Reconciling OO with Turing Machines

G. BERRISFORD* AND M. BURROWS

*Model Systems, 1 Wendle Court, 135 Wandsworth Road, London SW8 2LY, UK
†Aspen Lake Software Ltd, 5 Beech Farm Drive, Macclesfield, Cheshire SK10 2ES, UK

Two OO specification techniques have been developed almost independently. The 'inheritance paradigm' emphasizes the analysis of hierarchical structures of super and subtype objects, and reuse by inheritance. The 'state-transition paradigm' emphasizes the analysis of real world events and the state-changes they trigger in objects. Each paradigm has desirable properties that the other lacks. This paper shows ways resolve the structure clash between the paradigms. It also suggests that object processes or 'methods' are emergent properties of an object and event-oriented analysis, rather than an object-oriented analysis alone. The background of the main author is in database systems, but Section 10 shows the paper is relevant to other kinds of software engineering such as real-time process control systems.

Received July 28, 1994; revised October 3, 1994

1. INTRODUCTION

Booch (1994) probably speaks for most computer scientists when he says 'mapping an object-oriented view of the world onto a relational one is conceptually straightforward, although in practice it involves a lot of tedious details'. We cannot deny the 'tedious details', which means that even trivial examples of database systems take up a lot of space, but this paper offers a challenge to the 'conceptually straightforward'. It shows there are clashes between widely-accepted object-oriented concepts. Beside providing some resolutions of our own, we hope to stimulate further interest and research.

This paper is more discursive and less mathematical than most contributions to *The Computer Journal*, so we preface the paper with some remarks for those of a mathematical bent.

On notations and formality. We illustrate object class specification in Sections 2, 4 and 6 using informal pictures, rather than a specific programming language like C++ whose syntax is cryptic to the uninitiated. We have avoided mathematical symbols and expressions. We use the word 'process' rather than 'algorithm' or 'procedure that terminates'. But three of our notations, based on a standard promoted by the UK government (CCTA, 1990), are relatively formal. We use a regular expression to model the events and state transitions (and methods) of an object class. We use a formal syntax to model the one-to-one or one-to-many cardinality of an association between object classes. We use a formal syntax to model the one-to-one or one-to-many correspondence between a 'method' in one object class and the method it invokes in another class. The first and last of these notations evolved out of Jackson (1975) and they are described in Section 10.

Only the regular expression notation is vital to the main thrust of the paper. We prefer the graphical form of the Jackson structure notation. This widely-used standard notation can be mechanically translated and condensed into a more conventional mathematical notation. For some work on comparing 'formal methods' with Jackson's program design technique, see Latham (1990).

On unsupported claims. In a few places, notably those marked as such, we make statements that are unsupported by example or reasoning. Sometimes this is due to lack of space, since we have no room for richer examples or extracts from the references given. But occasionally, and this is where we would ask the reader to forgive us, we make an assertion merely to interest or challenge the reader. Like the mathematics lecturer who says 'and so it is obvious that' to mask a gap in a proof, we believe we can support all claims made, given more time and space to do so.

We conclude in Section 9 with some heuristics for choosing between different approaches.

2. ENCAPSULATION

Stefik and Bobrow (1986) say an object is 'an entity that combines the properties of processes and data', it 'performs computations and saves local state'. Perhaps the primary object-oriented idea is *encapsulation*. Encapsulated with an object are both its data variables and its processes.

Many use encapsulation to mean information hiding of attributes, meaning that the local variables of an object are private to that object. You can only access a variable by invoking a process of the object.

Some use encapsulation to mean information hiding of processes as well, meaning that some processes of an object (its 'implementation details' or 'method bodies') are private to that object. This is a major idea in writing production software and OO books, but a very minor idea in specifying a database system, where we expect every method body to be so simple it can be specified graphically on the object class and event class specification diagrams.

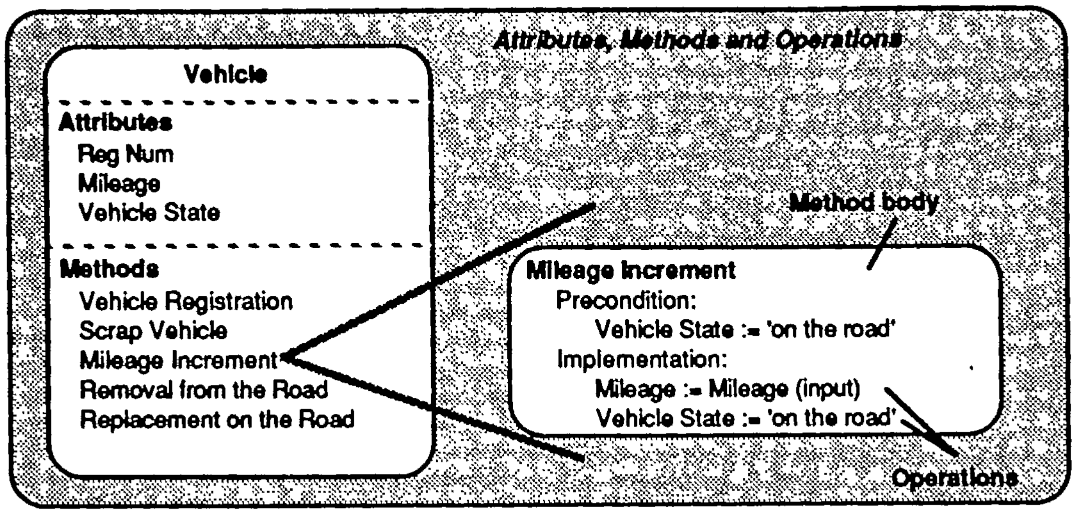

FIGURE 1.

Some use encapsulation at specification time to mean that when defining an object class, you must specify its variables and processes together. You specify the processes that define an object as a flat list of 'methods' in Smalltalk (Goldberg, 1981) or 'virtual member functions' in C++ (Stroustrup, 1986). We use *method* to mean a process property of an object, invoked by one or more of the events that affect that object. We reserve *operation* to mean a primitive executable statement, executed within a method. We use *attribute* to mean a data variable property of an object. Figure 1 is a picture to illustrate the terms.

Some use encapsulation to mean there are no free-floating methods, each belongs to an object and you can only invoke it via that object. You might use a format such as Object: Method (Parameter), e.g. Vehicle: Mileage Increment (Mileage).

2.1. Earlier incarnations of encapsulation

Encapsulation and information hiding were recognised in developments predating OO programming. Consider a 'called' routine in COBOL. This is like an object. You declare the parameters it receives and the results it returns. You declare its private variables in 'working storage'. You could declare its methods as 'performed' routines arranged under a case statement.

COBOL implementations did not emphasize encapsulation, but it was emphasized in methodologies developed contemporaneously. Information hiding was much discussed as a principle of modular programming. 'Structured design' (after Constantine and Myers and others) introduced notions of cohesion and coupling into modular design Jackson (1975) introduced a kind of OO programming methodology, later called JSP.

Of direct relevance to this paper, JSP described how to resolve structure clashes in a system by decomposing it into co-operating finite-state machines. In resolving an 'interleaving clash', the main process has:

- An input data structure representing the behaviour of an object.
- A 'state vector' to hold the private variables of each object instance.
- A 'state variable' to record the current position in an object instance's life.

Jackson promoted the idea that a database is nothing more or less than a place to hold the state-vectors of concurrent objects.

3. LIMITATIONS OF ENCAPSULATION

This Section expands on two themes. One is that there are clashing views of how to encapsulate, or aggregate, properties to form an object. A software architecture that reconciles these views should separate them by handling each view in a separate module.

The other theme is that the one-object-at-a-time view of system specification has its limitations. 'No object stands alone; every object collaborates with other objects to achieve some behaviour' (Booch, 1994). You need to think about the way that objects are related by the events that affect them. You need *event-oriented* analysis and specification techniques to help you specify the right or best set of methods and to design the message routing between objects.

3.1. Defining the centre around which to encapsulate

Although the individual data items of system specification are very important, we are more troubled in database systems by how to specify the rules and constraints governing larger objects, higher-level aggregates of data items.

There are different ways to group data items into objects, leading to different data models, that is different structures specified over the top of the data items. We need to understand the different possible approaches and their implications.

3.1.1. *The relational data model: object = relation*

Given you must design a system that consumes data and produces information, you may uncover the objects of interest by relational data analysis of the data items in already-specified system files and documents. An early step in relational data analysis is to spot the object identifiers or keys. Given an object instance, the value of each of its attributes is uniquely determined by the value of its key. In what is called a 'third normal form' relation, the value of each attribute is determined by first the key, second the whole key, and third nothing but the key.

Figure 2 shows the result, a relational data model. Each box is a relation, its key is underlined, its attributes are listed, and its associations to other relations are shown as lines connecting the boxes. The meaning of the different styles of line does not matter here.

There is little freedom of choice about what the relations are, given that you know the end-users' information requirements and you follow the idea that object instances are uniquely identifiable from each other by a key. But this is not the only logical view.

3.1.2. *The object data model: object = finite-state machine*

Another common logical view, after Jackson and others, is that an object is something that progresses through a defined series of states, from a beginning to an end. In this state-transition view, an object is a finite-state machine, governed by a state-variable (SV).

People who design systems with little or no persistent data, typically embedded or process control systems, often view objects as finite-state machines. They discover the objects by analysing states that objects pass through and the events that trigger state-changes. You can use similar techniques for database systems. You can transform a relational data model via object event modelling techniques into a specification of event processes to maintain the data. These event processes specify all the required business rules in the form of 'preconditions'.

People often assume you can draw one state-transition diagram or behaviour model for each relation. However, you need to draw separate behaviour models for the parallel aspects of a relation. Figure 3 shows that if you draw an object data model to back up the object event model (with one box per behaviour model) then this will differ from the relational view since you must divide the parallel aspects of a relation.

There is little freedom of choice about what the finite-state objects are, given that you know the business rules and constraints and follow the idea that each object is a finite-state machine controlled by one state variable. (The necessary behavioural analysis is not shown here, but an indication of it appears in Figs. 7 & 8.)

3.1.3. *Inheritance view: object = type*

There is one more logical view. The inheritance-oriented view is that an entity is something uniquely identifiable by a type or subtype. In our example, a Vehicle may be either a Car or a Truck. Should we show the subtypes as distinct objects in the data model? We are going to consider inheritance further in later sections.

3.1.4. *The internal data model: object = database table*

Given a system that maintains persistent data, you must design the record types or tables into which the database will be divided. The physical database designer's view is that an object is a record type or database table, that is the unit of input/output accessed by programs. There is enormous freedom of choice here. Designers may roll up several logical objects into one table, or split one logical object between tables.

