The Theory of X-Machines - Part 1

Technical Report CS-05-09

Mike Stannett
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom
`M.Stannett@dcs.shef.ac.uk`

9 February 2006

This is the first of four short reports describing the theory of X–machines. In this report we lay the foundations for the rest of the series. After a brief historical summary, we introduce the basic algebraic structure of the X–machine model, including the relationship between regular languages and X–machine computability. We show that the X–machine is essentially equal in computational power to the Turing machine, even though it is possible to construct hypercomputational X–machines with ease. The second report will extend our analysis to include Laycock's highly important *Stream X–machine* (SXM), together with the Sheffield School's *SXM Testing Methodology*, and Gheorghe's *generalised SXM with underlying distributed grammars*. The third report briefly describes the *Analog X–machine* (AXM) and its computational capabilities, before moving on to the *General-Timed* or *Temporal X–machine* (TXM). Finally, the fourth report summarises current progress in the difficult problem of encapsulating object oriented behaviour within the general X–machine framework. In particular, we discuss the relevance of types within X–machine theory.

1 Introduction

X–machines were introduced, appropriately enough, in chapter **X** of Samuel Eilenberg's 1974 textbook, *Automata, Languages and Machines* [Eil74]. The 'X' in 'X–machine' is a *type variable*; an X–machine is a machine or device for manipulating objects of type X. For example, a calculator could be described as a `float`–machine, since it lets us manipulate floating point numbers, and a lorry might be described as a `location`–machine, because it lets us manipulate where its cargo is located.

Eilenberg's original definition takes X to be an arbitrary set, which he calls the *fundamental data type* (FDT) of the X– machine. Technically, however, sets and types are different kinds of thing, and it is instructive to consider the extent to which the X– machine definition is compatible with X's being defined as an *abstract data type* (ADT). A set collects together an arbitrary collection of entities, whereas the elements of a type tend to share important properties in common. Later in this series we will examine the way X-machines can be used to model object-oriented systems, for which the distinctions between 'sets', 'types' and 'classes' are important. We will also consider the relationship of X– machines to behaviours defined by concurrent and mobile calculi, where formal notions of 'type' are again significant.

In the form originally described by Eilenberg, the X– machine has received relatively little attention. Holcombe [Hol88] introduced the model to the wider IT community, showing how the X– machine could be used as a system specification tool; his subsequent work, and that of his postgraduate students, has developed this theme considerably over subsequent years [HI98]. At around the same time, Stannett [Sta90] introduced an analog-time version of the model (the *Analog X– machine,* or AXM), and suggested that its computational power surpassed that of the Turing machine [Tur36]. He subsequently developed this theme with the introduction of the *general-timed*, or *temporal*, X– machine (TXM) [Sta01] before developing a more general interest in the field of hypercomputation [Sta06].

A major milestone in the development of X– machine applications was Gilbert Laycock's introduction of the *stream X– machine* (SXM) [Lay93]. Whereas the X– machine is based on the finite state machine (FSM), the SXM is based on the transducer (FSM with output); this distinction, though minor, has proven highly significant. For where the X– machine improves on the Turing machine by separating out the notions of *control* and *process*, the SXM goes further still; it makes the additional distinction between *memory* and *I/O*, thereby allowing the role of memory as an intermediary between control and process to be studied and exploited. In particular, since the theory of transducer testing is well established [Cho78], the relationship between transducers and SXMs can be exploited to generate a powerful theory of SXM testing [Ipa95, IH97, KEK00, BH01, IBE03, HH04]. Moreover, the transducer/translator basis of the SXM has other advantages: Gheorghe [Ghe00] has shown how to couple the control structure of the SXM with a set of formal grammars to generate a hierarchy of computational models. This approach (one of the few cases to make extensive use of label *relations* as opposed to functions) allows him to generate higher-level models from lower-level components, for example any language accepted by a Turing machine can be obtained by a suitable combination of regular and context-free systems. We describe this work in more detail later in the series.

