Article:
Stannett, M.P. orcid.org/0000-0002-2794-8614 (2014) Specification, Testing and Verification of Unconventional Computations using Generalised X-Machines. International Journal of General Systems, 43 (7). pp. 713-721. ISSN 0308-1079

https://doi.org/10.1080/03081079.2014.924203

To appear in the *International Journal of General Systems*
Vol. 00, No. 00, Month 201X, 1–10

Specification, Testing and Verification of Unconventional Computations using Generalised X-Machines

Mike Stannett
Verification and Testing Research Group
University of Sheffield, Department of Computer Science,
Sheffield S1 4DP, United Kingdom
m.stannett@dcs.shef.ac.uk

(*Received 00 Month 201X; final version received 00 Month 201X*)

There are as yet no fully comprehensive techniques for specifying, verifying and testing unconventional computations. In this paper we propose a generally applicable and designer-friendly specification strategy based on a generalised variant of Eilenberg's X-machine model of computation. Our approach, which extends existing approaches to SXM test-based verification, is arguably capable of modelling very general unconventional computations, and would allow implementations to be verified fully against their specifications.

1. Introduction

Unconventional computation is a relatively new research area which covers an extremely wide range of implementation strategies, including everything from slime-mould behaviours to the exploitation of cosmological singularities (Adamatzky, 2010; Stannett & Németi, 2012). Unsurprisingly, therefore, there are as yet no general techniques for specifying, verifying and testing arbitrary unconventional computers and their behaviours. This paper offers a way to address this problem, by describing how a generalised variant of Eilenberg's designer-friendly X-machine model of computation (Eilenberg, 1974) can be equipped with a *design-for-test* strategy, thereby allowing *unconventional* implementations to be verified against their intended specifications using complete finite test sets. Our approach extends, and is based upon, the well-established 'stream X-machine' testing methodology described by Ipate and Holcombe (1997).

2. Describing behaviours using X-machines

Introduced by Eilenberg (1974) some forty years ago, the X-machine is a general model of computation which allows behaviour to be specified using what are essentially just finite-state automata; this ensures that they are easy for system designers (and in particular, engineers) to understand and to use. Nonetheless, X-machines are extremely powerful, and can be used to specify any computable (or indeed, uncomputable) behaviour. Unfortunately, the original X-machine concept is inherently limited to discrete-time behaviours in which instructions are executed one after another, whereas unconventional computations often involve the evolu-

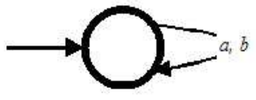


Figure 1. A single-state X-machine that recognises the non-regular language $\{a^n b^n \mid n \in \mathbb{N}\}$. In this example, we take $X = \{A, B, \mathit{Fail}\} \times \mathbb{N}$. The encoder and decoder are defined by $e(s) \equiv (A, 0)$ and $d(s, n) \equiv (s \neq \mathit{Fail} \wedge n = 0)$. The labelling is given by

$$a^\Lambda(s,n) \equiv \begin{cases} (A, n+1) & \text{if } s = A \\ (\mathit{Fail}, 0) & \text{otherwise} \end{cases} \quad ; \quad b^\Lambda(s,n) \equiv \begin{cases} (B, n-1) & \text{if } (s \neq \mathit{Fail} \wedge n > 0) \\ (\mathit{Fail}, 0) & \text{otherwise} \end{cases}$$

tion of a system in continuous time, or even the combination of analog and discrete components within a single hybrid system. We therefore begin this discussion by explaining the basic X-machine model and its uses, and then go on to describe our 'general-timed' variant of the model, which permits the use and combination of components with extremely general timing structures (Stannett, 2001).