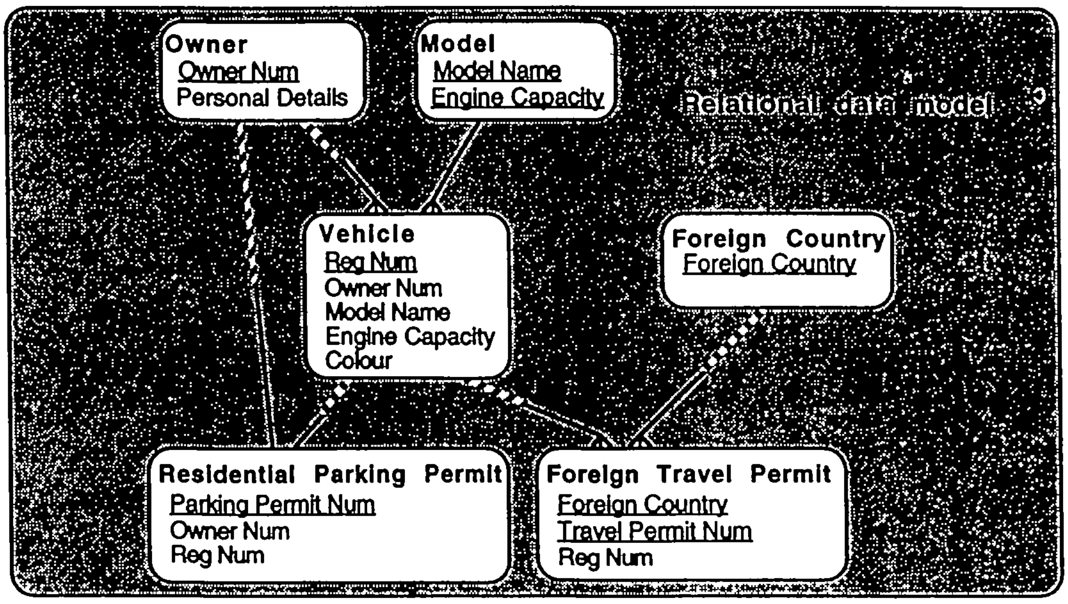

FIGURE 2.

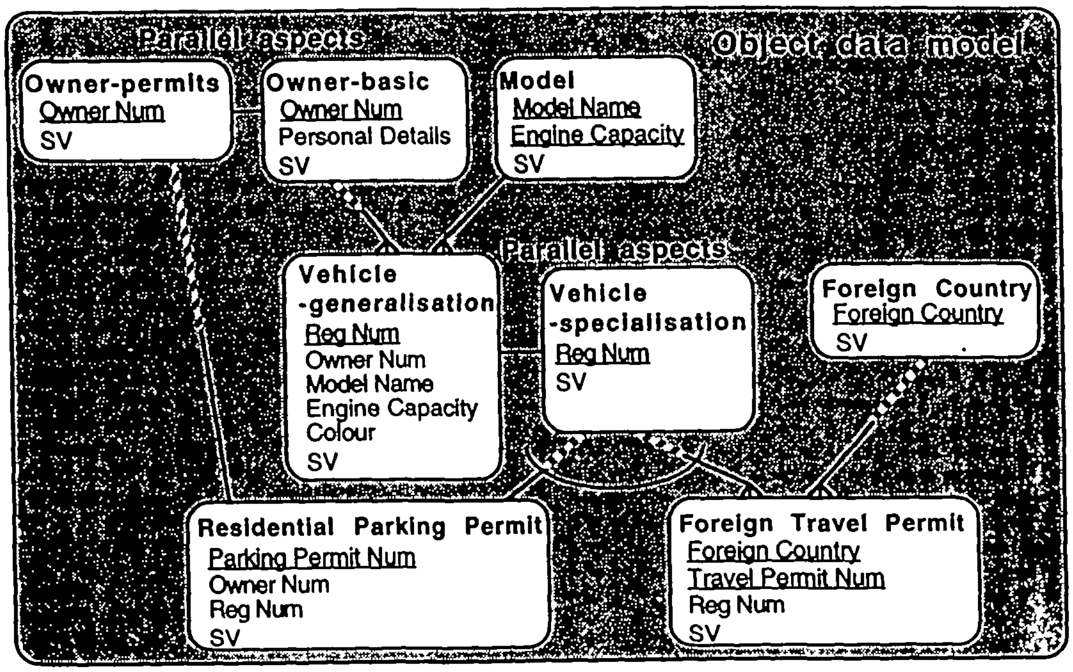

FIGURE 3.

3.1.5. The structure clash between aggregate objects

By concentrating on simple examples, authors have so far been able to gloss over the possible structure clashes between these views. In a simple simple system, there is no structure clash, the different definitions lead to the same set of objects. You will draw the same data model whichever definition you pick. You can specify many of the business rules in this data model. You can use an application generator to build a workable system from this data model.

For more complex systems, we can no longer pretend that the different definitions give the same answer. The more complicated the system, the more the logical views diverge from each other. You may reasonably design the physical database tables to match either the relations or the finite-state machines, but you cannot match both. The pictures you draw will depend on the definition you pick and the design decisions you make.

In non-trivial database systems, rather than select one or other logical view as the basis of encapsulation, we want to have it all ways. Current OO ideas are not enough; we need a richer theory of system specification. We need a software architecture that results in separates modules handling each logical view, and separate modules addressing the physical concern of designing efficient database tables. We have outlined a software architecture rich enough for this purpose (Robinson and Berrisford, 1994).

Whichever basis you choose for encapsulation, there are further questions about method specification.

3.2. The difficulty of specifying methods (unsupported claim!)

It is usually easy to list the data attributes of an object, especially where a business already maintains some persistent data that you can inspect. It is also easy to list the primitive operations that can operate on these attributes. You might think it will be just as easy to recognise and list the 'methods', but this is not so.

You do not want to clutter up your system with methods that are irrelevant, which fall into state of neglect and disrepair. You do want to specify methods that are meaningful and useful. To be meaningful and useful, a method must be invoked by at least one event. To be reusable, a method must be invoked by one than one event.

We address the question of where the methods come from by taking an event-oriented approach to requirements capture and knowledge acquisition. We name methods after the events that invoke them. To specify all the effects of events on one object, we use an object behaviour model. To specify one commit unit's worth of processing, we string all the effects of one event together in an event diagram. See Section 10 for details.

One event effect is close to the OO idea of a 'method', but each effect is unique to the event which invokes it. Small reuse occurs where two or more events invoke the same effect on one object type (see Section 10). Larger reuse occurs where two or more events can share a set of effects on several objects; we can show this set of effects in an event diagram for the 'superevent' which is invoked by the external events. This approach gives a significant advance in that it helps us to define useful and reusable methods via a rational analysis and design process (see Robinson and Berrisford, 1994).

3.3. Discovering and specifying the message routing between objects

'Our understanding of message routing tends toward the

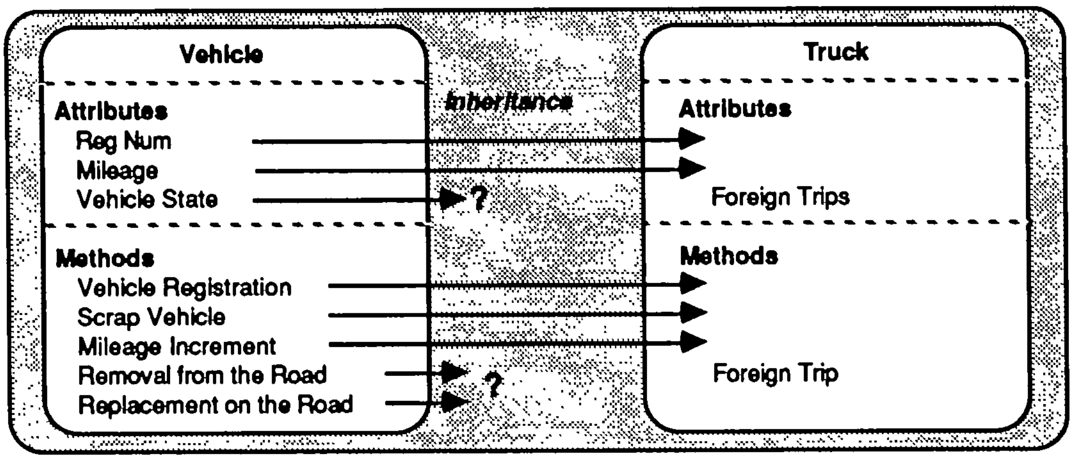

FIGURE 4.

magical. Somehow messages are directed towards the appropriate receivers. We end up relying on miracles. Message routing problems are resolved, often in a haphazard way, at coding time.' (Palmer, 1993)

The inheritance paradigm does not help you define the message routing between associated objects that is needed when an event hits the system. Section 10 shows a natural way to specify the message routing between objects is to supplement the inheritance paradigm by drawing a diagram for each event.

4. INHERITANCE

The idea of *inheritance* has come to predominate in discussions of object-orientation. This is probably because most OO books are written by production software engineers who find OO programming languages like Smalltalk and C++ very helpful.

Inheritance predates OO programming, but it was given little emphasis in earlier technologies and methodologies.

OO programming languages get their power from the facility to define each object as being of a type in a hierarchical structure of types (or several overlapping hierarchical structures in systems that allow multiple inheritance). An object can inherit methods from any object higher up the class hierarchy or *inheritance tree.*

We introduce below a reduced and modified version of the example used in Section 3 to illustrate different views of objects in data models.

Suppose you must keep track of Cars and Trucks. Figure 4. shows the supertype Vehicle can hold the common properties, leaving only one extra attribute and method to be defined for the subtype. (We will come back to the question marks shortly.)

We want to record the Mileage covered by both Cars and Trucks, but we only want to count the Foreign Trips made by Trucks. A Truck has some common properties shared with Car (the Mileage Increment method updates the Mileage attribute), and some unique properties (the Foreign Trip method increments the counter called Foreign Trips). The subtype Truck can inherit the Mileage attribute and the Mileage Increment method.

A subtype object defined at a low level in an inheritance tree can use methods defined in several different higher level types. It might inherit a method from the supertype at the top of the inheritance tree, or from an object at an in-between level. A subtype object may *override* an inherited method with its own variant, and it can have additional methods of its own, not at all relevant to any of its supertypes. *Multiple inheritance* means simply that an object type can belong to more than one classification of super and subtypes, and can therefore inherit from supertypes in different inheritance trees.

4.1. Benefits of inheritance

The inheritance paradigm provides a powerful and easily extensible programming system, at least for production software if not for information systems. The aim is to do only as much work as is absolutely necessary to define new object types. The inheritance mechanism encourages reuse of methods defined in supertype objects (although there are other kinds of reuse it does not encourage).

A further benefit is that programmers do not have to define the type of an object to be processed. The programmers' code does not test types; the run-time environment does this behind the scenes. The programmer sends an object a message, identifying the object instance and naming the method. The run-time environment works out what kind of thing the object is and invokes the required variant of the method.

5. LIMITATIONS OF INHERITANCE

Inheritance between object types can be very useful, but it is not always very relevant to the problem at hand and there are many other routes to reuse.

5.1. The limited applicability of inheritance in database structures

Inheritance is a relatively minor concern in database design. In practice, most objects in database systems are connected by associative or composition relationships rather than inheritance relationships, so the possibilities for reuse by inheritance between object types are limited.

Why? There are relatively few real-world inheritance trees stable enough to be built into a fixed database structure. For example, you cannot expect that every Vehicle in the world is either a Truck or a Car. Eventually you are bound to come across a Vehicle that exhibits the characteristics of both Truck and Car. Few class hierarchies are so certain that they can specified in object data and behaviour models.

In short, if reuse is the aim, database designers should not expect too much of inheritance between object types and should learn other techniques for increasing reuse.

5.2. Potential for other kinds of reuse

You can design inheritance between event types as well as between object types. Inheritance trees of event types are probably more useful than inheritance trees of object types. Later we will extend object modelling with the notion of superevents and show how reuse of methods can be spotted in a control-structured state-transition diagram or object behaviour model.

In the external design, entirely independent of the object event model, there are many further opportunities for achieving reuse.

5.3. Potential clash between types and database objects

It is not at all clear that a 'type' in an inheritance tree maps well onto a 'relation' in a relational data model or a 'table' in a physical database. An indication of this mismatch was given in Section 3.

5.4. Potential clash between types and object state management

The view taken by some OO theorists to date is that you can draw a state-transition diagram for each object in an inheritance tree and this will resolve all your state management problems. It does not turn out to be quite that simple, as we shall see.

