SPECIFICATIONS,

PROGRAMS

AND

IMPLEMENTATIONS

BY

C.A.R. HOARE

Technical Monograph PRG-29

June 1982

Oxford University Computing Laboratory
Programming Research Group
45 Banbury Road
Oxford OX2 6PE

Programming Research Group
Oxford University Computing Laboratory
45 Banbury Road
Oxford OX2 6PE
United Kingdom

Summary

A specification is a predicate describing all observations permitted of the system specified. Specifications of complex systems can be constructed from specifications of their components by connectives defined in the predicate calculus. A program is just a predicate expressed using only a restricted subset of such connectives, codified as a programming language. An implementation of the programming language is a mechanism that will accept any predicate of the language, and then behave as described by it. Given a proposed model of an implementation it is desirable to prove that every program expressible in the language is consistent and complete with respect to the model; furthermore, there should be no program which logically implies all the others. These points are illustrated by the design of a very simple programming language, describing the interactions of concurrent processes. It is suggested that the design of a realistic programming language requires, and is worthy of, the skills of a methematical logician.

Contents

SPECIFICATIONS, PROGRAMS AND IMPLEMENTATIONS

1. Introduction

A *variable* in a formula of applied mathematics stands for some directly or indirectly observable value. The correspondence between the variable and the observation must be established informally by some such phrase as "Let x stand for the position (in metres) of a body at time t (measured in seconds), and let v stand for its velocity (in metres per second)".
"Let *coin* stand for the number of coins inserted into a vending machine up to a certain moment, and let *choc* stand for the number of chocolates it has dispensed".

The *alphabet* of a system is the set of variables denoting those observations which are of current interest. The selection of a useful and relevant alphabet of observations is one of the primary characteristics of a successful scientific theory. A *specification* of a system S is defined by its alphabet αS together with a predicate, usually containing variables from αS, which describes all possible observations which may be made of the system. An observation *ascribes* a value to each variable in the alphabet; a specification *describes* the observation if the predicate evaluates to true when each variable is replaced by its ascribed value. We use S itself to stand for the predicate.

Example 1.

$$\alpha LINEAR \triangleq \{x, t, v\}$$

$$LINEAR \triangleq 1 \leqslant t \leqslant 4 \implies x = v \times t - 3 \;\&\; v = 3$$

LINEAR specifies a body that moves at a constant speed of 3 metres per second between time 1 and time 4. Some observations described by this specification are tabulated below:

x	t	v
0	1	3
3	2	3
6	3	3
12	0	17
14	5	12
0	5	0

The first three lines make the consequant of the specification true, and the last three make the antecedent false. When t is outside the specified range, the specification is indeterminate: it specifies nothing at all about the values of x and v.

Example 2.

$$PROFIT \triangleq (choc \leqslant coin)$$

$$BUFFER1 \triangleq (coin \leqslant choc+1)$$

$$VM1 \triangleq PROFIT \;\&\; BUFFER1$$

$$\alpha VM1 = \{coin, choc\}$$

The predicate PROFIT specifies that a vending machine shall be profitable in the sense that it never dispenses more chocolates than there have been coins inserted. BUFFER1 states that the machine will buffer only one coin, so it is impossible to insert two coins and then extract two chocolates. Some possible observations are:

coin	choc
0	0
1	0
1	1
13	12

By a bold abstraction, we now say that a system is fully defined by the strongest specification which describes its every possible behaviour. Thus two systems are regarded as the same if their alphabets are the same, and their specifications are logically *equivalent*. This is reasonable, for then there can be no observation of one of them which is not also a possible observation of the other, and by the principle of the identity of indiscernibles, they should be regarded as the same.

Let S be an arbitrary specification, and let A be the strongest specification of some actual system, and suppose that A logically implies S. This means that every observation described by A is also described by S. Thus we can claim that A is a correct implementation of the specification S. The suggestion of this and the previous paragraph is that questions of equivalence and correctness of systems can be treated within the traditional framework of mathematical logic without requiring any specialised axioms or proof rules. This suggestion is explored in the next section by defining predicates which describe the behaviour of systems composed from interacting concurrent processes.

2. Interacting Processes

We are interested in systems which engage in certain observable events, such as the insertion of coins into a vending machine, or extraction of chocolates. For each event individually, it is possible to record how many times that event has occurred up to any given moment. The alphabet of a process contains variables standing for these counts. For example, the alphabet of a vending machine may be declared as

$\alpha VM1 \triangleq \{coin, choc\}$

2.1 No action

Let A be the alphabet $\{a,b,c,\ ...\}$

We define the predicate

$$ZERO_A \triangleq (a = b = c = \ldots = 0)$$

$$\alpha ZERO_A \triangleq A$$

This is the specification of a rather useless system, which never does anything at all, so that its event counts remain forever zero.

Example $VMO \triangleq ZERO_{\{coin, choc\}}$

This describes the behaviour of a broken vending machine. Its alphabet indicates that it is equipped with the physical organs for accepting coins and dispensing chocolates, but its predicate states that it never uses them.

2.2 Arbitrary action

We define the predicate

$$CHAOS_A \triangleq true$$

$$\alpha CHAOS_A \triangleq A$$

This is a specification which places no constraint whatsoever on the behaviour of the specified system. Every system is correct in accordance with this specification.

Example

$$VMBR \triangleq CHAOS_{\{coin, choc\}}$$

This machine is even more badly broken than VMO; it accepts coins and dispenses chocolates with gay abandon.

2.3 First action

Let $a \in A$

Let $P(a)$ be a predicate with alphabet A

$(a;P(a)) \triangleq ZERO_A \ \vee \ a > 0 \ \& \ P\ (a-1)$

This specifies a system which first engages in the event counted by the variable "a", and then behaves as specified by P. On the first observation, all the counts are zero; on all subsequent observations, the count of "a" is positive; furthermore, on reducing the count of "a" by one, we get an observation described by P.

Example

Recall that $VM1 \triangleq (choc \leq coin \leq choc + 1)$

$$
\begin{aligned}
\therefore\ (coin;\ choc;\ VM1) \Leftrightarrow\ & (coin = choc = 0 \\
& \vee\ coin > 0\ \&\ (coin - 1 = choc = 0 \\
& \qquad \vee\ choc > 0\ \&\ choc - 1 \leq coin - 1 \leq choc)) \\
\Leftrightarrow\ & VM1
\end{aligned}
$$

where we allow ; to be right associative.