2.1 *The X-machine model of computation*

Given some set X (called the *fundamental data type*), an X-machine is essentially a finite-state machine whose labels are relations on X – traversing an arrow in the machine diagram corresponds to evaluating the associated relation. Formally, each X-machine, M, is a pair $M = (F, \Lambda)$ where F is a finite state machine over some arbitrary alphabet A, and $\Lambda \colon A \to R(X)$ is a function (called the *labelling*) where $R(X)$ is the semigroup of relations on X (Stannett, 2006). Writing a^Λ for $\Lambda(a)$ (where $a \in A$), we can extend Λ to A^* by defining $(a_1 \ldots a_n)^\Lambda$ to be the composition $(a_1 \ldots a_n)^\Lambda = a_1^\Lambda \circ \cdots \circ a_n^\Lambda$. Writing L for the language recognised by F, this allows us to associate a *path relation* s^Λ with each $s \in L$, and we define the *behaviour*, $|M|$, of M to be the union of its path relations, viz.

$$|M| = \bigcup \{s^\Lambda \mid s \in L\} \ .$$

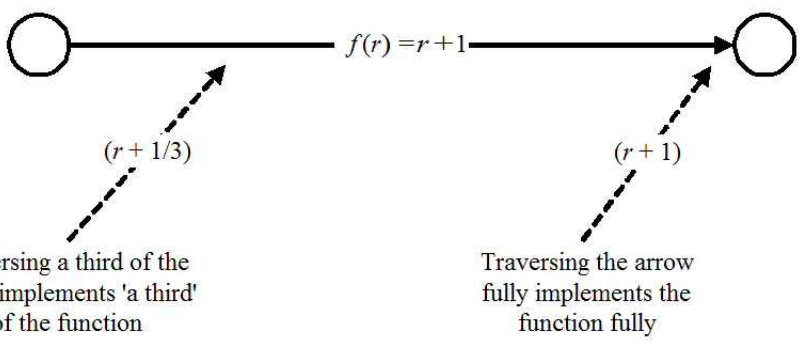
More generally, we can use an X-machine to compute relations of type $Y \to Z$ (for sets Y and Z), by applying encoding and decoding relations $e : Y \to X$, $d : X \to Z$ and evaluating $e \circ |M| \circ d$. As these definitions suggest, X-machines are closely related to finite state automata, and this relationship will be exploited in what follows.

As with finite state automata, we can use X-machines to recognise and generate formal languages, but we are not restricted to recognising only regular languages. Figure 1, for example, shows an X-machine capable of recognising the non-regular language $\{a^n b^n \mid n \in \mathbb{N}\}$. Indeed, X-machines are considerably more powerful than finite state automata, since there is no a priori restriction on the machine's labelling – technically speaking, we can label a transition with *any* relation on X, even an uncomputable one.

To simplify the following exposition, we will assume henceforth that all label-relations are in fact (possibly partial) functions, so that $a^\Lambda \in {X_\perp}^X$, the set of functions of type $X \to X_\perp$. Here, $X_\perp$ is the set obtained by adjoining a new element $\perp$ to X, and we write $a^\Lambda(x) = \perp$ to mean that $a^\Lambda(x)$ is undefined.

2.2 *The general-timed X-machine (TXM)*

Despite the evident computational power of the X-machine model, it suffers various limitations when considered as a vehicle for specifying and verifying unconventional



f(r) = r + 1
(r + 1/3)
(r + 1)
Traversing a third of the arrow implements 'a third' of the function
Traversing the arrow fully implements the function fully

Figure 2. Implementing the function $f(r) = (r + 1)$ on $\mathbb{R}$ via a continuous path $\alpha : [0, 1] \to \mathbb{R}_\perp{}^{\mathbb{R}}$, viz. $\alpha(z) = \lambda r.(r + z)$. For technical reasons, we assume that the function space $\mathbb{R}_\perp{}^{\mathbb{R}}$ carries the pointwise topology – this ensures that α is continuous, so that $\alpha(z)(r)$ converges to $\alpha(1)(r) \equiv f(r)$ as $z \to 1$ for each $r \in \mathbb{R}$.

systems. Foremost among these is the nature of its underlying timing system. Since the machine moves from state to state via discrete transitions, it is essentially impossible to provide a natural model of any system that implements analog or hybrid computation.

We can overcome this problem by using the *Timed X-machine* (TXM), a generalisation of Eilenberg's original model introduced in (Stannett, 2001). The motivation for the TXM was our observation that an X-machine could be made to model *analog* computations by a simple re-interpretation of the machine's transitions. When we picture X-machines, we typically draw a transition as an arrow labelled by the appropriate symbol α. However, we can also think of the arrow as indicating a *continuous process* by which inputs of type X are transformed. For example, if traversing some arrow in an $\mathbb{R}$-machine happens to implement the function $f(r) = r + 1$, we can think of this indirectly as a continuous operation: when we are a third of the way along the arrow we have computed $r + {}^1\!/\!_3$; at the half-way point we have computed $r + {}^1\!/\!_2$; and so on. That is, we interpret the label α as the (higher-order) function $\alpha(z) = \lambda r.(r + z)$, as shown in figure 2.