1.1 Notation and preliminaries

The notation we use throughout this series of reports is largely standard. We write $\varnothing$ for the empty set. Given sets X and Y, we write $r\colon X \rightsquigarrow Y$ to mean that r is a relation from

X to Y, i.e. $r \subseteq X \times Y$. If r is a function and we wish to stress this fact, we instead write $r \colon X \to Y$. The set of all such relations of type $X \rightsquigarrow X$ forms a monoid under relational composition, $r \circ r' = \{(x,z) \,|\, \exists y.\, ((x,y) \in r \wedge (y,z) \in r')\}$, with the identity function $1_X = \lambda x.x$ acting as the identity element. It is called the *relational monoid* on X, denoted $R(X)$.

If $r \colon X \rightsquigarrow Y$ and w is some entity, we define $r[w] = \{y \,|(w,y) \in r\}$. In other words, if $w \in X$, $r[w]$ is the set of possible images of w under r, while $r[w] = \varnothing$ otherwise. If r is a function and we wish to stress this fact, we will write $r(x)$ for $r[x]$. Given $A \subseteq X$, we define $r[A] = \bigcup\{r[a] \,|\, a \in A\}$. We often write A^r instead of $r[A]$. Notice that this defines a monoidal action on X, i.e. $x^1 = x$ and $x^{(r_1 \circ \cdots \circ r_n)} \equiv ((x^{r_1}) \cdots)^{r_n}$.

A *labelled transition system* (LTS) is a triple $S = (\mathit{State}, A, \mathit{Next})$, where State is a nonempty set (called the *state set*), A is a non-empty set (called the *alphabet* or *label set*), and $\mathit{Next} \colon \mathit{State} \times A \rightsquigarrow \mathit{State}$ is a relation (called the *next-state relation*). As usual, we normally write $s \xrightarrow{a} t$ rather than $t \in \mathit{Next}(s,a)$, and represent an LTS diagrammatically as a labelled directed graph. If I and T are subsets of State, the triple $F = (S, I, T) \equiv (\mathit{State}, A, \mathit{Next}, I, T)$ is a *state machine*; I is the set of *initial* states and T the set of *terminal* states. If, in addition, State is finite, F is a *finite state machine* (FSM) or *automaton.* The FSM F is *deterministic* provided the next-state relation is a function; otherwise it is *nondeterministic.*

As usual, we will write A^* for the set of finite strings over a set A. Strings are represented as lists of symbols delimited by angle brackets, e.g. $\langle a_1, \ldots, a_n \rangle$. The *empty string* of type A will also be written ϵ^A, or just ϵ if the context is clear. Concatenation of two strings $\mathbf{a}$ and $\mathbf{a}'$ is written $\mathbf{a} \cdot \mathbf{a}'$.

If $p \equiv s_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n$ is a path in F, starting at some $s_0 \in I$ and ending at some $s_n \in T$, we'll call it a *valid path of length n*, and say that it *generates*, *recognises*, or *is associated with* the string $|p| \equiv a_1 \ldots a_n$. A valid path p has length *zero* (or is *trivial*) if and only if $p = s_0$ for some $s_0 \in I \cap T$. If p is trivial, then $|p| = \epsilon^A$. If we write $[\![F]\!]$ for the set of valid paths through F, then the *behaviour* of F is the set $|F| = \{|p| \,|\, p \in [\![F]\!]\}$. Equivalently, $|F|$ is the *language L recognised by F*; such languages are *regular* by definition. Write $\mathit{Reg}(A)$ for the set of all regular languages over A.

2 The Structure of X– machines

An X– machine is essentially a finite state machine whose labels are elements of $R(X)$. In an ordinary FSM, each valid path p generates a string $|p| \equiv a_1 \ldots a_n$. In an X– machine, each label a_j is replaced by a relation $r_j \in R(X)$, and the path is considered to generate the composite relation $r_1 \circ \cdots \circ r_n$. Clearly, if we want to understand the behaviour of an X– machine, we need to know two things: what is the structure of F, and what mapping $a_j \mapsto r_j$ is used to substitute relations for labels?

Definition 2.1 *An X–* machine *is a pair $M = (F, \Lambda)$ where* *a)* $F = (State, A, Next, I, T)$ *is an FSM, called the* underlying *FSM (or* underlying automaton*);*

b) $\Lambda\colon A \to R(X)$ *is a function, called the* labelling.

We typically write F^Λ *in place of* (F, Λ).

