1,\ldots,n\}}(\mathcal{R}))$ directly as data structure for vector fields. Then both scalar fields and tensor fields embody the discretisation method, meaning that discretisation code is duplicated in the library. When expanding the library with another discretisation method most of the library has to be reprogrammed for the new method. In extreme cases, the user programs will have to be rewritten for the improved library as well [BPR92].

Our case study of the Sophus software architecture for PDEs does not show these symptoms. The Sophus library is designed to mimic the abstract structure of the mathematics of partial differential equations, as used in the description of many natural and industrial phenomena. Its apparatus supports coordinate free numerics in the formulation and development of solvers to the problems. By requiring strict adherence to specified interfaces, we have been able to achieve that different implementations of the same mathematical concepts are basically interchangeable. The Sophus library components can be organised in different layers of abstractions. The interchangeability of modules within a layer allows software developers to experiment with different solution strategies and high performance computer architectures without the need for extensive reprogramming. For example, the change from a sequential to a parallel version of a seismic simulator does not involve any reprogramming of the rest of the solver application, just a small change in the configuration. Currently the Sophus library is implemented using C++.

7 Summary

We have proposed a methodology that focuses on developing domain specific languages (DSL) as an important basis for software library development. The software library is seen as providing a domain specific embedded language (DSEL) [Hud96]. The DSL will aid when formulating problem descriptions, while the DSEL is useful for the design and implementation of application programs. In our opinion development of a DSL and its implementation as a software library has a software lifecycle model, see Section 1, which is distinct from that of application software development [Boe76].

Software itself is a formal entity, with strict rules for syntax and a mathematically precice interpretation, hopefully as a formal semantics, but ultimately in the form of a computer execution. This leads us to propose a discipline of formal software engineering, i.e., engineering of the formal side of software. Formal software engineering does not imply the use of formal methods throughout. We see this as especially critical when developing the DSL and when investigating the library structure. We strongly believe that algebraic methods are appropriate for this, using them extensively in our case study.

The major part of this paper is a case study from the field of computational modeling, specifically that of partial differential equations (PDE), an area which is of significance for the industrial sector. First we have used algebraic specification techniques to identify types and operations from the problem domain, see [HMMK92]. This gave us a coordinate free language for PDEs[7]. The fruitfulness of the DSL and the appropriateness of the library architecture was checked with the development of a collection of application programs. Then we studied the DSL with the purpose of structuring the concepts in order to find an optimal architecture for the software library that would provide these concepts. The library architecture was designed to eliminate a large part of the implementation work by focusing on reusable template classes.

In summary the concepts of universal algebra allow us to identify the basic entities – the types and functions – of the problem domain, giving them a syntax and a meaning, thus yielding a domain specific language. This language should be used to investigate the problem domain, formulate the algorithms, and to check its usability for writing software – programming in the small. Then we use categories and functor concepts to pinpoint how the language element definitions should be organised into a collection of modules. By carefully structuring the modules our coding effort can be reduced, and reusable software components be defined. This is work at the software architecture level – programming in the large.

Later, after the initial development of the DSL library and the application programs, additional applications were specified, and the impact of the new requirements on the library were studied. The case study showed that the library architecture remained stable throughout these changing requirements. It was possible to incrementally add interchangeable versions of new implementations as the need arose, and then use these with the other constructors to build the needed software. This seems to yield a radical improvement in software development and maintenance productivity: the development of a family of 16 application programs, being so different from the existing applications that they traditionally would have required a full development process to be initiated, was now completed at 5% of the development cost of a single application. Moreover, the library architecture makes it clear exactly which components need to be reconsidered when requirements change. In most cases a component would relate to

[7] Incidently, the problem domain specific language we discovered *is not* the language commonly used for implementing PDE software, but a much more abstract language with much better software structuring properties.

a specific requirement, but in some cases a component was influenced by several independent requirements and required reimplementation whenever one of them changed. Further study of structuring techniques is needed to see if these influences can can be factored out in a more satisfactory way.