More generally, suppose $T \equiv (T, \leq)$ is any *directed set* – i.e., a partially ordered set in which every pair of elements has a supremum – containing a least element 0 and greatest element 1. Given a function $f : X \to X$, we define an *X-process for f over T* (or "T-process for f", if X is understood) to be any function $F : T \to X_\perp{}^X$ satisfying $F(0) = id_X$, the identity function on X, and $F(1) = f$. Such spaces T, equipped with their natural order topology, are what we mean by *timing structures*.

To use a T-process as the label for an arrow, we first interpret the arrow-as-drawn as a picture of T (with the source state corresponding to 0 and the target as 1), and then say that traversing the arrow corresponds to a continuous transformation from $F(0)(x) = id_X(x) = x$ to $F(1)(x) = f(x)$. For this to be mathematically meaningful, however, we first need to equip the function space $X_\perp{}^X$ with a topology, and we do so step-by-step as follows. In general, the choices of X we deal with ($\mathbb{N}$, $\mathbb{R}$, etc.) carry a natural topology which is both Hausdorff and locally compact, and if X is a product set we can equip it with the corresponding product topology. The main exception occurs where an unstructured set of values is introduced for 'book-keeping' purposes (for example, the set $\{A, B, \mathit{Fail}\}$ in Fig. 1), in which case we assign it the discrete topology in which all subsets are open. With this convention in place, we will always assume henceforward that the topology on X is Hausdorff and locally compact (as is the case for all spaces discussed in this paper).[1]

[1]The space $\mathbb{Q}$ is not locally compact. If we wish to include $\mathbb{Q}$-based computations in the scheme proposed

- If X is finite (and hence discrete, given that it is Hausdorff), we assign $X_\perp$ the discrete topology also. This ensures that $X_\perp$ is a compact Hausdorff space.
- If X is infinite, we define the open neighbourhoods of $\perp$ in $X_\perp$ to be sets of the form $X_\perp \backslash K$, where K is a compact subset of X. This makes $X_\perp$ homeomorhic to the Alexandroff one-point compactification αX of X, whence $X_\perp$ is again a compact Hausdorff space.
- Next, we assign ${X_\perp}^X$ the standard product topology, in which convergence satisfies: $\langle f_\mu \rangle \to f$ if and only if $\langle f_\mu(x) \rangle \to f(x)$ for all $x \in X$.
- Finally, we note that ${X_\perp}^X$ is itself compact Hausdorff by Tychonov's Theorem.

Technically, these definitions ensure that the T-process F is a convergent net in ${X_\perp}^X$ with unique limit f, whence it is meaningful to think of traversing an F-labelled arrow as implementing the T-continuous transformation $x \rightsquigarrow f(x)$.

2.2.1 Example: Discrete and analog computations

Every transition function f in a (standard, i.e. discrete) X-machine can be modelled as a 2-process, where we equip $2 \equiv \{0, 1\}$ with the obvious ordering and define the associated 2-process by $F(0) = id_X$, $F(1) = f$. Since 2 is discrete in the order topology, F is indeed a continuous mapping of 2 into ${X_\perp}^X$. Similarly, every transition function in an analog X-machine can be modelled as a $[0, 1]$-process.

2.2.2 Example: Stream processing

The *stream X-machine* (SXM) – an input/ouput variant of Eilenberg's original concept – was first described by Laycock (1993), and has diverse applications ranging from cellular processing (Bell & Holcombe, 1996) and swarm satellite systems (Hinchey, Rouff, Rash, & Truszkowski, 2005) to web service testing (Ma, Wu, Zhang, & Hu, 2010); as we shall see below, its properties will be key to our ability to test and verify complex unconventional behaviours. In the model, the fundamental data type X is assumed to be of the form $\mathit{In}^* \times \mathit{Mem} \times \mathit{Out}^*$, where

- In is a finite non-empty *input alphabet*;
- Out is a finite non-empty *output alphabet*;
- Mem is an arbitrary non-empty set, called the *memory.* As always, we will assume that Mem is equipped with a locally compact Hausdorff topology.

