Invariants as Design Templates in Object-based Systems

S.J. Goldsack and K. Lano

Imperial College of Science, Technology and Medicine,
London SW7 2BZ, England
{sjg, kcl}@doc.ic.ac.uk

Abstract. Recent work using VDM^{++}(see [1]) has shown how objects in an object-oriented language can be refined by a process of transformation of an initial class into structures of (usually) simpler objects. The process has been nick-named "annealing" by analogy with the physical process which changes the crystaline structure of a material while retaining its chemical properties. This process is considered an important aspect of system development in relation to object oriented systems, especially where formal correctness is an objective. It was shown in [2] that it is a mechanism by which model splitting (see Jian Lu, 1995 [3]) may be implemented. In [1] it was applied to the development of *design patterns* (see for example Gamma [4]) to give them a formal basis. In this paper we pay special attention to the way in which the structures of the specified system depends on the logical structure of the invariant

1 Introduction

Recent papers [2] and [1] described a process of object refinement based on *transformations* of an initial object specification to structures with equivalent properties. Class structures can be transformed into (usually more elaborate) structures composed of objects of (usually simpler) classes. Such transformations constitute an important process in the development of a formal basis for system design. In particular, it is often appropriate to start from a system description using a single class, capturing the abstract requirements of the complete system, (which we refer to as model$_0$) and refine it by property-preserving transformations into more complex structures of simpler classes.[1] For the purpose of the present discussion we find it useful to represent object structured programs using VDM^{++}, a simple object-oriented extension of the specification language

[1] We are pleased to acknowledge the contribution of Eugene Dürr to the ideas developed in this paper. Quite early in his work on VDM^{++} he suggested the role of such transformations as steps in the reification process.He gave the nick-name "annealing" to the process, in analogy with the chemical process which develops a material of finer granularity but identical chemical composition.

VDM-SL. A toolset for the support of VDM^{++}has been developed by IFAD and full details of VDM^{++}, including the language reference manual, can be found at the IFAD website, http://www.ifad.dk.

1.1 The Essence of VDM^{++}

Types, functions, values and instance variables use familiar VDM-SL notations. Almost all notations allowed in VDM-SL are allowed also in VDM^{++}. Instance variables are extended to include reference types, written with a prefix @. Thus a declaration

myobj : @ClassName

declares an item, myobj, which stores a pointer to some external object of class ClassName.

Methods are closely modelled on VDM operations, and may be given procedural definitions or be defined more abstractly with pre and post conditions. A procedurally defined method may have statements executed in loops or sequences in a way close to familiar programming notations, but one or more of the statements may be abstractly defined as a *specification statement* written

[ext < external clauses >
pre < preconditions >
post < postconditions >]

An operation which is abstractly specified is identical to a procedural method having just one statement, presented as a specification statement.

To support concurrency a class may define a thread part delimited with the key word **thread** and the execution of its methods may be controlled by synchronisation rules in a part introduced by the keyword **sync**. For real-time specifications in postconditions keywords now and $\overleftarrow{\text{now}}$ give access to times (on a system clock) of the start event and the completion event of a method while a **whenever** expression enables events and their consequences to be described. Details of most of these features are given in a chapter of the book [5].

2 The Basic Transformation Step

2.1 A class with a single data item

We start from the most basic possible example. In this a class C, has some *instance variables* to represent the state of the objects of the class and *methods*

to provide operations on that state and form its interface with client objects. This class is transformed into two classes, the first C_1 supplying the same interface for clients as before, but with the instance variables moved to an associated object. This is illustrated on the left side of figure 1. The annealing of the class C, having some internal data, leads to a class C_1 which instead references the data stored in an external object. One could say they were related by the client/server relationship. However, this name does not seem to capture well the use which is being made by the class C_1 of its dependent object. To make the distinction clearer, we shall mainly use the term *component* class for the server. The client class may be called the *framing* class and an object of the framing class may sometimes be called the *frame*.

