Proc RE'93, pages 56-64; IEEE CS Press, 1993

Domain Descriptions

Michael Jackson
101 Hamilton Terrace
London NW8 9QX
England

Pamela Zave
AT&T Bell Laboratories
Murray Hill NJ 07974
U S A

Abstract

Specifications should describe the domain explicitly; they should distinguish domain properties that are independent of the system from those that the system is required to enforce. A common semantics based on a simple phenomenology allows composition of partial specifications expressed in different languages. Descriptions should be based on explicit identification of relevant domain phenomena, and so separate assertion from definition of terminology. Explicit structures over descriptions are of interest in their own right. Current formal specification techniques are deficient in some important respects.

1 Introduction

We are working to develop a technique of multi-paradigm specification. Our technique relies on composing partial specifications expressed in different languages, the language for each partial specification being chosen according to the demands of the properties or constraints to be expressed. We believe that this emphasis on composition will offer, as a by-product, insights into incremental development and into the problems of reuse. A general account of our aims and approach is given in [Zave 91].

We concentrate our primary attention on the description of domains (or 'real worlds') and requirements (or 'problems'), because that is where we believe that the need for multi-paradigm techniques is most pressing and least well served. This means paying serious attention to the reality that is being described, rather than focusing chiefly on the syntax and formal semantics of the descriptive language.

From this point of view, we have found that specifications, both formal and informal, are often hard to understand. In the large, it is often hard to determine whether a particular description is describing a property of the system or of its environment. In the small, it is often hard to determine what is being said about the reality described. An informal language may cause difficulty by sheer imprecision of both syntax and semantics. A formal language may cause difficulty in spite of well-defined syntax and semantics: some things can not be said directly in the language and must be obscurely encoded; and a formal semantics may be too abstract to capture meaning effectively.

A central aim of our specification technique is, therefore, to state explicitly what is being described, and it is this aspect of our technique that forms the central theme of this paper. Several examples of multi-paradigm composition may be found in a companion paper [Zave 92].

2 Problem Frames

The usual view of a software development problem recognises two principal parts: the system and the environment, communicating across a transparent noiseless interface. Such a view constitutes a problem frame — a framework for understanding the problem and developing its solution. With this problem frame it is natural to suppose that the content of a specification should be the properties and behaviour of the system that are observable at its interface with the environment. The properties and behaviour of the environment are largely implicit, being essentially those of the system reflected about the interface. To the extent that they differ from this reflection of the system they will demand explicit attention: but this is a secondary concern, and according to [Wing 90] often neglected in formal specifications. At the very least there is often doubt about what is being described. For example [Hayes 87, p144] states that a Z specification gives 'a formal system description', but also that 'the informal text ... can be consulted to find out what aspects of the real world are being described'.

We want to modify the usual practice in several ways. We consider that a careful and explicit description of the environment is an essential first stage in a serious development. This description should capture the behaviour and properties of the environment that are independent of the presence and operation of the system to be constructed. In a library, for example, 'books', 'copies', and 'volumes' stand in certain relationships that must be understood by the developers of a system but can not be changed by them; another independent truth about the library is that a copy of a book, once borrowed, can not be borrowed again until it has been returned.

In place of the traditional term 'environment' we prefer the

term 'domain'. One reason is that an environment is, by implication, secondary: it is merely what surrounds the object of real interest. Since we want it to become a focus of serious attention we prefer to use a different name. Another reason is that 'environment' connotes a tangible, physical surrounding; we think this is too narrow a concept. Domain is a broader concept. The domain is the subject matter of the system's computations, and provides the context in which those computations have useful meaning or effect. A domain may be intangible, such as the texts of Shakespeare's plays for a concordance system; it may be tangible but static, such as the road network for a route planning system; it may be tangible and dynamic, such as a chemical plant for a process control system or the customers of a company for a sales processing system. It is often convenient to divide the domain of a development into several domains or subdomains; we will use the singular term unless we want to emphasise this subdivision.

We regard a domain, then, as a topic for description in its own right, independently of any description we may eventually make of the system to be constructed[1]. Requirements, in our view, are also domain descriptions, but of a special kind. Grammatically, an ordinary domain description is in the indicative mood: it asserts certain truths about the domain. A requirement, on the other hand, while also describing the domain, is in the optative mood. It describes the desired state of affairs that the machine is required to bring about: for example, that no library user has more than 6 books on loan at any one time, and that no reference book is ever borrowed. Clearly, the purpose of a system is to bring about observable effects in the domain: only to this extent does a requirement description describe the system that is to be constructed. In general, it is desirable to separate domain descriptions from requirement descriptions, rather than to mix the optative and indicative moods in one description.

