

Computation over arbitrary models of time

A unified model of discrete, analog, quantum and hybrid computation

Mike Stannett

Verification and Testing Research Group
Dept of Computer Science, Regent Court, University of Sheffield
211 Portobello Street, Sheffield S1 4DP, United Kingdom
Email. m.stannett@dcs.shef.ac.uk

ABSTRACT

This technical report is an updated record of a talk given by the author at the 1998 “X-machines Day” hosted by Sheffield University Computer Science Department. Where appropriate the introductory section has been streamlined, but the underlying content is otherwise unchanged.

We introduce several new variants on Eilenberg's X-machine. We show by direct construction how the first of these, the *operational X-machine*, is related to our 1990 analog X-machine (AXM). We give the first published operational semantics of the AXM, showing how concurrent execution across (possibly infinitely many) simultaneous computation paths is synchronised and co-ordinated. We demonstrate simple one-arrow AXMs for (computable) Lebesgue integration and differentiation of real-valued functions, and observe that the same principles allow us to model the unitary evolution underpinning quantum computation.

We introduce a general model of time, *duration space*, and extend the AXM to a second new model, the Temporal (or Timed) X-Machine (TXM). We argue that the TXM can model every computation compatible with classical or quantum physics as they are currently understood. We demonstrate that every X-machine (and hence that every Turing machine) has TXM-computable behaviour, and that every AXM is automatically a TXM in its own right. We demonstrate TXM models of physical processes which are known not to be Turing-computable. We illustrate a TXM process in which uncountably many distinct computation paths evolve concurrently.

In conclusion we argue that the TXM is, in and of itself, a demonstrably hypercomputational and fully unified model of discrete, analog, quantum, and hybrid computation which goes some way towards allowing the specification of *physics itself* as a parameter in computational models.

Keywords. Theory of computation, hypercomputation, analog computer, hybrid computation, quantum computer, real-time computation, concurrent semantics.

1 Motivation

We begin with a brief summary of Eilenberg's 1974 model of computation, the *X-machine* [EIL74]. This was originally presented in a purely theoretical framework, but re-emerged [HOL88] in 1980s Sheffield as an educationally useful computational model – *X-machines* are capable of encoding Turing machine behaviours, but have the advantage that they cleanly separate the *control* and *processing* aspects of computation, thereby making the underlying processes of computation more intelligible to students and researchers alike.

The ideas we present below are not always in the form presented by Eilenberg, though the concepts are essentially equivalent. In particular, Eilenberg follows standard practice by including the initial and terminal state sets as intrinsic components of an automaton. We do not, because the computational framework we introduce is more succinct if these sets are regarded instead as parameters of the model.

A *state machine* is a triple $(States, Alphabet, Arrows)$ where *States* is a set whose elements we call (*control*) *states*, and *Alphabet* is a non-empty set whose elements we call *letters* or *symbols*. The elements of the set *Arrows*, called *arrows*, are triples of the form (s_0, a, s_1) where s_0, s_1 are states and a is a letter. What we refer to as a state machine is also called a *Labelled Transition System (LTS)*.

If both *States* and *Alphabet* are finite, in which case *Arrows* is also finite, we call the machine a *finite state machine (FSM)* or *automaton*. Automata are often depicted as directed labelled graphs in which each state becomes a node, and each arrow (s_0, a, s_1) an arc $s_0 \xrightarrow{a} s_1$. These arrows can be concatenated in the usual way to form paths labeled by words. We write $s_0 \Rightarrow^{\mathbf{a}} s_n$ to denote such a path from one state s_0 to another (potentially the same) state s_n . Such a path exists in an automaton precisely when there exist n arrows $s_0 \xrightarrow{a} s_1 \xrightarrow{b} \dots \xrightarrow{c} s_n$ with $\mathbf{a} = ab\dots c \in Alphabet^*$. The length of the word \mathbf{a} , written $|\mathbf{a}|$, determines that of the path. In particular, each state is associated with a trivial path $s \Rightarrow^{\lambda} s$ of length 0 (called the *null path* at s), where λ denotes the empty string over *Alphabet*.

Suppose we identify two subsets $I, T \subseteq States$, whose members we call *initial* and *terminal* states, respectively. We will write

$$F(I,T) =_{\text{def}} \{ i \Rightarrow^{\mathbf{a}} t \mid i \in I \text{ and } t \in T \}$$

for the set of all paths in F which lead from initial to terminal states. Any path $|i \Rightarrow^{\mathbf{a}} t|$ in $F(I,T)$ is labelled by the word $\mathbf{a} \in Alphabet^*$ (we write $|i \Rightarrow^{\mathbf{a}} t| = \mathbf{a}$), and this lets us view F as a program schema for constructing languages. The language associated with any particular computation $F(I,T)$ is the set

$$|F(I,T)| =_{\text{def}} \{ |\mathbf{p}| \mid \mathbf{p} \in F(I,T) \}$$

and is called the *behaviour* of F (given these particular choices of I and T). Languages that can be generated in this way are said to be *regular*. Clearly, if $L = |F(I,T)|$ is a

regular language, then L contains λ if and only if $I \cap T$ is non-empty. If this is not the case, so that $\lambda \notin L$, we say that L is λ -free.

The step from finite state machines to X -machines is not a difficult one. We use the term X -machine (where X is some nonempty datatype) to mean a computational machine whose run-time configurations can be represented as members of the datatype X . If the configuration is currently x and we apply some concrete operation ϕ_{concrete} to the system, it will take up some new configuration x_{new} . In general there may be several potential configurations x_{new} because the system's behaviour may be nondeterministic, and we represent the concrete behaviour ϕ_{concrete} by the abstract relation $\phi: X \leftrightarrow X$, where $y \in \phi(x)$ if and only if the concrete operation ϕ_{concrete} can cause the system as represented by x to migrate to the new configuration y . We say that ϕ is an *action* on X , and typically write x^ϕ (or $x\phi$) in place of $\phi(x)$.

The set of actions associated with the machine will be written *Actions*. Each *action* $\in \text{Actions}$ is a relation on X . The set $R(X)$ of all relations $X \leftrightarrow X$ is called the *relational semigroup on X* . It is a monoid (a semigroup with identity element) whose identity is the identity relation 1_X on X , i.e. $1_X(x) = \{x\}$ for each $x \in X$.

Because we are interested in computational behaviours, we follow Eilenberg in assuming that concrete operations are not applied at random, but are subject to some scheme of application whose control mechanism can be represented as a finite state machine. Following Holcombe [Hol88] we call this control mechanism the *associated automaton* of the X -machine. Traditionally the symbols used to decorate arcs in the associated automaton are just the associated action relations, but this description is unsuitable for our purposes. We suppose instead that the associated automaton is defined over some alphabet A of atomic symbols, which are associated with relations on X via some 1-1 labelling $\Lambda: A \rightarrow R(X)$ – relative to this description we have $\text{Actions} = A^\Lambda$. We extend Λ so that it applies to words as well as letters by defining $\lambda^\Lambda = 1_X$, and $\mathbf{a}^\Lambda = (a_1^\Lambda) \dots (a_n^\Lambda)$ for each $\mathbf{a} = a_1 \dots a_n$. We call such a composition of actions a *path relation* in the machine.

When modelling the evolution of the system, the presence of an arrow \rightarrow^a in the control system implies the corresponding capability to apply the operation $\phi = a^\Lambda$. The existence of a path $\Rightarrow^{\mathbf{a}}$ in the machine ($\mathbf{a} = a_1 \dots a_n$) indicates that the application of the operations $a_1^\Lambda, \dots, a_n^\Lambda$ (in that order) is permitted by the control system, or in other words, that the operation \mathbf{a}^Λ can be applied.

For our purposes, then, an X -machine is a pair $M = (F, \Lambda)$, where F is a finite state machine (over some alphabet A) and Λ is a 1-1 mapping from A into $R(X)$. We normally think of Λ as a morphism from finite state machines to X -machines, and write $M = F^\Lambda$. As usual we shall abuse notation, and think of Λ as being defined on subsets as well as points of A . For $S \subseteq A$ we define $\Lambda(S) = \cup \{ \Lambda(s) \mid s \in S \}$.

In general there will be some configurations in which the system can meaningfully be initialised, and others in which a computation can meaningfully be said to have

reached completion. These determine the basic sets I and T for a given run of the machine. Each path $i \Rightarrow^a t$ through the associated automaton corresponds to a possible operation on the system, and the overall operation computed is represented by the composite relation \mathbf{a}^\wedge , provided we interpret relational application as taking place “on the right”. That is, the relation $\phi\psi$ is defined by $x(\phi\psi) \equiv (x\phi)\psi$.

If we initialise the machine to be in initial configuration i , the set of possible terminating configurations is the set $\{\mathbf{a}^\wedge(i) \mid \mathbf{a} \in |F(I,T)|\}$. This is the outcome of a relation in $R(X)$ acting on the state i . The relation in question is called the *behaviour* of the X -machine, written $|M(I,T)|$, and is given by

$$|M(I,T)| = \cup\{ \mathbf{a}^\wedge \mid \mathbf{a} \in |F(I,T)| \}$$

If we drop I and T , and recall our notation $M = F^\wedge$, we can express this rather succinctly as

$$|F^\wedge| = |F|^\wedge$$

because $|M| = \cup\{ \mathbf{a}^\wedge \mid \mathbf{a} \in |F| \} = \cup\{ |\mathbf{p}|^\wedge \mid \mathbf{p} \in F(I,T) \} = \Lambda(\{ |\mathbf{p}| \mid \mathbf{p} \in F(I,T) \}) = \Lambda(|F|)$.

Any X -machine can be equipped with an encoding relation $E: Input \leftrightarrow X$ and a decoding relation $D: X \leftrightarrow Output$ so that computations can be carried out between arbitrary *Input* and *Output* types – if the X -machine normally computes $\phi: X \leftrightarrow X$, then equipping it with E and D allows it to compute the relation $E\phi D$.

Thinking of $|M|$ in terms of the language $L = |F(I,T)|$ reveals the relationship $|M| = L^\wedge$. This relationship is particularly useful if we want to investigate the computational power of X -machines, because it reminds us that Λ is a morphism mapping the set REG of regular languages to X -machine behaviours. Since this morphism is defined for all languages, not just those in REG , we can also investigate the images of other language families, *e.g.* the context-free and computably enumerable language families. However, it is easy to see that these extensions cannot, of themselves, increase the power of X -machines, as this is already unlimited.

For example, any Turing machine can be modelled as an X -machine. We take X to be the Turing machine’s configuration space, and choose our alphabet large enough to represent the finitely many instructions present in the Turing machine’s program. This example also explains why X -machines are so much more powerful than the finite state automata on which they are based – the extra power comes from the arbitrary nature of the relation mapped onto by Λ , which can be as complex as we wish.

Indeed, if we define *ONE* to be the regular language $\{a\}$ whose only string is the single-letter word a , then every set-theoretic relation can be computed by a^\wedge for an appropriate choice of Λ . For suppose $\phi_{\text{concrete}}: Y \leftrightarrow Z$ is a relation between sets Y and Z . Define $X = Y \times Z$, and introduce the encoding and decoding relations $E(y) = \{y\} \times Z$, and $D(y, z) = z$. Let $\phi \in R(X)$ be the relation $(y, z)^\phi \equiv \{y\} \times \phi_{\text{concrete}}(y)$, and take $a^\wedge = \phi$. Then

$$|M| = ONE^\Lambda = a^\Lambda = \phi$$

and for each $y \in Y$,

$$(y)(E\phi D) = (\{y\} \times Z)(\phi D) = (\{y\} \times \phi_{\text{concrete}}(y)) D = \phi_{\text{concrete}}(y)$$

as required.

This result should make us wary, as we clearly need to take special care that the components we use are sensible – we cannot expect to obtain a meaningful theory of computation if we allow arbitrarily complex behaviours to be “hidden” in our choice of Λ . The fact that arbitrarily complex behaviours are possible must mean that the basic X -machine model is not constructive. It allows us to *describe* but not *prescribe*, whence the behavioural descriptions the model gives are purely *observational*.

The observational nature of the model is also apparent if we ask *why* a given path through the machine should have been selected, as we now illustrate. Since we are interested in generating a constructive model, our first task will be to understand the nature of this observability. We will then be in a position to introduce a more operational description of X -machine computations.

1.1 Example of the problem #1

A familiar example may make the observational nature of X -machine semantics clearer. Think of a word-processor as a DOC-machine. The states of the machines indicate what mode we are currently in, for example, EDITING, HELP, SAVING, and so on. The labels on each arc may as well be the characters on a standard keyboard, suitably adapted and interpreted to take account of the state in which they are applied. Thus *Editing_a* might be the operation "insert the character a at the current position".

Given this example, we can ask ourselves two closely related, but subtly distinct, questions:

- What happens if I insert-the-character- a -at-the-current-position?
- What happens if I type- a ?