6. STATE MANAGEMENT IN THE INHERITANCE PARADIGM

The earlier sections have suggested you need to place OO ideas within an analysis methodology that helps you with problems such as: designing the mapping of logical objects onto persistent data, discovering and specifying the 'right' set of methods, discovering and specifying the message routing between objects and discovering, and specifying object state management.

The rest of this paper is concerned with the last of these problems. The inheritance paradigm works best where the methods are independent of each other, independent of the context in which they are called, independent of any control structure. It becomes less attractive when the types in an inheritance tree need to maintain and test some kind of state variable. It does not help you to discover or show that methods are constrained by an object's state in ways like these:

- 'Replacement on Road' can only follow 'Removal from Road'.
- 'Removal from Road' can precede only 'Replacement on Road'.
- 'Mileage Increment' cannot happen while a Vehicle is off the road.
- 'Foreign Trip' cannot happen while a Truck is off the road.

6.1. State management

A well-established principle of program design (perhaps the oldest idea of encapsulation) is that a state variable is private property. It is private to the finite-state machine which manages it. How does this apply to super and subtype processes?

When the Foreign Trip event fires the Foreign Trip method in a Truck, how does the Truck check whether it is on or off the road? There are at least three possibilities, which we illustrate in Figure 5 and discuss below.

6.2. Meaning of private and protected

By 'private' we mean not directly inspectable by a method of any other object class. By 'protected' we mean 'inheritable and inspectable by subtypes'. We use 'protected' in the C++ sense. We don't mean 'public', that the state variable can be directly inspectable by any of its clients (objects connected by associative relationships rather than subtype relationships). This would clearly break the principle of encapsulation and be against the spirit of OO design.

6.3. Private state variables in both super and subtype

You might maintain a Vehicle state variable and a Truck state variable, and declare variants of the Removal from Road and Replacement on Road methods in Vehicle and Truck to maintain these two state variables separately. OO programmers don't like this approach. The duplication of maintaining and testing two state variables that do the same job is wasteful. It will lead to difficulties in maintenance.

Note in Figure 5 that the Vehicle Registration event invokes a method in both super and subtype objects. In Vehicle it creates an instance of the class. In Truck it initializes both the Truck State and the Foreign Trips attributes.

6.4. Private state variable in supertype, inspectable via enquiry method

You could design the Foreign Trip event to fire an enquiry method in Vehicle (to test whether it is on the road) and an update method in Truck (to increment the Foreign Trips counter) as illustrated below. OO

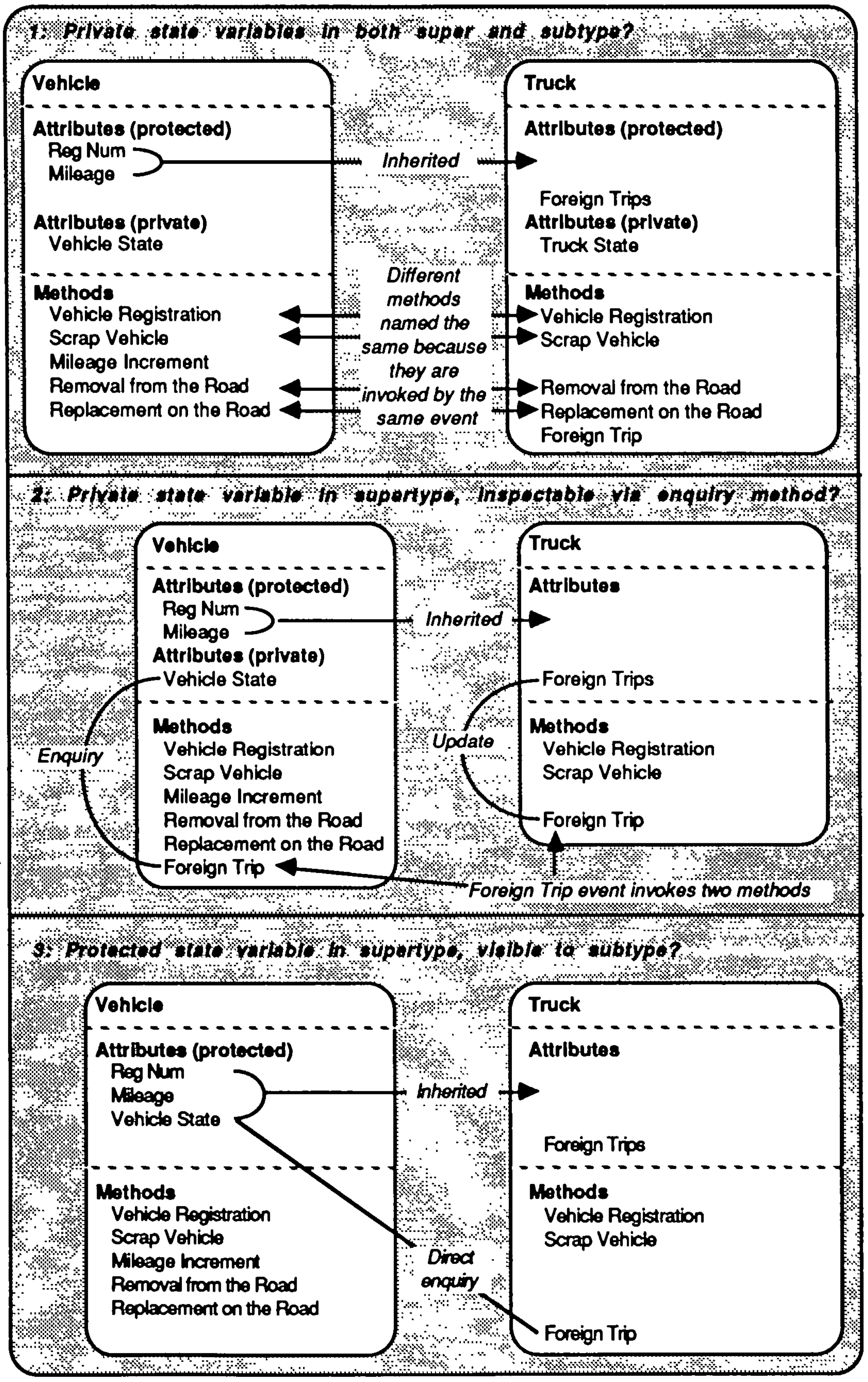
1: Private state variables in both super and subtype?
Vehicle
Attributes (protected)
Reg Num
Mileage
Attributes (private)
Vehicle State
Methods
Vehicle Registration
Scrap Vehicle
Mileage Increment
Removal from the Road
Replacement on the Road
Inherited
Different methods named the same because they are invoked by the same event
Truck
Attributes (protected)
Foreign Trips
Attributes (private)
Truck State
Methods
Vehicle Registration
Scrap Vehicle
Removal from the Road
Replacement on the Road
Foreign Trip
2: Private state variable in supertype, inspectable via enquiry method?
Vehicle
Attributes (protected)
Reg Num
Mileage
Attributes (private)
Vehicle State
Methods
Vehicle Registration
Scrap Vehicle
Mileage Increment
Removal from the Road
Replacement on the Road
Foreign Trip
Enquiry
Inherited
Update
Truck
Attributes
Foreign Trips
Methods
Vehicle Registration
Scrap Vehicle
Foreign Trip
Foreign Trip event invokes two methods
3: Protected state variable in supertype, visible to subtype?
Vehicle
Attributes (protected)
Reg Num
Mileage
Vehicle State
Methods
Vehicle Registration
Scrap Vehicle
Mileage Increment
Removal from the Road
Replacement on the Road
Inherited
Direct enquiry
Truck
Attributes
Foreign Trips
Methods
Vehicle Registration
Scrap Vehicle
Foreign Trip

FIGURE 5.

programmers do not like this approach either. 'We run the risk of having to rewrite Vehicle if Truck needs an unforeseen manipulation of the state variable.' They would rather choose the third option below.

6.5 Protected state variable in supertype, visible to subtype

Most OO programmers would design the method called Foreign Trip in Truck to directly test the Vehicle State inherited from Vehicle. They argue this does not break the principle of encapsulation. Encapsulation (they say) means enclosing the attributes and methods of an object instance rather than an object type. They assure us it is OK for the Vehicle State to be 'hidden from clients, but visible to derived classes'.

However, something funny is going on here, not quite in the spirit of OO design. How we avoid the problem of 'having to rewrite Vehicle if Truck needs an unforeseen manipulation of the state variable'? Several if not all of the methods of Truck must inspect the Vehicle state variable as a precondition. In other examples, might they also have to update it? The super and subtype objects seem hopelessly entangled.

Using the inheritance paradigm, as in current OO programming environments, it is not obvious how best to do state testing and updating. But it is not just a question of designing the optimum solution. It is hard to *visualize* state management problems in terms of inheritance structures alone. Whichever way you look at it, the inheritance paradigm makes it awkward to discover, specify and maintain object state management.

Also, there is important information missing from the diagrams above, not only the detail of the preconditions and operations, but also the names of the events that invoke the methods. Events and methods are not always in one-to-one correspondence. We need the event-oriented view to define message routing.

7. THE STATE-TRANSITION PARADIGM

Tsvi Bar-David (1994) says an object is 'an entity that is born, lives and dies. Over its lifetime, the object has a state that varies'. Actually, some objects do not get to change their state before they are destroyed, but this is minor quibble with a useful time-oriented definition of objects.

For some objects, notably persistent objects of the kind stored in a database, 'the event- and time-ordering of operations is so pervasive that we can best characterize the behaviour of such objects in terms of an equivalent finite-state machine' (Booch, 1994).

Several graphical notations have been proposed for modelling object behaviour. Unstructured state-transition diagrams (see Section 10) are simple and there is a lot to be said for using the simplest theory that will work. The trouble is, they are not a theory, they are just a descriptive tool. Analysts need more than diagrams, they need techniques to make the process of systems analysis hang together, to guide the analyst in building the diagrams. A limitation of unstructured state-transition diagrams is that they do not:

- Play an active role in helping you to discover the rules.
- Formally link analysis with design and process specification.
- Encourage stereotypical representation.

However, Section 10 shows a state-transition approach that does these things, developed from JSP. The JSP idea that each finite-state machine represents the behaviour of an object has been refined over a period of almost 20 years. From Infotech Systems Technology (Robinson, 1977, 1979) to SSADM (CCTA, 1994), you *visualize* the process that models an object as a single control-structured state-transition diagram, entity life history or 'object behaviour model'. Ignoring inheritance for the moment, Figure 6 models the process that is a Truck.

Our experience is that you can best visualize and recognise a finite-state pattern of events by representing the control structure graphically in an object behaviour model. An object behaviour model shows all the events affecting one object. It gives a dynamic view of the object in terms of the events that update its attributes and relationships, and the sequential constraints on these events.

A control-structured state-transition diagram 'encapsulates' all the object's behaviour. Methods appear as the leaves of the structure, as the 'effects' of 'events'. These methods are invoked from outside the object by the events.

The value of the state variable after each method appears in the bottom-right-hand corner of the event effect box (1 = on the road, 2 = off the road). We 'optimize' the states as shown in Robinson and Berrisford (1994). The valid prior states or preconditions before each method are not shown on the diagram, since both prior and successor states are deducible from the structure of the diagram. Given a control-structured diagram like this, at least two current CASE tools can automatically generate the necessary state variable setting and testing.

7.1. Benefits of the state-transition paradigm

A control-structured state-transition diagram provides at least two important benefits. First, it graphically shows the *pattern* of invocations that an object will respond to (not just the individual invocations). It shows which invocations are valid and when (in which object state) they are valid. Because of this, it is easier to design the processes that maintain the object. You can design a program according to the simplest possible view of its task, then transform this program into a callable package of routines. The transformation is done via 'program inversion' in JSP and via event diagrams in our method.