We regard the system itself as consisting of two parts: the machine and the connection. The machine is described by program texts, and realised by a computer — possibly distributed — that accepts those texts and behaves as they describe. But there must also be a connection between the domain and the machine: enough information about the domain must somehow be transferred to and from the machine to allow its computations to have their required effect and meaning. Sometimes this connection will be almost transparent and noiseless. But often it will be complex, forcing a sharp distinction between what happens at the machine end of the connection and what happens at the domain end. The customer for a process control system is interested in material flows and in the operation of valves in the domain; the programmer is interested in the machine end of the RS-232 or IEEE-488 interfaces.

It is important also to recognise that use of a single problem frame, partitioning the substance of the problem into these constituent principal parts, suffices only for simple problems: it is analogous to regarding every program as one sequential process when in reality many programs demand to be understood as sets of communicating processes. For example, the connection may pose a complete development problem in its own right, with its own problem frame; or the behaviour of the machine in one problem frame may furnish the domain for another problem. A tentative attempt to recognise and consider this kind of complexity is presented in [Jackson 90].

In this paper we will draw our examples from the requirements specification phase, not from design or implementation (although we believe that what we have to say is applicable to those phases also). Within the requirements phase all descriptions are descriptions of the domain, differing only in grammatical mood: hence the title of this paper.

3 Phenomenology

To compose descriptions written in different languages we need a common semantic framework within which to ascribe meaning to those descriptions. We do not regard this common semantics as a purely formal matter. Specifications and programs are, indeed, formal mathematical objects. But they are much more than that: they are descriptions of various domains, and it is these domains that are the developer's primary concern. We therefore base our common semantics on a common general account of what is of interest in domains: in short, on a phenomenology. Our common semantics is then a formalisation of this phenomenology.

Each particular specification language and paradigm favours, or even imposes, a particular phenomenology. CSP [Hoare 85], for example, recognises events as the primary domain phenomena, and participation in shared events as the only means of communication. VDM [Jones 90], by contrast, emphasises the phenomena that constitute a domain state: events appear only as operations that cause state changes. In Z [Spivey 88], simple entities may appear as domain phenomena, but must be classified in static and mutually exclusive base types.

For multi-paradigm composition we adopt a phenomenology that is minimalist but sufficient. It is minimalist in the sense that it contains nothing incompatible with the phenomenologies of the particular paradigms we

1 Some approaches to software development [Arango 89, Iscoe 89, Jackson 83] already pay explicit attention to describing the domain separately from any description of the system.

have considered; and sufficient in the sense that it allows us to describe everything that we have so far found to be of importance in domains. The only base phenomena we recognise are individuals, which can be named, and facts about those individuals. Nothing else is built in: individuals are known only by the facts in which they play a role; there is no built-in notion of existence, of distinction between entities and values, of typing, or of the formation of aggregates from parts.

This minimalist phenomenology is directly formalised in first-order untyped predicate logic with equality and recursion[2]. The logic is first-order because we do not regard facts, or classes of fact, as individuals. It is untyped because we regard types as nothing more than ordinary predicates. Equality is essential to the notion of an individual (an individual is equal to itself and to no other individual). And recursion is necessary for many descriptive purposes. Each predicate is either true or false for all instantiations of its arguments by individuals: a truth value is never undefined or unknown.

Our treatment of time is entirely within this minimalist phenomenology and accordingly is formalised in predicate logic. Events and intervals (periods of time during which no event occurs) are individuals of which predicates *event(e)* and *interval(v)* are respectively true. Events and intervals are disjoint:

forAll e • not (event(e) and interval(e))

Time ordering is formalised as a predicate *earlier(x,y)* that gives a total ordering of events and intervals (no two events can occur simultaneously). Events and intervals alternate; the first element in the ordering is an interval, and if there is a last element it, too, is an interval.

An event is associated with the two intervals on either side of it by the predicates *begins(e,v)* and *ends(e,v)*. This association is, as usual, the basis for specifying preconditions and postconditions of events. Regarding time as a total ordering of events and intervals, we can always add to our specification descriptions of further (previously undescribed) events intervening between events that were previously consecutive. But we can still, when we wish, refer within a description to the locally next event — local in the sense that it is drawn from the particular subset of events that is the topic of the description.

