Programming and formal topology

Peter Hancock

1st-May-2003

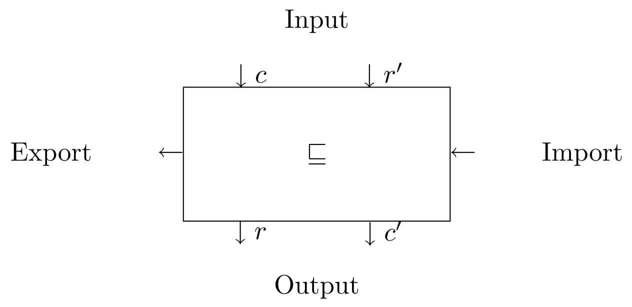
1 Introduction

In practice, what programmers produce are not entire, self-standing programs that run on a "bare" machine, but are rather parts or components of systems. Commonly, this takes the form of a file (for example the text of a program, a '`.o`') file) that can be linked together with two kinds of things:

- implementations of required interfaces, such as the operating system interface or that of a specialized library. These provide services *to* our module.
- client modules, that require services *from* our module.

The linking can be done at various times and in various ways, depending on the details of the interfaces involved. The interface events most often take the form of procedure calls of some kind: system traps, control transfers, calls over a network to remote services. A call is comprised of an ordered pair of communication events: entry and exit, or (to use the word 'call' in another way) call and return.

A suitable icon therefore for 'the' task of a programmer is therefore this:



In this picture the horizontal dimension shows interfaces. The more abstract, high-level exported interface is at the left, and the more concrete, low-level imported interface at the right. The vertical dimension shows communication events, with data flowing from top to bottom: c and r' communicate data from the environment, while r and c' communicate data to the environment. The labels c and r constitute events in the higher level interface, while c' and r' are at the lower level. The pattern of communication is that first there is a call c, then some number of repetitions of parts $c'r'$, then finally a return r. One may think of the c's and the r's as opening and closing brackets (square or round) in a language of balanced strings denoted by the regular expression:

$$(c(c'r')^*r)^*$$

However, the regular expression abstracts completely from the nature of the calls and returns, and the way they depend on the previous history of interactions.

The task of the programmer is to fill the box. We may perhaps imagine a painting like Edvard Munch's "Scream" in the box instead of the scientific-looking symbol $\sqsubseteq$. The latter can be thought of as a reflexive and transitive order between interfaces, meaning the exported interface can be implemented, or emulated given only an implementation of the imported interface.

In order to describe an interface, in the strong sense of a contract between a client and a server, which the server needs to rely upon, it may be necessary to refer to a *state*. The calls that are acceptable in an interface depend on the state of that interface. Moreover the returns that may be made to an outstanding call may depend not only on the state at the time of the call, but also on the value input: the input value typically selects the procedure to be called (or method receives control) and the values passed as via input parameters.

To tie down mathematically the notion of an interface, or more precisely a call-return interface, we define the notions of protocol and interface as follows.

Definition 1 *A protocol (or alternatively, an interaction structure) is given by four things:*

1. *A set S of states.*
2. *A (set valued) function C that assigns a set of acceptable calls $C(s)$ to a given state s.*
3. *A (set valued) function R that assigns to a state s and a call $c \in C(s)$ acceptable in state s a set of permitted returns $R(s, c)$.*
4. *A (state valued) function n that assigns to a state s and a call $c \in C(s)$ acceptable in state s and a return $r \in R(s, c)$ the next state of the interface $n(s, c, r)$.*

An interface consists of a protocol and a given current or initial state $s_0 \in S$.

We could also define a protocol to consist of a set S, and a function which assigns to each state s a (doubly indexed) family of families of states:

$$\{\{s_{cr} \mid r \in R(s, c\} \mid c \in C(s)\}$$

Here s_{cr} is a state determined by first a $c \in C(s)$ and then $r \in R(s, c)$. Instead of substricts, I shall use the 'square-brackets division' notation $s[c/r]$ for the new state determined by first a call $c \in C(s)$ and then a return $r \in R(s, c)$.

A simple example that illustrates a number of points is provided by the interface provided by a memory cell capable of holding a value $s \in S$.