Consider the class defined as:

class C

instance variables
store : X_type;
init store $\underline{\triangle}$ init (store);
inv store $\underline{\triangle}$ invar (store)

methods
MP (a : atype) $\underline{\triangle}$
[**ext wr** store
pre preMP (a, store)
post postMP (a, store, $\overleftarrow{\text{store}}$)];

MF (b : btype) **value** r : **N** $\underline{\triangle}$
[**ext rd** store
pre preMF (b, store)
post postMF (b, r, store)]

end C

Here the data repesenting the state of objects of the class is stored in a variable store of type X_type. The keywords **init** and **inv** define an initialisng operation and an invariant property respectively. There are also typical operations MP which is a mutating operation, changing the internal state, and MF which is an enquiry operation with no side effects on the state[2]. For brevity we shall consider only a single method MP in the rest of this paper.

The transformation moves the instance variable of C into a new class, which we shall call X_class since it is to serve as a container class for the state store of type X_type, and replace the instance variables of C by a reference to an object

[2] The notation is mnemonic: MP is like a procedure in say Pascal while MF is like a function. We use an obvious notation in which, for example, preMP denotes the precondition for the method MP.

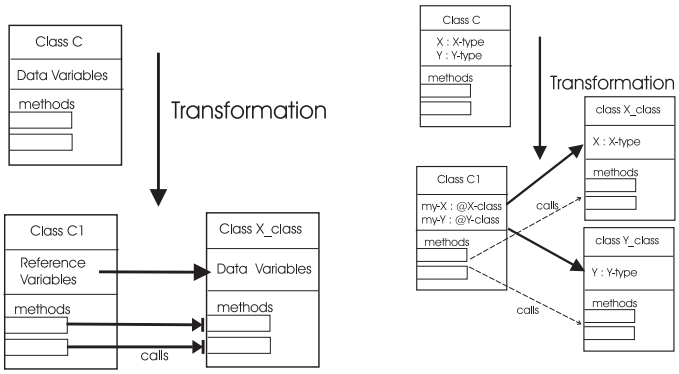

Fig. 1. The basic transformation steps: left a single state item, right two items

of the new class. To provide access to this state for use by the new version of C, it is necessary to provide methods which will permit the state X to be modified and inspected. In fact the methods MF and MP or their equivalents, will both have to be provided. The state store must also be initialised and satisfy the same invariants as in class C. Thus, in this trivial case, the class X_class is no different from a renaming of C:

class X_class

instance variables
store : X_type;
init store $\underline{\triangle}$ init (store);
inv store $\underline{\triangle}$ invar (store)

methods
MP (a : atype) $\underline{\triangle}$
[**ext wr** store
pre preMP (a, store)
post postMP (a, store, $\overleftarrow{\text{store}}$)]

end X_class

We wish to show that we can form C1, a refinement of C, by constructing a new class with an instance variable which is a reference to an object of class X_class.

Any invariant property or any initialising action associated with the state of objects of class, must apply also to the data in the annealed version. To ensure this the **init** and **inv** clauses have been moved to the component class, where they become properties of the data stored there. In this simple case this makes the class X_class identical with C_class.

The correctness of this transformation scarcely needs proof. It is discussed in more detail elsewhere [1] [6].

2.2 A More Complex Case

Suppose a class C2 has more than one state variable, and of different types.

class C2

instance variables
X : X_type;
Y : Y_type;
init objectstate $\triangleq$ init (X, Y);
inv objectstate $\triangleq$ invar (X, Y)

methods
MP2 (a : atype) $\triangleq$
[**ext wr** X, Y
pre preMP2 (a, X, Y)
post postMP2 (a, X, $\overleftarrow{\text{X}}$, Y, $\overleftarrow{\text{Y}}$)];

MF2 (b : btype) **value** r : **N** $\triangleq$
is not yet specified

end C2

We again wish to "anneal" this class to store the data in external objects referenced from the framing object. Of course, the state items X and Y could be moved together into a single component, and there would be no significant difference between this and the previous example. More interestingly, the designer may decide to anneal the class C2, using references to *separate* component objects to hold the state values for X and Y. This might be because he felt such a structure to be the most logical representation of the system being designed, or it might be in order to reuse some class or classes which already exist for one or other of the types X and Y or both. The transformation we are now considering is illustrated on the right hand side of figure 1

In attempting to form a class in which the state of the original class is split between more than one component object, we encounter some interesting new concerns.

1. The `init` statement and `invar` predicate now relate state information from the two separate objects, which know nothing of each other's internals.
2. Because the state information about the servers is now encapsulated, it is not even directly observable by the framing object, so it is not possible to write an invar clause in the frame relating the states of the servers.