Thus VM1 specifies a machine that first accepts a coin, then dispenses a chocolate, after which it behaves again like VM1. Such a machine will alternately accept coins and dispense chocolates for as long as there is any call upon it to do so.

2.4 Recursion

Let P be the name of a predicate, and let F(P) be an expression denoting a predicate, and possibly containing occurrences of P. Furthermore, let

$$\alpha P = \alpha F(P)$$

Thus the equation

$$P \triangleq F(P)$$

may be regarded as a recursive definition of the predicate named by P. Problems of existence and uniqueness of this solution are postponed to section 4.

Example

$$P \triangleq (\text{coin};\ \text{choc};\ P)$$

$$\alpha P \triangleq \{\text{coin},\ \text{choc}\}$$

We already know that a possible solution of this equation is:

$$P = VM1 = (\text{choc} \leqslant \text{coin} \leqslant \text{choc} + 1)$$

In fact, this is the only solution, as will be shown in section 4.4.

2.5 Alternatives

Let P and Q be specifications, with

$$\alpha P = \alpha Q$$

We stipulate that $\alpha(P \vee Q) = \alpha P$.

$(P \vee Q)$ specifies a system which will behave like P or like Q (or like both). The choice between the alternatives is not determined; it may be made by the environment within which this system is embedded, as will be described in 2.6.

Example 1.

$$P \triangleq (\text{choc};\ \text{coin};\ P \vee \text{coin};\ \text{choc};\ P)$$

This describes a vending machine which allows its customer to sample a chocolate, and trusts him to pay after. The solution to the defining equation is:

$$P = (\text{choc} \leqslant \text{count}+1 \leqslant \text{choc} + 2)$$

Not surprisingly, the customer should pay for the privilege of using such a machine, as in the next example.

Example 2.

$$VM2 \triangleq \text{coin};\ P$$

$$\Longleftrightarrow \text{choc} \leqslant \text{coin} \leqslant \text{choc} + 2$$

This machine allows its customer to insert up to two coins before extracting up to two chocolates.

Example 3.

$$\begin{aligned} CUST \triangleq (&\text{choc};\ \text{rejoice};\ CUST \\ &\vee \text{coin};\ \text{choc};\ CUST \\ &\vee \text{kisswife};\ CUST) \end{aligned}$$

This describes the behaviour of a customer for the vending machine. He is, of course, very willing to take a chocolate without paying, after which he will greatly rejoice. He is also willing to kiss his wife, and whether he does so depends on whether she too is willing. He is also willing to pay for his chocolate in the normal way. The unique solution of the equation is

$$CUST = (choc \leqslant coin + rejoice + 1 \leqslant choc + 2 \ \& \ rejoice \leqslant choc)$$

Note that the specification of the customer does not prevent him from engaging in the following sequences of actions, since in each case, it correctly describes the values of the counts before and after each event of the sequence:

coin, kisswife, choc, ...
choc, rejoice, coin, coin, choc

The reason for these possibly unexpected sequences is that our specification language is too weak to describe such constraints as:

"he can't kiss his wife between inserting a coin and extracting a chocolate".

"he never inserts two coins in a row".

The weakness of the specification language is a deliberate decision that we do not wish to observe the exact relative timing of events; thus we allow events to occur "simultaneously", in the sense that any attempt to place them in order will lead to a non-determinate result. Some further consequences of this decision will become more apparent later.

2.6 Concurrency

Let P and Q be specifications with disjoint alphabets, i.e.,

$$\alpha P \cap \alpha Q = \{\}$$

we then stipulate that

$$\alpha(P \ \& \ Q) = \alpha P \cup \alpha Q.$$

(P & Q) describes a system composed from the two subsystems separately described by P and Q, as they evolve together. Each event that occurs is in the alphabet of only one of the components; and its occurrence requires participation of that component, leaving the other component unaffected. Each observation of the complete system splits uniquely into two parts; the values of variables in αP are described by P and are not mentioned by Q; the value of variables in αQ are described by Q and are not mentioned in P. Thus the full observation is exactly described by (P&Q).

Now let us relax the restriction of disjointness of alphabets. We stipulate that each event in the intersection of the two alphabets requires simultaneous participation of <u>both</u> subsystems described by P and Q. Thus the count of events recorded by one subsystem must always be the same as that recorded for the other subsystem; so each possible value of this count must be described by both P and Q, and therefore by (P&Q).

Example 1.

$VM1 \triangleq PROFIT \;\&\; BUFFER1$

A simple vending machine can be constructed from two concurrent subsystems, one of which is profitable and the other refuses to allow insertion of a second coin until the first chocolate has been dispensed. Each event that occurs is an event in the life of both subsystems.

Example 2.

$$
\begin{aligned}
VM2 \;\&\; CUST \iff\; & choc \leq coin \leq choc + 2 \\
& \& \; choc \leq coin + rejoice + 1 \leq choc + 2 \\
& \& \; rejoice \leq choc \\
\iff\; & rejoice = 0 \;\&\; choc \leq coin \leq choc + 1 \\
\vee\; & rejoice = 1 \leq choc = coin
\end{aligned}
$$

In this system, the customer can still kiss his wife at any time - that is of no concern to the vending machine. The customer never allows the number of coins inserted to reach (choc + 2) and the vending machine never allows the number of chocs to exceed the number of coins inserted. It may therefore seem surprising that the customer should ever have cause to rejoice. This can happen when the events coin and choc occur simultaneously; the vending machine behaves as if the coin dropped first; but the customer thinks that choc was dispensed first and rejoices prematurely.

If we wish to exclude such strange happenings, we shall have to strengthen our language for specifications, or use a more complex composition operator than & ; these topics will be pursued in later **sections.**

3. Example: the Dining Philosophers

In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. The names of the philosophers were $Phil_0$, $Phil_1$, $Phil_2$, $Phil_3$, $Phil_4$ and they were disposed in this order anticlockwise round the table. To the left of each philosopher there was laid a golden fork, and in the centre a large bowl of spaghetti, which was constantly replenished.

A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went to the dining room, sat down in his own chair, picked up his own fork on his left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second fork is required to carry it to the mouth. The philosopher therefore had also to pick up the fork on his right. When he was finished he would put down both his forks, get up from his chair, and continue thinking. Of course, a fork can be used by only one philosopher at a time. If the other philosopher wants it, he just has to wait until the fork is available again.