Second, the approach separates events from the 'methods' they trigger in objects. If you name methods in different objects after the events that invoke them,

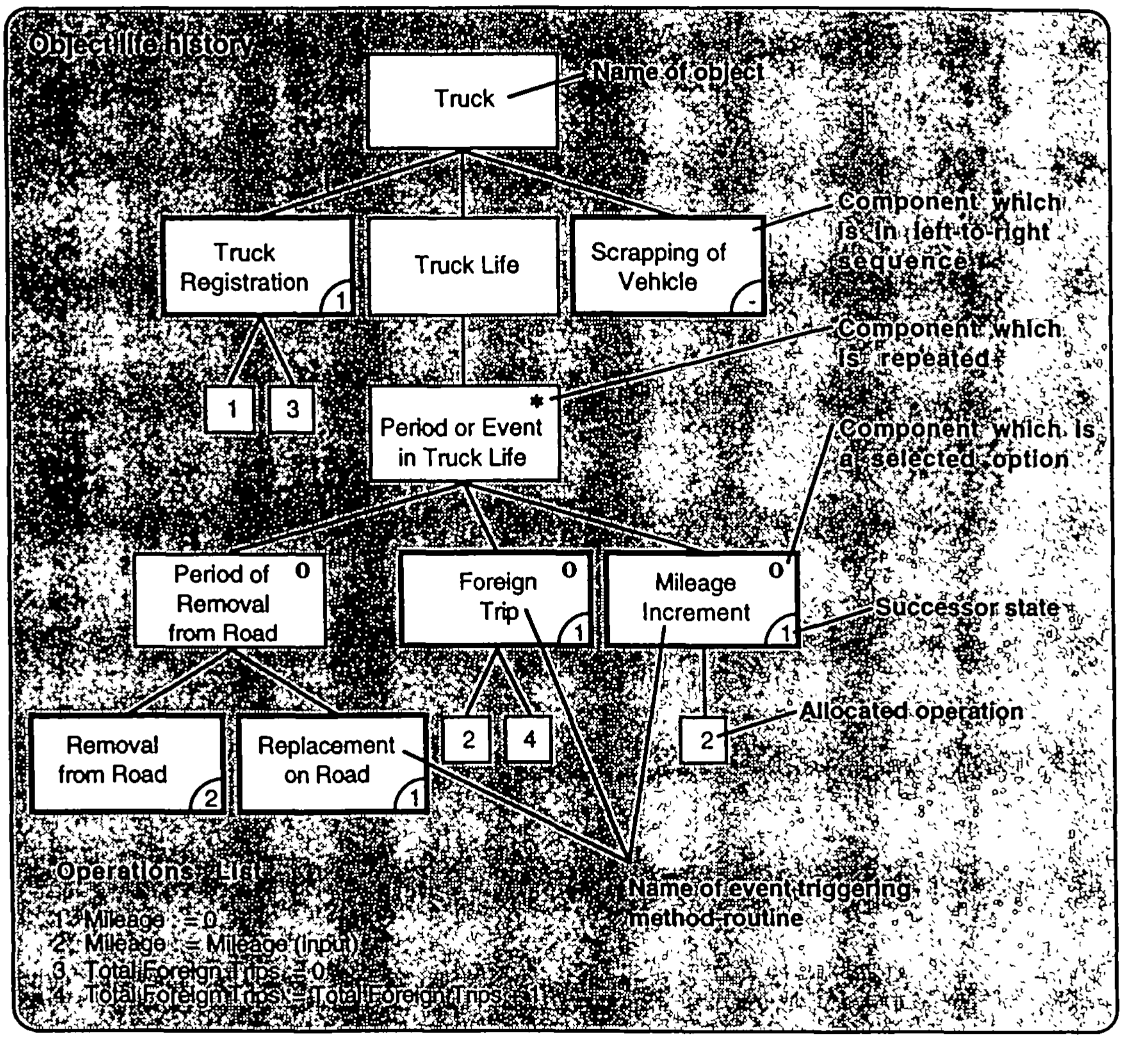
Object life history
Truck
Name of object
Truck Registration
1
Truck Life
Scrapping of Vehicle
-
Component which is in left-to-right sequence
Component which is repeated
1
3
*
Period or Event in Truck Life
Component which is a selected option
Period of Removal from Road
o
Foreign Trip
o
1
Mileage Increment
o
1
Successor state
Removal from Road
2
Replacement on Road
1
2
4
2
Allocated operation
Name of event triggering method-routine
Operations List
1 Mileage := 0
2 Mileage := Mileage (input)
3 Total Foreign Trips := 0
4 Total Foreign Trips := Total Foreign Trips + 1

FIGURE 6.

then it is relatively easy to extract an event-oriented view of the system. It becomes natural to define the message routing between objects, a mysterious process in OO programming, by drawing a diagram for each event as shown in Section 10.

7.2. Limitations of the state-transition paradigm

The control-structured state-transition paradigm above has its own set of limitations.

7.2.1. Backtracking

A process hits a 'recognition problem' when, in consuming an input data structure one-record-or-event-at-a-time, the process has insufficient data to test a condition on entry to a selection or iteration in the formal grammar defining the input data structure. It cannot recognise what path to take. It cannot parse the grammar of its input data structure as it goes along. Jackson defined a 'backtracking' technique to resolve this kind of problem.

A theoretical limitation that troubles us is that the backtracking constructs of 'posit', 'admit' and 'quit' are not universally accepted. We cannot quote a mathematical treatise that accommodates them within the theory of formal grammars.

A practical limitation to successful teaching has been ignorance of when and how the backtracking technique must be applied to control-structured diagrams. We overcome this obstacle by teaching stereotype control structures, easily recognisable standard models for defining the business rules and constraints associated with state-changes (unsupported claim!).

7.2.2. Independent aspects of an object

If you regard the whole of a normalized relation or a database table as being one object, a minor complication of the state-transition paradigm is that one object may have several independent aspects, or parallel object behaviour models, as shown in Section 10. It turns out this idea is helpful for specifying inheritance (see Section 8 below).

7.2.3. Reuse of methods by events

An apparent limitation is that each method is defined as being specific not only to the object but also to the *event* that calls it. However, we can easily extend the theory (at least it looks easy in retrospect) so that two or more different events can reuse the same method in one object (see Section 8 below).

7.2.4. Inheritance of methods between objects

Some OO authors dismiss state-machine oriented approaches as being merely 'object-based', meaning that one object type cannot share the methods of another. However, you can get the benefits of inheritance. The next section shows how to reconcile the paradigms outlined in this paper, the type-oriented and state-oriented views of objects.

8. INHERITANCE IN THE STATE-TRANSITION PARADIGM

We do not find inheritance between object types to be a very significant part of database design, but we must know how to specify inheritance when it does occur. Your first thought might be to draw three object behaviour models: Vehicle, Car and Truck. But to represent inheritance of methods you need only two behaviour models, both for the supertype Vehicle, one called Vehicle-generalization and the other called Vehicle-specialization.

Figure 7 shows how to specify the subtyping of Vehicle by a high-level selection in the Vehicle-specialization diagram. By the way, the value of the state variable above can be used to indicate the subtype.

Figure 7 shows how to specify the methods inherited by both subtypes in the Vehicle-generalization life.

Figure 8 shows that several events can reuse one method. The Car Registration and Truck Registration events invoke one method called Vehicle Registration. The method here is not a 'leaf' of the structure; it emerges above a low-level selection of events with the same operations.

Note also, one event may invoke several methods. The Foreign Trip event fires an update method in Vehicle-specialization (to update the Foreign Trips attribute) and an enquiry-only method in Vehicle-generalization (to check the Vehicle is on the road). (We have dropped an operation shown on the previous Truck diagram, just to illustrate an enquiry-only method.)

8.1. The possibility of distinct subtype behaviour models (unsupported claim!)

The more complex the situation, the greater the number of parallel behaviour models. If Car and Truck have independent aspects you will have to draw distinct behaviour models for them, anchored to the Vehicle-specialization life by their birth and death events.

8.2. Multiple inheritance (unsupported claim!)

So far the example shows only single inheritance. Suppose we develop a multiple-inheritance structure. Say Truck inherits from the two supertypes Vehicle and Taxable Asset. You would then show Truck as an option in the specialization lives of both its supertypes. And if Truck has properties that are independent of both its supertypes, you would have to draw a further parallel object behaviour model for Truck on its own.

9. CONCLUSIONS AND REMARKS

9.1. Polymorphism

Scrap Vehicle (see Figs. 7 & 20) is a *polymorphic* event.

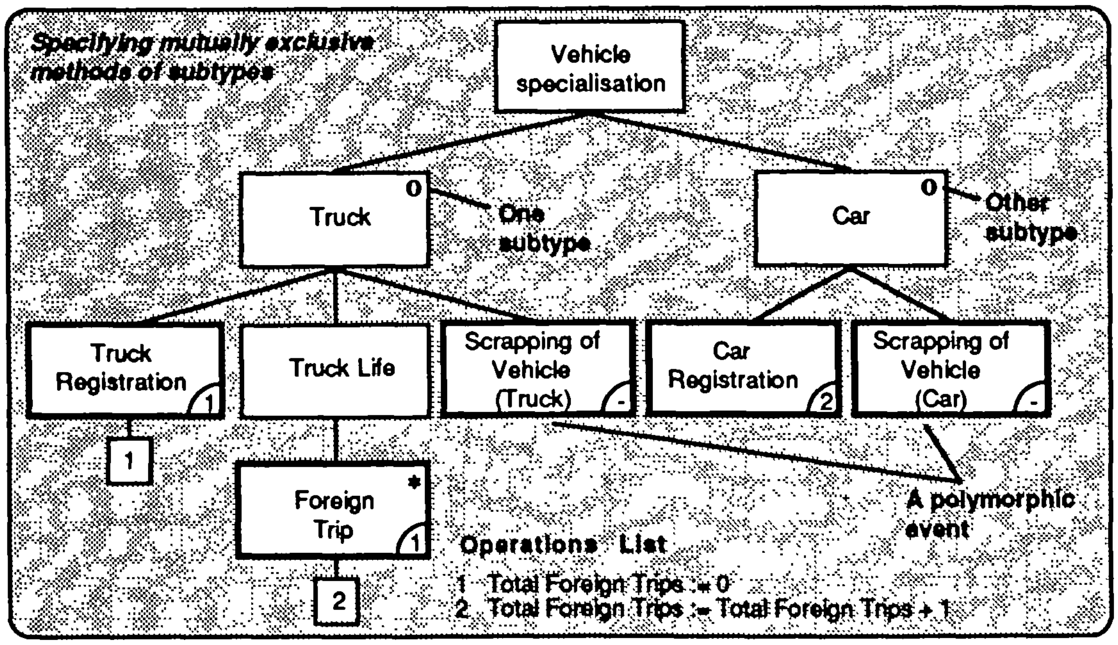
Specifying mutually exclusive methods of subtypes
Vehicle specialisation
Truck
O
One subtype
Car
O
Other subtype
Truck Registration
1
Truck Life
Scrapping of Vehicle (Truck)
-
Car Registration
2
Scrapping of Vehicle (Car)
-
1
Foreign Trip
*
1
2
A polymorphic event
Operations List
1 Total Foreign Trips := 0
2 Total Foreign Trips := Total Foreign Trips + 1

FIGURE 7.