The *Next-state* relation takes the form "In *state*, doing *action* takes you to potential new *states*", *i.e.* $States \rightarrow (Actions \leftrightarrow States)$. In particular, then, we would expect to see a program rule of the form

- $EDITING \mapsto (Editing_a \mapsto EDITING)$

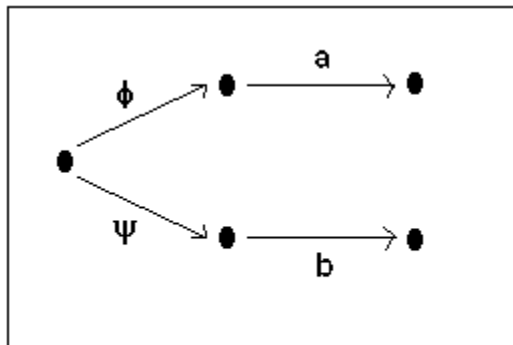
To answer our questions then: What happens if I insert the character a at the current position? In terms of the model, this question translates into "what happens if I activate *Editing_a*?", and the answer is clear, the machine ends up back in the state EDITING.

But what happens if you type *a*? There is absolutely no way to answer this question, because nowhere in the model is it recorded that state-changes occur in response to inputs. The standard model explains correctly that letters may appear in the document, but at no point does it explain how they come to be there.

1.2 Example of the problem #2

The second area where the observational nature of the standard X-machine is evident is in its treatment of determinism (an X-machine is deterministic provided each state has at most one arrow coming from it labelled with any given action ϕ). No matter which *state* we are currently at, there must never be a choice of arrows, each with the same label attached. The problem with this definition is that it does *not* capture the intuitive meaning of determinism, as we experience it in everyday programming. It bans from consideration a multitude of functionalities which are entirely deterministic in practice.

For example, imagine that a programmer has written the following code fragment, and decides to model the function `X& func(X&)` using a simple X-machine component. By definition, the machine represents the *possible behaviours* of the program, and does not concern itself with the reason why one branch of the `if(bCond)` statement should have been chosen in preference to the other. This program is clearly operationally deterministic, for at any time during any run of the system, the value of `bCond` will be known, and no ambiguity will arise. Similarly, the X-machine representation is that of a deterministic X-machine. No state is given a choice of similarly-labelled exit arrows from which to choose.

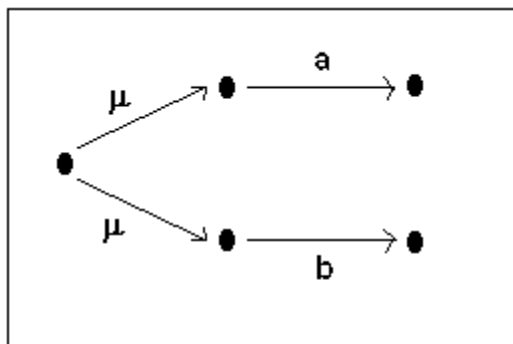


`X& func (X&)`

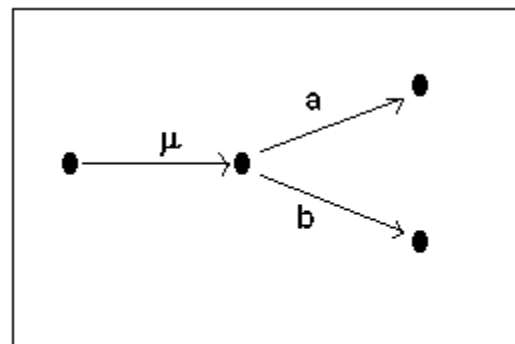
```
extern struct X;
extern bool bCond;
```

```
X& func (X &x) {
    if ( bCond )
        { x = phi(x); x = a(x); }
    else
        { x = psi(x); x = b(x); }
    return x;
}
```

Now suppose a second programmer is introduced onto the team, and decides to



`X& new_func_1 (X&)`



`X& new_func_2 (X&)`

create the following two functions, each functionally equivalent to the function above. These code fragments are functionally equivalent to one another, but the X-machine representation of `new_func_1` is non-deterministic, while those of `func` and `new_func_2` are both deterministic.

Clearly, there is something wrong here.

```
extern struct X;
extern bool bCond;

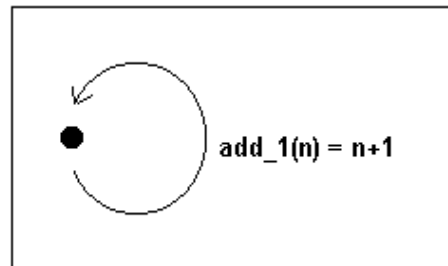
inline X& μ(X &x) { return bCond ? φ(x) : ψ(x); }

X& new_func_1 (X &x)
{
    if ( bCond )
        { x = μ(x); x = a(x); }
    else
        { x = μ(x); x = b(x); }
    return x;
}

X& new_func_2 (X &x)
{
    x = μ(x);
    x = bCond ? a(x) : b(x);
    return x;
}
```

1.3 Example of the problem #3

At the other extreme, we also have a problem with *non-determinism*. This **Z**-machine (whose sole state is both initial and terminal) generates all functions $\mathbf{Z} \rightarrow \mathbf{Z}$ of the form $add_n : m \mapsto m+n$ ($n > 0$) and as such is a random-addition generator. But it's well-known (and in any event easy to prove) that such a machine cannot be constructed as a standard program, since every statement in C++ (say) has fully deterministic semantics. Thus, while non-deterministic machines may be useful abstractions, they should be used only with extreme caution when they are intended to be accurate representations of physically programmable entities.



1.4 Overcoming the problem

The roots of these problems lie in what we loosely termed the *control structure* of the machine - the underlying finite-state machine. In using this as the basis for the X-machine's control structure, Eilenberg is appealing to an *analogy* between the behaviour of finite-state machines and the behaviour of general control structures. But analogies are frequently dangerous, and the definition of determinism is a case in point.

The constraint that emerging arrows should not share the same label makes perfect sense in a finite-state machine, because the finite-state machine really *is* a control structure. It tells us how a system's behaviour responds to changes in its environment, as signalled by the inputs it receives. It is these inputs which label the arrows, and these inputs which control the course of the computation.

But in a standard X-machine, the labels have no relationship to run-time changes in the system's environment. Instead, we are forced to embed environmental information within the fundamental datatype itself. This is why the Eilenberg/Laycock *stream X-machine* [LAY93], which requires the use of an $X \equiv Input^* \times Memory \times Output^*$ structure, has proven so very useful for representing realistic control systems. Without this recognition of the role played by inputs we are left with too loose a definition of behaviour, and cannot meaningfully represent choice-resolution. Nonetheless, even the stream X-machine works by default rather than by design. Arcs are not directly selected by the current input - rather, all of the arcs emerging from a given state are of equal status, but only one of them is labelled with an action that is meaningfully defined for the current input. So, as with the standard X-machine, we select paths at random, and eventually find that unique path which provides a non-empty output in response to our particular input stream.

The second observation we need to make is that there is no formal provision within the standard model for *re-labelling*. Because labels are actual relations on X, rather than labels representing these actions, an action is its own semantics – the idea that both ϕ and ψ might be unified into a single module μ is simply not catered for. In *practical* terms, this means that the model cannot encapsulate *within itself* the modular upgrading of a system. Instead, we have to "step outside" the model, and apply the concept of *refinement* [IPH98, BFHIJ97] thereby giving an *external* representation of the hierarchical design process. Refinement considers the relationship *between one X-machine and another*, rather than the effect of relabeling and re-interpretation within a single machine. For a fully consistent modelling environment, we need to distinguish between *labels* and the *actions* denoted by those labels. Using this approach, we would not have replaced ϕ and ψ in the X-machine diagrams with the new function μ - instead, we would simply have updated our denotational information to record the fact that ϕ and ψ were now both to be regarded as labels for the single function $X \& \mu(X \&)$.

This is not to say, of course, that the standard X-machine has no role to play. It remains the model that underpins each of its generalisations, and so a consideration of its properties gives us an insight into the entire X-machine family. Moreover, provided we understand X-machine behaviours in the same observational manner as (say) CCS process evolutions [MIL89], the model is perfectly suited to our needs. We experience problems only when observation is no longer enough - when we want to model run-time interactions.

In short, then, the standard X-machine is essentially an *observational model*, whereas specification of physical systems, whose behaviour varies crucially from run to run according to the local context, requires a more operational approach. We want to know *how* decisions are made *as and when inputs arrive for processing*, and what effect changes in the environment will have.

2 The operational X-machine

I have already mentioned the Eilenberg/Laycock stream-machine model as a valuable addition to the X-machine family, but it is not suitable in its standard form to the task ahead, which requires considerable focus on the nature of arrows. Instead, therefore, I shall introduce a new variant on the standard theme, the *operational X-machine* (OpXM). This embraces the control/processing separation of the standard model, but adopts a more rigorously operational attitude towards individual run-time behaviours. Nonetheless, the family resemblance to the stream X-machine is clear.

By construction, the OpXM models the standard X-machine, and so must be at least as powerful. However, the purpose of the OpXM lies not in its computational power, but in the ease with which run-time behaviours can be followed through the machine, and the ease with which the description can be extended to give a meaningful and complete definition of the *analog X-machine* (AXM).

The principal innovations in the definition of an OpXM are the formal distinction between labels and the actions they denote, and the replacement of the FSM with a general-purpose state machine - essentially, we require the concept of path traversal, but there is no physical justification for requiring the underlying state-set to be finite. Accordingly, we assume simply that a state-set exists and that state are connected by labelled arrows. Also, because we are focussing on specific operations, we shall assume that the sets of initial and terminal states are specified up-front.

Moreover, we can distinguish not just two sub-domains within the model (processing and control), but rather *five* distinct components.

2.1.1.1 Definition (OpXM)

An *operational X-machine* (OpXM) is a 5-tuple comprising 12 components in all,

$$OpM \equiv \langle \langle States, Initial, Terminal, Labels, Arrows \rangle, \langle Input, Import \rangle, \langle X \rangle, \langle Actions, Denotes \rangle, \langle Output, Export \rangle \rangle$$

where the components have the basic interpretations listed below.

LTS Backbone

<i>States</i>	a set of system states
<i>Initial</i> \subseteq <i>States</i>	valid initial system states
<i>Terminal</i> \subseteq <i>States</i>	valid terminal system states
<i>Labels</i>	a set of labels
<i>Arrows</i> : $\wp(States \times States \times Labels)$	a set of labelled arrows $a \xrightarrow{l} b$

Input Interface

<i>Input</i>	run-time input type
--------------	---------------------

As the machine receives inputs, these are used dynamically to select arrows. This is very different to the "try an arrow until you find one that works" semantics of the stream X-machine, although the observational outcome is superficially identical. A major difference between the two models is that the potential for non-determinism is formally represented within the operational X-machine. If a given input can be used to activate more than one arrow, we assume that this is *deliberate*, and that concurrent execution of *all* relevant paths is intended. Consequently, the machine may be in several *configurations* simultaneously, where a configuration is specified as an $\langle x, state \rangle$ pair. This affects the way that behaviour is defined, because we no longer require a given input to be meaningful for *all* configurations - it is entirely sensible for inputs to be supplied to each thread separately, but via the same keyboard (say). The important point is that *some* thread can process the input. More generally, of course, we cannot even assume that the relevant thread is one of those within the particular model we're considering, since this might simply be a sub-component of a larger system. Instead, each thread simply ignores all inputs that have no arrow associated with them.

Suppose the *input* is received, and one of the current configurations is $\langle x, state \rangle$. The *local behavioural equations* of the operational X-machine tell us the *output*, *new_state*, and *new_x* values associated with that particular input/configuration pair.

- $state \xrightarrow{label} new_state \in Import(state, input)$
- $output \in Export(x, Denotes(label))$
- $new_x \in Denotes(label) (x)$

2.3 Error handling

A *failure* occur when an *input* is received which cannot be processed given the current configuration $\langle x, state \rangle$. In general, such a failure may (but need not) indicate that the designer or specifier of a system has failed to consider every relevant possibility.

Looking at the three behavioural equations

- $state \xrightarrow{label} new_state \in Import(state, input)$
- $output \in Export(x, Denotes(label))$
- $new_x \in Denotes(label) (x)$

we see immediately that errors can occur in many ways:

- There is no arrow $state \xrightarrow{label}$
- *label* is defined, but $action \equiv Denotes(label)$ is undefined
- *label* and *action* are defined, but $Export(x, action)$ is undefined

- *action* is defined, but *action(x)* is undefined

Whether or not a failure is actually an *error* depends upon the contract between the creator of the model and his client. This is a commercial decision – we cannot make a ruling on *theoretical* grounds. In general, it is reasonable to expect, for example, that not *every* input is expected to generate an output immediately. During a lengthy processing operation, for example, the system may be *designed* not to refresh the screen after every input, since doing so would slow processing down intolerably; instead an update is produced every 1000th input (say).

Moreover, it may be that the designer knows that only a limited subset of the input type is actually relevant for the current system, so that certain paths, though theoretically no different from others, are actually unobtainable given the available inputs. In such circumstances, there may be a valid commercial reason not to associate every label with an action; if an arrow can never be traversed, its label can never be activated, and no action need be associated with it.

Nonetheless, it is sensible to allow for situations in which failures *are* errors, and for this reason we typically require each of the relevant types *Arrows*, *Labels*, *Actions*, *States*, *Output* and *X* to contain an "undefined" element, \perp . Where a failure would otherwise occur we extend the definition of the relevant relation to ensure that the value \perp is returned instead. As usual, any function which takes \perp as one of its inputs is expected to return the result \perp in its turn, so that failures propagate cleanly through the system. In such circumstances, we will often write *e.g.* *Output* \perp to indicate that \perp has been included as a valid *Output*, and has the intended meaning "undefined".

