ON THE SPECIFICATION OF BEHAVIOURAL CONSTRAINTS

Flaviu Cristian
Computing Laboratory, University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, England

1. Introduction

When designing information processing systems, the key problem is to find what symbols and constructs (available in some given language) should be used so as to be able to answer the questions that the users want to ask and to perform the state transitions that the users have to perform in order to keep in step with some reality that is being modelled. If the language that is being used cannot be interpreted directly by the software of a computer (e.g. set theory, algebra, semantic networks), the resulting model is called an abstract model (e.g. a specification, a data base schema). If all of the symbols and constructs used to express the model can be interpreted by a computer, the model is a concrete implementation (e.g. a data base system). Both the abstract and concrete models capture some aspects of the reality that is modelled. They differ with respect to the languages in which they are expressed.

The choice of the languages which are appropriate for writing such models is a subject of intensive debate. The recent workshop on Data Abstraction, Data Bases and Conceptual Modelling has shown that this debate is not likely to diminish in the near future. In the context of this debate, however, a common opinion was expressed several times by data base participants: the data models used to specify data base schemas are lacking to support the specification of operations tailored to particular application environments. It is therefore expected that in the context of data base modelling, the integration of operations (behaviour) with data (structure) will be one of the most fertile research areas for the next few years.

The aim of this paper is to present a "data abstraction" approach to the specification of operations. A simple example will be used to illustrate the main ideas. The possible states of the "reality" which is modelled are simple, and the emphasis is placed on state transitions. However, the author believes that the approach can be used to specify operations on complex state spaces, as encountered in data base applications. Whenever possible, attempts will be made to establish a correspondence between the different terminologies used by the data base and the programming language communities.

2. Structural and Behavioural Constraints

During the past decade, significant progress has been made in the data base area with respect to describing the structural constraints (e.g. functional dependencies, hierarchical dependencies, etc.) which characterise the "allowable" states of a data base. Such structural constaints (called invariants in the programming language area) are needed to restrict the basic modelling space (provided within the language used to specify the data base schema) to some subspace of interest which corresponds to the "possible" states of the reality which is modelled. Let us call A the set of such allowable states.

The effect of an operation OP producing a new state $a \in A$ from an old state $a' \in A$ can be described by giving a state transition relation $S \subset AxA$ between old and new states. Traditionally [1,2] state transitions are not defined by enumerating all the pairs (a',a) which are in S, but by giving a predicate $post \in AxA \rightarrow \{true,false\}$ called the postcondition of the operation: $post(a',a) \Leftrightarrow (a',a) \in S$. Another predicate $pre \in A \rightarrow \{true,false\}$, called the precondition of the operation, is used to define the domain of S: $pre(a') \Leftrightarrow a' \in dom(S) = \{a' \mid \exists a \in A : (a',a) \in S\}$. Pre- and postconditions express behavioural constraints imposed upon a model by stating which are its "allowable" state transitions. These have to correspond to the "possible" state transitions of the reality being modelled.

A set of states A together with a set of operations {OP} which can be invoked to produce transitions between the states in A are called a data type in the programming language community. A set of abstract states can be described as being the equivalence classes obtained by restricting some free word algebra with some equations [3], or as being a restriction of some basic modelling space borrowed from set theory plus relations and functions with some "structural" invariant [2], etc. The operations of a data type (which should not be confused with the operators used by algebraists to define abstract states [3]) can be

specified by restricting the "chaotic" state transition relation $A \times A$ with postconditions.

Traditionally [1,2,8] the effect of invoking an operation OP in a state $a' \in A$ is specified only if a' is in the domain defined by pre. In that case, the new state $a \in A$ is such that post(a',a). If $\neg$ pre(a'), there exists no state $a \in A$ such that post(a',a) and the result of the invocation is left unspecified. In such a case it is usual to say that an exception or a run-time error (constraint violation - in data bases) occurs. In [4] it is argued that the specification of operations for a data abstraction should contain not just the description of one (standard) effect, but also of several possible exceptional effects, providing well defined responses to possible attempts to violate the invariant properties inherent to that data abstraction. For this purpose, it is proposed that the semantics of an operation OP should not be described only by a pair of standard pre- and postconditions, but by a conditional form [5] composed of a finite set of pairs of pre- and postconditions, describing a set of possible state transitions

$(E_1: pre_1 \rightarrow post_1, \ldots, E_k: pre_k \rightarrow post_k, pre \rightarrow post)$