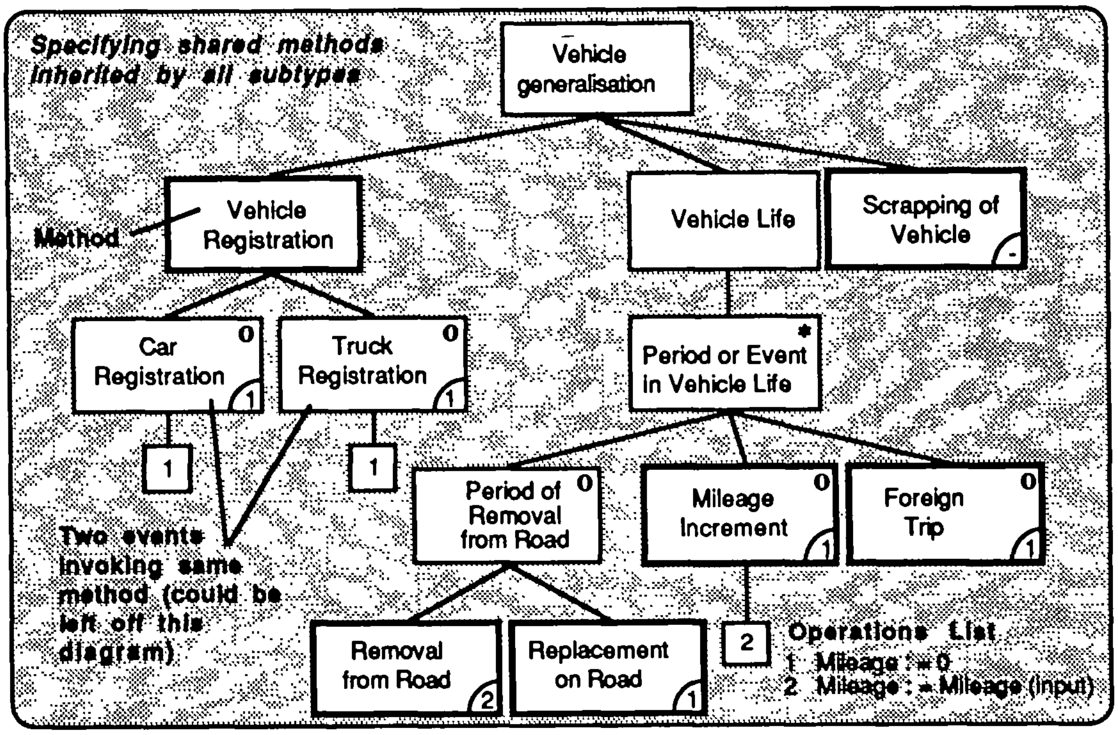

FIGURE 8.

The input message on the event is apparently careless of the sub-type (Car or Truck) it affects. It carries a Vehicle object's identity but not its type.

Some of the type-independence you can achieve in a process control system that sends messages directly to real-world objects is lost in a database system that has to retrieve stored information about real-world objects.

Given that the sub-types of a class draw their keys from the same range of identifiers, you don't need to know the type of an object when you send it a message or read its database record, but *you do need to know the type before you can interpret the data in the record you have retrieved.*

The approach in this paper leads us to construct a single-machine that relates all the sub-types of a class. The benefit is that we don't later have to add any extra code to work out the type affected by a polymorphic event. The different effects are already recorded in the state machines and are automatically generated as mutually exclusive options of one 'method' into the event diagram.

9.2. Different approaches for different applications

The diagram notations of the two paradigms have similar semantics, but you can see from the books that an OO designer like Booch typically draws diagrams with a different *shape* or *pattern* from those of a database analyst. This difference stems not from the methodologies but from the characteristics of problem domains being studied.

Real-time systems
(CASE, CAD, telecommunications & process control)

- mostly transient objects
- mainly one-to-one associations
- many class hierarchy relationships
- few objects undergo state-transitions
- few events hit several object instances
- single-user
- little historic data maintenance

Database systems
(inventory, billing and record keeping systems)

- mostly **persistent** objects
- mainly **one-to-many** associations
- **few** class hierarchy relationships
- **most** objects undergo state-transitions
- **many** events hit several object instances
- **multi**-user (hence locking of data et al.)
- **much** historic data maintenance.

The combined effect of these differences is profound! Objects persist longer in database systems. Almost every object is affected by three state-change events (birth, death and deletion). This means that the state-machine view of an object class (which, curiously, is often presented by academics as a feature of real-time systems) is usually *more complicated* and therefore *more important* in database systems.

The rules controlling the interaction between objects on these state-change events can be quite complex. The longer objects persist, the more complex the rules and the interactions tend to become, and the more you need event-oriented analysis techniques and notations of the kind in this paper.

9.3. Five proposals

The arguments in this paper, and our experience with modelling finite-state machines in database systems, lead us to conclude as follows.

9.2.1 Relational data analysis

For database systems, RDA is a useful way for database system analysts to reach a first-cut identification of object classes (though only a partial definition of course).

9.2.2. Method body specification on diagrams

For database systems, the method bodies of methods are so simple they can be fully documented and displayed on the graphical specification of:

(a) the state-machines for the classes
(b) the object/interaction diagrams for the events

Event diagrams show not only the principal message routing between objects, but the method bodies themselves. Our long-term aims are to generate (b) from (a), and generate working code from (b). Our CASE tool can do some limited generation at the moment.

9.2.3. Specification of mutual exclusion between state-machines

The above aim means that we have to specify the mutual exclusion between the state-machines of sub-types in a class hierarchy. The easiest way is to join these state-machines together as options of a selection.

Those OO authors who distinguish 'object-based' from object-oriented methods realise it is not a trivial task to reconcile finite-state machine theory with OO theory. But it turns out that a grand unified theory is conceivable, we have shown in Section 8 that the state-transition paradigm does provide the tools we need for designing inheritance into a system.

9.2.4. Class to state-machine correspondence

For database systems, it is convenient (at least) for some purposes (at least) to equate the notion of 'class' with 'state-machine'.

What is an object? In our example there are three object types (Vehicle, Car and Truck) but only two object state machines (Vehicle-generalisation and Vehicle-specialisation). There is a structure clash between the notion of an object as a type in an inheritance tree (with a flat list of methods) and the notion of an object as a finite-state machine (with a control structure sitting over the methods).

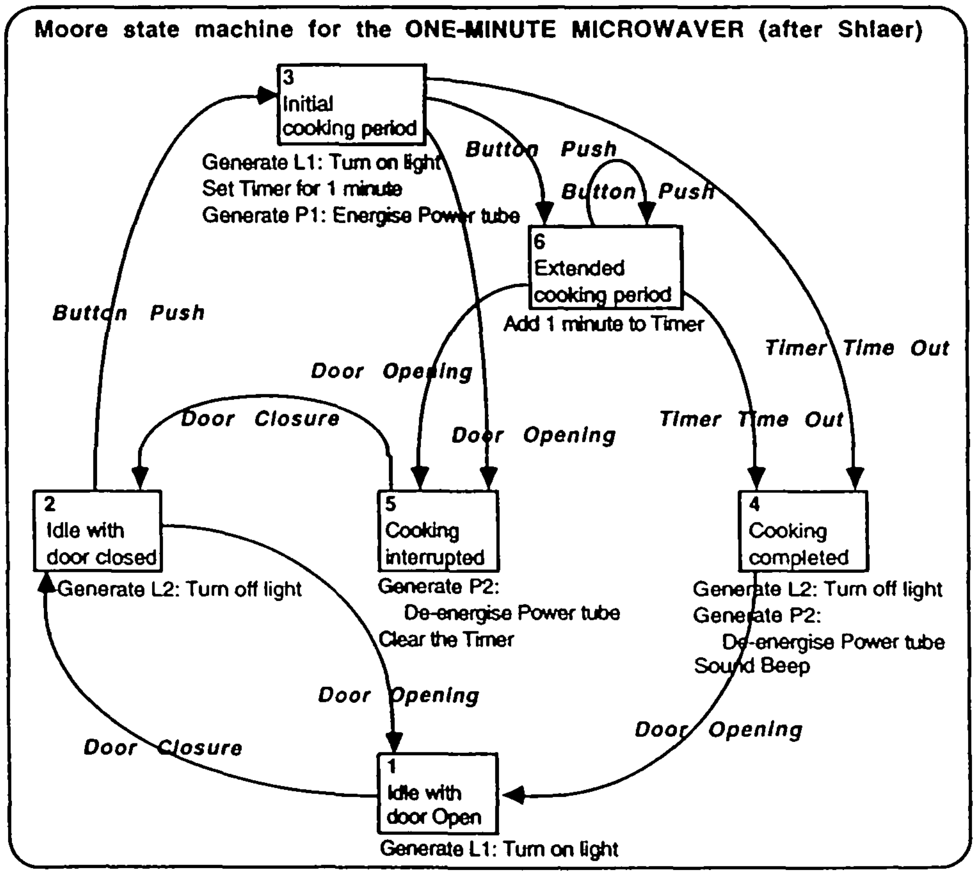

FIGURE 9.

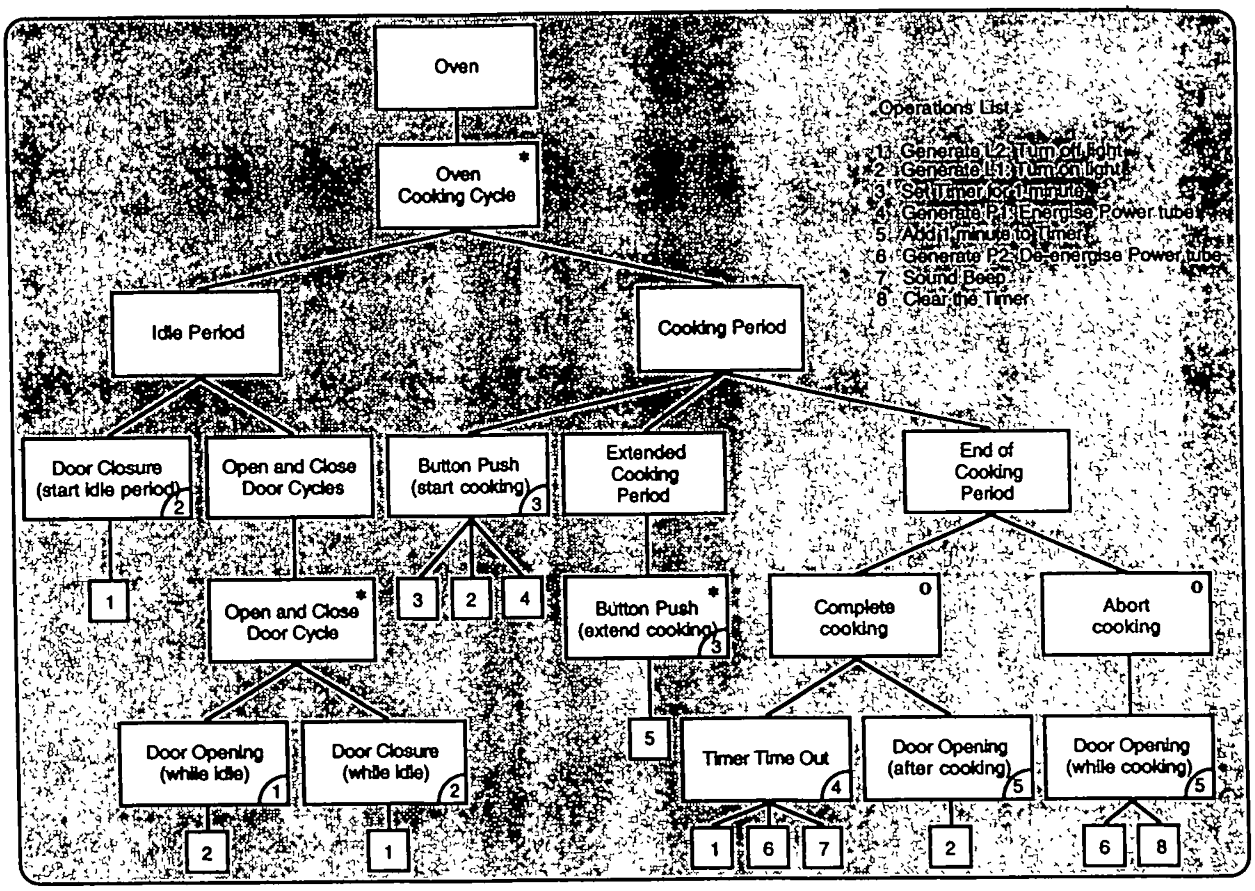
Oven
Oven Cooking Cycle *
Operations List
1 Generate L2: Turn off light
2 Generate L1: Turn on light
3 Set Timer for 1 minute
4 Generate P1: Energise Power tube
5 Add 1 minute to Timer
6 Generate P2: De-energise Power tube
7 Sound Beep
8 Clear the Timer
Idle Period
Cooking Period
Door Closure (start idle period) 2
Open and Close Door Cycles
Button Push (start cooking) 3
Extended Cooking Period
End of Cooking Period
1
Open and Close Door Cycle *
3
2
4
Button Push (extend cooking) * 3
Complete cooking o
Abort cooking o
Door Opening (while idle) 1
Door Closure (while idle) 2
5
Timer Time Out 4
Door Opening (after cooking) 5
Door Opening (while cooking) 5
2
1
1
6
7
2
6
8

FIGURE 10.