This strategy is useful, but does require designers and modellers to pay careful attention to their designs. For example, the requirement that \perp propagate through the system obviously makes it unsuitable as the representation of 'waiting conditions', as when output is deliberately suppressed for efficiency reasons. The designer would need to introduce a notional WAITING output, which represents to the model that a result has been produced but which has no observational profile.

Such precautions ensure that the given machine is *complete* - no matter what the configuration or the input, a new configuration and an output will always be generated.

2.4 Determinism

Because the *OpXM* takes explicit account of the input, it is considerably easier to decide when a computation is truly non-deterministic. There is still room, however, for debate.

It is clearly reasonable to require that a deterministic machine be complete. The user should not be left in a situation where it is impossible to distinguish between a system

which has stopped running altogether, and one which is still running, but which has stopped producing observable indications of this fact.²

The user also has a right to expect the observable output-stream of a deterministic machine to depend deterministically upon the input-stream. This essentially means that each individual *output* must be well-defined. In other words, the relation

$$Export : X_{\perp} \times Actions_{\perp} \rightarrow Output_{\perp}$$

should be a (total) *function*.

Completeness implies that the values *new_state* and *new_x* calculated by the behavioural equations necessarily exist, but does not guarantee that they are uniquely defined. A naïve approach to determinism would be to require these values to be uniquely determined. Again, however, this misses the point.

Determinism operates at two levels - *local* and *global*. Internally, a machine may still be deterministic even if its individual steps along a computation path are *non-deterministic*, because the overall behaviour of the machine - the overall transformation $X \rightarrow X$ that it computes - is the composition of the actions at each arrow. It is entirely possible for a collection of proper relations to have a composition which is a function; if this occurs across every computation path, the machine is indeed *globally deterministic*, despite the *local non-determinism* in individual configurations.

Incidentally, we might also require the relation *Denotes: Labels* ↔ *Actions* to be a function. Indeed, readers might wonder why this should ever be allowed to be a proper relation. The general answer to this question will be illustrated in the following sections, where we demonstrate that relations are required to model hierarchical specification of component architectures. But even in existing commercial software projects, any given function name and signature may have multiple implementations. This was reflected in the introduction of the C++ namespace keyword. In commercial development, it is entirely possible for two code fragments of a system to be created by different project groups or even different companies, and the risk always exists that two distinct functions, classes, or global constants will be given the same name by different programming groups. In general, we can assume that such conflicts will be eliminated in a fully specified system, but until this elimination has occurred, it is sensible to include a mechanism for modelling the conflicting label usage.

2.5 Example: Application to Component Architecture

Though not of direct relevance to this paper, it is perhaps useful to consider a practical application of the *OpXM* model.

² In practical terms, it is entirely acceptable that no screen refresh occur. Nonetheless, we would expect the *theoretical* model to produce (say) WAITING outputs which might *feasibly* be interrogated.

2.5.1 COMPONENT ARCHITECTURE

Component architecture refers to the construction of large systems from small, reusable components that are selected and linked dynamically, *at run-time*. In order to keep this discussion concrete, I shall base my examples and terminology on COM, but it is not my intention to endorse any particular manufacturer's system over another, and the argument should be understood to apply generally. This is not the place to go into the inner working of COM, but it is useful to highlight the three main advantages that components bring to industrial applications, together with the requirements these impose upon programmers, and to show how the operational X-machine supports their use.

The most obvious commercial advantage of component architecture is that it delays obsolescence and reduces construction costs. Rather than produce a monolithic application that stands or falls as a single unit, we generate a number of co-operative but identifiably separate components. When a particular function becomes obsolete, perhaps because a better algorithm has been discovered, we simply replace the component in question, leaving the rest of the system unaffected. Thus, if the only part of my word-processing suite that needs replacement is the section that handles bitmap display, I should be able to add a new bitmap-handler to my PC and throw the old one away, without in any way changing any other component of my word-processor. This is very different from traditional commercial practice, where each upgrade of a system requires the *entire* system to be re-installed, rather than just the parts that are actually affected.

Of course, the idea that functionality should be compartmentalised into re-usable and replaceable objects is nothing new, and it's a reflection of the close link between objects and components that C++ compilers typically offer the easiest direct means of compiling COM components [ROG97]. The distinction between a component, as opposed to any other type of object, is that its re-use and replacement can happen *while the system is running*. We typically re-use a C++ class, or replace its internal definitions, as a programming exercise. Having completed the required development work we recompile the files affected, and re-link the program. But the whole point of component architecture is that this shouldn't be necessary.

As always, this added flexibility doesn't come free. For this kind of re-use and replacement to be possible, it is necessary that components be defined not in terms of their internal behaviour, but in terms of their *interface behaviour*. It must be possible for the rest of the word-processor to interact with the new bitmap-handler as if it were the old one; the new interface must support the old one. Rather than define a class by distributing its header file, we need to distribute information about the interfaces it supports. Nonetheless, the benefits are widely considered to be worth the effort involved, because of the other major advantages that components allow – *embedding* and *distribution*.

As soon as you throw away the header file and define an object in terms of its interfaces, it becomes a simple task to embed one object within another. You simply equip the outer object with the means to pass relevant requests to the inner object. It is for this reason that Microsoft re-wrote the whole of its OLE technology using the

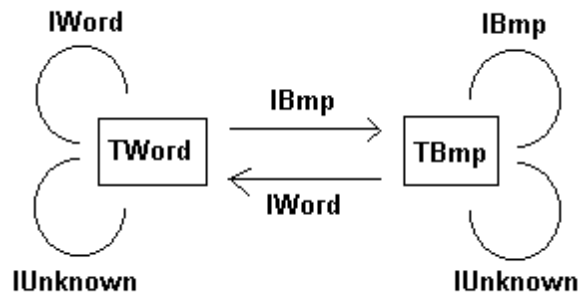
component object model, as soon as COM had been formalised (hence OLE2). Moreover, the fact that an application has been constructed from components means that we can easily pull it apart into its separate functional subsystems. As components, these subsystems are totally defined by their interface behaviour - consequently, it is irrelevant *where* they reside physically. Provided the interface is cleanly supported, the component can be on the same machine, elsewhere on the same LAN, or even on an orbiting satellite. These two simple observations have made possible much of the internet technology, which, though generally unavailable just six years ago, is so taken for granted today.

2.5.2 HIERARCHICAL DESIGN AND THE OPXM

But what has this to do with the *OpXM*?

First of all, consider the consequences of component architecture. A monolithic application has been neatly divided into functionally distinct components. In X-machine terms, this means that the machine has been neatly carved up into a number of sub-machines - one for each component.

Now consider what it means for a component to become activated. For this to be possible, recent user inputs must have resulted in the need for an action to be carried out, and the component has been selected to perform the action on behalf of the application. But the component will only be invoked if it responds favourably to a *QueryInterface* enquiry. In *OpXM* terms, the current input has caused an arrow to be traversed into the component sub-machine, where the label on that arrow corresponds to an action identified in one of the component's interfaces.



What does this mean in practice? At the most abstract level, it means that the labels on arrows connecting the component sub-machines necessarily correspond to various interfaces supported by that component. So, in addition to specifying the run-time interactions of the components, the interfaces *also* specify the arrows connecting sub-machines in the *OpXM* model of the application. And as we make the model more concrete, we break the single interface-labelled arrow into a collection of arrows, each one labelled with the name of one of the functions supported within the interface.

We can refine the model recursively (hence the claim for hierarchical design) in two obvious ways, both related to the concept of refinement as explained in *e.g.* [BFHIJ97]. First, we regard the *TWord* and *TBmp* "states" as place-holders for sub-machines. Before substituting the machine for the state (and matching up arrows in the usual way) we would continue the component-wise decomposition of the structure - for example, it is likely that *TWord* would itself comprise a number of components joined by interfaces. Secondly, we would deconstruct the "interface arrows" into a

family of arrows, one for each action specified in the interface. As it stands, for example, this *OpXM* is highly non-deterministic, because the single label *IBmp* actually represents *every* action listed within the interface *IBmp*. By introducing a new arrow for each action, and attaching it to the appropriate "exit point" of the expanded *TWord* sub-machine, we gradually increase the determinism of the machine, until eventually we generate a fully deterministic model.

Replacement of a component is now much easier to model. By definition, the new component must interact with the rest of the system in exactly the same way as the old one - only its internal working can differ. This means that the high-level diagrams remain unchanged. Only the sub-machine corresponding to the actual component will change. Nonetheless, we shall argue below that here are good reasons for considering the underlying actions themselves to change when a component changes, even though the interface remains unaffected. But even this is catered for - the label (and hence the machine diagram) remains unchanged; only the map *Denotes* is updated.

For this reason, it is convenient to think of *Actions*, *Export* and *Denotes* as *parameters* of the *OpXM* model, and we can write *e.g.* $OM(\textit{Actions}, \textit{Export}, \textit{Denotes})$ to indicate a general *OpXM* "engine" whose ultimate functionality depends upon the user's parameter selection.

3 Concurrency Issues

The behavioural nature of the standard X-machine hides another problem that becomes apparent as soon as one moves to the operational X-machine. Because we "run" a standard X-machine by choosing a single valid path through the underlying finite state machine, concurrency is never an issue.

In realistic situations, however, concurrency is *always* an issue, and has presumably been raised in discussions of stream X-machines, where the same considerations must apply. We have mentioned this problem only briefly. If non-determinism is present in the machine when a given input is provided, do we choose *one* of the possible pathways, or *all* of them? Because we intend the model to be a tool for modelling real systems, the answer has to be the latter - we follow *all* valid pathways at all times. The reason for this choice is purely pragmatic. We assume that users are capable of defining deterministic machines if they wish to (although any "X-machine Development Environment" had better be equipped with a tool for spotting accidental non-determinism!). Therefore, any non-determinism that remains can safely be deemed an intended behaviour of the system. This is not unreasonable. In physical systems, it is *normal* for a single stimulus to result in the simultaneous activation of one or more behavioural modes - indeed, we would not be able to apply this model to quantum computation without this assumption.

The problem with this assumption is that our definition of *action* is apparently inadequate to model it. We currently deem an *action* to be a relation $X \leftrightarrow X$, and we will normally insist that this relation be a function. However, it is well known that

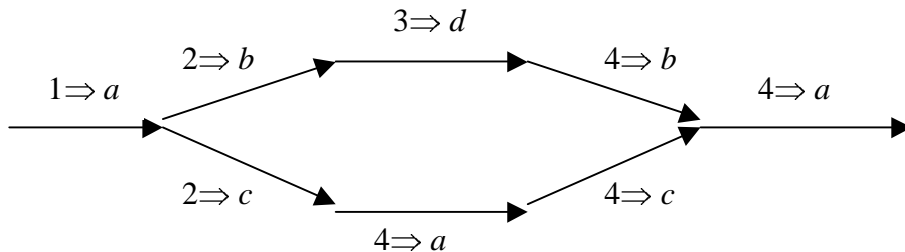
functional equivalence is not enough to guarantee concurrent equivalence, as this familiar counter-example recalls. Define $a(x) \equiv 10$, $b(x) \equiv x+10$, $c(x) \equiv 20$ and $d(x) \equiv 40$. Then $\xrightarrow{a} \xrightarrow{b}$ and \xrightarrow{c} are functionally equivalent. But they are *not* equivalent as concurrent processes, because they can interact in different ways with the process \xrightarrow{d} . The possible interleavings of d with ab include the function $adb(x) \equiv 50$, while those of d with c ($cd(x) \equiv 60$ and $dc(x) = 20$) do not.

This failure is, however, only *apparent*. The problem arises because the fundamental datatype is insufficiently sensitive to take account of the internal structure of composite relations. In situation where concurrency is a real problem, we can maintain our description of actions as functions (more generally relations) $X \rightarrow X$ by requiring the fundamental datatype itself to hold a representation of process evolution. To see how this might be achieved, consider how we might give an operational semantics to some particular *OpXM* run. It is convenient for these purposes to write

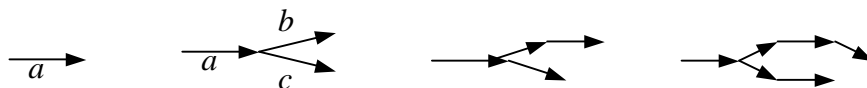
$$state_0 \xrightarrow{input \Rightarrow action} state_1$$

to indicate that receiving *input* in $state_0$ results in *action* being applied, and $state_1$ being attained - in other words, $Denotes(Import(state, input)) \equiv action$.

As we traverse the machine diagram, we label each node with a tree showing the process evolution to-date. To give a simple example suppose that the user supplies the input stream 1234 to this machine.



The evolution of this machine, given the input stream, is represented by the sequence of trees (where I've ignored the labels in later trees for clarity)



These trees provide a record of the machine's behaviour, including its branching structure in response to local non-determinism. It seems sensible, therefore, to *include these trees within the datatype X* itself. By doing so, we automatically distinguish between $\xrightarrow{a} \xrightarrow{b}$ and \xrightarrow{c} , because the embedded trees are different, despite the fact that they give the same functionality on \mathbf{N} (say). If we write $Proc(Actions)$ to denote the set of *Actions*-labelled trees (since they are essentially *CCS* process trees), we are really asking for concurrency to be catered for by choosing X of the form $X \equiv Proc \times Value$, where *Value* is the actual type originally being operated upon.

