Invariants as Design Templates in Object-based Systems

S.J. Goldsack, K. Lano,

Department of Computing
Imperial College of Science, Technology and Medicine,
180 Queens Gate, London SW7 2BZ, England
Tel (+44) 171 594 8246
email sjg@doc.ic.ac.uk
email kcl@doc.ic.ac.uk

E. Dürr
Utrecht University
Dept. Computational Physics
P.O.Box 80.000
3508 TV Utrecht, The Netherlands.
fax : +31-38-4654511
email: durr@fys.ruu.nl

Abstract

Recent work using VDM++ (see [4, 5]) has shown how classes in an object-oriented language can be refined by a process of transformation of an initial class into a structure 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.

The annealing 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 [8]) may be implemented. In [4] and [7] it was applied to the development of *design patterns* (see for example Gamma [3]) to give them a formal basis. In this paper we pay special attention to the way in which the structure of the specified system can be derived from the logical structure of the invariant.

Keywords: Object Structures, formal methods, invariants, annealing, VDM^{++}, design patterns.

Workshop Goals: Advance the formal basis of design structures in component based programming. Increase awareness in the programming community of VDM^{++}, of structural transformations as a design approach and of the dependence of system structure on the logical structure of the invariant requirements.

1 Background

1.1 Introduction

Recent papers [4] and [2] have introduced a concept of “annealing”, a process of class/object refinement based on *transformations* of an initial class specification into a structure with equivalent properties. The initial class structure is transformed into a (usually more elaborate) structure composed of (usually simpler) classes. Annealing is an important concept in formalising 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). This model is then refined by property preserving transformations into more complex structures of simpler classes.

There is no doubt about the importance of object-oriented structure in system development. However, it should be recognised that its use can introduce its own form of design bias. In the use of a technique such as OMT[9] the designer presents class structures which include classes for objects perceived in the world which constitutes the application domain. Often classes are included such as a class *Controller* whose function is to coordinate the activities of the effective classes but whose existence is an unrecognised design decision. By developing the system from an initial class specifying an abstract model of the complete system, the annealing steps develop an object system in which any necessary coordinating objects arise naturally.

1.2 The Essence of VDM^{++}

Full details of VDM^{++}can be found in the language reference manual[1].

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 $myobj : @ClassName$ declares an item, *myobj*, which holds a reference 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 such statements may be abstractly defined as a *specification statement* written

$$\begin{array}{ll}[\textbf{ext} & < external\ \ clauses > \\ \ \textbf{pre} & < pre\text{-}conditions > \\ \ \textbf{post} & < post\ \ conditions > \ ]\end{array}$$

A method which is abstractly specified is identical to a procedural method having exactly 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 $\overleftarrow{\mathsf{now}}$ 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 [6].

The language is supported by a toolset, known as the Venus toolset, developed by cooperation between the industrial partners Verilog, IFAD and CAP. [1]

1.3 Annealing : The Basic Idea

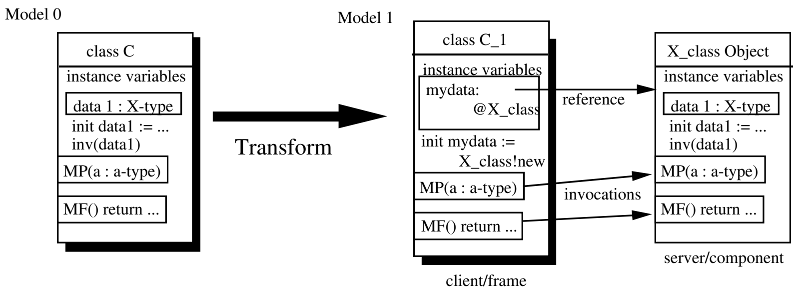

Figure 1: The basic idea of annealing.