Label relations in an SXM are typically abbreviated as functions of type $\mathit{In} \times \mathit{Mem} \to \mathit{Mem} \times \mathit{Out}$, where we take $(\mathit{in}, \mathit{mem}) \mapsto (\mathit{mem}', \mathit{out})$ to be shorthand for the behaviour

$$(\mathit{in} : s, \quad \mathit{mem}, \quad t) \mapsto (s, \quad \mathit{mem}', \quad t : \mathit{out})$$

which strips the head in from the input stream (s is its tail), performs a memory update $\mathit{mem} \mapsto \mathit{mem}'$, and then appends out to the output stream t.

This concept extends immediately to generalised X-machines, although care must be taken when considering the input-output relation in isolation. The arrows in the generalised version of an SXM are labelled with functions of the form

$$\alpha : T \to (\mathit{In} \times \mathit{Mem} \times \mathit{Out})_\perp{}^{(\mathit{In} \times \mathit{Mem} \times \mathit{Out})}$$

here, we first re-interpret each function in ${\mathbb{Q}_\perp}^{\mathbb{Q}}$ as a partial function in ${\mathbb{R}_\perp}^{\mathbb{R}}$. Since $\mathbb{R}$ is locally compact, the current analysis can then be applied.

so that when we are at position $t \in T$ along the arrow, it has computed the value

$$\alpha(t)(\mathit{inStream}, \quad \mathit{mem}, \quad \mathit{outStream})$$

where *inStream* and *outStream* describe the input and output streams *when the arrow traversal began.* So when modelling continuous stream processing, for example, the model does not capture directly how $\alpha(t)$ converts the 'input at time t' into its corresponding output. This must be described using auxiliary functions, keeping track of how much of *inStream* has been processed by time t, and how this has affected *outStream* and *mem*.

2.2.3 Example: Cosmological hypercomputation

Cosmological hypercomputation concerns the use of cosmological singularities to circumvent Turing-computability limits. In certain, arguably reasonable, cosmological settings it is possible to find an observation point e which lies entirely to the future of some infinite worldline w, and this can be exploited to solve formally undecidable questions like the Halting Problem (Etesi & Németi, 2002; Stannett, 2013a, 2013b). As we now show, we can use our modelling scheme to represent the execution of a cosmological hypercomputer as the traversal of a suitably-parametrised transition arrow.

For our purposes, the key components of the system are a computer which 'runs forever' (i.e. while traversing w), together with the observation event e which lies to the future of every execution step. Since the computer's timing structure is modelled as a discrete sequence of instruction executions, its entire computation can be modelled as a function on the first infinite ordinal ω, so adding the subsequent observation event means we need to take T to be the transfinite ordinal $\omega + 1$, where the '1' (maximum element) in this case is the ordinal ω.

Formally, suppose $h : \mathbb{N} \to \mathbb{N}$ is the halting function

$$h(n) = \begin{cases} 1 & \text{if } \exists a, b \in \mathbb{N} \text{ such that } n = 2^a 3^b \text{ and } P_a(b) \text{ eventually halts} \\ 0 & \text{if } \exists a, b \in \mathbb{N} \text{ such that } n = 2^a 3^b \text{ and } P_a(b) \text{ never halts} \\ \perp & \text{if } n \neq 2^a 3^b \text{ for any } a, b \in \mathbb{N} \end{cases}$$

where P_a is the a^{th} program in some computable enumeration of deterministic one-input programs, and b is the input to be supplied. The corresponding (ω+1)-process H on $\mathbb{N}$ is then given by $H(0) = id_{\mathbb{N}}$, $H(\omega) = h$ and, for $1 \leq \iota < \omega$,

$$H(\iota)(n) = \begin{cases} 1 & \text{if } n = 2^a 3^b \text{ and } P_a(b) \text{ has halted in fewer than } \iota \text{ steps} \\ 0 & \text{if } n = 2^a 3^b \text{ and } P_a(b) \text{ is still running at step } \iota \\ \perp & \text{if } n \neq 2^a 3^b \text{ for any } a, b \in \mathbb{N} \end{cases}$$

where a 'step' refers to the execution of a single program instruction.