9.2.5. State-machine representation

For database systems, there are good reasons (not listed here) to represent a state-machine using a regular expression rather than a statechart. And we propose that the Jackson notation (together with the concept of resolving structure clashes by parallel aspects related via event diagrams) is a significant advance on the notations of Moore, Harel and others. Section 10 says more about separating parallel aspects of an object to resolve structure clashes.

The only drawback, not revealed by the example, is that need for Posit, Admit and Quit constructs in some object behaviour models, but we have recently made improvements in the theory and teaching of the backtracking technique.

9.4. How is message routing specified?

The message routing issue is discussed in Section 10. Each arrow in an event diagram represents one-to-correspondence rather than a message. But loosely, the arrows indicate where the identifier comes from to read an object, so in most cases a message naturally follows the arrow. The messages necessary to implement an event usually travel along the arrows in the direction of the read access path, but sometimes they flow in both directions.

The message routing in event diagrams was discussed by Robinson and Berrisford (1994) and more recently Hall (1994). Simply, there are two strategies. You may delegate the message routing task to each of the objects affected by an event (usual in OO programming). Alternatively, you may give the message routing task to a distinct, though transient, event manager object.

In a typical database system, each transaction is written as a process. You may regard the transaction process as being an event manager, with methods from various objects embedded within it. So, the old-fashioned transaction process is a kind of aggregate set of OO methods, optimized in that they can communicate via the working storage of the process rather than sending messages to each other!

10. MORE ABOUT OBJECT AND EVENT MODELLING

This final section, to be published in *ROAD* magazine, Jan–Feb 1994, gives a more complete view of:

- modelling object behaviour as regular expressions
- separating the parallel aspects of an object
- specifying the interactions between objects.

This section may be viewed as an appendix, but many readers will find it necessary to follow some of the arguments in earlier sections, or to satisfy themselves that state-transition diagrams and object behaviour models are semantically equivalent. (Grady Booch tell us that he thinks this can be mathematically proven).

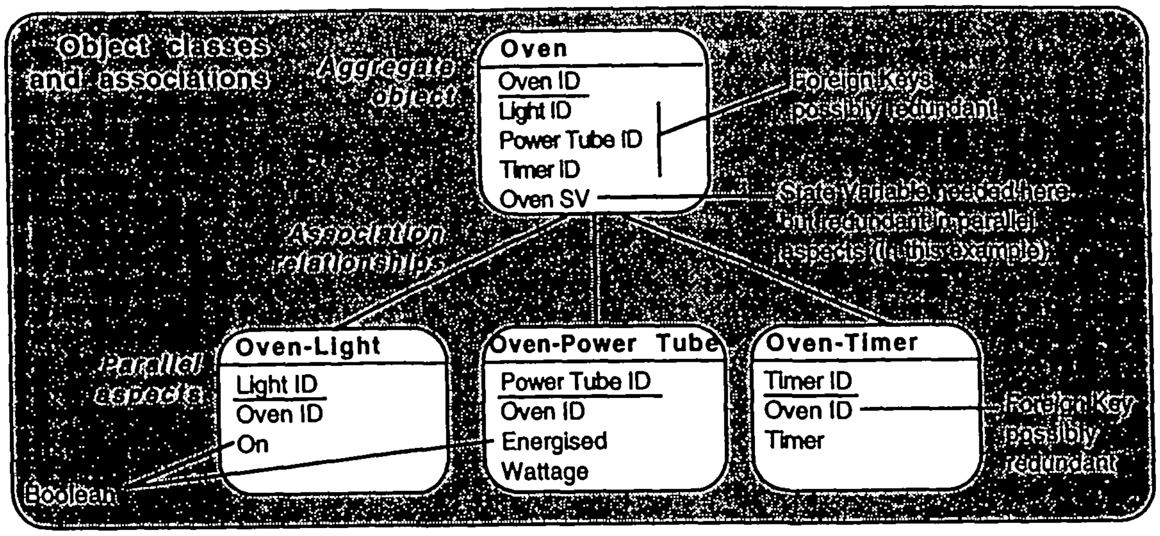
Object classes and associations
Aggregate object
Oven
Oven ID
Light ID
Power Tube ID
Timer ID
Oven SV
Foreign Keys possibly redundant
State Variable needed here but redundant in parallel aspects (in this example)
Association relationships
Parallel aspects
Oven-Light
Light ID
Oven ID
On
Oven-Power Tube
Power Tube ID
Oven ID
Energised
Wattage
Oven-Timer
Timer ID
Oven ID
Timer
Foreign Key possibly redundant
Boolean

FIGURE 11.

Since the case study is a real-time process control system, we can avoid a lot of the 'tedious detail' Grady Booch complains of in database systems. In terms of what we know as the 3-schema architecture, we can concentrate on the *conceptual model*, assuming that the *external design* and *internal design* will be trivial.

10.1. Event-oriented specification of object behaviour

Shlaer (1994) discusses Turing machines and their representation using the notations of Mealy, Moore and Harel. Figure 9 is our version of the diagram she draws to represent an oven, given a specified requirement.

Figure 9 shows six states, four event types and eight operation types. But how many object types are there? Sally goes on to model the Light and the Power Tube as separate and relatively trivial state machines. We are not sure from the article why the Timer and the Beeper are not selected for modeling, and for completeness we model all object aspects below.

Figure 10 shows the same finite-state machine as a formal grammar or regular expression. Sequence is left to right, * marks an iterated component and an open circle

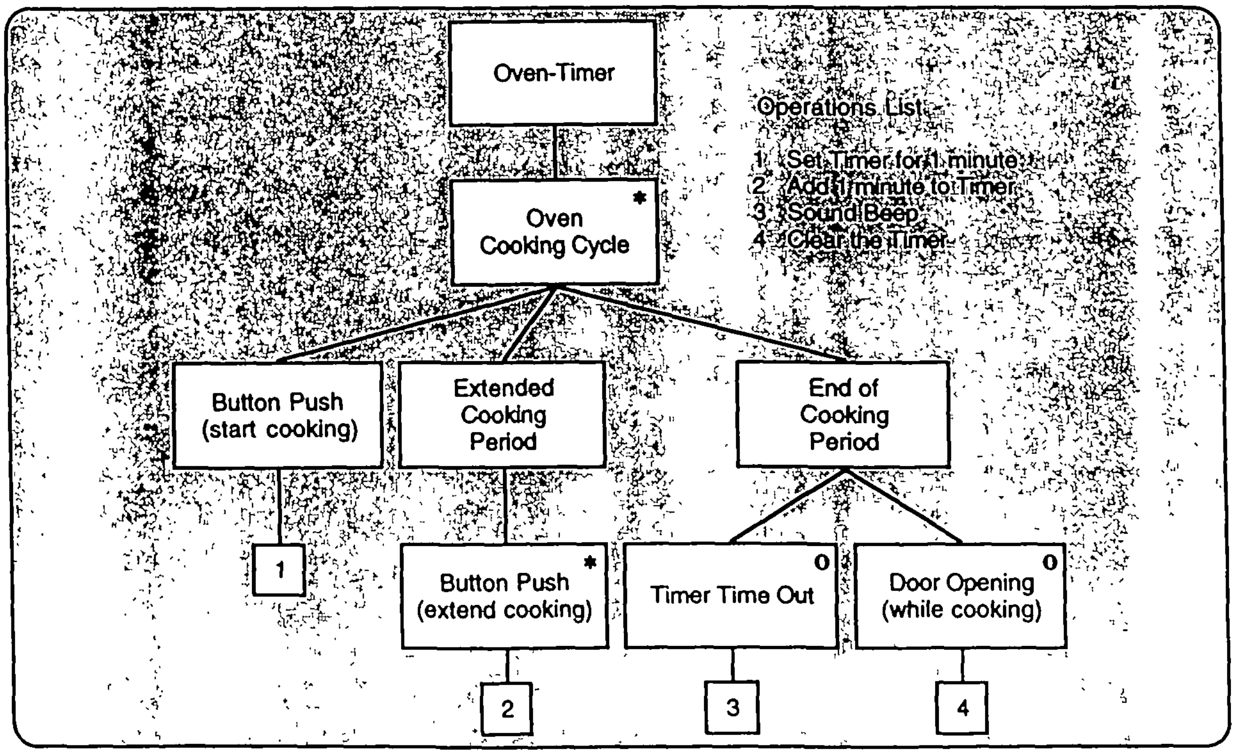

FIGURE 12.

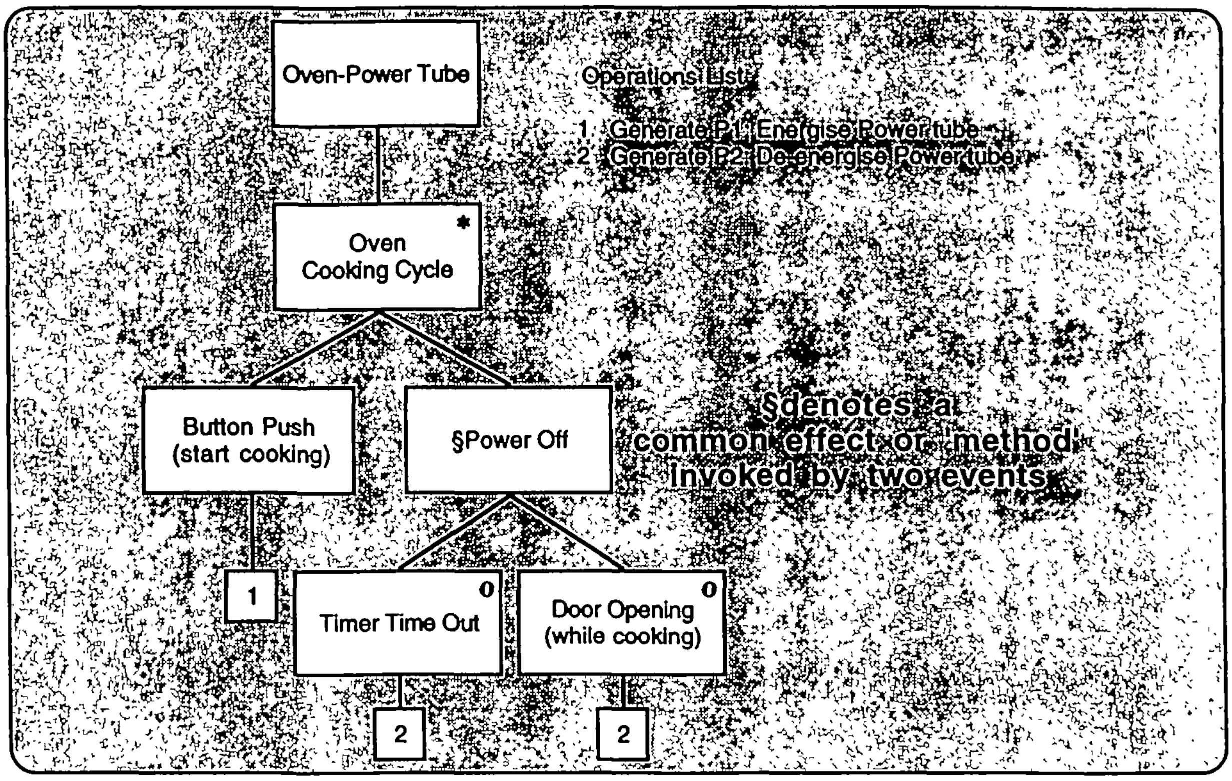
Oven-Power Tube
Operations List:
1. Generate P1: Energise Power tube
2. Generate P2: De-energise Power tube
Oven Cooking Cycle *
Button Push (start cooking)
§Power Off
§ denotes a common effect or 'method' invoked by two events
1
Timer Time Out o
Door Opening (while cooking) o
2
2

FIGURE 13.

marks a selected option. Primitive operations are listed and allocated under the event effects. Events with different effects at different states are qualified with an effect name in brackets.