Figure 1 shows a conventional class C with state variables and two typical methods. The data representing the state of objects of the class is stored in an instance variable *data* of type *X_type*. There are typical operations $MP(a)$ 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 annealing step 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 *data* of type X[3]. We wish to show that we can form a refinement of C, by constructing a new class C_1 with an instance variable which holds a reference to an object of class *X_class*, and with method bodies which are invocations of the methods in *X_class*. This is illustrated in Figure 1. The relationship between C_1 and the *X_class* instance which it references is commonly called the client/server relationship. However, in connection with annealing we sometimes call the client the *frame* class and refer to the server as a *component*.

The correctness of this transformation scarcely needs proof. Programmers of object oriented systems would not hesitate to use it. It is discussed in more detail elsewhere [4] [5]. To be correct it is essential that the reference variable *my_data* in the frame class is not void, but references an actual instance of *X_class*; this is ensured by the initialisation specified in the frame class. It is essential also that there is no aliasing, so the reference must not appear on the right of an assignment, nor be passed as a parameter to any method which might copy it. Finally, and crucially in connection with the position of this paper, the invariant and initialisation required by the data in the class C are moved with the data to its new location, to ensure the same properties in the annealed structure.

2 Position

During work on system development using VDM^{++}we have become aware that, because the overall system is required to maintain the invariants of all the components of a component-

[1]Information about Venus, including the syntax for VDM^{++}can be accessed on the World Wide Web at the address:
http://www.ifad.dk/products/venus.

[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*.

[3]One might say it defines an abstract data type for *X_type*. It would be simple to change the actual representation of the type.

based system, the logical structure of the overall requirements dictates (or is dictated by) the actual component structures. This paper attempts to make this apparent by treating some basic examples.

The paper notes how the overall requirement of a system may be expressed in terms of the totality of its invariant properties, and how its natural decomposition is into components each responsible for maintaining a part of the invariant condition. The structure is related to the *logical* structure of the requirements statement. Design proceeds by forming a single class whose invariant(s) describe the system requirements; this is then "annealed", usually step by step, to define its construction as a system of cooperating objects, each responsible for maintaining a part of the overall invariant.

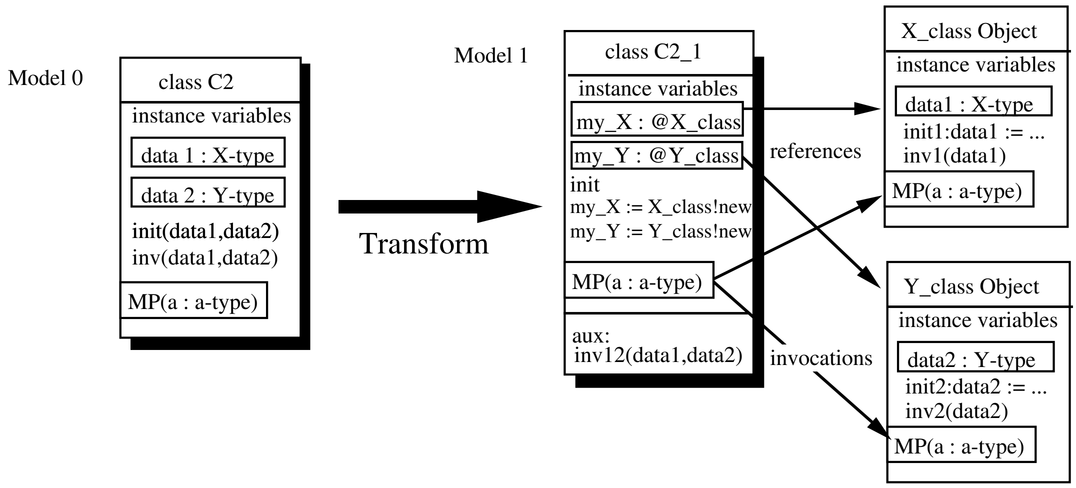

Figure 2: Annealing more complex data.

In the previous section we presented a simple case where the data of the initial class is moved to a component. Now we look at a slightly more complex case. Here the data in the initial class has two elements of possibly different types. Each of these can be moved as before to a container class. This case is illustrated in Figure 2[4].