This requires the actions themselves to be more complex than before, since they must process both the current *Value* and at the same time extend the process evolution tree, but the technology already exists - it is no different in principle to that used for constructing *CCS* process trees in the first place. It is clear, incidentally, that the inclusion of *Value* as a factor space in *X* is for convenience only - the information is already encoded (and in more complete form) within the accompanying process diagram. Accordingly, we could meaningfully define simply $X \equiv Proc(Actions)$, whence a concurrent operational X-machine can equally be considered an operational *Proc(Actions)*-machine, and one which is, moreover, deterministic.

The behaviour of such a machine is slightly less directly determined than that of a standard operational X-machine. First we traverse the tree in response to inputs received at run-time. This generates a process tree, which we regard either as a *CCS* description of the concurrent process generated, or equally as a standard(!) *Value*-machine, the (full) behaviour of which determines the relation to be applied to any given input. Despite the seeming complexity of this two-step process, it seems a small price to pay for a true member of the X-machine family which nonetheless caters for full *CCS*-style concurrency within a single coherent structure.

This illustrates, by the way, that it is entirely meaningful for the intermediate output of an operational X-machine to be another X-machine. In particular, one would expect to find a process of this form resulting in a *Universal Operational X-Machine*, capable of modelling any other X-machine (operational or otherwise) via this two-step process.

4 The Analog X-Machine

In this section we are interested more in describing the analog extension of the X-machine model, the AXM, and illustrating it with simple examples, than with discussing detailed semantics or computational power - these important issues will be addressed more completely in a later section.. Accordingly, we shall assume that *Labels* and *Actions* are in 1-1 correspondence and we can regard the concepts as equivalent, and that all "actions" on *X* are actually functions. The prefix "analog" refers to our choice of real numbers to model *duration*, rather than some more general partial order.

4.1 Timing and analog descriptions

It is important to recognise the significant complication that continuous time introduces into the description physical computation. Typically, analog descriptions of computation assume that time can be modelled by the continuous space \mathbf{R} , and that such operations as integration and differentiation are consequently meaningful. In the same way that discrete operational components combine to yield recursive functions, so *analog-generable* functions, *i.e.* those realised as the behaviour of standard analog circuitry, are characterised by differential equations relating the behaviours of basic analog components.

But how valid is the assumption that time can be modelled by \mathbf{R} ? Remember, we want to ensure that all of the constructions we use in this paper are valid physically. If it should turn out that a number of time values are not physically computable, this will throw doubt upon any construction which relies on continuous variation.

In general terms the assumption is valid, because we can describe an experiment which can potentially generate any positive real number. For example, connect a Geiger-counter to a measuring system, start the system running, and measure the interval before the first observed radioactive particle generates a click. Given our current understanding of physics, any positive interval might be observed.

However, this example also serves to warn us about the pitfalls of making physical assumptions. For in the Standard Model of physics (the model of everyday research, based on quantum theory, and including the Big Bang and Inflation), there are strong grounds for arguing that intervals shorter than roughly 10^{-34} *sec* are physically meaningless, because they are associated with energies so huge that they would generate a black-hole which would swallow up the interval at the instant of its creation. In contrast, a representation of time as \mathbf{R} assigns a meaning to *all* intervals, no matter how short. Since our Geiger-counter experiment reports the length of just such an interval, it is possible that our set of observable values will not include *all* positive reals, but only those which are sufficiently large.

Of course, this particular shortcoming is easily overcome. By constructing a system which subtracts one temporal length from another, we can potentially still generate any real number, despite the problems inherent in measuring very small intervals.

Nonetheless, we should recognise that this is only possible because of the stochastic nature of the processes under discussion. This ‘probability factor’ allows us to ignore the logical circularities that would otherwise arise when we ask *how* we are to choose the instant at which a given input is to be supplied. Clearly, any such decision would require a computation, but it is precisely the nature of this computation that is under scrutiny. Fortunately, this circularity is not insurmountable, since the classes of Turing computable and analog-generable functions must surely still be regarded as computable in any valid model of physics (*i.e.*, any model of physics consistent with known experimental results), and as such can be used as the starting point from which to develop our more comprehensive theory of full physical computability.

Topic for further research. We have observed that all real numbers can be generated by observation of a physically realisable *stochastic* process. But can real numbers always be generated if we limit ourselves to *deterministic* real processes?

Topic for further research. The classes of Turing-computable and analog-generable functions are in some sense *absolutely computable*. Any valid model of physics must render members of these classes physically computable. Is there a maximal class of absolutely computable functions?

4.2 Notation

We shall frequently want to convert the domain of a function from $[0,1)$ to $[0,\infty)$ or vice versa. Accordingly, we shall make frequent use of the mutually inverse computable order-isomorphisms

$$\text{Tan} : [0,1) \rightarrow [0,\infty), x \mapsto \tan(\pi x/2)$$

$$\text{Tan}^{\leftarrow} : [0,\infty) \rightarrow [0,1), x \mapsto 2.\text{tan}^{\leftarrow}(x)/\pi$$

Any function $\phi: [0,\infty)_c \rightarrow X$ can be converted to an associated function on $[0,1)_c$ by composition with Tan , and vice versa. For example, $\phi: [0,\infty)_c \rightarrow X$ is associated with the function $\phi \circ \text{Tan}: [0,1)_c \rightarrow X, y \mapsto \phi(\text{Tan}(y))$.

4.3 Generalised paths

An operational X -machine is based ultimately upon its LTS backbone, the fundamental components of which are arrows

$$\text{state}_0 \xrightarrow{\text{input} \Rightarrow \text{action}} \text{state}_1$$

An OpXM arrow is by construction *atomic* - it is either traversed or not. There is no middle ground, and no concept of *duration*. In order to model analog systems, it is useful to introduce timing information. A standard approach would be to associate a real value (or sometimes a pair of such values) with each arrow, indicating the time required to traverse it, and then use timing information of this kind to say how far along a path a given computation has reached. The approach we introduced in [STA90] is somewhat more general.

We assume that it requires a positive amount of time to perform the action associated with an arrow, and that it is meaningful to say something like "when the arrow has been half-traversed, the work is half-done". Formally, we are saying that each point along the arrow represents a function $X \rightarrow X$, or more generally $X_{\perp} \rightarrow X_{\perp}$, and that this function varies continuously as we traverse the arrow. In other words, an arrow is actually a continuous function

$$\text{arrow}: [0,1) \rightarrow X^X$$

Notice the extremely important point that we have done away with the state-set. Because every point along the arrow corresponds to an action, we are really saying that *every state* corresponds to an action. Consequently, we may as well dispense with state-space, and consider the diagram to be drawn directly on function space itself.

It is still useful to model the idea that continuous computation occurs between branching points where two or more arrows 'meet', and so we retain the concept (though not the reality) of a state-set by identifying a set of *Junctions* - by definition, a *junction* is the initial point of an arrow, and so is a function in X^X of the form $\text{arrow}(0)$ for some *arrow*. Similarly, it can sometimes be useful to select certain *junctions* and declare them to be valid places to start and finish computations. Notice,

however, that although *terminal* points are junctions, they are sometimes junctions of a special kind, as we shall see below when we discuss the domain of an arrow in more detail. The model does not specify a general temporal type *Time*, except in so far as progress along a given arrow can be parameterised by values in $[0,1)$. This is because progress along a path is defined operationally. The operational semantics of the AXM focuses upon the receipt of inputs, and asks, *How far along the arrow must we be, given that it is physically appropriate for the current input to be arriving at this time?*³

Why is *arrow* defined only on $[0,1)$ rather than the whole of the unit interval? Two reasons. As soon as we move into the world of timed operations, we can contemplate processes which are designed never to terminate, and where the arrow effectively takes for ever to traverse. Requiring *arrow* to be defined at 1 would be demanding too much, and might bar the definition of realisable systems for no good reason. For example, if we define the function *arrow*(x) to be the constant real-valued function taking the value *Tan*(x), then *arrow*(x) is indeed a continuous path in $\mathbf{R}^{\mathbf{R}}$, but cannot be extended continuously to yield a meaningful *arrow*(1). Secondly, it is technically simpler to use the fact (for $a < b < c$) that $[a, b) \cup [b, c) \equiv [a, c)$ when describing the consecutive traversal of two arrows – we don't have to keep writing conditions concerning the behaviour at the intersection of the two arrows.

Nonetheless, there are also times when we specifically *want* an arrow to contain its end-point, and on those occasions, we use a simple trick. Suppose the 'missing' endpoint is the function $f \in X^X$. The constant function *arrow*(x) $\equiv f$ is necessarily continuous, and so is a valid path. We now compose the two arrows, and so generate a combined behaviour which is equivalent to traversal of the closed path. In particular, terminal points are often generated in this way. The nature of inputs and outputs needs careful consideration. Whether inputs are received at discrete intervals or in a continuous stream, they are intimately associated with decision points, or in other words, *Junctions*. Notice, incidentally, that we do not ask *how* the input is generated. It could be generated by a user, or by a continuous monitoring program. Our interest is to merely explain how the arrival of input is handled. It is possible, of course, that the handling of inputs may be time-dependent. Even though we do not model time directly, time-dependence of this kind can be modelled by including a suitable type, *Time*, as a component either of *Input* or of X .

Taking these observations on board, the definition of the *Analog X-machine (AXM)* is considerably *simpler* than for an operational X-machine, since we no longer need a separate state space, and the space of *Actions* is fully determined by the arrows in X^X .

4.4 Definition

An *analog X-machine (AXM)* is 9-tuple,

³ We could, if we wished, also introduce a *temporal semantics* for modelling time-critical systems. The pertinent question would then be, *Given that processing is this far along an arrow, can the current input meaningfully be handled?*

$AXM \equiv \langle X, Arrows, Junctions, Initial, Terminal, Input, Import, Output, Export \rangle$

where the components have the basic interpretations listed below.

Representation

X the fundamental datatype

LTS Backbone \equiv Process Interface

$Arrows : \wp([0,1) \rightarrow X^X)$ a set of function paths

$Junctions \equiv \{ a(0) : a \in Arrows \}$ a set of functions

$Initial \subseteq Junctions$ the valid initial system states

$Terminal \subseteq Junctions$ the valid terminal system states

Input Interface

$Input$ the run-time input type

$Import : Junctions \times Inputs \rightarrow Arrows$ gets arrow corresponding to input

Output Interface

$Output$ the run-time output type

$Export : X \times X^X \rightarrow Output$ determines appropriate output

Constraints

The *arrow* selected by *Import* must emerge from the specified *junction*:

$$Import(junction, input) \equiv arrow \Rightarrow arrow(0) \equiv junction$$

4.5 Behaviour

The guiding question is always the same: *given that we are currently traversing such and such an arrow, and such and such an input has arrived, how far along the arrow must we be, given that it is physically appropriate for the current input to be arriving at this time?* Sometimes, this question will have no answer, because the input was intended for some other concurrently active computation path. But if there is an answer, we require that answer to be uniquely defined. This means that any set of potential junctions must have a unique minimal member, where the ordering is induced from the parameterisation of the arrow itself. We therefore discuss the *local behaviour* of the machine. The global behaviour is found in the usual way by integrating over all local behaviours. Because inputs determine behaviour, and are always imported at junctions, we can assume without loss of generality that computations are fully determined by the string of junctions visited as the thread evolves.

Suppose that the machine is currently at some junction $f \in X^X$ on some arrow $a \in Arrows$, so that $f \equiv a(0)$, and that *input* is received. Let

$$J(a, input) \equiv \{ t \in (0,1) \mid a(t) \in Junctions \ \& \ Import(a(t), input) \text{ is defined} \}$$

- If

$J(a, input)$ has a minimum value, m , then computation moves to $a(m)$

Else If

a has a terminus, *i.e.* there is at least one arrow b such that $a(t) \rightarrow b(0)$ as $t \rightarrow 1$ and $Import(b(0), input)$ is defined, computation moves to $b(0)$.

Else

computation remains where it is and nothing further happens until a new input is received, at which time this procedure is re-invoked.

Assuming we haven't terminated until the next input, computation is now at a valid junction j for *input*. We now know that the intent of the preceding computational session has been completed satisfactorily, so we generate any relevant output and update the internal value of the fundamental datatype accordingly:

- Generate the output $Export(x, j)$ – this may be empty
- Calculate $new_x \equiv j(x)$

This is a rather unfamiliar model for most readers, so we now illustrate the basic principles with a number of worked examples.

4.6 Examples of Analog Computation

Recall that we use the term *computable* in its purely physical sense, to mean "computable by some physically realisable algorithm" – this need not coincide with Turing-computable. Similarly, we use the term "computably enumerable" rather than "recursively enumerable". In this section we shall focus on two operations familiar from "standard" analog computation, namely integration and differentiation. Intuitively, it is simplest to think of $\mathbf{R}_c \equiv \mathbf{R}$ throughout this section, but we retain the more general notation to ensure compatibility and applicability of our results later on.

4.6.1 COMPUTABLE LEBESGUE INTEGRATION

Since the AXM is fundamentally an analog model, it is not surprising that it can be used to implement the archetypal analog operation, (*Lebesgue*) *integration* [APO78, WEI73]. Because we're interested in physical examples only, we shall re-express the standard definitions underlying Lebesgue integration to ensure they don't violate computability. The definitions we introduce apply specifically to \mathbf{R} , but extend in the obvious way to any other measurable space, and in particular \mathbf{C} , in the obvious way.