3. An invariant written at the usual place, following the instance variables in the framing class, would relate the object references not the objects referenced.

In VDM^{++}each class may have an optional section called the "auxiliary reasoning" part, available for stating properties which are global to the class and all its components and transitively, to their components. In this section of the class description, the encapsulation of the component objects is weakened to allow read access to their internal state as may be required to specify the overall system behaviour. A conventional dot notation is used to name the items in the extended scope.[3]

The predicates `invar(X, Y)` and `post-init(X, Y)` (established by execution of the init action) may, in general, consist of terms constraining the separate variables X and Y compounded with additional terms expressing the relationship between them.

It is always possible to transform a predicate such as `invar(X, Y)` into an expression of the following form:

$$\text{invar}_{\text{X}}\,(\text{X}) \wedge \text{invar}_{\text{Y}}\,(\text{Y}) \wedge \text{invar}_{\text{X,Y}}\,(\text{X}, \text{Y}) \tag{1}$$

in which the terms of the conjunctive normal form of `invar(X, Y)` have been collected into those depending only on X, those depending only on Y and those relating X and Y. Any of these could be void, and one may expect that in cases when splitting the state data seems an attractive option, it will be the case that the data components will be fairly independent so that the term $\text{invar}_{\text{X,Y}}(\text{X}, \text{Y})$ will be empty or of small importance.

Responsibility for maintaining those parts of the invariant which depend on one state item may now be given to the appropriate component objects while the term *relating* them may be placed in the auxiliary reasoning part of the framing class, which will be responsible for maintaining its truth. If the two component classes are specified to maintain certain invariants, then it is evident that the overall invariant will include the conjunction of these separate invariants. In its auxilliary reasoning part, the framing class must state those additional properties which relate them.

We shall see later that it can also happen that the terms of the *conjunctive* form do not separate well, but that the *disjunctive* form does. This indicates that the design using *alternative* classes to be selected (one or other) using polymorphic substitution will make a more appropriate design. It is illustrated in section 3 and 4 below.

The post-init predicate may be similarly treated. The initial system construction must ensure observance of the initial conditions of the component state variables,

[3] Auxilliary reasoning as such may not form part of the standard VDM^{++}, but there will be a means of referring to the contents of component objects.

and of the initial relation between them as defined in the framing object. The initialisation of an object is required to establish the truth of the invariant at system construction.

2.3 Splitting the Methods

The treatment then follows roughly that of the previous section. We shall again consider only the mutating method MP2. In the class C2 it is given as:

$$\begin{array}{l} \mathrm{MP2}\,(\mathrm{a}:\mathrm{atype}) \triangleq \\ \quad [\ \textbf{ext wr}\ \mathrm{X},\mathrm{Y} \\ \quad\ \ \textbf{pre}\ \mathrm{preMP2}\,(\mathrm{a},\mathrm{X},\mathrm{Y}) \\ \quad\ \ \textbf{post}\ \mathrm{postMP2}\,(\mathrm{a},\mathrm{X},\overleftarrow{\mathrm{X}},\mathrm{Y},\overleftarrow{\mathrm{Y}})\ ] \end{array}$$

In general this may change the values of both X and Y. In the class C2_1 it will be implemented with calls to methods of the classes X_class and Y_class. We therefore need to divide the actions of MP2 into parts responsible for updating the separate variables. We give here an outline of the theory of splitting. In the following the method is written as two specification statements, defining actions which may occur in either order, each updating only one of the variables. In general, also, each may use the initial value of the other's variable, so to ensure that it gets the value before it was changed by the other action we have introduced local declarations to provide temporary storage.

