Integration testing of heterotic systems

M. Stannett\textsuperscript{1} and M. Gheorghe\textsuperscript{2}

\textsuperscript{1}Department of Computer Science, University of Sheffield, Sheffield S10 2TN, UK
\textsuperscript{2}Department of Computing, University of Bradford, Bradford BD7 1DP, UK

Computational theory and practice generally focus on single-paradigm systems, but relatively little is known about how best to combine components based on radically different approaches (e.g. silicon chips and wetware) into a single coherent system. In particular, while testing strategies for single-technology artefacts are generally well developed, it is unclear at present how to perform integration testing on heterotic systems: can we develop a test-set generation strategy for checking whether specified behaviours emerge (and unwanted behaviours do not) when components based on radically different technologies are combined within a single system? In this paper, we describe an approach to modelling multi-technology heterotic systems using a general-purpose formal specification strategy based on Eilenberg’s X-machine model of computation. We show how this approach can be used to represent disparate technologies within a single framework, and propose a strategy for using these formal models for automatic heterotic test-set generation. We illustrate our approach by showing how to derive a test set for a heterotic system combining an X-machine-based device with a cell-based P system (membrane system).

1. Introduction

A central problem when building a computational system is establishing whether it meets its specification—does the system do what is required of it? Two complementary approaches to solving this problem are formal methods and empirical testing.

Formalists generally start with the construction of a mathematical specification of the system’s desired behaviour. Where cost is a factor, the specification may simply be used to guide informal program construction,
but where correctness is paramount, for example in safety- or mission-critical systems, the specification may instead be transformed (‘refined’) step-by-step into the specification of a physically implementable system whose behaviour matches what is required. In 2014, for example, more than 25% of the world’s automatic metro-train systems included embedded software developed using Atelier B [1], a commercial tool system for deploying Abrial’s refinement-based B-method [2]. Central to the effectiveness of such methods is the notion of formal verification, the idea that each refinement step can be formally proven to preserve correctness. Since the initial specification is a priori deemed correct, verifying each of the finitely many refinement steps logically ensures that the generated physical implementation will also be correct.

By contrast, the test-based approach focuses on identifying input sequences (‘test cases’) that can be supplied to the completed system so that analysis of the corresponding outputs would reveal the presence of errors. In general, we can identify three testing stages, corresponding to the idea that a computational system generally comprises a collection of interconnected sub-assemblies (‘design items’), each of which is constructed from basic components. Unit testing investigates the behaviour of each basic component in isolation; this allows problems to be identified early in the development cycle, with corresponding benefits in terms of costs and repair times. Integration testing checks whether the components within each design item interact correctly, under the assumption that the components themselves are behaving correctly. System testing checks whether the complete system, formed by interconnecting the design items, meets its original specification [3,4].

The two approaches should not be seen as mutually exclusive, but rather complementary approaches to establishing system correctness. Indeed they can be combined to give formal specification-based approaches to test-set generation [5]. For example, one might derive both the implementation and its test set from the specification at the same time, with each refinement step being accompanied by an associated test-generation step [6,7].

In this paper, we consider systems whose sub-assemblies combine components based on radically disparate technologies. Rather than specify in advance which technologies we are willing to consider, we assume that the component manufacturers working in each technology have already devised adequate methods for testing their products. Our focus, therefore, is on integration testing of the sub-assemblies made by combining those components. But before explaining the special difficulties associated with this task, we first need to discuss the properties of mixed-technology computing systems more generally.

Modern technologies allow computation to be defined and implemented relative to a wide variety of paradigms and physical substrates, and it is natural to ask whether any advantage is to be gained by combining components based on radically different technologies to form a heterotic system. Stepney et al. [8] describe several instances of this idea, which at its most basic involves a system $\mathcal{H}$ comprising two interacting components, Base and Control. The two components, possibly based on different computing paradigms, interact in a step-by-step manner. At each stage, the Base component performs an action, thereby generating an output. This is interpreted by Control, which then tells Base what action to perform next.

The computational power of heterotic systems has been studied for many years. Towards the end of the twentieth century, Siegelmann showed that no analogue device computing in polynomial time can compute more than the non-uniform complexity class $\text{P/poly}$ [9], while Bournez & Cosnard [10] had previously argued that an idealized hybrid analogue/discrete dynamical system could in principle achieve this bound. More recent analyses by Tucker, Beggs and Costa have described a series of models that use experimental systems (Base) as oracles providing data to an otherwise computable algorithm (Control)—the Control layer observes the outcome of each Base-level experiment, and uses this information to reconfigure Base prior to the next experiment [11]. Their results show that ‘interesting and plausible’ model systems can, in principle, compute the smaller non-uniform complexity class $\text{P}/\log^*$, and they postulate [12, p. 872] that this is essentially an upper limit for efficient real-world computation (‘physical systems combined with algorithms cannot compute more in polynomial time than $\text{P}/\log^*$’).
Kendon et al. [13] have likewise pointed to the work of Anders & Browne [14], who observed that the combination of (efficiently) classically simulable Control and Base layers in a quantum cluster state computer results in a model which cannot be simulated efficiently. This implies that the interactions between two layers in a heterotic computer can contribute fundamentally to the power of the combined system, and this in turn has important consequences for anyone interested in the practicalities of integration testing of such systems, since it confirms that the correctness of a heterotic system’s behaviour cannot be assessed simply by examining the behaviours of its various components in isolation.