3.1 Alphabets

We shall now construct a mathematical model of this system. First we must select the relevant sets of events. For $Phil_i$, the set is defined:

$$\alpha Phil_i = \{ i \text{ sits down}, i \text{ gets up}, \\ i \text{ picks up fork } i, i \text{ picks up fork } (i \oplus 1), \\ i \text{ puts down fork } i, i \text{ puts down fork } (i \oplus 1) \}$$

where $\oplus$ is addition modulo 5.

Note that the alphabets of the philosophers are mutually disjoint. There is no event in which they can agree to participate jointly, so there is no way whatsoever in which they can interact or communicate with each other – a realistic reflection of the behaviour of philosophers of those days.

The other actors in our little drama are the five forks, each of which bears the same number as the philosopher who owns it. A fork is picked up and put down either by this philosopher, or by his neighbour on the other side. Its alphabet is defined

$$\alpha \text{fork}_i \triangleq \{\text{i picks up fork i, } (i \ominus 1) \text{ picks up fork i,}\\ \text{i puts down fork i, } (i \ominus 1) \text{ puts down fork i}\}$$

where $\ominus$ denotes subtraction modulo 5.

Thus each event except sitting down and getting up requires participation of exactly two adjacent actors, a philosopher and a fork.

3.2 Behaviour

Apart from thinking and eating which we have chosen to ignore, the life of each philosopher is described:

$$\begin{aligned}\text{Phil}_i \triangleq\ &(\text{i sitsdown; i picksup fork i; i picks up fork } (i \oplus 1);\\ &\text{i puts down fork i; i puts down fork } (i \oplus 1);\ \text{i gets up;}\\ &\text{Phil}_i)\\ \Longleftrightarrow\ &(\text{i sitsdown} \succcurlyeq \text{i picks up fork i} \succcurlyeq\\ &\text{i picks up fork } (i \oplus 1) \succcurlyeq \text{i puts down fork i} \succcurlyeq\\ &\text{i puts down fork } (i \oplus 1) \succcurlyeq \text{i gets up} \succcurlyeq\\ &(\text{i sits down} - 1))\end{aligned}$$

The rôle of a fork is a simple one; it is repeatedly picked up and put down by one of its adjacent philosophers:

$$\begin{aligned}\text{Fork}_i \triangleq\ &(\text{i picks up fork i; i puts down fork i; Fork}_i\\ &\vee\ (i \ominus 1) \text{ picks up fork i; } (i \ominus 1) \text{ puts down fork i; Fork}_i)\end{aligned}$$

$$\begin{aligned}\Longleftrightarrow\ &(i \ominus 1) \text{ picks up fork i} = (i \ominus 1) \text{ puts down fork i}\\ \&\ &\text{i picks up fork i} \succcurlyeq \text{i puts down fork i} \succcurlyeq ((\text{i picks up fork i}) - 1)\\ &\vee\ \text{i picks up fork i} = \text{i puts down fork i}\\ \&\ &i \ominus 1 \text{ picks up fork i} \succcurlyeq (i \ominus 1) \text{ puts down fork i} \succcurlyeq (((i \ominus 1) \text{ puts}\\ &\qquad \text{down fork i}) - 1)\end{aligned}$$

The behaviour of the entire college of philosophers is simply the conjunction of the behaviour of these components:

$$\text{College} \triangleq (\mathop{\&}_{i=0}^{4} \text{Phil}_i)\ \&\ (\mathop{\&}_{i=0}^{4} \text{Fork}_i)$$

3.3 Deadlock!

When this mathematical model had been constructed, it revealed a serious danger. Suppose all the philosophers get hungry at about the same time; they all sit down; they all pick up their own forks; and they all reach out for the other fork - which isn't there. In this undignified situation, they will all assuredly starve. Although each actor is capable of further action, there is no action which any pair of them can agree to do next.

A possible observation of this sad outcome is

$$\mathop{\&}_{i=0}^{4} \text{ i sitsdown } = \text{ i picks up fork i } = 137$$

$$\& \quad \mathop{\&}_{i=0}^{4} \text{ i puts down fork i } = \text{ i puts down fork } (i \oplus 1)$$
$$= \text{ i picks up fork } (i \oplus 1) = \text{ i gets up } = 136$$

This observation is described by the specification of the College; but there is no way of adding unity to just one of the counts so that this successor observation is also described by the specification. If we are not willing to allow the simultaneous occurrence of several events in the life of a single actor, nothing further can happen.

3.4 Deadlock averted

However, our story does not end so sadly. Once the danger was detected, there were suggested many ways to avert it. For example, one of the philosophers could always pick up the wrong fork first - if only they could have agreed which one it should be! The purchase of a single additional fork was ruled out for similar reasone, whereas the purchase of five more forks was much too expensive.

The solution finally adopted was the appointment of a Footman, whose duty it was to assist each philosopher into and out of his chair. His alphabet was defined:

$$\alpha\text{Footman} = \bigcup_{i=0}^{4} \{\text{i sits down, i gets up}\}$$

This footman was given secret instructions never to allow more than four philosophers to be simultaneously seated:

$$\text{Footman} \triangleq \sum_{i=0}^{4} (\text{i sits down} - \text{i gets up}) \leq 4$$

Let F_j (for $0 \leqslant j \leqslant 4$) denote the behaviour of the footman with j philosophers seated. Then

$$F_4 \triangleq \bigvee_{i=0}^{4} (i \text{ gets up}; F_3)$$

$$(\text{for } j = 1,2,3)\ F_j \triangleq \bigvee_{i=0}^{4} (i \text{ gets up}; F_{j-1} \vee i \text{ sits down}; F_{j+1})$$

$$F_o \triangleq \bigvee_{i=0}^{4} (i \text{ sits down}; F_1)$$

We need to prove that

$$F_j \Longrightarrow \sum_{i}^{4} (i \text{ sits down} - i \text{ gets up}) \leqslant 4-j$$

and hence

$$F_o \Longrightarrow \text{Footman}$$

This establishes that F_o is a correct implementation of the behaviour specified for the Footman.

The edifying tale of the dining philosophers is due to Edsger W. Dijkstra; the Footman is due to Carel Scholten.