Although the labelling Λ is only defined on A, we can easily lift it to functions $\Lambda^*\colon A^* \to R(X)$, and $\Lambda^{**}\colon Reg(A) \to R(X)$. We define

$$\begin{aligned}
\Lambda^*(\epsilon^A) &= 1_A \\
\Lambda^*(\langle a_1, \ldots, a_n\rangle) &= \Lambda(a_1) \circ \ldots \Lambda(a_n) \\
\\
\Lambda^{**}(L) &= \bigcup \{\Lambda^*(\mathbf{w}) \mid \mathbf{w} \in L\}
\end{aligned}$$

All three of the sets $R(X)$, A^* and $Reg(A)$ are monoids. For A^*, we take the product of strings $\mathbf{a}$ and $\mathbf{a}'$ to be their concatenation $\mathbf{a} \cdot \mathbf{a}'$, with identity element ϵ^A. In $Reg(A)$, the identity element is the regular language $\left\{\epsilon^A\right\}$, and multiplication is given by $L \otimes L' = \{\mathbf{a} \cdot \mathbf{a}' \mid \mathbf{a} \in L \wedge \mathbf{a}' \in L'\}$.

The following properties all follow by direct calculation, but are nonetheless important.

Theorem 2.2 *Let* $\imath\colon A \to A^*$ *and* $\jmath\colon A^* \to Reg(A)$ *be the functions* $\imath(a) = \langle a\rangle$ *and* $\jmath(\mathbf{w}) = \{\mathbf{w}\}$, *where* $a \in A$ *and* $\mathbf{w} \in A^*$.

a) The function $\imath$ *is an injection of* A *into* A^**;*

b) The function $\jmath$ *is a monoid embedding of* A^* *into* $Reg(A)$*;*

c) The function Λ^* *is a monoid homomorphism, and for each* $a \in A$, *we have* $\Lambda^*(\imath(a)) = \Lambda(a)$*;*

d) The function Λ^{**} *is a monoid homomorphism, and for each* $\mathbf{w} \in A^*$, *we have* $\Lambda^{**}(\jmath(\mathbf{w})) = \Lambda^*(\mathbf{w})$. $\square$

Convention. In view of Theorem 2.2 we shall write $A \subseteq A^* \subseteq Reg(A)$, identifying A with its image under $\imath$, and A^* with its image under $\jmath$. Relative to these embeddings, we can regard Λ^{**} as an extension of Λ^*, and Λ^* as an extension of Λ. Consequently, we can safely abuse notation by writing Λ for all three functions. As before, it is sometimes convenient to use action-theoretic notation, writing e.g. $\mathbf{a}^\Lambda$ for $\Lambda^*(\mathbf{a})$ and $L^\Lambda = \Lambda^{**}(L)$. See Fig. 1.

Definition 2.3 *Let* $M = F^\Lambda$ *be an* X*–machine. If* $p \in [\![F]\!]$, *the relation* $|p|^\Lambda$ *is the* path relation *associated with* p. *The* behaviour *of* M *is the relation* $|M| = \bigcup\left\{\mathbf{w}^\Lambda \mid \mathbf{w} \in |F|\right\}$. *We say that* M computes *this relation.*

$$\begin{tikzcd}
A \arrow[r, "\imath"] \arrow[dr, "\Lambda"'] & A^* \arrow[r, "\jmath"] \arrow[d, "\Lambda^*"] & Reg(A) \arrow[dl, "\Lambda^{**}"] \\
& R(X) &
\end{tikzcd}$$

Figure 1: The relationships between $\imath$, $\jmath$, Λ, Λ^* and Λ^{**}.

3 The Computational Power of X– machines

Given an X– machine M, its behaviour is by definition a relation in $R(X)$. In order to allow computation of relations of type $Y \rightsquigarrow Z$, where $Y \neq Z$, Eilenberg allows the user to supply suitable encoding and decoding relations. There is no specific terminology associated with this construction, so we introduce the following definition and terminology.