An open (or closed or half-open) interval $I \equiv (a, b)$ is computable provided a and b are computable. The *c-measure* μ_c of the computable interval I is the (computable) number $\mu_c(I) = b - a$.

Let $F \subseteq \mathbf{R}$ be any subset, which need not be computably enumerable. A collection $C \equiv \{ U_\alpha \}$ of computable intervals is a *c-cover* of F provided the index set A is computable, the "enumeration" $\alpha \mapsto U_\alpha$ of the collection is computable, and there is a proof that $F \subseteq \bigcup U_\alpha$. Typically, we shall use $A \equiv \mathbf{N}$ so that "summation" needs no further consideration. If the value

$$length(C) \equiv \sum_{\alpha \in A} \mu_c(U_\alpha)$$

is computable (and so finite) or else provably infinite, we say that C is measurable.

If $\{ C_n \}$ is a c-enumeration of measurable c-covers of F , and there exists a proof that

$$\bigcap_n \left(\bigcup C_n \right) = F$$

then we say that F has computable measure (indeed, we claim that F is actually computable, since this is clear demonstration of an algorithm for computing it), and define

$$\mu_c(F) \equiv \inf_n \{ length(C_n) \}$$

For example, let CANTOR denote the Cantor middle-third set. This set is certainly not recursive in the standard sense because it is *uncountable*, and so cannot be enumerated, computably or otherwise. Nonetheless, it has c-measure 0, since its very construction yields the c-covers required to compute its measure, *viz.*

- $C_0 \equiv \{ [0, 1] \}$
- $C_1 \equiv \{ [0, 1/3], [2/3, 1] \}$
- $C_2 \equiv \{ [0, 1/9], [2/9, 1/3], [2/3, 7/9], [8/9, 1] \}$

and so on. By *definition*, $\bigcap_n \left(\bigcup C_n \right) \equiv \text{CANTOR}$, and moreover $length(C_n) \equiv (2/3)^n \rightarrow 0$.

If a property can be proven to hold for all points of \mathbf{R} , with the exception of a subset of c-measure 0, we shall say that it holds *computably almost everywhere (c.a.e.)*.

A function *step*: $\mathbf{R} \rightarrow \mathbf{R}$ is a *c-step-function* provided there is a finite collection U_n of disjoint open c-intervals, each of finite length, such that *step* is constant with computable value s_n on each U_n , and takes the value 0 outside the U_n 's. (We don't actually care what happens at the point-boundaries between intervals, as long as f is defined there, but we may as well require the value 0 for definiteness). It is easy to integrate a c-step-function, because all the terms are computable, and there are only finitely many of them.

$$\int_{\mathbf{R}} \text{step} = \sum_n \text{length}(U_n) \cdot \text{step}(U_n)$$

A c-enumerable sequence f_n of such step-functions is non-decreasing provided $f_0(x) \leq f_1(x) \leq \dots$ c.a.e., and has limit f provided $f_n(x) \rightarrow f(x)$ c.a.e.. We define

$$\int_{\mathbf{R}} f = \sup \int_{\mathbf{R}} f_n$$

If this value is finite, we say that f is c-integrable. The set of all c-integrable functions obtainable in this way as the limit of a c-enumerable increasing sequence of c-step functions is denoted L^{inc}_c . In particular, all computable step-functions are computably integrable. We now extend the definition by saying that whenever g and h are in L^{inc}_c , then the function $f = g - h$ is c-(Lebesgue) integrable, with integral

$$\int_{\mathbf{R}} f = \int_{\mathbf{R}} g - \int_{\mathbf{R}} h$$

and write $f \in L^1_c$.

In order to define integration between limits, we shall use the notation $f|_K$ (for $K \subseteq \mathbf{R}$) to denote the function $f|_K \equiv f$ inside K , $f|_K \equiv 0$ outside K . Then

$$\int_a^b f = \int_{\mathbf{R}} f|_{[a,b]}$$

This tells us how to calculate *definite* integrals, whenever $a, b \in \mathbf{R}_c$. To compute *indefinite* integrals, we simply define the indefinite integral $F: \mathbf{R}_c \rightarrow \mathbf{R}_c$ of f to be the function

$$F(x) = \int_0^x f$$

Since \mathbf{R}_c is certainly *dense* in \mathbf{R} , and the definitions we have given for integration coincide with those normally used, we know that whenever $x \in \mathbf{R}_c$, $F(x)$ can be continuously extended to the whole of \mathbf{R} . As usual, changing the lower bound from 0 to some other fixed c-value simply varies F by a constant amount; and we denote the indefinite integral F by dropping the limits from the integration sign, $F \equiv \int f$.

We shall demonstrate a AXM that outputs the value $F(x)$ in response to the input $x \in \mathbf{R}_c$. The function to be integrated, f , is held as the value of the fundamental datatype. Since the c-integral of a function is again c-integrable, we shall take $X \equiv L^1_c$.

Let **Int**: $[0,1)_c \rightarrow L^1_c \rightarrow L^1_c$ denote the function $\mathbf{Int}(x)(f) \equiv \lambda y. \int_{\text{Tan}(x)}^y f$. Then

$$\mathbf{Int}(t)(f) \equiv \lambda y. \int_{\text{Tan}(t)}^y f \equiv \lambda y. \left(\int_0^y f - \int_0^{\text{Tan}(t)} f \right) \equiv \lambda y. (F(y) - F(\text{Tan}(t)))$$

where $F \equiv \int f$. Consequently, $\mathbf{Int}(x)$ is an action $L^1_c \rightarrow L^1_c$.

We observe for future reference that, given any $x \in [0, \infty)$,

$$\mathbf{Int}(\mathbf{Tan}^{\leftarrow}(x))(f) \equiv \lambda y. (F(y) - F(x))$$

Notice also that \mathbf{Int} is a continuous function of x . For suppose δx is some c -variation in x . Then

$$[\mathbf{Int}(x + \delta x) - \mathbf{Int}(x)](f) \equiv \lambda y. (F(\mathbf{Tan}(x)) - F(\mathbf{Tan}(x + \delta x)))$$

is independent of y . By the Fundamental Theorem of Calculus, we know that F is differentiable, and hence F is continuous. But \mathbf{Tan} is also continuous, whence $F \circ \mathbf{Tan}$ is continuous, and

$$\forall f \in X : \quad \lim_{\delta x \rightarrow 0} \mathbf{Int}(x + \delta x)(f) \equiv \mathbf{Int}(x)(f)$$

This tells us that $\mathbf{Int}(x)$ is a continuously varying action $L^1_c \rightarrow L^1_c$, *i.e.*, a valid arrow in a Analog L^1_c -machine.

Recall that a AXM is a 9-tuple. We need to identify the nine components of this example. We have already identified X . It is the set L^1_c .

The Process Interface

We want to be able to stop at any point along the integration arrow. Since we can only stop at junctions, we define every point of the arrow to be a junction, by making it the image of a constant path. For the purposes of this example, *Arrows* contains uncountably many paths, all but one of which are constant. The only non-constant path is \mathbf{Int} . For each point $x \in [0, \infty)$, we define a constant path $stop_x : t \mapsto \mathbf{Int}(\mathbf{Tan}^{\leftarrow}(x))$. *Junctions* is fully determined by *Arrows*. In this case, the junctions are precisely the points along the only non-constant arrow. So $Junctions \equiv \text{im}(\mathbf{Int})$. We shall take all junctions to be terminal, but only the first junction to be initial.

$$Arrows \equiv \{ \mathbf{Int} \} \cup \{ stop_x \mid x \in [0, \infty) \}$$

$$Junctions \equiv \text{im}(\mathbf{Int})$$

$$Initial \equiv \{ \mathbf{Int}(0) \}$$

$$Terminal \equiv Junctions$$

The Input Interface

We shall use the input set to determine the upper limit for the integration. Since this can be any computable non-negative real number, we'll take $Input \equiv [0, \infty)_c$. The *Import* function has to select the appropriate point along the integration path. By construction, these points are all junctions, and are automatically labels by the path parameter itself, so we can use this for synchronisation purposes.

$$Input \equiv [0, \infty)_c$$

$$Import : \langle stop_x(0), x \rangle \mapsto stop_x$$

The Output Interface

We're going to generate the integral of f (whatever the value of the fundamental datatype happens to be) between 0 and the input value x .

$$Output \equiv \mathbf{R}_c$$

$$Export : \langle f, operator \rangle \mapsto -(operator(f))(0)$$

Does this AXM representation actually work? Suppose that the machine starts at the point $start \equiv integrate(0)$ on the arrow $integrate$, and that the initial value of the fundamental datatype is f . Suppose the input x is received.

To determine the subsequent behaviour, we have to examine the set

$$J(start, \mathbf{Int}, x) \equiv \{ t \geq 0 \mid \mathbf{Int}(t) \in Junctions \ \& \ Import(\mathbf{Int}(t), x) \text{ is defined} \}$$

The only *junction* for which $Import(junction, x)$ is defined is $junction \equiv stop_x(0)$. By definition, $stop_x(0) \equiv \mathbf{Int}(Tan^{\leftarrow}(x))$, so the only viable value of t is $t \equiv Tan^{\leftarrow}(x)$. This does indeed satisfy $t \geq 0$, and so

$$J(start, \mathbf{Int}, x) = \{ Tan^{\leftarrow}(x) \}$$

The minimum element of the set is t , and computation moves to $j = \mathbf{Int}(t)$. This identifies the intended effect of the computation, so we generate the appropriate output and update the fundamental datatype accordingly:

- Generate the output $Export(f, j)$ – this may be empty
- Calculate $new_x \equiv j(f)$

Direct calculation shows that

$$j \equiv \mathbf{Int}(Tan^{\leftarrow}(x))$$

so we update the fundamental datatype to

$$new_x \equiv \mathbf{Int}(Tan^{\leftarrow}(x))(f) \equiv \lambda y. (F(y) - F(x)),$$

after first generating the output

$$Export(f, \mathbf{Int}(Tan^{\leftarrow}(x))) \equiv -\lambda y. (F(y) - F(x))(0) \equiv F(x) \equiv \int_0^x f$$

Q.E.D.

4.6.2 DIFFERENTIATION

If $F(x)$ has a derivative at the c -value a , the value of $f \equiv \frac{dF}{dx}$ at a can be written

$$\lim_{(1-t) \rightarrow 0} \frac{F(a + (1-t)) - F(a)}{1-t}$$

Suppose, then, that we define the operator $\mathbf{D}: [0,1) \rightarrow \mathbf{R}^{\mathbf{R}} \rightarrow \mathbf{R}^{\mathbf{R}}$ by

$$\mathbf{D}(t)(F) \equiv \frac{F(a + (1-t)) - F(a)}{1-t}$$

Given any differentiable F , we have $\mathbf{D}(t)(F)(a) \rightarrow f(a)$, and since we're using the pointwise topology on $\mathbf{R}^{\mathbf{R}}$, it follows that $\mathbf{D}(t)(F) \rightarrow f$. By the same token, the operator $\mathbf{D}(t)$ is a continuous function of t . Consequently, \mathbf{D} is a valid arrow in a Analog $\mathbf{R}^{\mathbf{R}}$ -machine, and traversing this arrow to its terminus results in differentiation of the function stored as the fundamental datatype. The rest of the machine definition is straightforward and we leave it as an exercise.

4.6.3 UNITARY EVOLUTION

Unitary evolution lies at the heart of quantum computation, although it is not the whole story. Essentially, a quantum computation begins by applying a standard macro-process to initialise the states of various q-bits to be used in the computation proper. Unitary evolution then occurs – that is, the quantum states of the various q-bits change in such a way that they become and remain entangled – until the wave-function is made to collapse by observation into the required result. We shall deal with the hybrid nature of this 3-part process below, but for now we observe that continuous unitary evolution of the state of a q-bit can be thought of as the application of a continuously varying invertible matrix to a complex 2-vector. Each matrix along the evolution is related to the function associated with its application, whence unitary evolution corresponds to traversal of a continuous path in the space of functions $\mathbf{C}^2 \rightarrow \mathbf{C}^2$. Accordingly, the central process of quantum computation can be modelled as an AXM in the usual way.

5 General-Timed X-Machines

The key to defining any generalisation of the X-machine is the definition of arrow traversal, since this encapsulates the actual work performed by an action. In this section, we focus solely upon this issue, and ask what the unifying feature is that allows us to recognise the standard (discrete) X-machine and the AXM as variants on a common theme. This investigation leads us directly into the foothills of general topology.

5.1 Duration spaces

The AXM derived its prefix from the fact that arrows were defined as functions of the real interval $[0,1)$, while the underlying models of time in discrete X-machine models are subsets of the first transfinite ordinal ω . The discrete nature of ω makes it easy to misunderstand the role played by this timing-set in the X-machine context, but this role is much easier to elucidate in the AXM.

In each case, the ordered set allows us to represent three basic concepts: *continuity*, *duration*, and *convergence*. Continuity and duration are implicit in such analog statements as "when we are partly along the arrow, we have carried out some of the work involved in traversing that arrow", and as we shall shortly see, the same role is played by ω in the discrete models. The contribution to convergence is more subtle, and is used when we join arrows together to form diagrams – it allows us to attribute meaning to the idea that the end of one arrow coincides with the beginning of another.

These concepts resonate strongly with standard ideas from general topology, and in particular with the definition of convergence in a general topological space in terms of so-called *directed sets*. These are posets in which any two elements are bounded above. Combining the standard topological concepts with the need for an arrow to have a clearly defined initial point, we introduce the notion of a *duration space*.