4. Mathematical Properties

A major advantage of specifying complex systems in terms of some familiar domain like the predicate calculus is that the operators enjoy a number of elegant mathematical properties. Algebraic properties are those that can be expressed as simple equations, or (in the case of predicates) as equivalances. Ordering properties are those which are expressed as inequalities, or (in the case of predicates) as implications.

4.1 Algebraic properties

(1) "&" and "v" are associative, commutative, and idempotent.

(2) "&", and "v" and "c;" distribute through v.

However, because the operands of "v" must have the same alphabet, it is not generally true that "v" distributes through "&". Nevertheless any predicate which uses only "&" and "v" (and constants ZERO and CHAOS) can be reduced to disjunctive normal form, in which & is the innermost operator.

The "&" operator may then be eliminated, using:

(3) $ZERO_A \ \& \ ZERO_B = ZERO_{A \cup B}$

$CHAOS_A \ \& \ CHAOS_B = CHAOS_{A \cup B}$

$ZERO_A \ \& \ CHAOS_B = ZERO_{A \cup B}$

We would now like to develop a normal form for predicates containing the prefixing operator "c;". Since prefixing distributes through "v", it is necessary only to show how it distributes through "&".

(4) $(a;P) \ \& \ CHAOS_A = a;(P \ \& \ CHAOS_A)$

$(a;P) \ \& \ ZERO_A = a;(P \ \& \ ZERO_A) \text{ if } a \tilde{\in} A$

$= ZERO_A \text{ otherwise}$

(5) $(a;P) \ \& \ (b;Q) \iff b;((a;P) \ \& \ Q) \quad \text{if } \begin{cases} a \in \alpha Q \\ b \tilde{\in} \alpha P \end{cases}$

$\iff a;(P \ \& \ (b;Q)) \quad \text{if } \begin{cases} a \tilde{\in} \alpha Q \\ b \in \alpha P \end{cases}$

$\iff a;(P \ \& \ (b;Q)) \vee b;((a;P) \ \& \ Q) \quad \text{if } \begin{cases} a \tilde{\in} \alpha Q \\ b \tilde{\in} \alpha P \end{cases}$

$\iff a;(P \ \& \ Q) \quad \text{if } a \in (\alpha P \cap \alpha Q) \text{ and } a = b$

This theorem permits prefixing to be moved outside "&" in all cases except for $(a;P) \ \& \ (b;Q)$ when $a,b \in \alpha P \cap \alpha Q$ and $(a \neq b)$. In this case each operand is attempting an action which requires simultaneous oarticipation of the other, but they disagree on which action it shall ba. As a result, one might expect that nothing can happen, and the conjunction should reduce to ZERO. But this fails to take into account the possibility of simultaneous occurrence of events; for example:

$(a;b;ZERO) \ \& \ (b;a;ZERO)$

$\iff (a = b = 0 \vee a = b = 1)$

If we wish to eliminate this possibility, we must introduce the restriction that the alphabets of the two operands of "&" may have at most one event in common.

Under this restriction, every expression can be reduced to a normal form in which "v" is the outermost operator, and the "&" operator has been eliminated.

4.2 Ordering properties

The set of all predicates is partially ordered by the relation of logical implication, which we write " $\Longrightarrow$ "; this is shown by the following familiar metatheorems:

$$P \Longrightarrow P$$

$$\text{if} \quad P \Longrightarrow Q \quad \text{and} \quad Q \Longrightarrow P \qquad \text{then} \quad P \Longleftrightarrow Q$$

$$\text{if} \quad P \Longrightarrow Q \quad \text{and} \quad Q \Longrightarrow R \qquad \text{then} \quad P \Longrightarrow R \ .$$

A function F is said to be <u>monotonic</u> if it "respects" the ordering of its operand, i.e.

$$\text{If} \quad P \Longrightarrow Q \quad \text{then} \quad F(P) \Longrightarrow F(Q) \quad \text{for all } P,Q.$$

The definition extends to functions with many operands, if they are monotonic in each of their operands separately, e.g.,

$$\text{If} \quad P \Longrightarrow Q \quad \text{then} \quad F(P,R) \Longrightarrow F(Q,R)$$
$$\text{and } F(R,P) \Longrightarrow F(R,Q) \quad \text{for all } P,Q,R.$$

Any operator that distributes through "v" is monotonic, so all the operands we have defined so far enjoy this property. Furthermore any composition of monotonic operators is also monotonic, so any function or expression defined in terms of these operators will be monotonic in all its operands.

There is a good reason why operators used in recursive definitions should be monotonic - it ensures the <u>existence</u> of a predicate satisfying each recursive definition. This assurance is given by the Tarski-Knaster theorem, provided we accept the assumption that the space of predicates is <u>complete</u>. A partial ordering is complete if <u>every</u> set S of predicates has an infinite conjunction $\forall S$, such that (for any Q):

$$Q \Longrightarrow P \quad \text{for all } P \text{ in } S$$

$$\text{if and only if} \qquad Q \Longrightarrow \forall S.$$

The limit points $\forall S$ are related to ordinary predicates in much the same way as the reals are to the rationals. They may be uncountable in number, but that should not be taken as a reason for denying their existence.

If there is more than one solution for a recursion equation, it is necessary to decide which one is meant. The usual technique here is to take the weakest solution, i.e. the disjunction of all possible solutions. The fact that this is itself a solution is also assured by the Tarski-Knaster theorem.

4.3 Continuity

Let $S = \left\{ S_n \mid n \geqslant 0 \right\}$ be a countable set of predicates such that

$$S_{n+1} \Longrightarrow S_n \qquad \text{for all } n.$$

Such an S is known as a chain, and its limit $\forall S$ is written

$$\forall n \geqslant 0.\ S_n$$

If F is a monotonic function of predicates, and S is a chain, then

$$\left\{ F(S_n) \mid n \geqslant 0 \right\}$$

is also a chain, and has as limit

$$\forall n.\ F(S_n)\ .$$

Because F is monotonic, we have the implication:

$$F(\forall n.\ S_n) \Longrightarrow \forall n.\ F(S_n) \qquad \text{for all chains } S.$$

If this implication can be strengthened to equivalence, then F is said to be continuous.

A function of several predicates is continuous if it is continuous in each argument separately.