- Suppose $h(n) = 1$. Then $n = 2^a 3^b$ for some (unique) $a, b \in \mathbb{N}$, and $P_a(b)$ eventually halts (at step j, say). So $H(\iota)(n) = 1$ whenever $j < \iota < \omega$.
- Suppose $h(n) = 0$. Then $n = 2^a 3^b$ for some (unique) $a, b \in \mathbb{N}$, and $P_a(b)$ never halts. So $H(\iota)(n) = 0$ whenever $0 < \iota < \omega$.
- Suppose $h(n) = \perp$. Then $n \neq 2^a 3^b$ for any $a, b \in \mathbb{N}$, whence $H(\iota)(n) = \perp$ whenever $0 < \iota < \omega$.

In all three cases, we have $H(\iota)(n) \to H(\omega)(n)$ for arbitrary n, whence $H(\iota) \to H(\omega)$ in the product topology and H is indeed an (ω+1)-process for h, as required.

2.2.4 *Example: Multiply-accelerated membrane systems*

Multiply-accelerated membrane systems form a hierarchy of biologically inspired hypercomputational models $\{P_\alpha\}$, where α ranges over ordinals, extending the *accelerating P system* model of Calude and Păun (2004). The extended model allows computation of problems at all levels of the Arithmetic Hieraracy (Ash & Knight, 2000) — P_{2n}-systems solve problems in Π^0_{2n}, P_{2n+1}-systems solve those in Σ^0_{2n+1}, and there exist hyperarithmetic problems which can be solved using a P_ω-system (Gheorghe & Stannett, 2012, Theorems 3 & 5). As with cosmological hypercomputation, the additional power comes from the underlying timing structure of the computation: each P_α-system is, in essence, an $(\omega^\alpha + 1)$-process (Gheorghe & Stannett, 2012, Theorem 4).

2.3 *Hybrid systems*

Suppose we want to specify a system whose various components are based on radically different technologies. Such a specification isn't intrinsically different to that of a conventional system – we simply allow each arrow in the specification machine to be implemented using a different technology. As with standard X-machines, therefore, we can easily model such a system using a finite state machine diagram, but whereas all arrows in a standard X-machine are 2-processes, here we allow each arrow to be parameterised by a different timing structure, thereby reflecting the underlying timing structure of the intended unconventional component.

Formally, we can define a *generalised X-machine* to be a tuple

$$M = (F, A, \{T_a \mid a \in A\}, \{\alpha_a \mid a \in A\})$$

where F is a finite state machine over alphabet A, and each label $a \in A$ is interpreted as a T_a-process α_a on X, for some suitable timing structure T_a. Writing 1_a for the maximum element in T_a, and recalling that the traversal of any individual arrow $\to^a$ corresponds to a continuous T_a-computation of $\alpha_a(1_a) \in {X_\perp}^X$, we define *the X-machine associated with M* to be the X-machine

$$\overline{M} = (F, A, \{a \mapsto \alpha_a(1_a)\})$$

We take the behaviour of M to be identical to that of $\overline{M}$. This reflects the intuition that what makes different components equivalent from a functional viewpoint is *what they compute.* Their internal structure is immaterial.

3. Test-based verification of unconventional systems

Chow (1978) considered the *complete testability* of systems whose specification and implementation could both be expressed as finite state automata. Provided we have some upper bound on how many extra states are present in the implementation (the specification machine can be assumed minimal, without loss of generality), it is possible to identify a *complete finite test set* $\{\tau_1, \ldots, \tau_n\} \subset A^*$ — if the two machines behave identically on each of the test inputs τ_i, they are certain to behave identically on *every* input. In other words, it is possible to *verify* the implementation's correctness via testing.

While this is an important result, it relies on the limited computational power of finite state automata. In contrast, it is well known that no algorithm exists which can determine whether two arbitrary computable functions are equal. Nonetheless, Ipate and Holcombe (1997) showed that Chow's results could be extended to SXM behaviours (sect. 2.2.2) by applying *design-for-test* (DFT) criteria. These place restrictions on the specifications that are considered acceptable, thereby overcoming the undecidablity problem.