Facts are changed only by events: their truth values are constant within intervals. A fact that varies over time is formalised as a predicate with at least one argument to be instantiated by an interval individual. So *customer(c,v)* may mean that individual *c* is a customer in interval *v*, while *customer(c)* would mean that *c* is a customer in every interval. The type constraint on the second argument of the first predicate would be stated explicitly:

forAll c,v • customer(c,v) => interval(v)

Events[3] and intervals are first class individuals. They can be named, and play roles in facts just like other individuals. For example, *startRaining(e)* may mean that event *e* is an event in which it starts to rain, and *raining(v)* may mean that it is raining in interval *v*; again, the type constraints would be explicitly stated. These two predicates might be associated by the formula:

forAll e,v • (startRaining(e) and begins(e,v)
=>*raining(v))*

A more elaborate example of a predicate involving an event individual is *sell(e,s,g,b,p)*, which may mean that in event *e* an individual *s* sells some goods *g* to a buyer *b* at price *p*.

Structures are induced by facts that associate two or more individuals. If a player's hand in a card game consists of the Jacks, Queens, and Kings of Hearts and Spades, it may be regarded as a set of cards:

cardSet(MyHand) and *memberOf(MyHand,HeartJack) and memberOf(MyHand,HeartQueen) ...*

as three pairs:

threePairs(MyHand) and *pairIn(MyHand,P1) and pairIn(MyHand,P2) and pairIn(myHand,P3) and arePair(P1,HeartJack,SpadeJack) and ...*

or as two runs:

twoRuns(MyHand) and
runIn(MyHand,R1) and runIn(MyHand,R2) and areRun(R1,HeartJack,HeartQueen,HeartKing) ...

The predicates *cardSet(x)*, *threePairs(x)*, and *twoRuns(x)* may be regarded as types; but there is no built-in type structure and, as the example shows, the types need not be disjoint.

Temporal structures over events and intervals are induced in exactly the same way, by the predicates *begins(e,v)*, *ends(e,v)*, *earlier(x,y)*, and any other relevant predicates. For example, an 'On-Off Pair' might be defined as an episode (a structure over events) in which the first event is an *on*, the last event is an *off*, and there is no intervening *on* or *off* event:

forAll e,f,p • onOffPair(p,e,f) <=>
(firstInEpisode(p,e) and lastInEpisode(p,f) and event(e)
and event(f) and earlier(e,f) and on(e) and off(f)
and (not forSome g • (earlier(e,g) and earlier(g,f)
and event(g) and (on(g) or off(g)))))

An individual instantiating the argument *p* in this predicate is a temporal structure — an onOffPair — in exactly the same way as MyHand is a structure — a cardSet — over playing

2 The work reported in [Niskier 89] is an earlier attempt to accomplish similar goals.

3 Peter Ladkin has pointed out to us that our view of events is very similar to the view of the philosopher Donald Davidson (see [Davidson 91]).

cards. Generally, our treatment of time is on all fours with our treatment of any other structure over a totally ordered collection of elements. The companion paper [Zave 92] presents time as an instance of a 'marked sequence': that is, a totally ordered bipartite set in which items and inter-item markers alternate. Other possible treatments (for example, permitting simultaneous event occurrences or allowing events to overlap or to be composed of other events) can be similarly accommodated within the phenomenology and its expression in predicate logic.

4 Identifications and Description Scope

Our basic tool for ensuring that we know what our descriptions are about is an identification. An identification consists of a number of rules. Each rule has a Left Hand Side, which is a definition of a domain phenomenon, and a Right Hand Side, which is a predicate or the proper name of an individual, by which a description may refer to the domain phenomenon. Here is an identification[4] with five rules:

(x *is a human being* <=> HUMAN(x);
w *is a literary work* <=> WORK(w);
p*wrote the work* w <=> authorOf(p,w);
the play "Hamlet, Prince of Denmark" <=> hamlet;
the William Shakespeare who lived in Stratford <=> ws)

An identification is somewhat like an interpretation in logic. But its motivation is in the contrary direction: its aim is not to give meaning to an existing description, but to make unambiguous description possible.

The LHSs of the rules are informal because the domain is informal. A LHS must allow the defined phenomenon to be unambiguously recognised in the domain, within the tolerances that flow inevitably from the informality. If 'Stratford' might reasonably (but wrongly) be taken to refer to Stratford in Ontario instead of to Stratford-on-Avon, the fifth rule must be amended accordingly; but we can not expect to define the predicates *HUMAN(x)*, *WORK(w)*, and *authorOf(p,w)* precisely enough to eliminate doubt in every case. This informality should not be taken as an excuse for vagueness in identification rule LHSs. In particular, care must be taken in identifying the individuals involved in facts. The rule:

w *is a word in the play Hamlet* <=> hamletWord(w)

is unacceptably vague, leaving us in doubt whether 'To be or not to be' contains four or six individuals *w* of which *hamletWord(w)* is true.