It is easy to see that the operators "c;", "&" and "v" are continuous. It follows that every function of predicates defined in terms of these operators is also continuous. The main advantages of continuity will emerge later; but here we note that continuity of a function F greatly simplifies the search for a solution to a recursion equation of the form:

$$P = F(P) .$$

Define $F^0(\text{true}) = \text{true}$

$$F^{n+1}(\text{true}) = F(F^n(\text{true}))$$

$$= \underbrace{F(F(\dots F}_{n+1 \text{ times.}}(\text{true})))$$

Now the weakest solution of the equation is

$$\forall n.\ F^n(\text{true})$$

Proof. (1) that it is a fixed point:

$$F(\forall n\ F^n(\text{true})) = \forall n(F^{n+1}(\text{true})) \quad \text{by continuity}$$

$$= \text{true} \ \& \ \forall n.F^n(\text{true})$$

$$= \forall n.F^n(\text{true})$$

(2) that it is the weakest fixed point:

Let $P = F(P)$

$\therefore P = F^n(P)$ by induction

$\therefore P = \forall n\ F^n(P)$

$\Longrightarrow \forall n\ F^n(\text{true})$ by monotonicity

Examples

(1) $P \triangleq I(P)$ where I is the identity function.

$I^n(\text{true}) \Longleftrightarrow \text{true}$

$\therefore P \Longleftrightarrow \forall n.\ I^n(\text{true}) \Longleftrightarrow \text{true}$

(2) $P \triangleq (P \vee Q)$

$F^n(\text{true}) \Longleftrightarrow \text{true} \vee Q \vee Q \vee \dots \vee Q$

$\Longleftrightarrow \text{true}$

$\therefore P \Longleftrightarrow \text{true}$

(3) $P \triangleq P \,\&\, Q$

$F^n(\mathrm{true}) \longleftrightarrow Q$

$\therefore\ P = Q$

(4) $P = a;P$ where $\alpha P = \{a,b\}$

$F^1(\mathrm{true}) = (a = b = 0 \vee a > 0)$

$F^2(\mathrm{true}) = (a = b = 0 \vee a = 1 \,\&\, b = 0 \vee a > 1)$

$F^n(\mathrm{true}) = (b = 0 \vee a \geqslant n)$

$\therefore\ P \longleftrightarrow \forall n.(b = 0 \vee a \geqslant n)$

$\Longleftrightarrow (b = 0)$

As shown by these examplee, the explicit solution of each equation raquires an induction to recast $F^n(\mathrm{true})$ as a predicate explicitly containing n as a free variable.

4.4 Unique solutions

In all our examples of sections two and three, the solutions of the recursive equations have been unique. It is useful to recognise cases of unique solution, because they are simpler to reason about. Consider the equation

$$P = F(P).$$

F(P) is said to be <u>guarded</u> if it can be written in the form

$$a;G(P) \vee b;H(P) \vee \ldots\ldots \qquad \ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$

where G(P), H(P) are expressed using only the notations "c;" and "v". If F(P) is guarded, then the equation above has an unique solution.

Proof. An expression is said to be guarded to depth n+1 if each of G(P), H(P), ... in formula (1) is guarded to depth n. Using distributivity of "c;" such formula can be written

$$a;b;\ \ldots;\ G'(P) \vee c;d;\ \ldots;\ H'(P) \vee \ldots$$

where the length of each of the prefix sequences "c;d;...;" is greater than n.

If $F(P)$ is guarded, then $F^n(P)$ is guarded to depth n. For any observation, we define its <u>length</u> as the sum of all the values ascribed to its variables; thus the length is a count of the total number of events that have occurred, and is necessarily finite. If an observation is of length n, and a predicate $F^n(P)$ is guarded to depth n, then one can determine whether the predicate describes the observation by merely looking at the prefix sequences, independent of the value of P.

Now let

$$P = F(P) \quad \text{and} \quad Q = F(Q)$$

$$\therefore \quad P = F^n(P) \quad \text{and} \quad Q = F^n(Q) \quad \text{by induction.}$$

Consider an observation described by P. Let its length be n. This observation is also described by $F^n(P)$ and therefore by $F^n(Q)$, which has all the same prefixes of length n. Thus all observations described by P are also described by Q, and vice versa. P and Q are therefore equivalent as predicates.

5. <u>Programming</u>

As discussed in the introduction, a specification is an arbitrary predicate describing all possible observations of some system. The task of the scientist is to <u>discover</u> the strongest specification describing some aspect of the behaviour of the natural universe. The task of the engineer is different: he has to <u>construct</u> some mechanism which meets a specification describing the needs of a potential user of the mechanism. The engineer's duties would be much simplified if he had at his disposal a stock of <u>universal</u> mechanisms. An universal mechanism is one that will first accept the text of any desired specification, and will then automatically transform itself into a mechanism which behaves in accordance with that specification, for as long as it is wanted. Such a marvellous mechanism might be a bit expensive; and if so, it should have a switch to turn it back into a universal mechanism when there is no need for it to continue to satisfy its current specification.

For the civil engineer or naval architect, an universal mechanism of this kind is nothing but a pipe-dream. But for the computer programmer, it is a reality which he takes for granted - it is the stored program digital computer. The only problem is that the computer will not accept an arbitrary predicate as a specification; the predicate must be written in a highly restricted notation known as a programming language. Such predicates are known as programs. In the remainder of this paper we shall define a program as a predicate expressed solely in terms of the constants and operators introduced in section 2 together with recursion. Later, we shall consider some further restrictions on this notation.

5.1 Programming methodology

The task of the programmer is now clear. First, he must use his best judgement to formulate a specification of the desired product. The specification should be expressed as a predicate, taking advantage of the full power of the concepts and notations of logic and mathematics to keep the formalisation simple and clear. Clarity is of the utmost concern, since a misunderstanding at this stage can have a severe impact on the quality of the product. The programmer now has to reformulate the specification as a program P, expressed in the restricted notations of his programming language; and this may involve a gross expansion in the size of the text. Furthermore, he must find a program that is adequately efficient when executed on a mechanism of affordable capacity and speed. His task may be slightly simplified by the fact that it is sufficient to find a program P that merely implies its specification S; there is no need to achieve exact equivalence.

Thus the task of programming is rather like that of finding an explicit definition of a function which satisfies given differential equations. Just as some equations have no explicit solutions, some predicates cannot be programmed because they are incomputable or even inconsistent. For soluble equations, mathematicians have discovered many techniques for finding and checking proposed solutions, though their application usually demands some mathematical skill and insight. A collection of methods for constructing a program to meet an arbitrary specification is known as a programming methodology.