Although our software relies on concepts packaged under the term object-orientation, and thus the elastic wave simulator is *object-oriented numerics* (OON), [WBPR93, ABL97], we end up with a software structure which deviates much from most packages claiming to be OON, such as [VHBR93, BL97]. The OON approaches represent a clear improvement over the traditional numerical software strategy, but many of the OON approaches still remain within that tradition when it comes to the conceptual decomposition of the software architecture.

The development of software libraries for important industrial problems is no simple task. One of the most important aspects is to find a mature library structure [Rac95]. Maturity means the library structure is robust faced with new demands and further development. We approached library development using formal software engineering, specifically algebraic methods. The software library produced this way has shown a high degree of maturity. This manifests itself by the fact that the structure of the library, and a large part of its code, is unaltered when the basic discretisation method are changed, and that a radical change in the assumptions of the problem (moving from elasticity to poro-elasticity) requires modifications in one of the smaller components only. Moreover, only one component needs to be changed when the software is moved onto a parallel machine architecture.

Formal software engineering is just a small part of the field of software engineering, but a sharper focus on this small area seems to be more than worthwhile in the benefits that may be gained in reduced software cost and improved reusability.

It may be argued that our example is taken from an area, partial differential equations, with a highly developed formal theory, and that this example does not indicate any general applicability of the technique. Our reply is that we promote algebraic technology as a language discovery device, and that any domain that is to be analysed for the purpose of developing software needs its language concepts to be discovered and formalised. Secondly, we promote the reasoning about software as a formal entity. This again is a property of software as such, and is independent of the problem domain the software is developed for. Thus both of these arguments should motivate the investigation of the potential of formal software engineering in other application areas.

Acknowledgments

SeisMod was developed in close cooperation with Åsmund Drottning[8], who also provided data for the example illustrations in cooperation with Roger Bakke. Numerous people have contributed to the design and implementation of Sophus and SeisMod, of which we will especially mention Hans Munthe-Kaas who was among the initiators of this work, Victor Aarre, Kristin G. Frøysa, Helge Gunnarsli, Kristian Stewart and Steinar Søreide.

References

[ABL97] Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen. Object-oriented numerics. In Morten Dæhlen and Aslak Tveito, editors, *Numerical Methods and*

[8]Contact address UNIGEO, Thormøhlensgt. 55, N-5008 BERGEN, Norway; E-mail info@unigeo.no

Software Tools in Industrial Mathematics, pages 7–26. Birkhäuser, Boston, 1997.

[AR80] Keiiti Aki and Paul G. Richards. *Quantitative Seismology Theory and Methods*, volume 1. W.H.Freeman and Company, 1980.

[Bac81] R.J.R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23:49–68, 1981.

[Bar83] D. Barstow, editor. *The Programming Language Ada - Reference Manual.* Springer LNCS 155, 1983.

[Bio55] M.A. Biot. Theory of elasticity and consolidation for a porous anisotropic solid. *Journal of Applied Physics*, 26(2):182–185, 1955.

[Bio56a] M. A. Biot. Theory of Propagation of Elastic Waves in a Fluid-Saturated Porous Solid. I. Low-Frequency Range. *J. Acoust. Soc. Am.*, 28(2):168–178, 1956.

[Bio56b] M. A. Biot. Theory of Propagation of Elastic Waves in a Fluid-Saturated Porous Solid. II. Higher Frequency Range. *J. Acoust. Soc. Am.*, 28(2):179–191, 1956.

[BL97] Are Magnus Bruaset and Hans Petter Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In Morten Dæhlen and Aslak Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 61–90. Birkhäuser, Boston, 1997.

[Boe76] B.W. Boehm. Software engineering. *IEEE Trans. Comput.*, C-25(??):1226–1241, 1976.

[Boe81] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[Bou96] Luc Bougé. The data parallel model: A semantic perspective. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 4–26. Springer, 1996.

[BPR92] K.G. Budge, J.S. Peery, and A.C. Robinson. High-performance scientific computing using C++. In *Proceedings from C++ Technical Conference, August 10–13 1992*, pages 131–150. USENIX Association, 1992.

[BST99] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In Armando Martín Haeberer, editor, *Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 341–357. Springer, 1999.