If the operation OP is invoked in a state $a' \in A$ such that $\exists i\ 1 \leq i \leq k\ \neg(\bigvee_{j=1}^{i-1} pre_j) \wedge pre_i$ then the relation between $a' \in A$ and the new state $a \in A$ is defined by the exceptional postcondition $post_i$ and the execution continues (exceptionally) by an invocation of the exception handler associated with the exception identifier E_i and the activation point of OP (for more details see [6]). Such handlers are known in the data base community as "violation actions" [7] or "failure actions" [8]. If, at the invocation of OP, all the exception preconditions pre_i are false but the standard precondition pre is true, then the relation between a' and the new state a is defined by the standard postcondition post and the continuation of the execution is standard, i.e. sequential. Otherwise (if all exceptional and standard preconditions are false) the state transition produced by the invocation of OP is left undefined.

If, for every state $a' \in A$ the effect of invoking OP is defined, i.e. $\bigvee_{i=1}^{k} pre_i \vee pre = true$, then the operation is called total. If an operation is total and all its exceptional postconditions $post_i$ are of the form a = a' then the operation is called atomic (in the sense that it either produces its standard effect or leaves the state unchanged*). The conditional forms which specify atomic operations can be simplified to

$(E_1: pre_1, \ldots, E_k: pre_k, post)$.

Indeed, as all the exception postconditions express the fact that the state does not change, it is possible to omit them. Moreover, as the standard precondition can be derived from the exception precondition by the formula $pre = \bigwedge_{i=1}^{k} \neg pre_i$ it is possible to omit it also.

* The term atomic is also used in a multiprocessing context to qualify parallel activities which do not interfere. Atomicity with respect to synchronisation and atomicity with respect to exceptions have clearly to be studied in inter-relation in parallel systems in which exception occurrences can signal attempts to violate invariant properties to be maintained within single activities, or between cooperating activities.

```
data type EXAM;
states e ∈ [1,99] → {NE,E,S}
invariant dom(e) = [1,99]   "e is total"
initially ∀i ∈ [1,99]  e0(i) = NE
operations
    procedure ENROL(st:1..99); signals ENROLLED;
        ENROLLED: e'(st) ≠ NE ,
        e'(st)=NE∧e(st)=E∧∀i∈[1,99]:(i≠st)⇒(e(i)=
                                            e'(i))
    procedure COMPLETE(st:1..n);
              signals NOT-ENROLLED,COMPLETED;
        NOT-ENROLLED:e'(st)=NE ,
        COMPLETED    :e'(st)=S  ,
        e'(st)=E∧e(st)=S∧∀i∈[1,99]:(i≠st)⇒(e(i)=
                                            e'(i))
  function SUCCESS-RATE:real;
      SUCCESS-RATE = (∀i∈[1,99] e(i)=NE → 0,

          true → card{i∈[1,99] | e(i)=S} / card{i∈[1,99] | e(i)≠NE})
```

Figure 1

3. An example

The example in Figure 1 (taken from [9] will be used to illustrate the ideas discussed above. Suppose that for statistical and administrative purposes it is decided to design a data base which keeps track of a class of students who have to pass some examination. There can be at most 99 students, and they are identified by integers from 1 to 99. The specification of Figure 1 (the "data base schema") defines a data base state as being a function e (exam) which associates students with their status: not enrolled (NE), enrolled (E) or successful (S). However, not every function belonging to the space $[1,99] \rightarrow \{NE,E,S\}$ models a "legal" state. Indeed, from a mathematical point of view, a function f in $[1,99] \rightarrow \{NE,E,S\}$ is just a set of pairs $(i,s) \in [1,99] \times \{NE,N,S\}$ such that if $(i,S_1) \in f$ and $(i,S_2) \in f$ then $S_1 = S_2$. What is required in reality is that at every moment all the students have a well defined status, i.e. every state e must be a total (and not a partial) function. This restriction over the basic modelling space (too "rich", because it contains also "undesirable" partial functions) is stated [2,4,9] as a structural invariant which completes the definition of the "allowable" states. The initial state of the data base is the total function $e_0(i)=NE\ \forall i \in [1,99]$, and all the (standard and exceptional) state transitions specified for operations yield new total functions if the old were total. Therefore, the truth of the invariant can be assumed before any operation invocation.

The operations that the administration can perform on the data base are to enrol a student and to record that a student successfully completes the exam. The statisticians can interrogate the data

base and ask about the success rate of the class. The behavioural constraints that have to be enforced (and which correspond to administrative rules governing the exam) are the following: a student can be enrolled only once and no student can complete the exam twice or without having previously been enrolled. These constraints are expressed in the standard postconditions of the operations ENROL and COMPLETE. The standard effect of SUCCESS-RATE is always available in our example. All the operations have been specified to be total and atomic.