3.1 *Design-for-test conditions*

Let us write *Spec* for the underlying SXM specification. We regard an arrow in *Spec* as being *enabled* by a given memory/input combination if that combination is in the domain of the associated label relation – this means that traversing the arrow will yield an observable output, which we can use for testing purposes. We require *Spec* to satisfy four basic conditions.

- **Minimal Specification**
  As indicated above, the underlying automaton of *Spec* should be *minimal* – it should contain as few states as possible. This condition is easily satisfied, since there exist well-known algorithms for minimising finite state automata.
- **Deterministic Specification**
  Whatever state the machine is in, and no matter what the current memory and input, there should be at most one enabled arrow, i.e. it should be clear which arrow (if any) should be traversed next.
- **Test Completeness**
  Given any arrow in *Spec*, it should be possible to traverse that arrow regardless of the current memory state (i.e. there is at least one choice of input for which traversal of the arrow is enabled). If necessary, we can augment the machine with special test inputs.
- **Output Distinguishability**
  The only information visible to us during testing is the output stream, and it is possible in general that two different arrows, located at the same source state, might generate the same output when presented with the same memory/input combination. For testing purposes we need to know which arrow was traversed, so we require that this situation never arises. It should always be possible to distinguish, using just the output information, which label relation was involved, for all memory-input pairs. If necessary, we can augment the machine with special test outputs.

3.2 *Generalised DFT conditions*

Given that these DFT conditions are satisfied, Ipate and Holcombe (1997) show that Chow's result extends to behaviours whose specification and implementations are described as stream X-machines, and since the behaviour of any generalised SXM is identical to that of the associated (standard) SXM, the same will be true of generalised SXMs also – *provided we can identify the appropriate generalisation of the underlying DFT conditions*. If we can do so, the verifiability of systems specified using appropriate generalised X-machines – including systems comprising interconnected unconventional computational components – will be assured, since the formal correctness of such systems can then be determined simply by carrying out a *finite* collection of clearly identifiable tests.

Our work in this area is still continuing, but some indications can already be provided.

3.2.1 Minimal Specification

This condition is again easily satisfied, since it only concerns the underlying automaton rather than the detailed machine construction.

3.2.2 Deterministic Specification and Test Completeness

These conditions say what should be true at states within *Spec*; in other words, they tell us what should be true concerning the specific unconventional components before they begin execution. For standard SXM testing, we can, if we wish, augment *Spec* by adding specific test inputs, but there is no guarantee that this concept will be meaningful when we move to the specific repertoires of unconventional components. For example, suppose we decide to implement an arrow using a comprehensive range of behaviours available to some *Physarum*-based system – in what way can we 'add' an extra test input to the system? We cannot necessarily *force* biological species to reliably respond in ways that are alien to their innate behaviours. This suggests that, under our proposed scheme, unconventional systems will need to have built-in inefficiencies. We need to ensure that enough distinguishable behaviours are left unexploited to enable the identification of additional test inputs and outputs as and when needed.

3.2.3 Output Distinguishability

When we refer to the 'output' in this DFT condition, we mean the output generated once an arrow traversal has been completed – but it is entirely possible for T-processes to generate continuous output streams. If we insist on using on-completion values to identify which arrow has been traversed, it will be necessary to recognise when traversal of an arrow has run to completion, and there is no obvious general mechanism by which this can be ensured.

This needs considerable further investigation, since there is clearly a danger of re-introducing undecidability limitations to the methodology.

3.2.4 Translation Correctness

This is a new DFT condition not required in the standard SXM testing methodology. One of the defining properties of a computational output is that it can be used as an input by some subsequent computation. Traditional computational schemes involving standardised sets of components cope with this requirement automatically, but if we are using a heterogeneous collection of unconventional components based on widely differing technologies, we obvious need to ensure that outputs can be 'translated' into inputs of the correct form wherever arrows meet.

Unfortunately, this auxiliary processing is entirely hidden within the proposed model, since behaviours occurring at the the junctions in question (the machine's states) are not addressed by the SXM testing methodology. Once again, further investigation is required here, since the correctness of an implementation obviously relies crucially upon the correctness of the embedded translation mechanisms.