5.2 Top-down development

One rather obvious programming method is the technique of "top-down development" or "divide-and-conquer"; it is rather similar to integration by parts, in that its injudicious use can require solution of subproblems more difficult than the original problem. Let S be the specification of the desired program. Then

(1) Let $F(T,U)$ be an expression containing the predicates T and U, but otherwise expressed wholly within the programming language.

(2) Prove that $F(T,U) \Longrightarrow S$.

(3) Find programs P and Q such that

$$P \Longrightarrow T$$

and $$Q \Longrightarrow U.$$

(4) The result you want is

$$F(P,Q).$$

The validity of this method depends on the fact that all operators of the programming language are monotonic. Its utility derives from the fact that F is proved correct before P and Q are programmed.

5.3 Introduction of recursion

One of the most significant tasks of the programmer is to construct correct loops or recursions. Suppose S is the specification which is believed to require a recursively defined program. Then the following steps are recommended:

(1) Find an expression E which maps the observations of the alphabet onto non-negative integers. This is known as a "variant function".

(2) Find a program $F(P)$, containing the predicate name P, such that

$$F(S \vee E \geqslant n) \Longrightarrow (S \vee E > n)$$

where n is a fresh variable.

Thus if the recursive call P establishes S in all circumstances when $E \leqslant n$, then $F(P)$ is <u>better</u> than P, in that it ensures S in the case when $E = n+1$ as well.

(3) Then the program you want is defined recursively:

$$p \triangleq F(p)$$

Proof. From step 2, by an easy induction

$$F^n(\underline{true}) \Longleftrightarrow F^n(S \vee E \geqslant 0) \Longrightarrow (S \vee E > n) \text{ for all } n$$

$$\therefore \quad (\forall n. F^n(\underline{true})) \Longrightarrow (S \quad \vee \forall n. E > n)$$

$$\Longleftrightarrow S$$

Since F is expressed wholly in notations which are known to be continuous, $\forall n.F^n(\text{true})$ is the weakest solution of

$$p \triangleq F(p)$$

6. Implementation

The main motivetion for writing specifications in a restrictive programming language is the existence of a universal mechanism that will automatically implement the program. In practice, such an implementation is constructed from silicon chips, boards, wires, etc., together with loaders, compilers, or interpreters for the given programming language. But for our present purposes, it is more convenient to construct an abstract mathematical machine, passing through a series of states; each state corresponds to a possible observation described by the program initially given to the machine. Here is an informal description of the behaviour of such a machine, intended to model the actions of an interacting system:

(0) Input the specification with given alphabet.

(1) Declare an integer variable corresponding to each event in the alphabet. Initialise all these variables to zero.

(2) Repeat the following steps as long as possible (or until switched off):

(2.1) For each variable in turn, add one to its value, test if the specification is true, and subtract one again.

(2.2) For all variables which have passed test (2.1), wait until the user/environment of the mechanism has selected which of the events is to occur. If no variable has passed the test, this wait will last forever.

(2.3) Add one to the variable selected in step 2.2.

We assume that an observation of this machine can be made just before each iteration of the loop (2); at that time, the current values of all its variables are observable.

6.1 Consistency

Clearly, this implementation does not correctly implement every predicate; in fact the predicate "a > 3" is false immediately after step (1). However, any predicate which is true after step (1) will remain true forever - that is the purpose of the tests of step (2.1). So, to show that the implementation correctly implements every <u>program</u>, it suffices to show that every program describes the observation when all the counts are zero, i.e., that

$$ZERO_A \Longrightarrow P \qquad \text{for all programs } P \text{ with alphabet } A.$$

This can be proved by structural induction on P, using the following lemmas.

(1) $ZERO_A \Longrightarrow ZERO_A$

(2) $ZERO_A \Longrightarrow CHAOS_A$

(3) $ZERO_A \Longrightarrow (a;P)$

(4) If $ZERO_A \Longrightarrow P$ and $ZERO_A \Longrightarrow Q$

then $ZERO_A \Longrightarrow P \vee Q$

(5) If $ZERO_A \Longrightarrow P$ and $ZERO_B \Longrightarrow Q$

then $ZERO_{A \cup B} \Longrightarrow P \ \& \ Q$

(6) If for all n $ZERO_A \Longrightarrow P_n$

then $ZERO_A \Longrightarrow \forall n . P_n$

The last clause is required to show that all programs defined by recursion are implied by ZERO; this conclusion depends also on continuity of all the connectives of the programming language, a property which has already been established in 4.3.

Thus we have shown that the proposed implementation works correctly for all programs expressed the language. This result may be reformulated: we have shown that the programming language is <u>consistent</u> with the given model implementation, in the sense that every observation produced by the implementation will be correctly described by its program.

6.2 Completeness

Suppose that a set I of one or more correct implementations has been proposed for a given programming language. Let P be a program submitted to these implementations. Now it may be that there is an observation described by P which can never in fact occur on any of the implementations. Thus the predicate P is not the <u>strongest</u> possible predicate describing the behaviour of its proposed implementations. In this case, we say the programming language is <u>incomplete</u> with respect to I, since it is not capable of expressing every true fact about I. Incompleteness is a serious fault, because it means that some correct programs cannot be proved correct solely in terms of their definition as predicates; their proof would require operational reasoning based on the implementations.

Unfortunately, our programming language is not complete with respect to the implementation described above. In that implementation, each observation except the first (all zeroes) has a <u>predecessor</u> which can be derived from it by subtracting unity from just one of its component variables. Consider the specification

$$\text{coin} = \text{choc}.$$

This is satisfied by the observation

$$\text{coin} = \text{choc} = 5$$

but not by the observation

$$\text{coin} = 4 \mathbin{\&} \text{choc} = 5$$

or by the only other predecessor observation

$$\text{coin} = 5 \mathbin{\&} \text{choc} = 4.$$

Thus the specification describes an observation which can never be reached by the intended implementation.