$$\begin{array}{l} \textbf{methods} \\ \quad \mathrm{MP2}\,(\mathrm{a}:\mathrm{atype}) \triangleq \\ \quad (\quad \textbf{dcl}\ \mathrm{localX}:\mathrm{X\_type} := \mathrm{X}\ , \\ \qquad\qquad \mathrm{localY}:\mathrm{Y\_type} := \mathrm{Y}\ ; \\ \\ \| \ (\ [\ \textbf{ext wr}\ \mathrm{X} \\ \qquad\ \ \textbf{pre}\ \mathrm{preMP2_X}\,(\mathrm{a},\mathrm{X},\mathrm{localY}) \\ \qquad\ \ \textbf{post}\ \mathrm{postMP2_X}\,(\mathrm{a},\mathrm{X},\overleftarrow{\mathrm{X}},\mathrm{localY})\ ], \\ \qquad [\ \textbf{ext wr}\ \mathrm{Y} \\ \qquad\ \ \textbf{pre}\ \mathrm{preMP2_Y}\,(\mathrm{a},\mathrm{localX},\mathrm{Y}) \\ \qquad\ \ \textbf{post}\ \mathrm{postMP2_Y}\,(\mathrm{a},\mathrm{Y},\overleftarrow{\mathrm{Y}},\mathrm{localX})\ ] \\ \quad) \\ \quad) \end{array}$$

Note that if the evaluation of the new X requires knowledge of the new Y it will have to evaluate it itself (redundantly). Again we see that if the data sets have much interaction this annealing would be a poor design. However, there are certainly many problems where this is not the case.

The evaluations have now been made independent, and can occur in either order, or indeed concurrently.[4]

In the class C2_1 there will be calls on each of the corresponding methods in X_class and Y_class, again in non-deterministic order, which will handle the changes in X and Y respectively. Also, if the old value of Y is actually used in evaluating X, then the value of localY will have to be set by a call on a speciallly provided enquiry method in Y_class and vice versa. Thus X_class is defined:

class X_class

instance variables
X : X_type;
init X $\triangleq$ init_X (X);
inv X $\triangleq$ inv_X (X)

methods
MP_X (a : atype, oldy : Y_type) $\triangleq$
[**ext wr** X
pre preMP_X (a, X, oldy)
post postMP_X (a, X, $\overleftarrow{\text{X}}$, oldy)];

giveX () **value** r : X_type $\triangleq$
[**post** r = X]

end X_class

and a similar class for Y. The frame class C2_1 for the annealed system is then of the form:[5]

class C2_1

instance variables
my_X : @X_class;
my_Y : @Y_class

methods
MP (a : atype) $\triangleq$
(**dcl** localX : X_type := my_X!giveX() ,
localY : Y_type := my_Y!giveY() ;

[4] Note: The non-deterministic operator || in VDM-SL introduces a list of actions which may be executed in any order. In VDM^{++}, if the actions involved are "durative" (consuming time) then, provided the calling mechanism is asynchronous, the start events can occur in any order, and the actions execute concurrently.

[5] Note: Recall that the optional "**auxilliary reasoning**" part of a class is provided in VDM^{++}to enable the specifier to add information not syntactically valid in other parts. In particular, statements relating to the internal states of dependent objects, breaking the object encapsulation, are permitted here.

```
        || (my_X!MP_X (a, localY),
            my_Y!MP_Y (a, localX) )
    )

  auxiliary reasoning
    invar_{X,Y}    (my_X.X, my_Y.Y )

end C2_1
```

3 Disjunction in the Invariant

It is evident that the structure, as in the right hand side of figure 1, in which a class has two component classes, both servers of the frame class, maintains a system invariant which is a conjunction of the form of equation 1. However, in general we might find cases in which the requirements specify an invariant which is a disjunction of the form:

$$\text{inv1 } \textbf{exor} \text{ inv2} \tag{2}$$

The objects are to satisfy one or other of two invariants. The 'or' is the exclusive or; it would be meaningless if both might be true.

To specify such a class at level 0, we must write:

```
class Disjunction

  types
    cases = < case1 > | < case2 >

  instance variables
    thiscase : cases;
    store : X_type;
    init store △ cases thiscase :
                  < case1 > → store := init1,
                  < case2 > → store := init2
                end;
    inv store, thiscase △
          (thiscase = < case1 > ∧ inv1 (store)) ∨
          (thiscase = < case2 > ∧ inv2 (store))

  methods
    MP (a : atype) △
     cases thiscase:
       < case1 > →
             [ ext wr store :
               pre preMP1 (a, store)
               post postMP1 (a, store, ←store) ],
```

```
        < case2 >→
              [ ext wr store :
                pre preMP2 (a, store)
                post postMP2 (a, store, ↙store) ]
        end

    end Disjunction
```

The invariant might have been written as a case expression, but the explicit use of the enumeration variable shows its disjunctive structure more clearly. Note that, since the role of the initialisation is to ensure that the invariant holds at the outset, it is natural that it should also have alternative definitions for the two cases.