The numbered states are an incidental by-product of the event-oriented analysis. We number the states entirely mechanically. Curiously, our CASE tool generates only five states rather than six. Readers can resolve this puzzle by spotting two equivalent states in Shlaer's diagram.

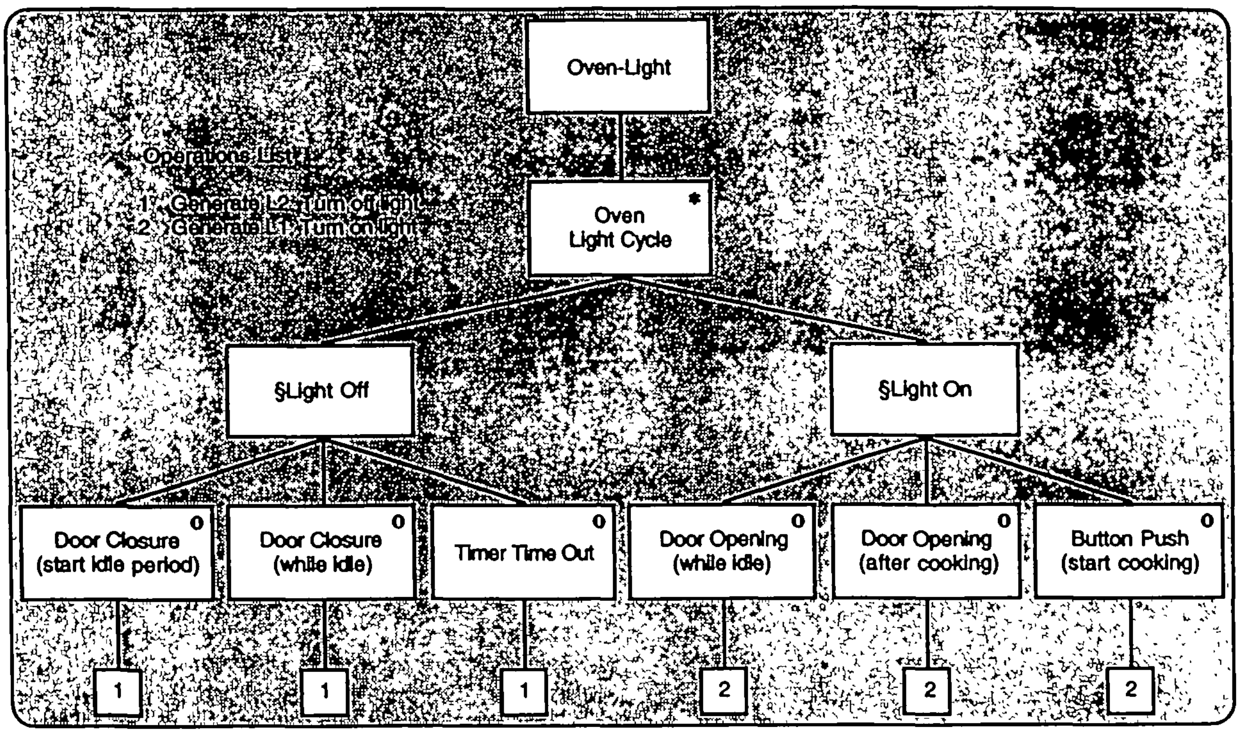
Oven-Light
Operations List:
1. Generate L2: Turn off light
2. Generate L1: Turn on light
Oven Light Cycle *
§Light Off
§Light On
Door Closure (start idle period) o
Door Closure (while idle) o
Timer Time Out o
Door Opening (while idle) o
Door Opening (after cooking) o
Button Push (start cooking) o
1
1
1
2
2
2

FIGURE 14.

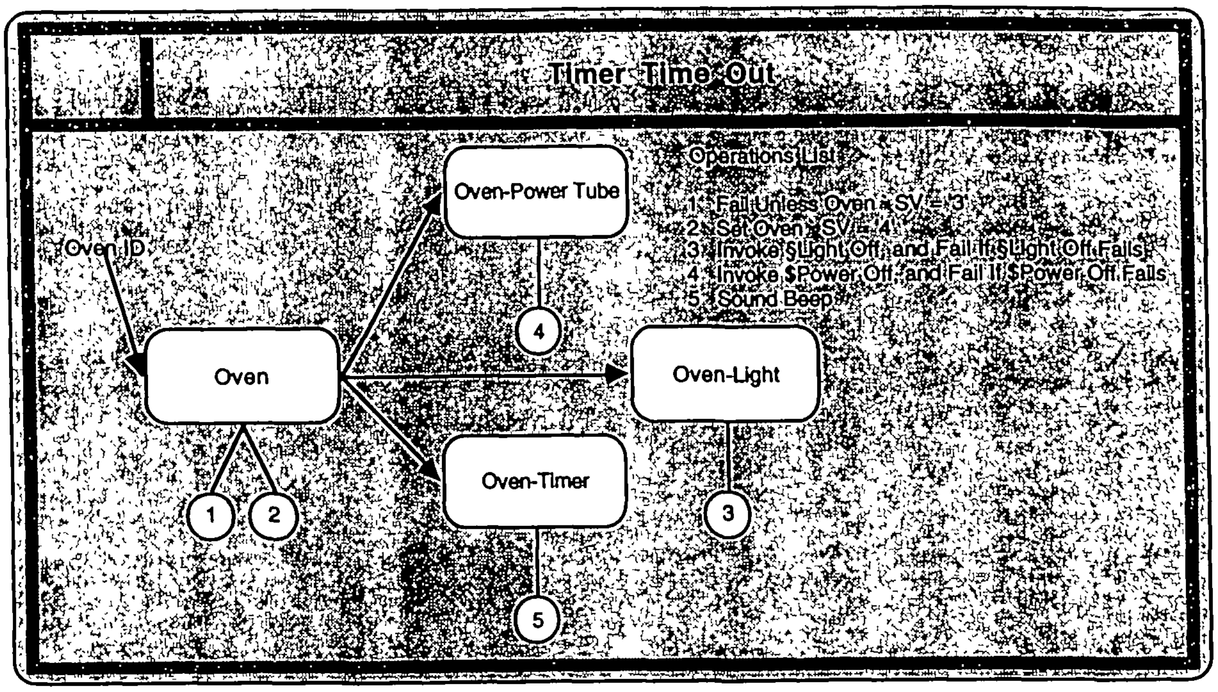
Timer Time Out
Oven.ID
Oven-Power Tube
Operations List
1 Fail Unless Oven.SV = 3
2 Set Oven.SV = 4
3 Invoke $Light Off, and Fail If $Light Off Fails
4 Invoke $Power Off, and Fail If $Power Off Fails
5 Sound Beep
Oven
4
Oven-Light
Oven-Timer
1
2
3
5

FIGURE 15.

We have our own syntax for operations, but here we have simply copied Sally's operations without altering their syntax. By the way, compare the alternation of operations 1 and 2 across the sequence in Figure 10 with Figure 14.

10.2. Parallel aspects of an object

Shlaer goes on to draw two alternative Harel statecharts. Each suppresses detail from one level of abstraction to another, by arranging the state transitions into super and substates. As she says, there seems no objective way to choose between them. There is a *structure clash* between different cycles in the Oven behaviour, the Idle-Cooking cycle and the Door Open–Door Closed cycle

My late and much-missed colleague Keith Robinson used to say that given an arbitrary choice between design options, either both are needed or a third way is better. We suggest what is really going on here is that the Oven is an aggregate object owning three parallel aspects and related to them by one-to-one associations. Figure 3 illustrates this in the form of something akin to a relational data model.

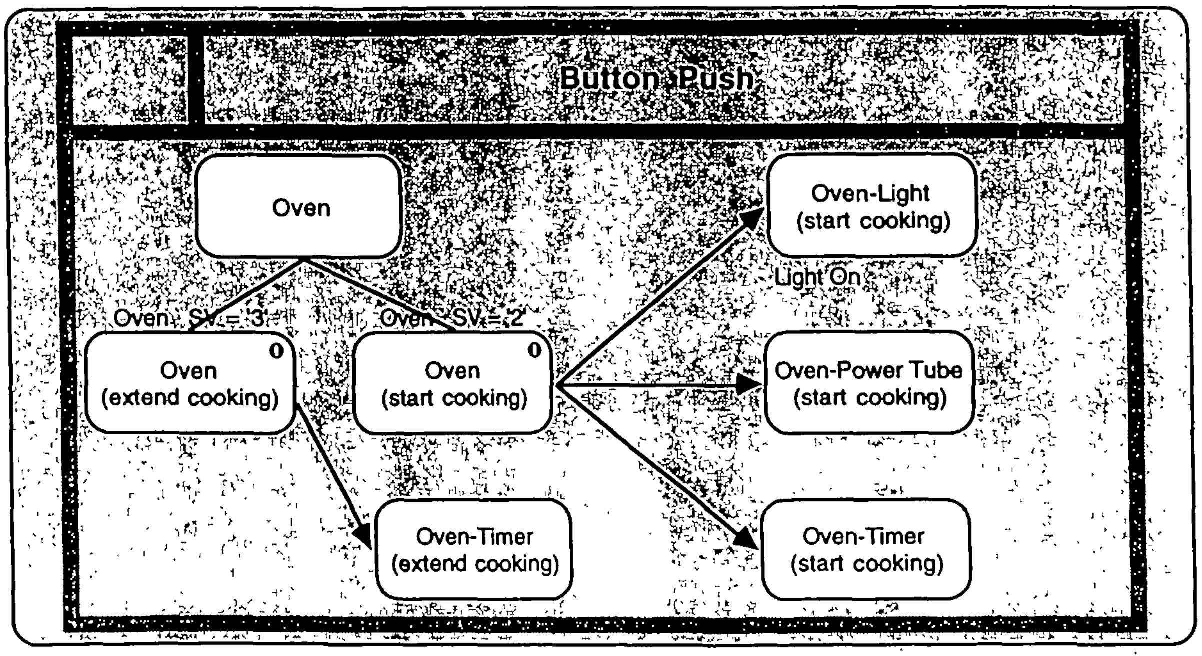
Button Push
Oven
Oven-Light (start cooking)
Light On
Oven.SV = 3
Oven.SV = 2
Oven (extend cooking)
0
Oven (start cooking)
0
Oven-Power Tube (start cooking)
Oven-Timer (extend cooking)
Oven-Timer (start cooking)

FIGURE 16.