Let us focus on the specification of COMPLETE. The first line provides syntactic information needed by an invoker of the operation, while the next three lines describe the possible effects of an invocation. Its standard postcondition indicates that its standard effect is to change the old state e' into a new state e in such a way that e' and e differ only in st: the value of e' in st was E while the value of e in st will be S. The domain of the standard state transition so defined is e'(st)=E : if e'(st) $\neq$ E there exists no pair (e',e) satisfying the standard postcondition. The invariant can be used to show that the disjunction of the standard and exceptional preconditions is always true, i.e. that COMPLETE is a total operation.

4. Some closing comments

This paper has presented a "data abstraction" approach to the specification of behavioural constraints. Although in the simple example which was given, the state space was defined in terms of concepts taken from set theory with functions, the approach can be applied also if other state definition methods are used. For example, if the relational model (restricted with appropriate structural invariants) is used to define the "allowable" state space, then the assertion language for writing pre and postconditions has to contain relational operators (instead of operators on sets or functions). Similarly, if an algebraic method [3] is adopted for the definition of the state space, then the assertion language has to contain the operators used in the state space definition. A clear distinction between the operators used to define the state space and the operations (called routines in [3]) which can be invoked by users in order to produce state transitions can remove some of the problems encountered in the algebraic specification community, like the need for "hidden" operators [10] and the need for including "error elements" in every state space [11]. Indeed, if such a point of view is adopted, then all the operators will become "hidden" (i.e. not invokable by user programs - they will be used just to specify the effect of operation invocations [3]). Moreover, if the operations are viewed as routines, there is no difficulty in designing them to be atomic [4] and there will be no need to produce "error elements" (i.e. to record the occurrence of exceptions by changing valid states into erroneous states).

Due to space limitations, nothing has been said about how such behavioural constraints can be enforced by the concrete models (e.g. programs, data base systems) that run on computers. Issues related to the concrete implementation of data types with exceptions in a programming language supporting data abstraction and exception handling are presented in [4] and a general discussion about exception handling in hierarchies of data abstractions can be found in [12]. Of course, nothing prevents the programming language used to implement "robust" data abstractions (i.e. with total operations) from providing among its predefined types a data type which is convenient for the implementation of data base systems (e.g. a type n-ary relation with powerful interrogation and manipulation operations). It seems therefore likely that results obtained in the programming language area concerning the integration of operations with data could be adapted so as to be usable in the data base area.

Of course, solutions proposed within some community have always to be "adjusted" so as to meet the needs of some other community. One important obstacle in the communication of ideas is provided by the different terminologies which are used. Meetings like the recent workshop on Data Abstraction, Data Bases and Conceptual Modelling can help in promoting this much needed communication.

References

[1] C.A.R. Hoare: Proof of correctness of data representations. Acta Informatica 1,4,1972.

[2] W. Wulf, R. London, M. Shaw: Abstraction and verification in Alphard, Introduction to language and methodology. TR, Carnegie-Mellon Univ. 1976.

[3] J. Guttag, J. Horning: Formal specification as a design tool. TR, Xerox PARC, 1980.

[4] F. Cristian: Specification and implementation of data types with exceptions. TR, Univ. of Newcastle upon Tyne, 1980.

[5] J. McCarthy: A basis for a mathematical theory of computation. In Computer Programming and Formal Systems, P. Braffort and D. Hirschberg (Eds.) North Holland Pub. Co., 1963.

[6] F. Cristian: Le traitement des exceptions dans les programmes modulaires. Doctoral Thesis, Univ. of Grenoble, 1979.

[7] M. Hammer, D. McLeod: Semantic integrity in a relational data base system. Proc. of the Int. Conf. on Very Large Data Bases, U.S.A. 1975.

[8] K. Eswaran, D. Chamberlin: Functional specification of a subsystem for data base integrity. Proc. of the Int. Conf. on Very Large Data Bases, U.S.A. 1975.

[9] C.B. Jones: Software development, a rigorous approach. Prentice-Hall, 1980.

[10] J. Thacher, E. Wagner, J. Wright: Data type specifications: parametrisation and power of specification techniques. Proc. 10th Annual Symp. on Theory of Computing, 1978.

[11] J. Goguen: Abstract errors for abstract data types. In Formal Description of Programming Concepts. E. Neuhold (Ed.) North-Holland Pub. Co., 1978.

[12] F. Cristian: Exception handling and software-fault tolerance. Proc. 10th Int. Symp. on Fault Tolerant Computing, Japan, 1980.