To anneal this class, we introduce an abstract class DisjunctiveCase with two alternative subclasses Case1 and Case2. These complete respectively the definition of the abstract parent class with the two alternative invariant behaviours.

The abstract class is of the form:

```
    class DisjunctiveCase

      methods
        MP (a : atype) △
        is subclass responsibility

    end DisjunctiveCase
```

The first subtype has the form:

```
    class Case1 is subclass of DisjunctiveCase

      instance variables
        store : X_type;
        init store △ store := init1;
        inv store △ inv1 (store)

      methods
        MP (a : atype) △
         [ ext wr store :
             pre preMP1 (a, store)
             post postMP1 (a, store, ↙store) ]

    end Case1
```

and the second is similar.

The annealed form of the level 0 class now holds its data as a reference to an object of the abstract class, to which may assigned polymorphically one or other of the alternative subclasses. The alternative method definitions in the level 0 definition have become the bodies of the methods in the respective cases.

The framing class now has the form:

class Disjunction

types
cases = $< \text{case1} > | < \text{case2} >$

instance variables
thiscase : cases;
mystore : @DisjunctiveCase

methods
MP (a : atype) $\triangleq$
mystore!MP(a)

end Disjunction

4 The Abstract Factory Design Pattern

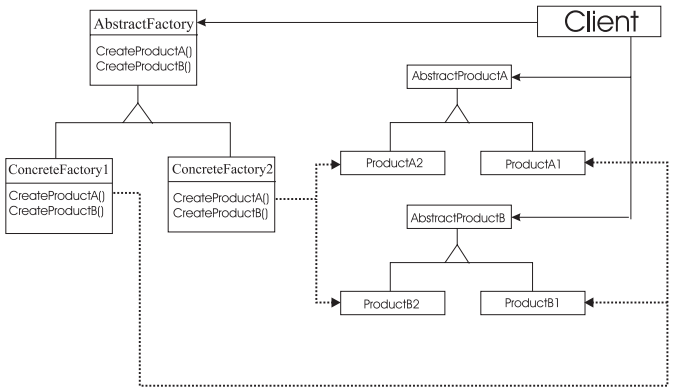

Fig. 2. The structure of the Abstract Factory Pattern

Design patterns, standard reusable designs for typical recurring problems, have recently become a popular field of study. Details of various examples are given in the book by Gamma et al[4]. The paper [7] describes how several such designs may be considered refinements of simpler ones.

In [1] we developed an example from the book [4] (page 87), referred to as the Abstract Factory, showing how it can be developed by annealing a monolithic initial model. Figure 2 is reproduced from the book. The example has a structure developed according to an invariant with both conjunctive and disjunctive elements.

5 Synchronised Classes

Finally, we show the way in which annealing can be used to separate concerns for the functionality of a class and its synchronisation rules. In VDM++ there are several ways to express concurrent behaviour, and to define the rules for synchronising the operations of a class which is subject to concurrent calls on its methods. The following class specification describes a simple database which allows multiple readers when there are no writers, and a single writer when there are no readers. This is the simplest form of readers and writers, and does not protect from writer starvation, and is used here for simplicity. More general cases can be handled in the same way.

```
class ReadWrite  -- level 0

  instance variables
    datastore : data;
    inv datastore △ ... -- is not yet specified

  methods
    read (a : atype) △
    is not yet specified;

    write (b : btype) △
    is not yet specified

  sync
    per read  ⇒  #active(write) = 0
    per write ⇒  #active(write) = 0 ∧
                 #active(read) = 0

end ReadWrite
```

Here we have instance variables defining a data set, with a normal data invariant. Then there are method definintions, here left for future development. Finally, in a sync part of the class, we have a specification of the synchronisation rules for the execution of the methods.