Definition 3.1 *An* augmented *X– machine is a triple $\langle E|\, M|\, D\rangle = (M, E, D)$ where*

- *M is an X– machine;*
- *$E\colon Y \rightsquigarrow X$ is a relation, called the* encoding*;*
- *$D\colon X \rightsquigarrow Z$ is a relation, called the* decoding*.*

The behaviour *of this machine is the relation $|\langle E|\, M|\, D\rangle| = E \circ |M| \circ D$ of type $Y \rightsquigarrow Z$. We say that the machine* computes *this relation.*

There are several ways to think of the encoding and decoding relations. For example, we can think of encoding as *initialisation* and decoding as *termination.* Or we can think of encoding as *input* and decoding as *output.* Whatever the underlying intuition, since we intend investigating the computational power of the basic X– machine model, it is important that we don't accidentally 'hide' difficult computations within the encoding and decoding relations. Clearly, if we allow encodings and decodings of arbitrary power, there is no limit to the computational power of an augmented X– machine. However, even if we allow only very simple encodings and decodings, there is no limit to what an arbitrary augmented X– machine may compute.

First, however, we show the close relationship between augmented and unaugmented machines. We show that every unaugmented machine has a computationally equivalent augmented analog, so that augmented machines are at least as powerful as their unaugmented cousins.

Definition 3.2 *Given any $r\colon Y \rightsquigarrow Z$, define $r^\dagger \in R((Y \cup Z) \times Z)$ by $r^\dagger[(w, z)] = r[w] \times r[w]$. We call $r^\dagger$ the* standard simulation *of r.*

Recall that $r[w] = \varnothing$ whenever r is undefined at w. Consequently, if $r\colon Y \rightsquigarrow Z$ and $w \in Z \setminus Y$, we have $r^\dagger[(w,z)] = \varnothing \times \varnothing = \varnothing$.

Definition 3.3 *Given arbitrary sets Y and Z, define $\mathcal{D}\colon (Y \cup Z) \times Z \to Z$ by $\mathcal{D}(w,z) = z$. If $z \in Z$, define $\mathcal{E}_z\colon Y \to (Y \cup Z) \times Z$ by $\mathcal{E}_z(y) = (y,z)$. We call $\mathcal{D}$ a* standard decoding, *and $\mathcal{E}$ a (family of)* standard encoding(s). *An augmented machine whose encoding and decoding are standard will be called a* standard augment.

Theorem 3.4 *Let $M = F^\Lambda$ be an X–machine, where $X \neq \varnothing$. Define $\Lambda^\dagger\colon A \to R(X \times X)$ by $\Lambda^\dagger(a) = (\Lambda(a))^\dagger$, and let $M' = F^{(\Lambda^\dagger)}$ be the associated $(X \times X)$–machine. Given any $x \in X$, we have*

$$|\langle \mathcal{E}_x |\, M' \,| \mathcal{D} \rangle| = |M| \ .$$

Proof. Let $a_1, a_2, \ldots, a_n \in A$ and $x, x' \in X$. We show first by induction on string length that $\mathbf{w}^{(\Lambda^\dagger)}[\mathcal{E}_x[x']] = (\mathbf{w}^\Lambda)^\dagger[\mathcal{E}_x[x']]$ for all non-trivial $\mathbf{w} \in A^*$. For the base case $n = 1$, the claim holds trivially. The induction step follows by calculation:

$$\begin{aligned}
\langle a_1, a_2, \ldots, a_n \rangle^{(\Lambda^\dagger)}[\mathcal{E}_x[x']] &= \langle a_2, \ldots, a_n \rangle^{(\Lambda^\dagger)}[(a_1)^{(\Lambda^\dagger)}[\mathcal{E}_x[x']]] \\
&= (\langle a_2, \ldots, a_n \rangle^\Lambda)^\dagger[(a_1^\Lambda)^\dagger[\mathcal{E}_x[x']]] \\
&= (\langle a_2, \ldots, a_n \rangle^\Lambda)^\dagger[(a_1^\Lambda)^\dagger[(x', x)]] \\
&= (\langle a_2, \ldots, a_n \rangle^\Lambda)^\dagger[(a_1^\Lambda)[x'] \times (a_1^\Lambda)[x']] \\
&= (\langle a_2, \ldots, a_n \rangle^\Lambda)[(a_1^\Lambda)[x']] \times (\langle a_2, \ldots, a_n \rangle^\Lambda)[(a_1^\Lambda)[x']] \\
&= \langle a_1, a_2, \ldots, a_n \rangle^\Lambda[x'] \times \langle a_1, a_2, \ldots, a_n \rangle^\Lambda[x'] \\
&= (\langle a_1, a_2, \ldots, a_n \rangle^\Lambda)^\dagger[(x', x)] \\
&= (\langle a_1, a_2, \ldots, a_n \rangle^\Lambda)^\dagger[\mathcal{E}_x[x']] \ .
\end{aligned}$$