[Coh65] Paul Moritz Cohn. *Universal algebra.* Harper and Row, New York, 1965.

[DHH98] T.B. Dinesh, Magne Haveraaen, and Jan Heering. An algebraic programming style for numerical software and its optimisation. Technical Report SEN-R9844, CWI, Amsterdam; NL, December 1998.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Automatic Computation. Prentice-Hall, 1976.

[EGW98] Hartmut Ehrig, M. Gaiewsky, and Uwe Wolter. From Abstract Data Types to Algebraic Development Techniques: A Shift of Paradigms. In Francesco Parisi Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 1–17. Springer Verlag, 1998.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume XIX of *Proceedings of symposia in applied mathematics*, pages 19–31. American Mathematical Society; Providence, Rhode Iland, 1967.

[For66] *USA standard FORTRAN: approved March 7, 1966*, 1966. Published by United States of America Standards Institute.

[GH78] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GHW98] Phil Grant, Magne Haveraaen, and Mike Webster. Tensor abstraction programming of computational fluid dynamics problems. Technical Report CSR3-98, Department of Computer Science, University of Wales Swansea, Swansea SA2 8PP; United Kingdom, October 1998.

[Gog91] Joseph A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1:49–67, 1991.

[Grä68] George Grätzer. *Universal algebra.* The University series in higher mathematics. Van Nostrand, Princeton, N.J., 1968.

[GTWW75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Abstract data types as initial algebras and the correctness of data representation. In *Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Representations*, 1975.

[Hav98] Magne Haveraaen. Abstractions for programming parallel machines. Technical Report 162, Department of Informatics, University of Bergen, Høyteknologisenteret, N-5020 BERGEN, Norway, December 1998.

[HMMK92] Magne Haveraaen, Victor Madsen, and Hans Munthe-Kaas. Algebraic programming technology for partial differential equations. In Arne Maus, Frank Eliassen, et al., editors, *Norsk Informatikk Konferanse – NIK'92*, pages 55–68. Tapir, Norway, 1992.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *Computing Surveys*, 28A(4):?, December 1996.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[Mey91] Bertrand Meyer. Design by contract. In Dino Mandrioli and Bertrand Meyer, editors, *Advances in object-oriented software engineering*, pages 1–50. Prentice Hall; Englewood Cliff N.J., 1991.

[MH83] J.E. Marsden and T.J.R. Hughes. *Mathematical Foundations of Elasticity.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[Mor73] J.H. Morris. Types are not sets. In *Proceedings of the ACM Symposium on the Principles of Programming Lanugages*, pages 120–124, October 1973.

[Mos97] Peter D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 115–137. Springer-Verlag, 1997.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[PCCW93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. The capability maturity model for software, version 1.1. Report CMU/SEI-93-TR-24 and ESC-TR-93-177, Software Engineering Institute, Carnegie Mellon University; Pittsburgh, Pennsylvania 15213, 1993.

[Rac95] James Tiberius Racko. Choose me. *Software Development*, pages 81–84, April 1995.

[Roy70] W.W. Royce. Managing the development of large software systems: Concepts and techniques. *Proc. WESCON*, ??(??):??–??, 1970.

[Sch80] Bernard Schutz. *Geometrical Methods of Mathematical Physics.* Cambridge University Press, 1980.

[SOHL+96] Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI – The complete reference.* MIT Press, Cambridge, Mass., USA, 1996.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3d edition, 1997.

[VHBR93] Diane A. Verner, Gregory L. Heileman, Kent G. Budge, and Allen C. Robinson. Development of generic field classes for finite element and finite difference problems. In *OON-SKI'93 Conference Proceedings*, pages 354–363, 1993.

[WBPR93] M. K. W. Wong, K. G. Budge, J. S. Peery, and A. C. Robinson. Object-oriented numerics: A paradigm for numerical object-oriented programming. *Computers in Physics*, 7(6):655–663, Nov/Dec 1993.

[Whi98] Alfred North Whitehead. *A Treatise on Universal Algebra, with Applications I.* Cambridge University Press, 1898. Reprinted by Hafner Publishing Company, New York 1960.