We may use the identification with a description:

[HUMAN, WORK]
authorOf : HUMAN <—> WORK
ws : HUMAN
hamlet : WORK
---
ws authorOf hamlet

The common semantics ascribed to the description language (in this case Z) associates language elements with the RHSs of the identification rules, and permits translation of the description into predicate logic. The meaning in the domain is that 'p wrote the work w' can be true only if p is a human being and w is a literary work; that the play "Hamlet, Prince of Denmark" is a literary work; that the William Shakespeare who lived in Stratford is a human being; and that the William Shakespeare who lived in Stratford wrote the work "Hamlet, Prince of Denmark".

The phenomena named in the RHSs of an identification constitute the scope of a description using that identification. A description may use more than one identification, and its scope is then the union of the phenomena they name. Domain phenomena not in scope can not be referred to, and nothing can be asserted about them. The identification above allows assertions that Shakespeare wrote other works, that no-one else wrote Hamlet, that other people wrote works, and so on; but it does not allow anything to be asserted about performances of Hamlet, or about its text. Since we are always concerned with partial specifications we are always dealing, not surprisingly, with partial identifications.

Assertions in descriptions are falsifiable. If the description is a domain description, the assertion is falsifiable from knowledge or examination of the domain; if it is a requirement description, the assertion is falsifiable by appeal to the customer, who may deny the requirement.

The whole weight of assertion is borne by the description. An identification rule defining a predicate asserts only that the truth of facts denoted by the predicate is recognisable in the domain. A rule defining a proper name of an individual asserts nothing beyond the recognisability of the individual. In particular, a rule never asserts type information, regardless of the wording chosen for the LHS. If the penultimate rule had been:

the literary work "Hamlet, Prince of Denmark"
<=> hamlet

it would still have been necessary for the description to assert explicitly that 'hamlet' is a literary work.

Conversely, the description bears none of the weight of

4 The use of upper and lower case letters in the examples is arbitrary; it is intended to match a common usage in Z, where the name of a base type is written in upper case.

identification: the domain phenomena must be recognisable from the definitions given in the identification rule LHSs. This places a salutary obligation on the specifier to choose to identify the most directly recognisable phenomena as the basis for the specification. Recognition of these phenomena then allows the user of the specification to orient it correctly with respect to the real domain, just as the user of a map orients it with respect to the territory by finding a few unambiguously identifiable features.

5 Assertion and Definition

A description, translated into the common semantics of predicate logic, uses names of predicates and individuals. In general, only some of these will be names of domain phenomena in scope: understanding the intended usage of the other, new, names is a central concern in understanding the description.

A description may define the meaning of a new name that can then be used in other descriptions. Some languages provide special syntax for definition, but many do not; distinguishing definition from assertion may depend on examining the description scope. Consider, for example, the following DFSA description of a button in a lift (the description applies to every button, describing a general case referred to as button *b*):

States: NoRequest, Request;
Transitions: (Press: NoRequest—>Request),
(Press: Request—>Request),
(Service: Request —>NoRequest);

The button is initially in the NoRequest state; a Press event takes it to the Request state, in which further Press events cause no state change; a Service event in the Request state causes a return to the NoRequest state.

If the event predicates *Press(e,b)* and *Service(e,b)* are in scope, but state predicates *NoRequest(b,v)* and *Request(b,v)* are not, then the description makes assertions about the events, and uses the events to define the meanings of the states. Its only falsifiable assertion is then that in the totally ordered set of all Press and Service events each Service event is immediately preceded by at least one Press. *Request(b,v)* is defined to be true of every interval *v* in which the most recent event of the set was a Press, and *NoRequest(b,v)* is defined to be true of every other interval.

If, by contrast, the state predicates are also in scope, then the description makes three additional, falsifiable, assertions: that *NoRequest(b,VI)* is true of the initial interval *VI*; that exactly one of *Request(b,v)* and *NoRequest(b,v)* is true of every interval *v*; and that *Request(b,v)* is true of an interval *v* if and only if the most recent event was a Press. The identification rules for *Request(b,v)* and *NoRequest(b,v)* show how to recognise these phenomena (for example, a light may be lit when *Request(b,v)* is true), and hence how the assertions might be falsified.

Such a distinction between assertion and definition is vital for composing partial specifications. Each term must be defined once, and only once (we can think of an identification rule as defining the term given in its RHS). But we must be able to make many assertions about the phenomena denoted by those terms, and to make them in many separate descriptions. This is why typing is not built into our phenomenology or its formalisation: having asserted in one description that MyHand is a cardSet, we must be free to assert in another description that it is also threePairs and in yet another that it is twoRuns. Strong typing is a programming idea, justifiable in programming because the constraint on expressive power, though severe, is an acceptable price to pay for static detection of certain errors. In domain description strong typing is unacceptable as a universal discipline: often an individual must be described in many different ways, at one time or at different times in its life.