Direct calculation now shows for non-trivial $\mathbf{w}$ that

$$\begin{aligned}
\mathcal{D}\left[\mathbf{w}^{(\Lambda^\dagger)}\left[\mathcal{E}_x\left[x'\right]\right]\right] &= \mathcal{D}\left[(\mathbf{w}^\Lambda)^\dagger[\mathcal{E}_x[x']]\right] \\
&= \mathcal{D}\left[(\mathbf{w}^\Lambda)^\dagger[(x', x)]\right] \\
&= \mathcal{D}\left[(\mathbf{w}^\Lambda)[x'] \times (\mathbf{w}^\Lambda)[x']\right] \\
&= (\mathbf{w}^\Lambda)[x'] \ .
\end{aligned}$$

For the trivial case, $\mathbf{w} = \epsilon$, direct calculation also shows that $\mathcal{D}[\epsilon^{(\Lambda^\dagger)}[\mathcal{E}_x[x']]] = (\epsilon^\Lambda)[x']$. Therefore,

$$\begin{aligned}
|\langle \mathcal{E}_x |\, M' \,| \mathcal{D}\rangle|\,[x'] &= \mathcal{D}[\,|M'|[\mathcal{E}_x\,[x']]] \\
&= \mathcal{D}\left[\left(\bigcup\left\{\mathbf{w}^{\Lambda^\dagger}\,|\,\mathbf{w} \in |F|\right\}\right)[\mathcal{E}_x\,[x']]\right] \\
&= \bigcup\left\{\mathcal{D}\left[\mathbf{w}^{\Lambda^\dagger}\,[\mathcal{E}_x\,[x']]\,|\,\mathbf{w} \in |F|\right]\right\} \\
&= \bigcup\left\{\mathbf{w}^\Lambda[x']\,|\,\mathbf{w} \in |F|\right\} \\
&= |M|[x']
\end{aligned}$$

whence $|\langle \mathcal{E}_x |\, M' \,| \mathcal{D}\rangle| = |M|$ as claimed. $\square$

Now we show that in fact *every* relation can be modelled as the behaviour of a suitable augmented machine.

Theorem 3.5 *Given any relation $r\colon Y \rightsquigarrow Z$ between arbitrary sets Y and Z, there exists a standard-augmented one-arrow $((Y \cup Z) \times Z)$–machine which computes r.*

Proof. If $Z = \varnothing$, then r is also empty. We can easily find a standard-augmented one-arrow machine computing the empty relation, by taking the set of terminal states to be empty. Without loss of generality, therefore, we can assume that Z is not empty, and choose some $z \in Z$. Let M be the one-arrow $((Y \cup Z) \times Z)$–machine $i \xrightarrow{r^\dagger} t$, where we define the initial and terminal state sets in the obvious way, viz. $I = \{i\}$ and $T = \{t\}$. Now $|M| = r^\dagger$, and given any $y \in Y$ we have $|\langle \mathcal{E}_z |\, M \,| \mathcal{D}\rangle|[y] = \mathcal{D}[r^\dagger[\mathcal{E}_z[y]]] = \mathcal{D}[r^\dagger[(y,z)] = \mathcal{D}[r[y] \times r[y]] = r[y]$. So $\langle \mathcal{E}_z |\, M \,| \mathcal{D}\rangle$ is a one-arrow standard augment which computes r, as claimed. $\square$

As Theorem 3.5 shows, there is no need to introduce complicated encodings and decodings. But because the encoding and decoding relations are so simple, it also tells us that to all intents and purposes the relations r and $r^\dagger$ compute the same relations, so that augmented and unaugmented machines have essentially the same computational power.

We can also give a more intuitive proof that X–machines can simulate each of the standard machine classes familiar to theoreticians.