Definition (Duration Spaces)

A partially ordered set $\langle Duration, \leq \rangle$ will be called a *duration space* provided

- There is a unique minimal element $0 \in Duration$.
- Given any $x, y \in Duration$, there is some z bounding both of them, $x \leq z$ and $y \leq z$.

In particular, all ordinals are duration spaces, as are $[0,1)$ and $[0,\infty)$.

5.2 Topological Issues

5.2.1 CONVERGENCE AND CAUCHY NETS

So far, we have used topological jargon without comment, because the meaning of words like "continuous" and "convergent" has always been intuitively clear. Nonetheless, we are rapidly reaching the point where formal definitions will be assumed and used throughout. The key definitions needed to understand the remainder of this paper are given in the appendices, but we shall mention a few of the concepts in the text itself, whenever this is particularly appropriate.

We remind readers of the importance of directed sets, and the nature of convergence in topological spaces. *Continuity* and *convergence* are closely related concepts, and we often see the continuity of a function described by such expressions as " $f(x_n) \rightarrow f(x)$ as $x_n \rightarrow x$ ". Such statements also illustrate the common misconception that convergence can always be defined in terms of sequences - a misunderstanding that has arisen through over-familiarity with the real line, \mathbf{R} .

A *sequence* of real numbers can be regarded as a function $\mathbf{x}: \omega \rightarrow \mathbf{R}$, where ω is just another name for the ordered set $0 < 1 < 2 < \dots$, with the convention that we typically write (say) x_n instead of $\mathbf{x}(n)$. When we say that the sequence $\langle x_n \rangle$ *converges* to the real number x , we are really saying that, no matter how small an open interval, U , we choose, if x is in U , then U will also contain some of the x_n 's. Unfortunately, not all spaces are as well-behaved as \mathbf{R} , and this definition of convergence doesn't always work. It is entirely possible for a topological space X to contain a point x and a subset A such that x is in the closure of A , but is *not* in the closure of any countable subset of A . In particular, since sequences are countable, no sequence of points in A can converge to x - despite the fact that points of A can be found "arbitrarily close to" x .

Topologically speaking, the answer is to replace ω with a general directed set Λ . Recall that Λ is directed if any two elements α, β of Λ are bounded above by a third member of Λ (which might be α or β itself). A function $\mathbf{x}: \Lambda \rightarrow X$ is called a *net* in X , and as with sequences we normally write x_α rather than $\mathbf{x}(\alpha)$, and write $\langle x_\alpha \rangle$ to denote the net in its entirety. If U is an open set in X , we say that $\langle x_\alpha \rangle$ is *eventually* in U if there is some $\lambda \in \Lambda$ such that $\alpha \geq \lambda \Rightarrow x_\alpha \in U$. The net *converges* to $x \in X$ provided $\langle x_\alpha \rangle$ is eventually in every open set containing x .

5.2.2 THE TOPOLOGY OF THE FUNDAMENTAL DATATYPE

We shall *always* assume that the fundamental datatype X carries a topology. Generally speaking, this topology will be obvious. For example, if X is an ordered space like \mathbf{N} or \mathbf{R} , the topology will be the standard order topology. Similarly, \mathbf{C} carries the standard $\mathbf{R} \times \mathbf{R}$ product topology. More generally, if X is defined as a product of subspaces, say $X \equiv X_1 \times X_2$, then we give X the product topology induced by the natural topologies on these factor spaces.

In the last resort, if no natural topology can be identified for X , we adopt the following convention. If \perp is not yet a member of X , we adjoin it to form the set $X_\perp \equiv X \cup \{\perp\}$. If X_\perp is finite, we give it the discrete topology. If X_\perp is infinite, we give X the discrete topology, and define the open neighbourhoods of \perp to be the sets of the form $X_\perp \setminus F$, where F is a finite subset of X . This makes X_\perp homeomorphic to αX , the Alexandroff compactification of X . In either case, the space X_\perp will be both compact and Hausdorff.

In general, all the spaces we are likely to meet in daily usage are Hausdorff. Occasionally, therefore, we will restrict attention to Hausdorff X 's only, but will always state clearly that we are doing so.

Whatever the topology on X , we shall always equip X^X with the pointwise topology. Convergence in this topology is defined pointwise (hence the name). That is, if $\langle f_\alpha \rangle$ is a net in X^X , then $f_\alpha \rightarrow f$ in X^X if and only if $f_\alpha(x) \rightarrow f(x)$ in X , for every $x \in X$.

Given this convention, we observe that X^X is compact and/or Hausdorff precisely when X is itself compact and/or Hausdorff.

6 Arrow Systems and Pre-Orders

Our goal in this section is to generalise the concept of *arrows* in an X-machine, and we do this by allowing the domain of the arrow to be an arbitrary duration space. In addition, we observe that any diagram comprising arrows arranged in relation to one another induces a number of important pre-orders, which reflect the temporal structure of computations that can be represented by the diagram. These will be used later to determine the nature of compatible input streams for generalised x-machine models.

6.1 Definition (arrows)

Formally, a *general-timed arrow* is a function $arrow: Duration \rightarrow X^X$ where *Duration* is a duration space.

It is often convenient to identify the duration space on which an arrow is defined, and we write $D(arrow) \equiv Duration$.

6.2 The arrow pre-order

There is a natural sense in which arrows can be said to follow one another, corresponding to the idea that the terminus of one arrow is the initial point of the next. This is, however, not the *only* way in which arrows can be ordered. For example, our AXM model of integration contained an arrow *all of whose points* were the initial points of secondary arrows, and computations followed paths which traverse only *part* of the main arrow before being diverted along a secondary arrow.

Accordingly, we define the natural pre-order on arrows to incorporate both of these relationships. Notice that this *is* a pre-order rather than an order, because the existence of loops ensures that it is entirely reasonable for two distinct arrows to follow one another. Putting these two relations together, we define the *arrow pre-order* corresponding to a given arrow system to be the smallest relation extending both *diverts* and *follows*.

6.3 The "diverts" relation

Suppose computation is proceeding along an arrow a , and is diverted along a secondary arrow b before reaching the terminus. We say that the second arrow *diverts* the computation, or that b *diverts* a . For this to be possible, the initial point of b must lie on a . That is,

$$b \text{ diverts } a \iff \exists d \in D(a) \text{ s.t. } a(d) \equiv b(0_{D(b)})$$

Notice that an arrow can easily be drawn to pass through its own initial point, whence diversion is also a pre-order rather than a partial order.

6.4 The "follows" relation

Now suppose that the "terminus" of a is the initial point of b . This concept is somewhat more subtle than diversion, because there is no guarantee that a actually has a "terminus" – just as the arrows in an AXM do not necessarily contain their endpoints. Instead, we recall that $D(a)$ is a directed set, whence a is automatically a net in X^X , and it is meaningful to talk of a converging to the point $b(0)$. Again, we have to be cautious, because an arbitrary net in a topological space may contain several convergent subnets, and these need not converge to the same limit. Nonetheless, all subsets of a *convergent* net converge to the same common limit. Accordingly, we shall say that

$$b \text{ follows } a \iff a \text{ converges in } X^X \text{ and } a \rightarrow b(0)$$

Essentially, this means that we require all "followed arrows" to be convergent nets. This is reasonable, since we can easily represent a net with multiple cluster points by drawing several arrows from the same initial point, one for each cluster point.

6.5 The computation pre-order

In addition to the arrow pre-order, we observe that any arrow system provides a prescription for merging the associated duration spaces into a composite pre-order, corresponding to all the possible temporal orderings of computation paths through the arrow system. This composite pre-order reflects both the orders of the individual duration spaces, and also the way in which those duration spaces have been "drawn" on the function space X^X . We call it the *computation pre-order*, denoted \prec .

The underlying set $Comp(Arrows)$ is just the set of all functions included in the arrow system diagram,

$$Comp(Arrows) \equiv \bigcup_{a \in Arrows} im(a) \subseteq X^X$$

Points within this set inherit their ordering from the arrow-pre-order. Suppose a and b are two arrows, and that are arbitrary positions in the associated duration spaces.

- If b follows a , we say that every function in $im(a)$ precedes every point of $im(b)$. That is,

$$b \text{ follows } a \implies \forall d_a \in D(a). \forall d_b \in D(b). (a(d_a) \prec b(d_b))$$

- If b diverts a at $d \in D(a)$, so that $a(d) \equiv b(0)$, then all points of a before the diversion point are considered to precede all points of b . The other points along a have no relationship to points of b , unless induced by some other pair of arrows. Because a might loop through the divergence point many times, we do not require d to be in any way minimal.

$$b(0) \equiv a(d) \implies \forall d_a \in D(a). \forall d_b \in D(b). ((d_a < d) \implies (a(d_a) \prec b(d_b)))$$

We shall write $Comp \equiv \langle Comp(Arrows), \prec \rangle$ to denote this pre-order.

7 General-Timed X-Machines

We are still not ready to describe everything relevant to the formal behaviour of X-machines defined with arbitrary general-timed arrows, but it is nonetheless useful to give the basic definitions at this time. These definitions refer to the "input stream" for a model, and it is clear that these streams must have quite a subtle structure, because they drive computation through a sometimes tortuous arrow-system structure. For the time being, we will simply consider what happens when an input arrives at the machine, without asking *how* that input happened to be delivered. Once the basic properties of the model are clear, we will return to a consideration of input streams.

7.1 Definition (General-Timed X-Machine (TXM))

A *general-timed X-machine* is an 8-tuple

$$TXM \equiv \langle X, Arrows, Initial, Terminal, Inputs, Import, Outputs, Export \rangle$$

where *Arrows* is an arrow system for the fundamental datatype *X*, *Inputs* and *Outputs* are spaces representing the input and output types, *Initial* and *Terminal* are sets of arrows, and

- *Import*: $Inputs \times Arrows \leftrightarrow Arrows$
- *Export*: $X \times X^X \rightarrow Outputs$

Although this definition seems very simple, the signature of *Import* may need explanation. For the purposes of this paper, we repeat that we are only interested in giving an *operational* semantics for the model. We once again rely on the input stream to determine the synchronisation properties of the model. Although it is too early to discuss the nature of input streams, it is enough for now to recall that inputs are always imported at junctions, and always at the earliest tenable junction. If no junction presents itself along the currently selected arrow, the input is assumed to be relevant to some other concurrently executing process and no action is taken. Using this protocol, the execution of a process is fully determined if we state the arrow we are currently traversing (we don't need to specify where we are *along* the arrow, because we can assume we are still at the initial point), and the new arrow whose junction we move to in response to the current input. Consequently, the signature of *Import* is simply $Inputs \times Arrows \rightarrow Arrows$.

7.2 Behaviour

As usual the behaviour is defined by taking the union of all valid path-computations through the arrow system. Suppose that we have so far reached the initial point of

arrow, and that we want to apply some action to some value x of the fundamental datatype, where the action is determined by the fact that *input* is being received.

We first identify the duration space D of the arrow a . Next we consider the set

$$J(a, D, input) \equiv \{ d \in D \setminus \{0\} \mid \exists b \in Arrows. a(d) \equiv b(0) \ \& \ Import(input, b) \text{ is defined} \}$$

```

If          J(a, D, input) has a minimum value, m, then
            computation moves to f ≡ a(m)
Else If     there is an arrow b following a and Import(input,b)
            is defined, computation moves to f ≡ b(0).
Else       nothing further happens until a new input is
            received.
    
```

Assuming we haven't terminated, we

Generate the output $Export(x, f)$ - this may be empty

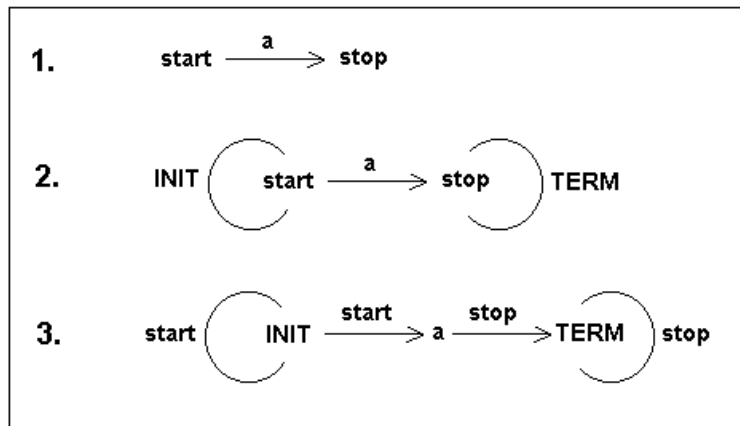
Calculate $new_x \equiv f(x)$

7.3 Simulation of X-Machines by Timed X-Machines

It's clear by construction that any AXM is a TXM, albeit a very restricted type of TXM in which every arrow has the same underlying duration space $[0,1)$. But how do we simulate standard X-machines? We cannot simply construct a simulation in which we take $States \equiv X^X$, because it is not the states that correspond to actions, but the arrows between them. To overcome this problem, we first regard the X-machine as a simple graph (corresponding to its FSM backbone), and then take the *dual* of this graph. This dual now has arrows labelled by states, and actions as nodes, exactly as required.