Terminology newly defined in one description can be made available to other descriptions by a renaming mechanism similar to an identification. In a renaming the LHS of each rule is not informal: it contains the newly defined predicate or proper name. The RHS contains the same predicate or proper name, possibly after renaming and changing the order or number of arguments. A renam ing of the newly defined terminology for the lift button might be:

Request(b,v) <=> demanding(v,b);
NoRequest(b,v) <=> quiescent(v,b);

The scope of a description using this renaming will then include the predicates *demanding(v,b)* and *quiescent(v,b)*, in addition to any others that may be in scope.

The ability to define and use new terminology freely is particularly useful in dealing with informal or complex domains. A traditional difficulty in informal domains is the ambiguity of many commonly used terms: in ten departments the word 'customer' is used in twenty different senses. The solution to this difficulty is to accept that twenty different terms are needed, and to build each different definition on the basis of the identification of the most directly recognisable domain phenomena; these phenomena might, for example, include events such as paying money, or placing an order, or receiving goods. The terms so defined can then be used without restriction in further assertions, exactly as if they denoted directly recognisable domain phenomena. Successive definitions made in this way need not interfere with one another: different terminology can coexist in different descriptions.

In complex domains an important use of definition is classification of individuals, especially of events. The DFSA description of the lift button can be modified to define another event predicate:

forAll e,b • effectivePress(e,b) <=>
(Press(e,b) and
forSome v • (ends(e,v) and NoRequest(b,v)))

A further description in which *effectivePress(e,b)* is in scope can then make assertions about just that subclass of Press events, uncluttered by the definition and by consideration of other Press events.

Consider a library in which books may be borrowed in the usual way. A predicate *borrow(e,m,b)* is defined in an identification rule: it is true of an event *e* in which member *m* borrows book *b*. Early partial specifications in the development sequence may be concerned with the predicate *borrow(e,m,b)*, describing the temporal ordering of borrow and return events, and restrictions on the number of books simultaneously borrowed by a member.

Books may also be borrowed against prior reservation; and a later partial specification, without disturbing the earlier specification, must describe the treatment of reservations. This will involve relationships among events including a subclass of borrow events: the predicate *borrowReserve(e,m,b,r)* is true of an event in which a book is borrowed against a reservation:

forAll e,m,b,r • borrowReserve(e,m,b,r) <=>
(borrow(e,m,b) and
forSome v • (reserved(r,m,b,v) and ends(e,v)))

Furthermore, books from other libraries may be borrowed through an inter-library sharing scheme: the predicate *borrowShare(e,m,b)* is true of an event in which the borrowed book belongs to another library from which it is currently held under the sharing scheme:

forAll e,m,b,s • borrowShare(e,m,b,s) <=>
(borrow(e,m,b) and
forSome s,v •
(belongs(b,s) and heldShare(s,b,v) and ends(e,v)))

Assertions about phenomena in scope introduce no new phenomena: they merely make falsifiable claims about the phenomena already recognisable. Defininitions of new names also introduce no new phenomena: the meaning of a new name is expressed in terms of phenomena already in scope. Freedom from implementation bias is guaranteed by careful observation of this restriction: we regard implementation bias [Jones 90] simply as the introduction into a domain description of a phenomenon that does not belong to the domain.

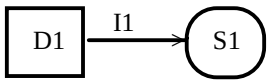
6 Description Graphs

In discussing development problems we have often found it convenient to make the dependencies of descriptions and domains explicit in a graphical form. Description graphs are directed graphs. Each node represents a domain or description; each arc represents an identification or a renaming or a combination of an identification and a renaming. An arc points from the node where the LHS names or phenomena are found to a node where their RHS names are in scope and may be used in assertions and definitions. A domain node has no incoming arcs; every description node must have at least one incoming arc; a description node has outgoing arcs if and only if it defines new names.

In this section we give some simple graph configurations corresponding to trivial domains and descriptions. For brevity, the treatment is highly simplified, and some details are omitted.

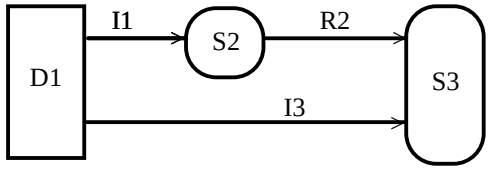
In graph (a) D1 is a domain in which projects have costs, managers, workers, and start and end dates. The identification I1 defines predicates *start(p,d)* and *end(p,d)*. The description S1 asserts that the start date for a project is earlier than the end date.

(a)



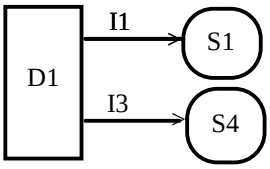
In graph (b) D1 and I1 are as in graph (a). S2 defines a predicate *short(p)*: a project is short if its end date is no more than 5 days after its start date. I3 defines the predicate *manage(p,x)*. S3 asserts that a short project has only one manager.