Example 1 *For a memory cell capable of holding a value $s \in S$, we can set $C(s)$ to be the disjoint union $\{Read\} \cup \{Write\, s' | s' \in S\}$, $R(s, c)$ to be defined by cases:*

$$\begin{aligned} R(s, Read) &= \{s\} = \{s' \in S | s' = s\} \\ R(s, Write\, s') &= \{Ack\} \end{aligned}$$

and the next state function to be defined by

$$\begin{array}{ll} n(s, \mathit{Read}, \_\,) & = s \\ n(s, \mathit{Write}\, s', \_\,) & = s' \end{array}$$

The above is an example of a memory protocol; for an example of an interface we add a distinguished initial state $s_0 \in S$.

The main point here is that the protocol requires that we have a *memory* cell, because we require that the output of a *Read* command is, if no *Write*s have occurred, the initial value and else the last written value. To accomplish this, it uses in an essential way the possibility that the set $R(s, c)$ depends on the state s, defining it to be the singleton set $\{s\}$. (The singleton may be defined in typetheory as the set of pairs $\langle s', \_ \rangle$ with $s' \in S$, and $\_$ a proof that the states s and s' are the same according to a given equivalence relation.) Thus we fully constrain the output value to be the right one if the device is to behave as a memory, and not merely some element of S.

Note that we do not require that all elements of S should be reachable in any sense.

We compare protocols and interfaces by means of the relation $\sqsubseteq$, which we call simulation. The intuition is that to show $\mathcal{P} \sqsubseteq \mathcal{P}'$, we have to define a relation between the states of $\mathcal{P}$ and the states of $\mathcal{P}'$ and prove that it satisfies certain closure properties. To show $\mathcal{I} \sqsubseteq \mathcal{I}'$, we have to find such a relation that holds between the two initial states.

Definition 2 *If $\mathcal{P}$ is the protocol $\langle S, C, R, n\rangle$ then we define an operator $\mathcal{P}[.]$ on the powerset of S as follows:*

$$\mathcal{P}[X] = \{s \in S \mid (\exists c \in C(s))(\forall r \in R(s, c))X(n(s, c, r))\}$$

To alleviate notation, we drop the square brackets, and so systematically confuse a protocol $\mathcal{P}$ with the corresponding predicate transformer.

Definition 3 *If A and B are sets, and R is a relation between A and B, ie a function from A into the powerset of B, then there are two predicate transformers $\{R\}$ and $[R]$ both from predicates P over B to predicates over A, defined as follows. We use Sambin's 'overlaps' notation $P \between Q$ to abbreviate the circumstance that two predicates have a joint solution.*

$$\begin{array}{lcl} \{R\}(P) & = & \{a \in A \mid R(a) \between P\} \\ [R](P) & = & \{a \in A \mid R(a) \subseteq P\} \end{array}$$

Definition 4 *If $\mathcal{P}$ and $\mathcal{P}'$ are protocols, which are represented by 4-tuples $\langle S, C, R, n\rangle$ and $\langle S', C', R', n'\rangle$, then we define a simulation of $\mathcal{P}$ by $\mathcal{P}'$ to be a binary relation Q between the sets S and S', taking a state $s \in S$ to a set of states $Q(s) \subseteq S'$ satisfying the following two conditions:*

1. *For all $s \in S$, $s' \in S'$, $c \in C(s)$, if $Q(s, s')$, then there exists $c' \in C'(s')$ such that for all $r' \in R'(s', c')$, there exists $r \in R(s, c)$ such that $Q(n(s, c, r), n'(s', c', r'))$.*

   We can express this condition more concisely with the formula

$$Q(s) \subseteq \mathcal{P}'[\bigcup_{r \in R(s,c)} Q(n(s, c, r))]$$

Here the free variables s and c are implicitly universally quantified. The condition says that $Q(s)$ must not be too big.

2. *For all s, the subset $Q(s)$ of S' is closed under the operation $\mathcal{P}'$. We can express this with the formula*

$$\mathcal{P}'[Q(s)] \subseteq Q(s)$$

Here the free variable s is implicitly universally quantified. The condition says that $Q(s)$ must not be too small. All states must be included which lead to a state in $Q(s)$ by one call and return in the lower level interface.

If $\mathcal{I} = \langle \mathcal{P}, s_0 \rangle$ and $\mathcal{I}' = \langle \mathcal{P}', s_0' \rangle$ are interfaces, we define $\mathcal{I}$ and $\mathcal{I}'$ to mean that there is a simulation Q of $\mathcal{P}$ by $\mathcal{P}'$ such that $Q(s_0, s_0')$.