Theorem 3.6 *Each standard machine model (FSM, transducer, pushdown automaton, Turing machine) can be simulated by an appropriately defined X–machine.*

Proof. Each standard machine model has two basic components: set C of configurations, and a relation *next* $\in R(C)$. For an FSM, C combines the states and input strings; *next* is the function that strips the first symbol from the input string, pairs it with the current machine state, and updates the configuration accordingly. For a transducer, we add details of the current output string to the input string and state details already stored in C. For a pushdown automaton, C combines the state, input string and stack. For a Turing machine, C represents the state and tape, and *next* encapsulates the machine's program. Clearly, however, standard models of the form $(C, next)$ can be represented as C–machines. □

Conversely, we can show that the standard X–machine is no more powerful than the standard Turing machine, except insofar as the individual label-relations are hypercomputational. There are various ways in which an X–machine computation might fail to be Turing computable. For example, the elements of the set X might be uncomputable in their own right. The label relations used by the machine, or their compositions, might be uncomputable. It might not be computationally possible to form the union of the various path relations contributing to the overall behaviour. Or it might not be computationally possible to identify which relation ϕ is associated with each label a. However, these are the *only* ways in which the behaviour can be non-recursive.

Theorem 3.7 *Suppose F^Λ is an X–machine, and let $\Phi = A^\Lambda$, the set of relations occurring as labels in M. Then $|M|[x]$ can be computed by a Turing machine which has oracular access to: the elements of X, the relations in Φ, relational composition in $R(X)$, relevant set-theoretic unions, and the determination of which relation in Φ is associated with each label in A.*

Proof. Because F is an FSM, the language $L = |F|$ is regular and hence computable; in particular, we can write a program P that enumerates the strings in F as a sequence $\langle \mathbf{w}_1, \ldots \rangle$ (possibly finite, possibly with repetitions). If $\epsilon \in L$, we note that $\epsilon^\Lambda = 1_X$ is computable. Any other string $\mathbf{w} = \langle a_1, \ldots, a_n \rangle$ is of finite length, so we can certainly write a program capable of translating it into the corresponding list $\langle a_1^\Lambda, \ldots, a_n^\Lambda \rangle$ of label relations. Since relational composition is also assumed computable, we can compute each $\mathbf{w}_n^\Lambda$. Therefore, for each $x \in X$ and each $n \in \mathbb{N}$ we can compute the set $\mathbf{w}_n^\Lambda[x]$, and hence $S_n = \bigcup_{m \leq n} \mathbf{w}_n^\Lambda[x]$. Given any $(x, x') \in |M|$, we will eventually encounter some S_n containing it; therefore $|M|$ is computable by this oracle-enhanced Turing machine. □

By convention, we usually take it for granted that we can identify the particular relation ϕ associated with each label $a \in A$, and also that we can perform relational compositions computationally. Similarly, it is usually assumed that if we can compute two sets U and V, then we can also compute their union $U \cup V$. Moreover, since an X–machine is specifically intended to operate on elements of X, it is typically assumed that these elements can be represented programmatically. If we make all of these semi-standard assumptions, Theorem 3.7 tells us that X–machines are no more powerful than Turing machines.

Corollary 3.8 *Let $\Phi \subseteq R(X)$. Then the set of relations that can computed by Φ-labelled*

X– machines is precisely the set of relations computable by Turing machines with oracular access to these label relations. The two models are essentially equivalent. □

Corollary 3.8 tells us that, if each of the label relations in an X– machine M is recursive, then so is $|M|$. In other words, uncomputability cannot be generated by a fortuitious choice of X– machine *structure*. On the other hand, we have total freedom over the *processes* making up our X– machines. As theorem 3.6 reminds us, a careful (or an injudicious) choice of label relations allows us to compute *any* set-theoretic relation whatever.