(b)



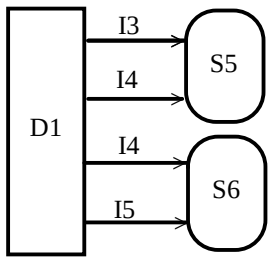
Graph (c) shows conjunction of assertions about one domain. D1, I1, S1, and I3 are as before; S4 asserts that each project has at least one manager. This conjunction is totally trivial: the scopes of the assertions S1 and S4 do not intersect, so the assertions are orthogonal.

(c)



A non-trivial conjunction is shown in graph (d):

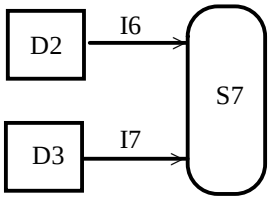
(d)



Identification I3 defines the predicate *manage(p,x)* as before; I4 defines *cost(p,c)*; I5 defines *work(p,x)*. S5 asserts that a project with cost below $500,000 has at most two managers; S6 asserts that a project with cost above $100,000 has at least three workers. (The constant dollar values have been omitted from the identifications.)

Relationships between two subdomains are expressed by descriptions for which phenomena of both subdomains are in scope:

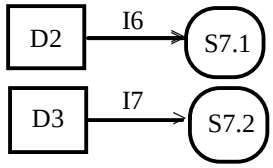
(e)



Domain D2 is the input stream to a simple program, with transaction records containing integers. Domain D3 is the output stream, with a total record containing an integer. Identification I6 defines the predicate *trans(r,i)*; identification I7 defines the predicate *total(r,i)*. Description S7 asserts that the integer in the output total record is the sum of those in the input transaction records.

Of course, this configuration expresses a relationship between the subdomains only if the description S7 is truly one description and not two: that is, if it is not decomposable into two descriptions S7.1 and S7.2 with disjoint scopes, as in the configuration:

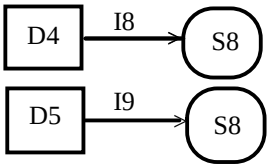
(f)



Here S7 asserts that the integers in the transaction records are all non-negative (S7.1), and so is the integer in the total record (S7.2): it asserts no relationship between D2 and D3.

An important relationship between two domains is the modelling relationship. We use the term modelling in a symmetrical sense: two domains are models of one another if there is some description that, with appropriate identifications, is true of both:

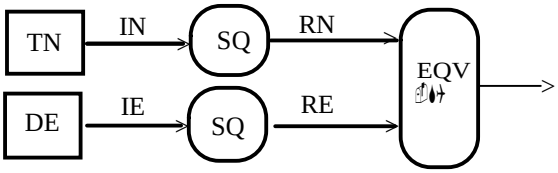
(g)



This modelling relationship is exploited in the JSP method of program design [Jackson 75]: the description of a message stream processed by a program is used as a description of the dynamic behaviour of the program.

An illuminating example of modelling arises in describing the dialling of telephone numbers[5].

(h)



TN is the telephone number domain. The identification IN defines predicates *str(n)* (*n* is a telephone number), *item(i,n)* (*i* is a character instance in the telephone number *n*), *precedes(i,j)* (*i* precedes *j* in a telephone number), and *charval(i,c)* (character instance *i* has the character value *c*). DE is the domain of dialling activity at a telephone. The identification IE defines the same terminology as IN, but based on phenomena of the domain DE: *str(p)* (*p* is a dialling episode at one telephone), *item(e,p)* (*e* is a 'dial' event in the dialling episode *p*), *precedes(e,f)* (event *e* precedes *f* in temporal order), and *charval(e,c)* (the dial event *e* is associated with the character value *c*).

The description SQ asserts that any individual *n* satisfying *str(n)*, having items *i* satisfying *item(i,n)*, also satisfies the constraints of a sequence: *precedes(i,j)* is a total ordering over the items, and *charval(i,c)* is a total function from items to individual values. All such *n* satisfy a newly defined

5 Dialling is often specified by a highly operational technique, using a string-valued variable that is initialised at the appropriate point and explicitly extended by concatenating each digit as it is dialled. For an example of this treatment, see [Woodcock 88].

predicate *sq(n)* (*n* is a sequence).

RN and RE rename the predicate *sq(n)*, allowing the two sequence individuals to be distinguished. The description EQV-SQ defines equivalence between the individuals, allowing other descriptions to make use of some predicate such as *hasDialled(p,n)*, where *p* is a dialling episode and *n* is a telephone number. The description EQV-SQ may be no more than string comparison; but it may need to define the correspondences between numeric dialled sequences and alphanumeric numbers (286-1841 is CUNningham-1841).