These rules are given by expressing a set of *permission axioms* defining for each method the circumstances in which a request by a client for its execution can be

dispatched. Each axiom is of the form

$$\textbf{per} \ < \text{methodname} > \Rightarrow < \text{condition} > \tag{3}$$

The conditions are usually boolean expressions over history counts, though they can include also data-items. Failure of a condition means the relevant method cannot be dispatched; if it holds it does not necessarily mean it can be or will be dispatched, as there could be other permission axioms for the same method which must also hold. Details can be found in [5]. All the permission axioms are effectively "anded" together, and form an invariant in the time axis which must always hold. Here an expression such as #active(m) is a count of the currently active executions of the method m. The synchronisation requirements could equivalently been expressed as the invariant condition:

$$\#\text{active(write)} \leq 1 \wedge \neg(\#\text{active(write)} > 0 \wedge \#\text{active(read)} > 0) \tag{4}$$

which must hold at all times.

By an annealing step, concerns for maintaining the data and temporal invariants can be separated into different component objects, in exactly the way we did before in section 2.2. In the usual way, we can form a container class DataStoreClass for the data; this provides the functionality of the database, with its read and write methods. The data in class ReadWrite is replaced by a reference to DataStoreClass. Control of synchronisation is provided by a class which we have callled ReadWriteControl, with methods startread and startwrite which block if the synchronisation invariant would become violated by admitting the operation. There are also methods endread and endwrite which record the completions of the respective executions. The object keeps record of the start events and completions, and maintains the history counters. Its role is to ensure the invaraint is maintained.

The necessary control class can be specified either again using permission predicates, or using a thread with answer statements, which is illustrated in the following class definition.

class ReadWriteControl

instance variables
readers : **N**;
writers : **N**;
init readers, writers $\triangleq$
(readers: = 0; writers: = 0
)

methods
startread () $\triangleq$;

endread () $\triangleq$;

```
    startwrite () △ ;

    endwrite () △

  thread
    while true
    do
        sel
          writers = 0 answer startread →
                        readers := readers + 1 ,
          readers = 0 ∧ writers = 0 answer startwrite →
                        writers := writers + 1 ,
           answer endread →
                        readers := readers -1 ,
           answer endwrite →
                        writers := writers -1


end ReadWriteControl
```

It is of interest that a class like this is conveniently implemented in Ada as a protected type.

The synchronised ReadWrite class now has the form:

```
class ReadWrite  -- level 1

  instance variables
    mydatastore : @DataStoreClass;
    mycontrol : @ReadWriteControl

  methods
    read (a : atype) △
    (    mycontrol!startread();
         mydatastore!read();
         mycontrol!endread()
    ) ;

    write (b : btype) △
    (    mycontrol!startwrite();
         mydatastore!write();
         mycontrol!endwrite()
    )

end ReadWrite
```

5.1 Some further examples

The paper in reference [1] treated in outline also some further examples of annealing, some at least of which are relevant to the main idea of this paper.

- A set refined as sequence of sets
- Recursive annealing
- Annealing of Maps
- Concurrently Active Parts
- factorials

6 Conclusions

We have shown how a concept of property preserving transformations in which a class is split into structures of objects of more elementary classes can be a powerful design approach in Object Oriented structuring. The process has been called "*annealing*".

The present paper has shown how annealing of an initial class structure can be a valuable approach to developing and justifying reusable design structures.

We consider of particular interest the way in which appropriate designs reflect the structure of the data in the abstract specification, and are greatly influenced by the invariants involved.

References

1. S. Goldsack, K. Lano, and E.H. Dürr. *Annealing, object decomposition and design patterns.* In Technology of Object Oriented Languages and Systems. (TOOLS Pacific), 1996.
2. S.J. Goldsack, K. Lano, and E.H. Dürr. *Annealing and data decomposition in VDM^{++}.* ACM Sigplan Notices, 31, July 1996.
3. Jian Lu. *Introducing data decomposition into VDM for tractable development of programs.* ACM Sigplan Notices, 30, September 1995.
4. E.Gamma, R. Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, Reading, Massachusetts : . IBSN 0-201-66361-2, 1995.
5. S.J Goldsack and E.H. Dürr. *Concurrency and Real-time in* VDM^{++}, chapter 6. Springer Verlag, ISBN 3-540-19977-2, March 1996.
6. S.J. Goldsack, K. Lano, and E.H. Dürr. *Refinement of object structures in VDM^{++}.* from www.doc.ic.ac.uk/~sjg get the file "annealing.ps", 1995
7. K.C. Lano, J.C. Bicarregui, and S.J. Goldsack. *Formalising design patterns.* In Northern Formal Method Workshop, Bradford. Springer Verlag EWICS, 1996.