Let $M = \langle X, Input, Output, Encode, Decode, States, Actions, Next, Initial, Terminal \rangle$ be a standard X-machine, where we restrict the members of *Actions* to be precisely those that occur as labels of arrows in the FSM backbone of M . We first augment the diagram for X by introducing two new actions, which we call INIT and TERM (both of which are actually equivalent to the identity function on X , and adding an INIT loop around every initial state and a TERM loop around every terminal state. Even with the addition of these two new actions, the set *Actions* remains finite, and forms the state set of the dual machine. The dual-labels (*i.e.*, labels in the dual machine) are just elements of *States*. The arrow $action_0 \xrightarrow{state} action_1$ is present in the dual machine if and only if the path-fragment $\xrightarrow{action_0} state \xrightarrow{action_1}$ is present in M .

We now make this dual graph into a TXM by assigning every arrow the same duration



space, $2 \equiv \{0, 1\}$, since any dual-arrow is then a valid continuous function $\{0 \mapsto action_0, 1 \mapsto action_1\}$. Arrows emerging from INIT are initial, and arrows terminating at TERM are terminal. For the purposes of this discussion, we can think of the input stream as being a set of clock pulses, where every initial point is valid for every input. Firing these pulses causes computations to traverse one arrow at a time - this is not actually a structurally correct description of an input stream (we'll complete the required analysis in section (2.7)), but the described input stream is *observationally* correct, and the outline of the proof is essentially valid. The output stream is irrelevant.

As an aside, it is interesting to note that we have effectively summarised all analog models as TXMs in which all arrows are defined over the single duration space $[0,1)$, and all discrete models as TXMs in which arrows are defined over $\{0,1\}$. Clearly, once we start constructing models over considerably more complex duration spaces, we might hope to develop some truly remarkable computational systems.

7.3.1 THEOREM

Any X-machine behaviour can be computed as the behaviour of a TXM.

Outline of Proof: Let M be an X-machine, suitably augmented M with INIT and TERM loops as described above. Let *path* be any successful path through M. By definition, this path must be of finite total length, so we can represent it as a chain of *N* arrows for some *N*, where the first arrow emerges from an initial state, and the last arrow arrives at a terminal state. The representations of M and the dual-graph TXM contain paths represented as follows:

$$M \quad s0 \xleftarrow{INIT} s0 \xrightarrow{a1} s1 \xrightarrow{a2} \dots \xrightarrow{aN} sN \xleftarrow{TERM} sN$$

$$TXM: \quad INIT \xrightarrow{s0} a1 \xrightarrow{s1} a2 \dots aN \xrightarrow{sN} TERM$$

The clock-pulse supply of inputs, and the fact that all inputs are valid for all arrows implies that each pulse causes computation to shunt along the chain one arrow-terminus at a time. Simple induction on the numbers of arrows traversed, coupled with the definition of INIT and TERM as equivalent to the identify function on X, shows that the appropriate path-behaviour is computed. Moreover, a path of this kind can occur

in the TXM *only if* the associated path were originally present in M, so this really is a bisimulation. The machines have *identical* behaviours.

Q.E.D.

7.3.2 COROLLARY

Any Turing machine, PDA and FSM can be bisimulated by a TXM.

Proof. Immediate, since X-machines bisimulate these models.

Q.E.D.

7.4 Temporal Definition of Computation Paths

We are modelling general computation in terms of paths through arrow systems. As always, we adopt the semantic device of allowing inputs to drive synchronisation. We have so far avoided the question of how inputs are supplied to the machine. For operational purposes this is, of course, irrelevant - we only care about observable behaviours, and the unobserved evolutions of the machine while we're not probing its behaviour with inputs are deemed irrelevant. Nonetheless, it is often a useful abstraction to assume the existence of some global temporal representation *Time*, and assume that inputs arrive in a *Time*-parameterised stream,

$$InStream : Time \rightarrow Inputs_{\perp}$$

The main problem associated with this strategy is the identification of constraints upon the structure of *Time*. The only obvious features are that *Time* should be a poset, and that whenever *InStream(time)* causes an arrow-migration, $\Phi(time)$ must be a junction (*i.e.* the initial point of an initial arrow, or else of an arrow which either follows or diverts an immediately preceding arrow).

7.5 Modelling time

Because computation is driven by inputs, the only way computation can migrate from an arrow *a* to an arrow *b* is if *b* either follows or else diverts *a*. In other words, every computation path is actually a linear chain in the computation pre-order. Using this intuition, we define a poset *Time* to be a *consistent model of time* for an arrow system provided there is some order-preserving function $\Phi: Time \rightarrow Comp(Arrows)$ with the property that every chain in *Comp* that joins an initial arrow to a terminal arrow corresponds to a chain in *Time* which is actually a subset of a maximal chain in *Time* with the same property. The forward image of that maximal chain then constitutes a *computation path* indexed by that chain.

Formally, we observe that whenever *C* is a chain in *Time*, $\Phi(C)$ is a chain in *Comp*. If *C* is a chain in *Comp*, we say it is a *valid Comp-chain* provided it has both a minimum element α and a maximum element β , where α is the initial point of an initial arrow, and $\{\beta\}$ is the singleton image of a constant terminal arrow. If *C* is a chain in *Time*, we say that *C* is a *valid Time-chain* provided $\Phi(C)$ is a valid *Comp-chain*.

7.5.1 DEFINITION (CONSISTENT MODELS OF TIME)

A pair $\langle \textit{Time}, \Phi \rangle$ is a *consistent model of time* for the arrow system *Arrows* provided

- $\Phi : \textit{Time} \rightarrow \textit{Comp}(\textit{Arrows})$ is a total function
- $t_0 < t_1 \Rightarrow \Phi(t_0) \prec \Phi(t_1)$
- Every valid *Comp*-chain is the image of at least one valid *Time*-chain.
- Every valid *Time*-chain is contained within a maximal valid *Time*-chain.

7.5.2 DEFINITION (COMPUTATION PATH)

If $\langle \textit{Time}, \Phi \rangle$ is a consistent model of time for the arrow system of a TXM, and C is a maximal valid *Time*-chain, then $\Phi(C)$ is called a *computation path* in TXM.

7.6 Topics for further research

7.6.1 PROPERTIES OF *INSTREAM*

Given a computation path, we know that computation migrates between arrows in response to inputs, and that these inputs are selected by time. Consequently, the map $\Phi : \textit{Time} \rightarrow \textit{Comp}$ induces maps $\textit{Time} \rightarrow \textit{Arrows}$ and $\textit{Time} \rightarrow \textit{Junctions}_\perp$. Since both of these range spaces carry pre-orders, we can ask how the order-preserving nature of Φ must be constrained if these induced relations are also to preserve ordering. In particular, what further constraints does this impose upon the function *InStream*: $\textit{Time} \rightarrow \textit{Inputs}_\perp$?

7.6.2 COMPUTATION PATHS AND DURATION SPACES

There is a natural sense in which the concatenation of duration spaces is again a duration space. Since we can obtain all computation paths by traversing successive arrows, each computation path might reasonably be expected to have the order structure of a duration space. Is this actually the case?

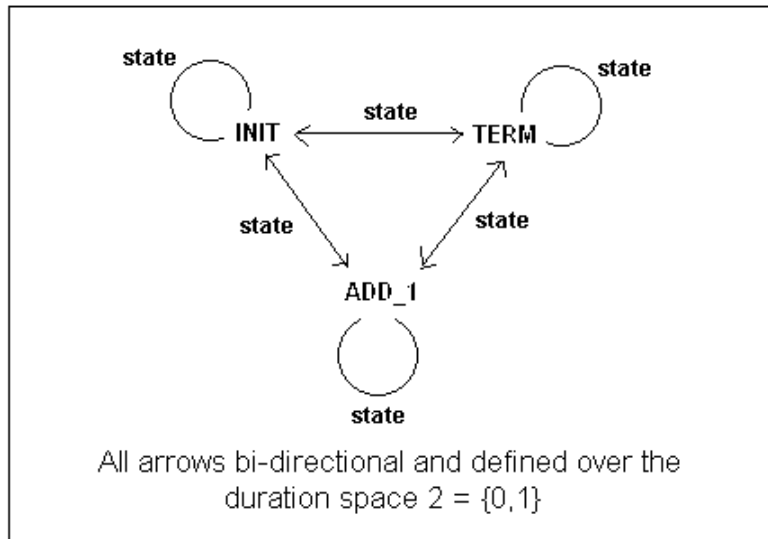
7.6.3 CANONICAL CONSISTENT MODEL?

There is a natural construction of a consistent model of time, obtained by "straightening out" all traversals of the TXM's arrow system. Is there any sense in which this model is *canonical*? For example, is there a projection from any consistent model of time onto this model (where we first restrict attention only to those elements of *Time* which lie in the range of Φ)? Is it the *smallest* consistent model, in some sense?

7.7 Every X-machine is a TXM (concluded)

7.7.1 EXAMPLE (A SIMPLE Z-MACHINE)

Consider the simple **Z**-machine of section (1.1.4) – this was a one-state machine, whose only arrow was a loop labelled by the action $n \mapsto n+1$. The state was both initial and terminal. By applying the rules of section 2.5, we construct the equivalent



TXM (shown left). Because we can construct a valid *Comp*-chain of positive length n , for every $n = 1, 2, \dots$ we need to find a poset *Time* in which each of these chains is contained within a maximal valid *Time*-chain. There are many solutions.

First, let's consider a *non-example*, where we take $Time \equiv \omega$ and define $\Phi(0) \equiv \text{INIT}$, and $\Phi(n) \equiv \text{TERM}$ for all $n > 0$. Every valid *Time*-chain is contained in the valid *Time*-chain ω , which is necessarily maximal. However, this is hardly an inspiring choice, since the machine performs no work. The problem here is that one of the criteria isn't satisfied - there are valid *Comp*-chains which cannot be represented as the image of a valid *Time* -chain.

First, let's consider a *non-example*, where we

7.8 A universal consistent model of time for X-machines

This is why I remarked that it was technically incorrect to think of the inputs as being clock pulses arriving in a stream isomorphic to ω - nonetheless, as we will now show, the description is *observationally* correct.

The model we construct, $Time_{XM}$, is *universal* in the sense that it works for *every* X-machine, although the actual function Φ has to be tailored to the machine in question.⁴

⁴ It is arguable that the loop at **TERM** can be traversed transfinitely many times, so that the union should be over all countable ordinals. We can restrict attention to *countable* ordinals on physical grounds. It is a fact that any ordinal which can be order-embedded within \mathbf{R} must be countable. While \mathbf{R} may or may not be valid as a model of physical time, it is sufficiently good an approximation that experiments rarely suggest otherwise. It is reasonable to expect that any model of physical time which subsumes \mathbf{R} will share many of the same topological properties, and the countable chain condition (on which the countability of embedded ordinals is based) is likely to be just such a property. However, extending the definition in this way subverts the definition of $InStream_{XM}$ in the next section, since we can no longer uniquely identify a meaningful finitary clock pulse. A possible solution would be to restart the clock at each limit ordinal.

First we take $Time_0$ to be a disjoint union of the ordinals $\leq \omega$

$$Time_0 \equiv \bigcup_{\alpha \leq \omega} \{\alpha\} \times \alpha$$

where (as usual) n is the n^{th} ordinal, defined by $0 \equiv \emptyset$ and $n+1 \equiv \{0, \dots, n\}$. It is necessary to include ω itself because the loop at TERM can be traversed indefinitely often without converting a valid chain into a non-valid chain. We impose an ordering on $Time_0$ by defining

$$\langle m, a \rangle < \langle n, b \rangle \quad \Leftrightarrow \quad m = n \ \& \ a < b$$

This construction allows us to define maximal valid chains of any length, because the set $\{\alpha\} \times \alpha$ is itself a maximal chain of length α .

However, we haven't yet constructed a rich enough set to cope with *any* X-machine, because we don't know how many valid *Comp*-chains there are of each length - and each such chain has to be the image of a *different* chain of the same length in *Time*. We *do* know, however, that there can be at most *finitely* many. Accordingly, we now define $Time_{XM}$ to be a countably infinite disjoint union of copies of $Time_0$,

$$Time_{XM} \equiv \bigcup_{n < \omega} \{n\} \times Time_0$$

and as before ordering is defined by

$$\langle m, a \rangle < \langle n, b \rangle \quad \Leftrightarrow \quad m = n \ \& \ a < b$$

This ensures that we have indefinitely many disjoint maximal chains of each finite length, and suitable projection Φ can be constructed for any X-machine M.

7.9 Interpretation of $Time_{XM}$ as a clock pulse sequence

Although $Time_{XM}$ is clearly very different to ω , we were nonetheless correct to state that the behaviour of TXM simulations of X-machines can be considered driven by a sequence of clock pulses. The reason lies in the *InStream* function. Recall that we define all inputs for a simulated X-machine to be valid arguments to the *Import* function. Consequently, we have enormous freedom in defining the *InStream* function, and we choose to complement the $Time_{XM}$ model of time by taking

- $Inputs_{XM} \equiv \omega$
- $InStream_{XM} : Time_{XM} \rightarrow \omega, \quad \langle m, \langle n, a \rangle \rangle \mapsto a$

No matter which of the infinitely many maximal chains we traverse, the input corresponding to the initial node is 0; the next node generates 1; then 2; and so on.

Consequently, no matter which computation path we select in *Comp*, the associated maximal chain in $Time_{XM}$ corresponds to an observed input sequence of the form 0, 1,

2, .. and so on. The input stream is *observationally* equivalent to that of a simple clock pulse.