3.1 Category Theory – a brief note

We finish this short primer with a brief note concerning category theoretical aspects of X– machine theory. In our proof of Theorem 3.6, we noted that many standard computational models can be described in the same way, as a pair $(C, \textit{next})$ comprising a set C of configurations, and a relation $\textit{next} \in R(C)$. This is, of course, a simplification; for example, we also need to include a projection operator on C that tells us the *values* that are considered to have been computed in each configuration. Nonetheless, it is clear that we can construct a category Mac of 'standard machine' instances, and that this category contains as objects all FSMs, transducers, pushdown automata, Turing machines and X– machines. The morphisms of this category can be chosen in many ways; perhaps the most natural choice is to consider standard notions of simulation; we might say that one machine simulates another if it can compute the same relation but is 'smaller' with respect to some complexity measure (say). Then a morphism exists from one object to another precisely when some such simulation exists. Similarly, we can think of regular languages, relations, and the like as members of a category Beh of *behaviours*. The map $|\cdot|$ can now be regarded as a functor from Mac to Beh. We can likewise regard the labelling Λ as a morphism in the category of machines, since it maps each FSM F to a corresponding X– machine M.

Theorem 3.9 *Each labelling* Λ *commutes with the functor* $|\cdot|$.

Proof. Since $M = F^\Lambda$, we have $|F^\Lambda| = |M|$. Definition 2.3 tells us that $|M| = \bigcup \left\{ \mathbf{w}^\Lambda \mid \mathbf{w} \in |F| \right\}$, or in other words $|M| = \Lambda^{**}(|F|)$. We have seen, however, that Λ^{**} is an extension of Λ; by convention, we use the same notation for both. Putting these three equations together gives the required result, viz. $|F^\Lambda| = |F|^\Lambda$. See Figure 2 on page 10. □

References

[BH01] K. Bogdanov and W.M.L. Holcombe. Statechart testing method for aircraft control systems. *Software Testing, Verification and Reliability*, 11:39–54, 2001.

$$\begin{tikzcd}
F \arrow[r, "\Lambda"] \arrow[d, "|\cdot|"'] & M \arrow[d, "|\cdot|"] \\
|F| \arrow[r, "\Lambda"'] & |M|
\end{tikzcd}$$

Figure 2: The commutativity of Λ and $|\cdot|$.

[Cho78] T. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[Eil74] S. Eilenberg. *Automata, Languges and Machines*, volume A. Academic Press, 1974.

[Ghe00] M. Gheorghe. Generalised Stream X–machines and Cooperating Distributed Grammar Systems. *Formal Aspects of Computing*, 12:459–472, 2000.

[HH04] R. Hierons and M. Harman. Testing conformance of a deterministic implementation against a non-deterministic stream X-machine. *Theoretical Computer Science*, 323(1–3):191–233, 2004.

[HI98] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution.* Springer–Verlag, Berlin, 1998.

[Hol88] W.M.L. Holcombe. X–machines as a Basis for System Specification. *Software Engineering Journal*, 3(2):69–76, 1988.

[IBE03] F. Ipate, T. Balanescu, and G. Eleftherakis. Testing communicating stream X-machines. In *1st Balkan Conference on Informatics, Thessaloniki, Greece*, pages 161–173, November 2003.

[IH97] F. Ipate and M. Holcombe. An integration testing method that is proven to find all faults. *International Journal of Computer Mathematics*, 68:159–178, 1997.

[Ipa95] F. Ipate. *Theory of X–machines with Applications in Specification and Testing.* PhD thesis, Department of Computer Science, Sheffield University, UK, 1995.

[KEK00] P. Kefelas, G. Eleftherakis, and E. Kehris. Communicating X–machines: A Practical Approach for Modular Specification of Large Systems. Technical Report CS-09/00, Department of Computer Science, CITY Liberal Studies, 13 Tsimiski Str., 54624 Thessaloniki, Greece, 2000.

[Lay93] G. Laycock. *The Theory and Practice of Specification Based Software Testing.* PhD thesis, Department of Computer Science, Sheffield University, UK, 1993.

[Sta90] M. Stannett. X-machines and the Halting Problem: Building a super-Turing machine. *Formal Aspects of Computing*, 2:331–41., 1990.

[Sta01] M. Stannett. Computation over Arbitrary Temporal Models. Technical Report CS–01–08, Department of Computer Science, Sheffield University, UK, 2001.

[Sta06] M. Stannett. Simulation Testing of Automata. *Formal Aspects of Computing*, 2006. In press.

[Tur36] A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936. Correction: Ibid. 43:544–546, 1937.