While the components’ correctness is obviously important, what Anders and Browne’s example shows is that important aspects of a heterotic system’s behaviour may depend less on the components per se than on the intricate choreography of their interactions. Consequently, even if we can generate test cases for the individual components within a heterotic system, it is unclear whether we could easily compose these to generate integration tests for the sub-assemblies in which they are used.

**Contribution and outline of the paper.** In this paper, we focus on the complex question of integration testing for heterotic systems: how can we test the sub-assembly obtained by combining Base and Control, given that unit tests are already available for these components taken in isolation?

Because the underlying technologies of Base and Control are undetermined, we will think of components in purely functional terms. That is, we consider which outputs are associated with which inputs but do not concern ourselves with the underlying physical processes leading to this transformation. At the same time, we can think of the flow of control from one component to the next in terms of a finite state machine (FSM), which identifies which component(s) should be in control at start-up and termination, and the way in which each component passes control to the next during system execution. This combination—a finite state machine in which traversing a transition arrow corresponds to executing a function (on some underlying data type X)—is known in the literature as an X-machine [15], and its simplicity lends itself well to integration test-set generation [16]. The stream X-machine testing (SXMT) test strategy is unusual in that it is capable of generating complete test sets, a feature that makes it ideal as our starting-point, since it allows us to determine by finite means (albeit subject to various constraints chosen to avoid undecidability problems) whether or not a system is behaviourally equivalent to its specification.

We illustrate our approach with a hybrid example drawn from the bioinspired area of P systems (membrane systems). The P system is a nature-inspired unconventional computing paradigm; while it draws its inspiration from (somewhat idealized) principles of cell biology, it is nonetheless known that there are strong connections between this formalism and other, more obviously mathematical, computational paradigms—examples include process calculi, brane calculi, Petri nets, L systems and cellular automata [17]. At the same time, applications in synthetic biology demonstrate the P system’s emerging capabilities with regard to modelling and specifying real-world biological behaviours [18].

In §2, we provide a review of X-machine testing strategies, which form the basis of our approach. In particular, we explain what a system of communicating stream X-machines (CSXMS) is, and how such a system can be tested. In §3, we show how the CSXMS approach can be used to model and generate a test set for a heterotic system combining a stream X-machine (SXM; Control) and a P system (Base). To make this example accessible to readers, we first describe the biologically based P system model in detail, and demonstrate how P system behaviours can themselves be unit tested. For simplicity, the proof-of-concept described below involves a basic deterministic system, but strategies also exist for non-deterministic X-machine testing [19], which could be extended in a similar way.

In §4, we identify shortcomings of our CSXMS testing approach and discuss ongoing research into extending the underlying theory accordingly. The use of X-machine specifications is especially useful for ‘model-based’ testing purposes [20] but is ultimately inadequate for our purposes in its current form. This is because transitions in an X-machine are assumed to take place discretely, one after another, whereas we are interested in hybrid machines whose components
may exhibit completely unrelated timing structures. For example, *Control* might be a traditional discrete-time digital system, while *Base* might be an analogue component. To overcome this problem, we propose a new and extended variant of the X-machine testing strategy, which is capable of testing computational behaviours defined over arbitrary models of time. We suggest in particular how a generalized theory of X-machine testing can be defined, which can be applied to heterotic systems in which the timing structures implicit in the system’s behaviour are radically more complicated than allowed by existing approaches.

Section 5 concludes the paper and outlines suggestions for theoretical and experimental research towards validating the approach.

### 2. The X-machine testing methodology

In this section, we introduce the basic concepts of the SXM and communicating SXM (CSXM) and describe what it means for an interacting collection of such machines to form a system (CSXMS). We explain what we mean by testing such a system and summarize the existing approach to SXM testing described in [16,21]. Finally, we discuss a testing strategy for communicating SXM systems derived from the SXM testing methodology. For simplicity, we only describe the procedures associated with testing deterministic machines, but a similar approach can also be developed for non-deterministic behaviours [19].