3 Conjunction in the Invariant

The invariant in class $C2$ has been written $inv(data1, data2)$. It is some predicate over the two state variables. Such a predicate can always be expressed in the form[5]:

$$inv1(data1) \wedge inv2(data2) \wedge inv12(data1, data2) \qquad (1)$$

where the terms of the conjunctive normal form of $inv(data1, data2)$ are collected into terms dependent only on $data1$, those dependent only on $data2$ and a residue of terms relating $data1$ and $data2$. When the data items are moved into separate classes, they each have their own invariant properties, and the terms of the invariant which refer to single data items can be moved to the respective components. The external frame class must maintain the term relating them. The method *MP* which acts on both the data items must now be split into parts which act on the separate items. This is discussed in [4].

We can note, then, that when objects are related by a client/server relationship (the association in OMT) the overall system is required to maintain the truth of the conjunction of

[4]It would also be possible to anneal one only of the data items, leaving the second in the 'frame'. The treatment would not be much different.

[5]Quantification would be needed if either data item was a set or sequence of items.

terms consisting of its own local invariants and the invariants of all its servers. It follows that the structure, such as Figure 2, 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.

4 Disjunction in the Invariant

We also find cases in which the requirements specify an invariant which is a disjunction of the form:

$$inv1 \;\vee\; inv2 \tag{2}$$

Objects of the class are to satisfy one or other of two invariants. The invariants will usually be disjoint. Any overlap would signify 'don't care' states in which it does not matter which invariant is maintained.

```
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))

end Disjunction

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
```

Figure 3: Model$_0$ for a disjunctive invariant.

An example of a class with such a structure is given in Figure 3. 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 *Case*1 and *Case*2. These complete respectively the definition of the abstract parent class with the alternative invariant behaviours given in equation 2.

The abstract class is of the form:

class *DisjunctiveCase*

methods
$MP\,(a : atype) \triangleq$
is subclass responsibility

end *DisjunctiveCase*

The first subtype has the form:

class *Case*1 **is subclass of** *DisjunctiveCase*

instance variables
$store : X\_type;$
init $store \triangleq store := init1;$
inv $store \triangleq inv1\,(store)$

methods
$MP\,(a : atype) \triangleq$
[**ext wr** $store$:
pre $preMP1\,(a, store)$
post $postMP1\,(a, store, \overleftarrow{store})$]

end *Case*1

The subclass *Case*2 is similar. Its method, maintaining a different invariant, might be quite different.

The frame class, 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 be 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.

As a more complex example, the so-called abstract factory design pattern, which uses both invariant structures, is developed in full in [4].

5 Synchronisation Rules as an Invariant Requirement

5.1 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 allowing 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.

class *ReadWrite* — level 0

instance variables
$datastore : data;$
inv $datastore \triangleq$ ... — **is not yet specified**

methods
$read\,(a : atype) \triangleq$
is not yet specified;

$write\,(b : btype) \triangleq$
is not yet specified

sync
per $read \Rightarrow \#active(write) = 0$

per *write* $\Rightarrow$ $\#active(write) = 0 \wedge$
$\#active(read) = 0$

end *ReadWrite*

Here we have instance variables defining a data set, with a normal data invariant. Then there are method definitions, 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

$$\mathbf{per}\ < methodname > \Rightarrow < condition > .$$

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, as there could be other permission axioms for the same method which must also hold. Details can be found in [6]. 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 have been expressed as the invariant condition:

$$\#active(write) \ \leq \ 1 \ \ \neg(\#active(write) > 0 \ \#active(read) \ > \ 0) \qquad (3)$$

which must hold at all times. Permission holds for the dispatch of a method at a given time provided the invariant holds at that time, and the dispatch does not change the history counts in a way which would then violate it. We believe that the permission form is easier to write and understand, and closer to the form of the implementation.

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 4. 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 called *ReadWriteControl*, with methods *requestread* and *requestwrite* which block if the synchronisation invariant would become violated by admitting the operation. A call on such a method is a request for permission. 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 invariant is maintained.