Adopting this description graphing approach, we view reuse as the reuse of description subgraphs. A subgraph is reused essentially by supplying new LHSs for incoming arcs of the subgraph and new RHSs for outgoing arcs; the reused descriptions of the subgraph, of course, are not changed.

7 Understanding Specifications

Emphasising domain meaning as we do, we have found many formal specifications very hard to understand. Partly this is for reasons already mentioned: the confusion of the machine with the domain and the indicative with the optative mood, and the absence of explicit indications of description scope. Use of a single formal specification language is not notably less constraining than use of a single programming language: meaning must be forced into the Procrustean bed, and any limbs of inconvenient length are lopped or stretched to fit. Writers of formal specifications are easily trapped by strong typing and by a desire for terse expression into surprising statements such as "A telephone subscriber is a sequence of digits". Like Assembler Language program texts, formal specification texts reveal little of their intent. An implementor who ignores the informal commentary on a formal specification is in no better position to proceed unaided than one who ignores the formal texts themselves.

From a certain point of view, this is not a criticism. Formal methods focus on exact expression of certain system properties that without their help might have been left vague; their users are content to leave other properties implicit or vaguely expressed, judging them to be less important or less in need of clarification. Our view is that as specifiers we can and should do better, especially if we have aspirations to achieve more effective reuse or to provide mechanised support for software development.

This vague relationship of a formal specification to its domain impedes detection of implementation bias. Essentially, implementation bias can not be usefully discussed without explicit description scope. Lamport, for example, defends his transition axiom method [Lamport 89] against the charge that it introduces internal system states when it should be concerned only with externally observable system behaviour. Our view is that if a requirement mentions either states or events they must be states or events of the domain. If, then, the states are not in scope, they must be defined on the basis of domain phenomena that are in scope; if, on the other hand, they are in scope, then their appearances in descriptions make assertions about the domain, and these assertions are falsifiable.

Questions of implementation bias often arise in data structure representation. A specification that declares

```
course : seq of SUBJECT
...
---------------------
...
```

invites the charge of bias: should *course* not be a set rather than a sequence? Again, the issue is easily settled. Explicit identification requires the specifier to say what is the ordering predicate of the sequence, and that predicate must be defined in terms of phenomena of the domain. The ordering of SUBJECTs within a course might be a time-ordering (the subjects are taught consecutively), an ordering by content (a certain SUBJECT must be mastered before some other can be studied), or any other ordering that might be found in the domain. But if no such ordering can be found, the sequence structure is merely a gratuitous invention of the specifier.

Confusion often surrounds the recognition of individuals. Partly, this arises from the common practice of using the same data structure types for domain individuals and for notational convenience in situations where no individual is intended. Consider, for example, the declaration

busy : set of PHONE

The name busy may denote a predicate in scope, an individual in scope, or neither. If it denotes a predicate in scope — busy(x) — we may understand the declaration to be an assertion:

forAll x •. busy(x) => PHONE(x)

If it denotes an individual in scope — busy — we may understand the declaration as an assertion about a membership predicate that relates this individual to certain others:

forAll x • isMemberOf(x,busy) => PHONE(x)

The predicate *isMemberOf(x,busy)* must also be in scope. The assertion that all members of the set *busy* are *PHONE*s is then, as it should be, falsifiable by examination of the domain.

If the name busy denotes no phenomenon in scope, then it is a new predicate name, and must be defined in the containing description on the basis of phenomena that are in scope. The declaration of *busy* is then an undertaking by the specifier that the predicate *busy(x)* will be so defined that:

forAll x •. busy(x) => PHONE(x)

By contrast, the declaration

hamlet : set of WORD

would be assumed by an informed reader to denote an individual, namely the play of that name. A predicate must also be in scope — *isMemberOf(x,hamlet)*, and, again, the assertion that all members of the set *hamlet* are *WORD*s is then, as it should be, falsifiable by examination of the domain.

It is hardly satisfactory that to understand specifications of telephony and of Shakespeare's plays we must rely on our prior knowledge of exactly those subjects.

Sometimes confusion about individuals goes a little deeper. Explaining a temporal logic specification of a bounded buffer, [Wing 90] says

> "For each message *m* currently placed on the input channel and for each previously placed message *m'* ..., *m* and *m'* are not equal. This property is ... an assumption of the environment. Without it, a buffer that outputs duplicate copies of its input would be considered correct."