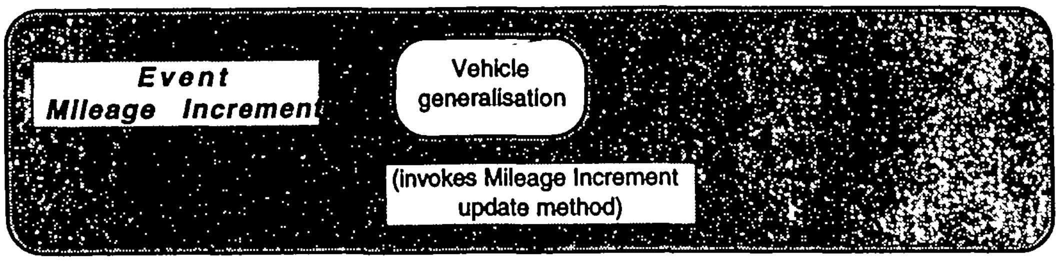

FIGURE 17.

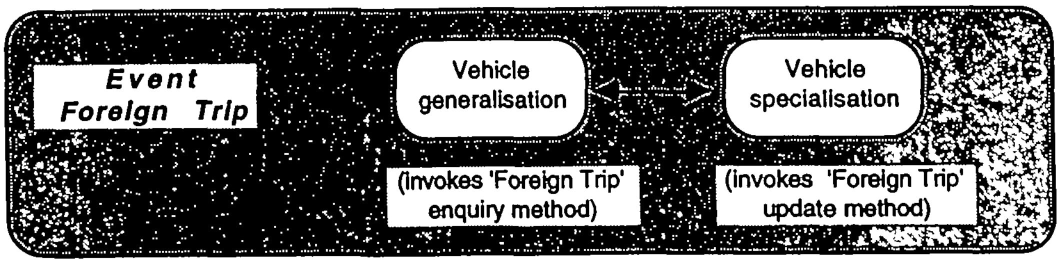
Event
Foreign Trip
Vehicle generalisation
Vehicle specialisation
(invokes 'Foreign Trip' enquiry method)
(invokes 'Foreign Trip' update method)

FIGURE 18.

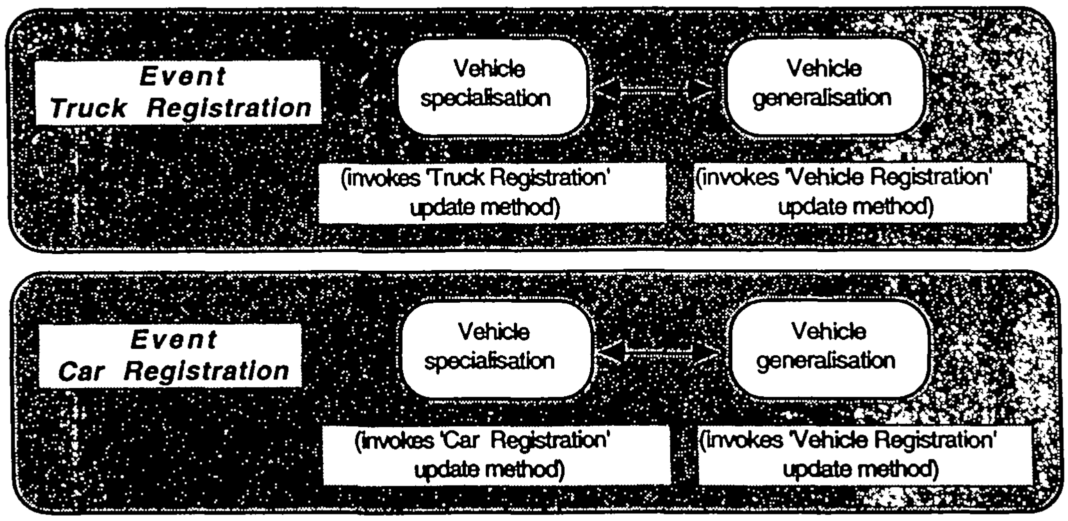
Event
Truck Registration
Vehicle specialisation
Vehicle generalisation
(invokes 'Truck Registration' update method)
(invokes 'Vehicle Registration' update method)
Event
Car Registration
Vehicle specialisation
Vehicle generalisation
(invokes 'Car Registration' update method)
(invokes 'Vehicle Registration' update method)

FIGURE 19.

This is such a simple system you might implement Figure 3 by rolling up all data into a single working-storage table and discarding the foreign keys. Or you might implement each object class as a table in relational database, using the foreign keys to locate the distributed objects when passing messages. This choice is a concern of the *internal design* you can hide in data management routines we call the process/data interface. The code specified in this article is that which implements the *conceptual model.*

The behaviour model for the aggregate object Oven is now as shown in Figure 11, but without the operations on it. The Oven will implement the rules, constraints or preconditions by testing and setting its state variable. The Oven will receive each event and pass it on to the relevant parallel aspect(s). This master-to-detail message-passing protocol is common but not always applicable.

Figure 12 is the subset of Figure 11 that models the behaviour of the Timer. The Beep aspect is so trivial it is easily controlled by the same finite-state machine. This behaviour needs no state variable, since it is a subset of the Oven behaviour that test and sets the only state variable in this example.

Figure 13 is the subset of Figure 10 that turns the Power Tube on and off. It features a common effect (or superevent) invoked (or reused) by more than one external event. We make a fuss about superevents in our book, not because they greatly simplify analysis or greatly increase reuse, but because they are a significant advance in reconciling an algorithmic theory of event effects with an object-oriented theory of 'methods'.

Figure 14 is the subset of Figure 10 that turns the Light on and off. Again, this behaviour needs no state variable. And since all the preconditions (fail-unless-valid-state tests) are tested by the Oven, you can reduce

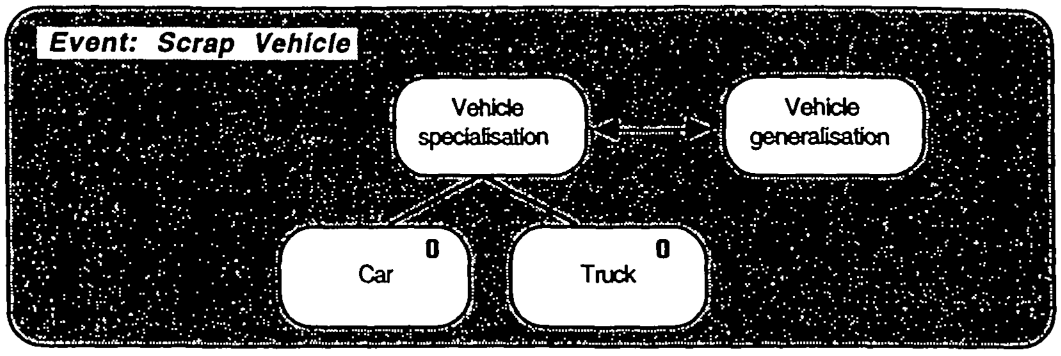

FIGURE 20.

the Oven-Light parallel aspect to a simple off–on cycle. The *structure clash* between this cycle and the idle-cooking cycle is resolved by the event diagrams that follow.

10.3. Event-oriented specification of message routing

Inter-object communication is naturally specified on an event-by-event basis. We generated the following event diagrams mechanically from the behaviour models shown above. Strictly, each arrow represents one-to-correspondence rather than a message. But loosely, the arrows indicate where the identifier comes from to read an object, so in most cases a message naturally follows the arrow.

Figure 15 shows the aggregate object receives the Timer Time Out event and passes it to all parallel aspects. Notice that where more than one event triggers the same 'method', the object invokes a common subroutine.

Figure 16 shows the Button Push event has two different sets of effects depending on the state of the oven. The aggregate object receives the event and passes it on only to the relevant parallel aspect(s). We have suppressed the detail of operations this time.

To be complete, the Button Push event diagram needs a third option. Shlaer did not model the case of the futile 'no effect' button push that may occur while the oven door is open. I am not saying she necessarily should have, but she could have. Such an 'enquiry effect' of an event must appear in the code that implements the event, and might be included in the object's behaviour model (it would appear twice, iterated, in Figure 10 and this duplication probably indicates some further parallelism too trivial to resolve) but I want to keep the illustration small and directly comparable with Shlaer's.

We have not fully described here what the event diagrams mean and we have ignored many important matters. The example does not include any events or methods that create or destroy object instances; this issue is addressed in our book, which is aimed at people building client–server database systems.

The example does not reveal another kind of structure clash between the parallel aspect (or finite-state machine) view of object classes and the subtype (or inheritance view) of object classes. This issue is clarified in the main body of the paper, where the case study includes an inheritance tree. Does the inheritance tree appear in an event diagram? We have to return the case study in the main body of the paper.

10.4. An inheritance structure in an event diagram

To define the message routing between objects, we start by drawing an event diagram for each event, showing one-to-one correspondences between the effects of an event. In the Vehicle Registration case study used in the main body of the paper, the Removal from Road, Replacement on Road and Mileage Increment events all invoke a method only in one object. Figure 17 illustrates the last of these.

Several of the other events invoke methods in both aspects of Vehicle. The event diagrams are almost as simple as the one above, simply showing both aspects of Vehicle in one-to-one correspondence. Figure 18 specifies the Foreign Trip event.

Using the object data model as our guide, we show the inheritance tree (Vehicle-specialization) as one object, one box, in these event diagrams. Event processes will create, read, write and delete instances of Vehicle-specialization making no distinction between the various subtypes. Figure 19 specifies two more example events.

So does the inheritance tree ever make itself evident an event diagram? In the case study, only the *polymorphic* Scrap Vehicle event reveals the inheritance tree. Fig. 20 shows Scrap Vehicle makes two appearances in the behaviour model of Vehicle specialization, invoking different methods for Car and Truck. Any event that has different effects on different subtypes will lead to a selection in the event diagram. We show two optional effects on one object type using the open circle notation.

If the inheritance tree is later implemented in the internal data model by delegation (separate tables for super and subtype objects) extra code must be written in the *internal schema* to stitch super and subtype tables together to satisfy a read operation. There is a strong analogy here with the way inheritance works in OO programming.

REFERENCES

Booch, G. (1994) *Object-Oriented Analysis and Design.* Benjamin Cummins.

CCTA (1990) *SSADM Version 4.* NCC Blackwell, Oxford.

Goldberg, A. (1981) *Smalltalk-80, The Language and its Implementation.* Addison Wesley, Reading, MA.

Jackson, M. (1975) *Principles of Program Design.* Academic Press, New York.

Palmer, J. (1993) Anti-hype. *Object Magazine.* May–June.

Robinson, K. (1977) An entity/event data modelling method. *Comp. J.*, **22**.

Robinson, K. (1979) *Infotech Systems Technology.* Infotech International, Maidenhead, Berks.

Robinson, K. and Berrisford, G. (1994) *Object-Oriented SSADM.* Prentice-Hall, Englewood Cliffs, NJ.

Latham, J. T. *et al.* (1990) *The Programming Process.* Addison Wesley, Reading, MA.

Tsvi Bar-David (1994) The elevator. *ROAD*, May–June.

Shlaer, A. (1994) Modeling dynamic behavior. *ROAD*, May–June.

Stefik, M. and Bobrow, D. (1986) OO programming: themes and variation. *AI Magazine*, **6**(4).

Stroustrup, B. (1986) *The C++ Programming Language.* Addison Wesley, Reading, MA.