This would not matter if such a specification could never be expressed as a program. Unfortunately it can. Consider a silly customer of the vending machine

$$\text{SILLY} \triangleq \text{choc};\text{coin};\text{SILLY}$$

$$\therefore\ \text{SILLY} \mathbin{\&} \text{VM1} \Longleftrightarrow (\text{coin} = \text{choc})$$

One solution to the problem would be to expand the set of implementations to allow any number of events to occur simultaneously. Another solution is to restrict the programming language still further, so that no specification expressible as a program will describe an unreachable observation.

Let us attempt a solution of the second kind. A predicate is said to be <u>grounded</u> if every described observation has a predecessor also described by it. Each observation therefore has a chain of predecessors reaching back to the initial zero observation. The intended implementation can follow this chain in the reverse direction, and can thus reach any observation described by a grounded predicate. So we need to restrict our programming language in such a way that it can express only grounded predicates.

An effective restriction is the one introduced in section 4.1 to obtain a normal form: forbid the use of the operator "&" except when the alphabets of its operands contain at most one event in common. The fact that all programs satisfying this restriction are grounded is proved by structural induction, based on the following lemmas:

(1) $ZERO_A$ and $CHAOS_A$ are grounded.

(2) If P and Q are grounded, so are $(a;P)$ and $(P \vee Q)$.

(3) Furthermore, if size $(\alpha P \wedge \alpha Q) \leqslant 1$ then $(P \& Q)$ is grounded.

(4) If for all n P_n is grounded and $P_{n+1} \Longrightarrow P_n$ then $\forall n . P_n$ is grounded.

Only the last clause requires any subtlety of proof. Consider an observation described by $\forall n . P_n$. The maximum possible number of its predecessors is finite (equal to the size of the alphabet). Since P_{n+1} logically implies P_n, the set of predecessors (of the given observation) described by P_{n+1} is a subset of those described by P_n. These finite subsets form a descending chain, whose intersection is therefore non empty. Furthermore this intersection is described by $\forall n . P_n$.

6.3 The excluded miracle

In section 3.1 we defined the task of the programmer as the discovery of a program P which logically implies a given specification S.

Now suppose he could find a program P that logically implies every other program Q. This P would be a miraculous program, since it could be used to implement <u>every</u> implamentable specification. With such a program, he would never need any other; and each of his tasks would be trivial. An example of such a miraculous program would be one that expresses the predicate <u>false</u>.

In practice, we suspect that a programming language in which there exists a miraculous program would be unrealistic or useless in some other way; it would certainly be unworthy of serious mathematical study. Unfortunately, our little programming language (as proudly proved in section 6.1) contains the miracle $ZERO_A$. So we need to exclude this constant from the language; but we also need to ensure that it can never be expressed in some other way, for example:

$$\begin{aligned} & (choc;SILLY) \ \& \ (coin;VM1) \\ \Longleftrightarrow\ & (coin = choc = 0 \vee choc > 0 \ \& \ coin \leqslant choc-1 \leqslant coin+1) \\ \& \ & (coin = choc = 0 \vee coin > 0 \ \& \ choc \leqslant coin-1 \leqslant choc+1) \\ \Longleftrightarrow\ & (coin = choc = 0) \end{aligned}$$

This problem can be solved by imposing the same restriction on the alphabets of the operands of & as has already been recommended in 4.1 and 6.2. Then no program is equal to the miracle $ZERO_A$. A proof of this uses structural induction based on the lemmas.

(1) $CHAOS_A \neq ZERO_A$

(2) If P and Q are programs distinct from $ZERO_A$ then so are (a;P) and $(P \vee Q)$, and $(P \,\&\, Q)$ provided size $(\alpha P \cap \alpha Q) \leqslant 1$.

(3) If for all n $P_n \neq ZERO_A$ and $P_{n+1} \Longrightarrow P_n$ then $(\forall n.P_n) \neq ZERO_A$.

The proof of the last clause uses the fact that an observation has only a finite number of possible successors.

Now we need to prove that there are no other miracles besides $ZERO_A$, which we have already excluded. Unfortunately there is. If the alphabet A contains only one event, then all expressible programs are equivelent to $CHAOS_A$. We must therefore insist that every alphabet A contains at least two variables, say "a" and "b". Then there are at least two different programs with alphabet A:

$$P \triangleq a;P \iff b = 0$$

$$Q \triangleq b;Q \iff a = 0$$

Furthermore the only observation they both describe is

$$(a = b = 0) \iff ZERO_A$$

But since we have shown that no program implies $ZERO_A$, no program can imply both P and Q. Thus there is no program which implies every other program.

6.4 Computability

The universal mechanism described above requires the ability to determine the truth or falsity of a predicate for an arbitrary value of its free variables. It is well known that for some predicates this is impossible. We must therefore ensure that our programming language can express only predicates which are in some sense computable.

In general, the appropriate sense of computability seems to be that the complement of the predicate should be recursive enumerable. Thus if an observation falsifies the predicate, it will be possible to prove that it does so. Such predicates are said to be falsifiable. According to Karl Popper, falsifiability is also a required property of all scientific theories. In accordance with this definition all our programs are falsifiable:

(1) $ZERO_A$ and $CHAOS_A$ are falsifiable.

(2) If P and Q are falsifiable, then so are $(a;P)$, $(P \vee Q)$ and $(P \& Q)$.

(3) If for all n, P_n is falsifiable, then so is $\forall n.P_n$.

Another very desirable property of a general purpose programming language is that its programs should be able to compute every computable function. The usual method of proving this is to program in the language a simulation of some known universal machine such as a Turing machine. But our language cannot do this; in fact the language can be implemented on a finite state machine. In a special purpose language, such a limitation may be an advantage, if it permits a mechanical check against certain undesirable occurrences such as non-termination or deadlock. A more practical defect of our language is that it cannot even describe the simple examples used to illustrate this paper. To solve these defects we would need to introduce more operators and perhaps variables denoting different kinds of observation.

6.5 Continuity again

The description of an universal implementation of our language essentially regards each complete program as a predicate which can be tested. In practice, programs are implemented in a structured fashion, and the complete implementation of a complex program is constructed from implementations of its parts, for example, a test of $(P \vee Q)$ or $(P \& Q)$ or $(a;P)$ can be constructed from tests of P and Q separately. However, the position with recursively defined predicates is less clear.