4. Summary

In this paper we have described a generalised version of Eilenberg's X-machine model of computation, and shown how it can be used to model a range of unconventional and hypercomputational behaviours exhibiting very different timing structures. The model allows for hybrid specifications, in which different machine transitions are implemented using components based on different timing structures;

this means that each transition can be implemented using whichever unconventional (or conventional) technology we consider most suited to the task at hand – as long as components produce the same on-completion behaviour, they can be used interchangeably.

We then considered SXM-based systems, in which we augment processes with specific input/output mechanisms. This is a promising technology, because – provided certain design-for-test conditions are satisfied – it is possible to determine the formal correctness of an implementation vis-à-vis its specification simply by examining the two systems' behaviours over a *finite* set of test cases. If they behave identically for these finitely many cases, they are guaranteed to behave correctly in *all* cases.

The case is not yet proven, however, since it is clear that the standard SXM design-for-test conditions need careful revision before we can be certain that these complete-testability results extend to include generalised unconventional computations. In particular, the model does not fully model the procedures involved in translating the outputs of one unconventional process into a form that can be processed by a second, radically different, such process. While we could simply impose correct-translation as an additional DFT condition, it would clearly be preferable to incorporate the translation processes within the model, so as to ensure that they, also, can be fully verified.

Acknowledgments

Our research is partially supported under the Royal Society International Exchanges Scheme (ref. IE110369).

References

Adamatzky, A. (2010). *Physarum machines: Computers from slime mould.* Singapore: World Scientific.

Ash, C. J., & Knight, J. F. (2000). *Computable structures and the hyperarithmetical hierarchy.* Elsevier.

Bell, A., & Holcombe, M. (1996). Computational models of cellular processing. In M. Holcombe, R. Paton, & R. Cuthbertson (Eds.), *Computation in cellular and molecular biological systems.* Singapore: World Scientific Press.

Calude, C., & Păun, G. (2004). Bio-steps beyond Turing. *BioSystems*, *77*, 175–194.

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

Eilenberg, S. (1974). *Automata, languages and machines, vol. a.* London: Academic Press.

Etesi, G., & Németi, I. (2002). Non-Turing Computations via Malament-Hogarth space-times. *International Journal of Theoretical Physics*, *41*, 341–70.

Gheorghe, M., & Stannett, M. (2012). Membrane system models for super-Turing paradigms. *Natural Computing*, *11*, 253–259.

Hinchey, M. G., Rouff, C. A., Rash, J. L., & Truszkowski, W. F. (2005). Requirements of an Integrated Formal Method for Intelligent Swarms. In *Proceedings of fmics'05, september 56, 2005, lisbon, portugal* (pp. 125–133). Association for Computing Machinery.

Ipate, F., & Holcombe, W. (1997). An integration testing method that is proved to find all faults. *Int. J. Computer Mathematics*, *63*, 159–178.

Laycock, G. (1993). *The Theory and Practice of Specification Based Software Testing.* Unpublished doctoral dissertation, Department of Computer Science, University of Sheffield, UK.

Ma, C., Wu, J., Zhang, T., & Hu, F. (2010). Web services testing based on stream x-machine. In J. Wang, W. K. Chan, & F.-C. Kuo (Eds.), *Qsic* (pp. 232–239). IEEE Computer Society.

Stannett, M. (2001). *Computation over arbitrary models of time* (Tech. Rep. No. CS-01-08). Sheffield, UK: Dept of Computer Science, University of Sheffield.

Stannett, M. (2006). *The Theory of X-Machines – Part 1* (Tech. Rep. No. CS-05-09). Sheffield, UK: Dept of Computer Science, University of Sheffield.

Stannett, M. (2013a). Computation and Spacetime Structure. *International Journal of Unconventional Computing*, *9*(1–2), 173–184.

Stannett, M. (2013b). Membrane systems and hypercomputation. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, & G. Vaszil (Eds.), *Membrane computing* (Vol. 7762, pp. 78–87). Berlin Heidelberg: Springer.

Stannett, M., & Németi, I. (2012). *Using Isabelle to verify special relativity, with application to hypercomputation theory.* Online: `arXiv:1211.6468`.