It is not clear what this environment assumption means or by what behaviour it would be invalidated. The buffer does not require that the character string of each message be unique; nor does it require the writer to supply a unique message-id. So it should surely treat a message currently placed and a message previously placed as *ipso facto* not equal. The assumption, if there is one, is a technical assumption intended to repair a specification defect; the appropriate inference is that the buffer should have been specified differently.

Event individuals present a number of difficulties. Some of these are simply examples of the failure to distinguish domain from machine and indicative domain descriptions from optative requirement descriptions. For example, when Z [Spivey 90] or VDM [Jones 90] is used to specify an operation, nothing indicates whether the operation is to be performed on the initiative of an agent in the domain or on the initiative of the machine. (In contrast, the Transition Axiom Method deals with this point quite explicitly [Lamport 89].) If the operation is to be performed on the initiative of the domain, nothing in the Z or VDM formal text indicates whether the precondition is in the indicative mood (the domain will not cause this operation to occur except when the precondition is true) or in the optative (the machine must inhibit this operation unless the precondition is true).

Other difficulties center on the individuality of events and their classification. For example, if a Z specification defines two event types E1 and E2, and also uses schema composition to give a schema

$$E3 \triangleq E1 >> E2$$

or schema piping to give a schema

$$\triangleq$$

it is not clear in each case whether an occurrence of the state change denoted by the more elaborate schema is to be regarded as one event or as two, nor whether the specifier, by defining E3, has thereby extinguished the possibility of an event described by E1 or by E2 occurring in isolation.

8 Conclusion

This paper has introduced some ideas about the meaning of descriptions, aimed at clarifying the relationship between a formal specification and the domain of the system to be specified. Understanding of specifications must rest on explicit statements of what they are about and what they assert. We believe that current formal specification techniques are inadequate in this respect, and therefore can not offer a satisfactory foundation for automated support of software development.

The companion paper [Zave 92] shows that these same ideas provide a basis for the composition of partial specifications. They enable us to translate specifications in a wide variety of languages into First Order Logic: conjunction serves as the fundamental composition operator, without introducing spurious inconsistency or spurious independence.

We also believe that our approach offers a possible basis for effective reuse of descriptions generally and of partial specifications in particular.

9 References

[Arango 89] G Arango; Domain Analysis: from Art Form to Engineering Discipline. In Proceedings of the 5th International Workshop on Software Specification and Design; IEEE Computer Society Press, 1989.

[Davidson 91] Donald Davidson; Essays on Actions and Events; Oxford University Press (paperback), 1991.

[Guttag 85] J V Guttag, J J Horning, and J M Wing; The Larch Family of Specification Languages; IEEE Software Volume 2 Number 5 September 1985.

[Hayes 87] Ian Hayes ed; Specification Case Studies; Prentice- Hall International, 1987.

[Hoare 85] C A R Hoare; Communicating Sequential Processes; Prentice-Hall International, 1985.

[Iscoe 89] N Iscoe, Ed; Proceedings of the Workshop on Domain Analysis; IEEE Computer Society Press, 1989.

[Jackson 75] M A Jackson; Principles of Program Design; Academic Press, 1975.

[Jackson 83] M A Jackson; System Development; Prentice-Hall International, 1983.

[Jackson 90] M A Jackson; Some Complexities in Computer-Based Systems and Their Implications for System Development. In Proceedings of CompEuro90; IEEE Computer Society Press, 1990.

[Jones 90] Cliff Jones; Systematic Software Development Using VDM; 2nd Edition; Prentice-Hall International,

1990.
[Lamport 89] Leslie Lamport; A Simple Approach to Specifying Concurrent Systems; Communications of the ACM Volume 32 Number 1 January 1989.
[Niskier 89] Celso Niskier, Tom Maibaum and Daniel Schwabe; A Look Through PRISMA: Towards Pluralistic Knowledge-based Environments for Software Specification Acquisition. In Proceedings of the 5th International Workshop on Software Specification and Design; IEEE Computer Society Press, 1989.
[Spivey 88] J M Spivey; Introducing Z: A Specification Language and its Formal Semantics; Cambridge University Press, 1988.
[Spivey 90] J M Spivey; Specifying a Real-Time Kernel; IEEE Software Volume 7 Number 5 September 1990.
[Wing 90] Jeannette M Wing; A Specifier's Introduction to Formal Methods; IEEE Computer Volume 23 Number 9 September 1990.
[Woodcock 88] Jim Woodcock and Martin Loomes; Software Engineering Mathematics: Formal Methods Demystified; Pitman, 1988.
[Zave 91] Pamela Zave and Michael Jackson; Composition of Descriptions: A Progress Report. In Proceedings of the Formal Methods Workshop '91; Springer Verlag, to appear.
[Zave 92] Pamela Zave and Michael Jackson; Conjunction as Composition; in preparation.