This control class can itself be specified either again using permission predicates, or using a thread with answer statements, as illustrated in the following class definition.

class *ReadWriteControl*

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

methods
$requestread\ () \triangleq\ ;$
$endread\ () \triangleq\ ;$
$requestwrite\ () \triangleq\ ;$

endwrite () $\triangleq$

thread
while *true*
do
sel
$writers = 0$ **answer** *requestread* $\rightarrow$ readers := readers + 1,
$readers = 0 \wedge writers = 0$ **answer** *requestwrite* $\rightarrow$ writers := writers + 1,
answer *endread* $\rightarrow$ readers := readers -1,
answer *endwrite* $\rightarrow$ writers := writers -1

end *ReadWriteControl*

It is of interest that a class like this is conveniently implemented in Ada95 as a protected type.

The synchronised read/write class now has the form:

class *ReadWrite* — level 1

instance variables
mydatastore : @*DataStoreClass*;
mycontrol : @*ReadWriteControl*

methods
read (*a* : *atype*) $\triangleq$
(*mycontrol*!*requestread*;
mydatastore!*read*;
mycontrol!*endread*
) ;

write (*b* : *btype*) $\triangleq$
(*mycontrol*!*requestwrite*;
mydatastore!*write*;
mycontrol!*endwrite*
)

end *ReadWrite*

5.2 Some further examples

The paper in reference [4] treated in outline also some further examples of annealing. Some at least have a bearing on the present position.

- A full treatment of the Abstract Factory pattern
- A set refined as sequence of sets
- Recursive annealing
- Annealing of Maps
- Concurrently Active Parts
- Distributed Computation

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 annealing of an initial class structure can be a valuable approach to developing and justifying reusable designs.

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.

7 Related Work

We are not aware of work which has addressed directly the role of invariants in determining the structure of a component based system. There is however an endless list of works which implicitly cover and relate to this topic. Not least is the growing volume of work on *Design Patterns* following the work of Gamma et al[3].

Related to our concept of development by annealing, one should mention again the paper on model splitting, by Jian Lu[8].

References

[1] E.H. Dürr and N. Plat (editor). *VDM^{++} Language Reference Manual.* Cap Volmac, Utrecht, Netherlands, afrodite (esprit-iii project number 6500) document afro/cg/ed/lrm/v9 edition, March 1994.

[2] S J Goldsack et al. Annealing and data decomposition in VDM^{++}. *ACM Sigplan Notices*, 31, July 1996.

[3] E Gamma, R Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, Reading, Massachusetts, 1995.

[4] S Goldsack and E H Dürr. Annealing, object decomposition and design patterns. In *TOOLS Pacific 1996*, 1996.

[5] S J Goldsack, K Lano, and E H Dürr. Refinement of object structures in VDM^{++}. Technical report, Dept. of Computing, Imperial College, address `theory.doc.ic.ac.uk`, get the file at `papers/Goldsack/annealing.ps`, 1996.

[6] S.J Goldsack and E.H. Dürr. *Concurrency and Real-time in VDM^{++}*, chapter 6. Springer Verlag, ISBN 3-540-19977-2, March 1996.

[7] K C Lano, J C Bicarregui, and S J Goldsack. Formalising design patterns. In *Northern Formal Method Workshop, Bradford.* Springer Verlag EWICS, 1996.

[8] Jian Lu. Introducing data decomposition into VDM for tractable development of programs. *ACM Sigplan Notices*, 30, September 1995.

[9] J Rumbaugh, M Blaha, W. Premerlani, F Eddy, and W Lorensen. *Object Oriented Modeling and Design.* Prentice Hall International, 1991.

8 Biography

About the authors

Dr Ir E.H.Dürr is Technical Manager at Cap Gemini, Assistant Professor at Utrecht University -both in the Netherlands- and visiting Professor at the Ecole Nationale Superieure des Techniques Industrielles et des Mines de Nantes, France. (http://www.fys.ruu.nl/~durr)

Prof S. Goldsack is Emeritus Professor at the Department of Computing, at Imperial College in London U.K. He was an active member of the team who developed the VDM^{++}language and method, bringing to it experience of work in distribution for Ada, OO languages with real time aspects (DRAGOON) and formal methods (the FOREST project).

Dr. K.C. Lano Kevin Lano has a long experience in the formal methods field, and is author of several books in the area.