Following Scott, we identify an implementation of the predicate

$$P \triangleq F(P)$$

as the infinite sequence of "finite approximations":

$$\left\{F^n(\text{true}) \,\middle|\, n \geqslant 0\right\}$$

such that the desired program P is the universal quantification of the sequence. Now if G is any <u>continuous</u> function of predicates to predicates, an implementation of $G(P)$ can be constructed from the implementation of P, thus

$$\left\{G(F^n(\text{true})) \,\middle|\, n \geqslant 0\right\}.$$

Because G is continuous, this is an infinite sequence of finite approximations to $G(P)$. Thus we have constructed an "implementation" of $G(P)$ out of an implementation of P.

There remains one outstanding question. A programming language defined by the methods described in the previous sections is a notation for specifying <u>falsifiable</u> predicates. However, the set of observations generated by any mechanism must be recursive enumerable. Since the complement of a recursive enumerable set is not necessarily recursive enumerable, there will in general be observations which cannot be generated by a correct implementation, even though they are described by the program which is being implemented. Thus no general-purpose programming language can be complete in the sense of 6.2. The problem is most acute in the case of a non-terminating recursion

$$P \triangleq I(P)$$

According to our definition

$$p \longleftrightarrow CHAOS_A \ .$$

However any "reasonable" implementation is likely to produce only the all-zero observation, so that it will behave as if

$$p \longleftrightarrow ZERO_A \ .$$

Thus <u>all</u> observations (except one) fall into the gap between what is described by the program and what is actually generated by its implementation.

I do not know whether this problem has a solution, or even whether it needs one.

7. Conclusion

The first step in the construction of a mechanism to meet some requirement is to obtain a very good understanding of that requirement. This understanding can be formalised as a predicate describing all possible acceptable observations of the behaviour of the mechanism. Because misunderstanding at this stage is so dangerous, a specification should be short, simple, and well-structured; and to this end it should take full advantage of all available concepts and notations of mathematics and logic. For effective description of certain kinds of dynamic system, it is convenient also to use certain concepts akin to those of computer programs; and this requires they be defined in terms of conventional predicates. Even so, the construction of a successful specification requires the same human skills and insights as are characteristic of an applied mathematician or engineer.

The task of the programmer remains to find some predicate, expressed only in the restricted notations of his programming language, which logically implies the specification. The reason for the restrictions is to enable the program to be run on some available implementation of the language.

This view of programs and their specifications is highly relevant for the designer of a programming language. Firstly, he must have a wide understanding of the application area of his language. Based on this, he should define a range of concepts to assist in the specification of programs within that area. These concepts should enjoy nice algebraic properties, preferably admitting a simple normal form. If recursion or repetition is required, then all the propositional connectives involved should be at least monotonic.

The designer should then define a class of model implementations, which will automatically conform to a subclass of specifications submitted to them. The implementations should be a reasonable abstraction of what can be built, perhaps in silicon, at acceptable cost. This will require some restriction on the generality of the notations used in the specification. These restrictions should not be so severe that they prohibit an exact description of the behaviour of the model implementations; but they should be severe enough to exclude the uniformly false predicate, or any program which logically implies all other programs. Finally, there should be some reasonably easy stepwise method for designing a program from its specification. All these tasks are considerably simplified if all the operators of the programming language are not only monotonic but continuous.

This account of the nature of programming and of programming language design suggests that both activities require and deserve the techniques and skills of the mathematical logician.

Acknowledgements

The original idea of equating a computer program with its strongest specification is due to E.C.R. Hehner. The treatment of Communicating Systems was inspired by the work of A.J.R.G. Milner. The technique of predicate transformers is due to E.W. Dijkstra. The development of specifications as predicates together with their alphabets is due to J.-R. Abrial. The treatment of recursion by fixed points and continuity is due to D.S. Scott. Valuable and continuing mathematical support has been provided by E.-R. Olderog.

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP TECHNICAL MONOGRAPHS

JULY 1982

This is a series of technical monographs on topics in the field of computation. Copies may be obtained from the Programming Research Group, (Technical Monographs), 45 Banbury Road, Oxford, OX2 6PE, England.

PRG-2 Dana Scott
Outline of a Mathematical Theory of Computation

PRG-3 Dana Scott
The Lattice of Flow Diagrams

PRG-5 Dana Scott
Data Types as Lattices

PRG-6 Dana Scott and Christopher Strachey
Toward a Mathematical Semantics for Computer Languages

PRG-7 Dana Scott
Continuous Lattices

PRG-8 Joseph Stoy and Christopher Strachey
OS6 - an Experimental Operating System for a Small Computer

PRG-9 Christopher Strachey and Joseph Stoy
The Text of OSPub

PRG-10 Christopher Strachey
The Varieties of Programming Language

PRG-11 Christopher Strachey and Christopher P. Wadsworth
Continuations: A Mathematical Semantics for Handling Full Jumps

PRG-12 Peter Mosses
The Mathematical Semantics of Algol 60

PRG-13 Robert Milne
The Formal Semantics of Computer Languages and their Implementations

PRG-14 Shan S. Kuo, Michael H. Linck and Sohrab Saadat
A Guide to Communicating Sequential Processes

PRG-15 Joseph Stoy
The Congruence of Two Programming Language Definitions

PRG-16 C. A. R. Hoare, S. D. Brookes and A. W. Roscoe
A Theory of Communicating Sequential Processes

PRG-17 Andrew P. Black
Report on the Programming Notation 3R

PRG-18 Elizabeth Fielding
The Specification of Abstract Mappings and their Implementation as B^+-trees

PRG-19 Dana Scott
Lectures on a Mathematical Theory of Computation

PRG-20 Zhou Chao Chen and C. A. R. Hoare
Partial Correctness of Communicating Processes and Protocols

PRG-21 Bernard Sufrin
Formal Specification of a Display Editor

PRG-22 C. A. R. Hoare
A Model for Communicating Sequential Processes

PRG-23 C. A. R. Hoare
A Calculus of Total Correctness for Communicating Processes

PRG-24 Bernard Sufrin
Reading Formal Specifications

PRG-25 C. B. Jones
Development Methods for Computer Programs including a Notion of Interference

PRG-26 Zhou Chao Chen
The Consistency of the Calculus of Total Correctness for Communicating Processes

PRG-27 C. A. R. Hoare
Programming is an Engineering Profession

PRG-28 John Hughes
Graph Reduction with Super-Combinators

PRG-29 C. A. R. Hoare
Specifications, Programs and Implementations

PRG-30 Alejandro Teruel
Case Studies in Specification. Four Games