7.10 Research topic: Universal temporal models in general

Just as the temporal model for discrete processes was somewhat more complex than might have been expected, so we cannot simply choose \mathbf{R} to represent general continuous time. Instead, we need to ensure that every bounded chain in $Time_{AXM}$ is contained in a maximal bounded chain (compatible with being a duration space), which is to say a half-closed interval. Accordingly, we take a disjoint union of all intervals of the form $[0,a)$, and then form a disjoint union of uncountably many copies the resulting ordered space, indexed by \mathbf{R} .

This is clearly a general procedure, and suggests the following exercise. Given a general duration space D , suppose that all arrows in a TXM are defined on D . Paths therefore generate concatenations of D , so the temporal model should be based on some finite, countable or even uncountable concatenation of disjoint copies of the underlying duration space. We then consider all subsets which are likely to occur as natural "maximal chains", and form the disjoint union of these sub-chains to obtain a primary "timed-space". Then we ask how many instances of each sub-chain might be needed in an arbitrary instance, and form a disjoint union of that number of "timed spaces".

Investigate the parameters of this construction. How many copies of each space are required in each disjoint union? How are the disjoint copies indexed? How does the procedure generalise to spaces with arrows on more than one type of duration space? What about the specific hybrid-computation scenario where just two duration spaces are involved: $[0,1)$ and $\{0,1\}$? Construct the universal hybrid consistent model of time, $Time_{HYBRID}$.

8 Examples: Non-Turing, Hybrid, Quantum Computational, and Uncountably-Concurrent Processes

We have already observed that the TXM bisimulates all X-machines, so that all Turing-computable processes are TXM-computable. In addition, every AXM is *de facto* a TXM, and this combined power which allows us to construct hybrid systems which are known to be *non-Turing*. Consequently, while the AXM and analog-generable functions are merely *non-Turing*, the TXM is truly *super-Turing*.

The two super-Turing examples we give are particularly simple, because the hard work was actually done by more ingenious authors some considerable time ago. In both cases they showed that certain analog processes, when applied to recursive inputs, gave non-recursive results. As we observed in [STA91], it follows that a *hybrid* system – one which first carries out the required recursive computation, and then the analog processing – can generate a non-recursive output from scratch.

The key observation is therefore that the TXM can easily model hybrid systems, something which neither the X-machine nor the AXM is capable of. This is really no more than the observation that two duration spaces can be concatenated to yield a third. Consequently, TXMs drawn in the same X^X function space can meaningfully be merged to form a coherent process which combines the two processes.

8.1.1 LEMMA (CONCATENATION OF ARROWS AND DURATION SPACES)

Suppose $a: D(a) \rightarrow X^X$ and $b: D(b) \rightarrow X^X$ are arrows, so that $D(a)$ and $D(b)$ are duration spaces, and suppose that b follows a . We can simulate the path ab by a single arrow c whose duration space $D(c)$ is just the concatenation of $D(a)$ and $D(b)$. We define $D(c)$ to be the set

$$D(c) \equiv \{a\} \times D(a) \cup \{b\} \times D(b)$$

with the ordering given by

- $\langle a, x_0 \rangle < \langle a, x_1 \rangle \iff x_0 < x_1$
- $\langle b, y_0 \rangle < \langle b, y_1 \rangle \iff y_0 < y_1$
- $\forall x \in D(a). \forall y \in D(b). \langle a, x \rangle < \langle b, y \rangle$

PROOF: We observe first that $D(c)$ has a unique least element, $\langle a, 0 \rangle$, and secondly that $D(c)$ is directed. Given $\alpha = \langle a, x_0 \rangle$ and $\beta = \langle a, x_1 \rangle$, the fact that $D(a)$ is directed shows the existence of some $m \in D(a)$ which bounds x_0 and x_1 , and now $\langle a, m \rangle$ is an upper bound for α and β . The same argument applies if α and β are both elements of $\{b\} \times D(b)$. Finally, we observe that $\max(\langle a, x \rangle, \langle b, y \rangle) \equiv \langle b, y \rangle$. This shows that $D(c)$ is a duration space. We now complete the proof by defining the arrow c in the obvious way, *viz.* $c(\langle a, x \rangle) \equiv a(x)$ and $c(\langle b, y \rangle) \equiv b(y)$.

Q.E.D.

8.2 Hybrid and Zeno computations

This tells us that hybrid and Zeno computations can be modelled very easily. If we denote concatenation of duration spaces by \oplus , it is clear that the following are all duration spaces:

$$[0,1) \oplus n, \quad [0,1) \oplus \omega \oplus [0,\infty), \quad \{0,1\} \oplus \{0,1\} \oplus [0,1) \oplus [0,2) \oplus \{0,1,2,3\} \oplus \dots$$

and it is easy to see how these duration spaces can be used as the basis for complex manoeuvres in the function space X^X . The half-closed intervals form the basis for continuous functional variations, while the ordinals allow for sequential jumps between behaviours. The actual effect of such manoeuvres depends on the way we define X , and the actual placement of the paths within function space.

Thus, if x is a continuous real variable, we can model piecewise-continuous functions of x with duration spaces of the form, *e.g.*, $[0,1) \oplus 2 \oplus [0,1) \oplus 2 \oplus [0,1) \oplus [0,1)$.

Alternatively, if X_{cts} is a set of continuous variables and X_{dis} a set of discrete variables, we can take $X \equiv X_{\text{cts}} \times X_{\text{dis}}$ and arrange for actions to be selected by arrows in such a way that discrete arrows manipulate only discrete variables and continuous arrows manipulate only continuous variables. Such an approach would, for example, provide the basis for an easy operational semantics of Duan & Holcombe's Hybrid Projection Temporal Logic [DuH97].

8.3 Non-Turing Example #1 [Myh71]

In 1971, Myhill [Myh71] published a simple construction of a function F , defined on a compact interval K , which is recursive, but whose continuous derivative f is *not* recursive. The TXM construction is a straightforward copy of the obvious hybrid system for computing f - we use a TXM-simulated X-machine to construct F , overlaid with an AXM to differentiate it, thereby generating f .

Obviously we need to identify a suitable fundamental datatype X , and there are a number of choices. The easiest solution is to select X so that F is itself of type X . Then we simply run the differentiation machine using F as the relevant data value. We already know that F is differentiable on K , so it must be continuous in K . We are told that f is also continuous, so we can take X to be the space of all continuous functions $K \rightarrow K$.

In general, we wouldn't normally construct an **R**-machine which calculates f directly. TXMs are designed to manipulate functions, and the natural structure of the fundamental datatype is that of a function space. Consequently, we typically encode constant values as functions, either by making them the bounds of an integration, or by considering them as constant functions whose image is uniformly the value in question. Nonetheless, it is feasible that a direct **R**-machine construction of f could be obtained by concatenating a variant of the general-differentiation machine to a specific TXM-simulated X-machine for computing $F(x)$. We leave this as an exercise at this time.

8.4 Non-Turing Example #2 [PoR81]

Again, there is nothing complicated about this TXM. All the hard work was done by Pour-El and Richards in their 1981 paper. They demonstrated that it is possible to set up the wave equation with computable initial conditions, set the system running, and observe the amplitude of the wave at the origin exactly one second later. Provided the initial conditions are correctly set, this amplitude will be non-computable. Again, this implies that a *hybrid* system can generate a non-computable value. First we use a TXM-simulated X-machine to construct the initial conditions, and then we use an AXM to model the evolution of the wave equation through continuous time.

8.5 Example of uncountably many simultaneous computation paths

This example is taken from my web-site [STA98]. Suppose that a small disk of magnetic fluid explodes at the origin (0,0) of the complex plane, so that the fluid spreads out as an ever widening disk of decreasing density. Assume also that a

magnetically sensitive probe has been placed at some point on the plane, and is monitoring the local field.

Assuming a symmetrical blast, any given element ∂S of fluid will follow a trajectory with constant radial angle θ . If its position before the blast was (r, θ) , it's position at time t later will be of the form $p_{r,\theta}(t) \equiv (r(t), \theta)$. In this position it causes the reading at the probe to be other than it would have been were the fluid not present. So if we write X to denote the field strength at the probe, the presence of the element at $p(t)$ can be represented by some function $\partial\Phi(p_{r,\theta}(t)) : X \rightarrow X$ representing the change caused. If the probe would have read x before, it is now reading $\partial\Phi(p_{r,\theta}(t))(x)$ instead.

But that's the effect due to this element alone. Every element of fluid is contributing to the field fluctuation at the probe. In other words, for every (r, θ) within the original disk, we have a computation path $p_{r,\theta}(t)$, and the transformation corresponding to each point along such a path is $\partial\Phi(p_{r,\theta}(t))$.

The total effect on the probe at time t will be some function

$$\Phi(t) = \iint_{r,\theta} \partial\Phi(p_{r,\theta}(t))$$

where the integral is over the entire pre-explosion disk. This computation is synchronised by the standard $Time_{AXM}$ -based input stream, which is observably equivalent to $[0, \infty)$.

8.6 Quantum computation

We have already observed that unitary evolution of entangled q-bit states can be modelled by a suitable AXM, and consequently by a TXM. Now quantum computation *per se* also requires two other processes to occur, one before unitary evolution, and one after. However, the initialisation of states required at the start of the computation is a simple matter of setting these states to a known constant (typically we'd set all the particles in a given system to *e.g.* spin-up), and can be represented by a constant arrow. The central evolutionary period terminates when the wave function collapses and observation occurs. Again, this is an essentially discrete action that converts the entangled state into a well-defined one, and can be modelled directly as a suitable *before*→*after* function defined on the duration space $\{0,1\}$. By concatenating the three process types, we generate a TXM model suitable for *any* quantum computation.

9 Summary

We have investigated an extremely powerful model of computation, the *Timed X-Machine*, and have shown that it extends the three main computational approaches currently under consideration, and thereby unifies them within a single over-arching theory. The theory of the TXM contains within itself the sub-theories of the finite

state machine, push-down automaton, Turing machine, analog generability, quantum computation, and their various hybrids.

But this is not all the model achieves. It allows the user to supply, as a parameter, whatever model of time he or she considers appropriate to the processes under investigation. It is well-known that time and space are intimately related, and that physical processes must evolve against the backdrop of the space-time in which they exist. Consequently, by allowing the specification of *time* within the TXM framework, we have gone along way to allowing the specification of *physics itself* as a parameter of the model.

In other words, whereas previous models have been updated as different models of physics and computational strategy have moved to the fore, the TXM should remain unaffected by such changes in scientific opinion. It is, in more than one sense, a truly *universal* model of computation.

10 References

- [APO78] Apostol, T. (1978) *Mathematical Analysis* (second edition). Addison-Wesley.
- [BFH+97] Bogdanov, K., Fairtlough M., Holcombe M., Ipaté, F. & C. Jordan (1997) *X-machine specification and refinement of digital devices*. Available online at <http://www.dcs.shef.ac.uk/~kirill/master7.ps>.
- [EIL74] Eilenberg, S. (1974) *Automata, Languages, and Machines*, vol. A. Academic Press.
- [DH97] Duan, Zhenhua & W.M.L. Holcombe (1997) *A Hybrid Projection Temporal Logic for Hybrid Systems*. Available online at <http://www.dcs.shef.ac.uk/~wmlh/paper/hpcomp.ps>.
- [HOL88] Holcombe, W.M.L. (1988) X-machines as a Basis for System Specification. *Soft. Eng. J.*, **3**(2), 69-76.
- [IH98] Ipaté, F. & W.M.L. Holcombe (1998) A method for refining and testing generalised machine specifications. *Intern. J. Computer Math.* **68**, 197-219.
- [LAY93] Laycock, Gilbert (1993) *The Theory and Practice of Specification Based Software Testing*. PhD Thesis, Dept of Computer Science, Sheffield University.
- [MIL89] Milner R. (1989) *Communication and Concurrency*. Prentice-Hall.

- [MYH71] Myhill J. (1971) A recursive function, defined on a compact interval and having a continuous derivative that is not recursive. *Michigan Math. J.* **18**, 97-8.
- [PR81] Pour-El, M.B., and I. Richards (1981) The wave equation with computable initial data such that its unique solution is not computable. *Advances in mathematics* **39**, 215-39.
- [POU71] Pour-El, M.B. (1971) Abstract computability versus analog-computability (a survey). *Cambridge Summer School in Mathematical Logic*, Springer Lecture Notes in Mathematics **337**, 345-60.
- [POU74] Pour-El, M.B. (1974) Abstract computability and its relation to the general purpose analog computer (some connections between logic, differential equations and analog computers). *Trans. AMS* **199**, 1-28.
- [ROG97] Rogerson, Dale (1997) *Inside COM*. Microsoft Press.
- [STA90] Stannett, Mike (1990) X-Machines and the Halting Problem: Building a super-Turing Machine. *Formal Aspects of Computing* **2**, 331-41.
- [STA91] Stannett, Mike (1991) *An introduction to Post-Newtonian and non-Turing computation*. Technical Report CS-91-02, Dept of Computer Science, Sheffield University, UK.
- [STA98] Stannett, Mike (1998) *An analog X-machine with uncountably many simultaneously executing computation paths*. [This document is no longer available in its cited form.]
- [WEI73] Weir, Alan (1973) *Lebesgue Integration and Measure*. Cambridge University Press.
- [WIL70] Willard, S (1970) *General Topology*. Addison-Wesley.