SXMs were introduced by Laycock [22] as a variant of Eilenberg’s *X-machine* model of computation [15], and we have recently described elsewhere how a generalized form of Eilenberg’s original concept might be used to describe hybrid systems of unconventional computations [23,24]. Our goal here is to expand on that description by showing in detail how the use of these models supports the identification of behavioural test sets.

**Notation.** Throughout this paper, we write ∅ for the empty set and $\mathbb{R}$ for the set of real numbers equipped with its standard algebraic and topological structures. Each natural number is interpreted to be the set of its predecessors, i.e. $0 \equiv \emptyset$, $n + 1 \equiv \{0, 1, \ldots, n\}$. In particular, we have $2 = \{0, 1\}$.

If $X$ and $Y$ are sets, the set of total functions from $X$ to $Y$ is denoted $Y^X$. The domain of a function $f$ is denoted $\text{dom}(f)$. Since each subset $S$ of $X$ can be identified in terms of its characteristic function $\chi_S : X \to 2$, we write $2^X$ for the set of subsets of $X$ (the power set of $X$).

Given any set $X$, we define $X_\perp = X \cup \{\perp\}$ where $\perp \not\in X$ is interpreted to mean ‘the undefined element of type $X$’. If ambiguity might otherwise arise, we write $\perp_X$ to indicate the set with which $\perp$ is associated. However, for historical reasons the ‘undefined memory’ value (below) is generally called $\lambda$ instead of $\perp_{\text{Mem}}$.

Given any alphabet $A$, we assume the existence of a symbol $\text{null} \not\in A$, with the property that prepending or appending $\text{null}$ to any string in $A^*$ leaves that string unchanged, and likewise, if a variable $x$ is of type $A$, then the assignment $x := \text{null}$ leaves the value of $x$ unchanged.

#### (a) Stream X-machines

We recall the definition of an SXM and some related concepts from [21].

**Definition 2.1.** An SXM is a tuple

$$\mathcal{P} = (\text{In}, \text{Out}, Q, \text{Mem}, \text{Procs}, \text{Start}, \text{Stop}, m_0, \text{Next}),$$

where

- $\text{In}$ and $\text{Out}$ are finite non-empty sets called the *input alphabet* and *output alphabet*, respectively, and $Q$ is a finite non-empty set of *states*; $\text{Start} \subseteq Q$ is the set of *initial states* and $\text{Stop} \subseteq Q$ is the set of *terminal states*;
- $\text{Mem}$ is a (possibly infinite) non-empty set of *memory values*, and $m_0 \in \text{Mem}$ is the *initial memory*;
— \textsc{Procs} is a finite set of \textit{processing functions}. Each of these is of type $\text{Mem} \times \text{ln} \rightarrow \text{Out} \times \text{Mem}$; and
— $\text{Next} : Q \times \procs \rightarrow 2^Q$ is a partial function, called the \textit{next-state function}.

Intuitively, a stream X-machine can be regarded as a finite state machine $A$, equipped with
transitions triggered by $\text{Next}$ and carrying labels of the form $o/\varphi/i$, where $i \in \text{ln}$, $o \in \text{Out}$ and $\varphi \in \procs$. Traversing such a transition is interpreted as consuming the input symbol $i$, updating the current memory from $m$ (say) to $\varphi(m)$ and producing the output symbol $o$. We call $A$ the automaton associated with $P$. This process is \textit{deterministic} if $\text{Start}$ contains just one element and $\text{Next}$ maps each state and processing function label onto at most one state, i.e. $\text{Next}$ can be regarded as a function $\text{Next} : Q \times \procs \rightarrow Q$. A \textit{configuration} of an SXM is a tuple $(m, q, \sigma, \gamma)$, where $m \in \text{Mem}$, $q \in Q$, $\sigma \in \text{ln}^*$ and $\gamma \in \text{Out}^*$. It represents the idea that the machine is currently in state $q$, the memory is currently $m$, the machine’s remaining input stream is $\sigma$, and it has so far produced the output stream $\gamma$. An \textit{initial} configuration is one in which $m = m^0$, $q \in \text{Start}$ and $\gamma = \epsilon$ (the empty sequence).

A \textit{final} configuration has $q \in \text{Stop}$ and $\sigma = \epsilon$.

We say that a \textit{configuration change} $(m, q, \sigma, \gamma) \vdash (m', q', \sigma', \gamma')$ can occur provided

— $\sigma = i\sigma'$ for some $i \in \text{ln}$;
— $\gamma' = \gamma o$ for some $o \in \text{Out}$; and
— there exists some $\varphi \in \procs$ with $q' \in \text{Next}(q, \varphi)$ and $\varphi(m, i) = (o, m')$.

The reflexive and transitive closure of $\vdash$ is denoted $\vdash^*$.

The \textit{relation computed} by an SXM $M$ is the relation $\llbracket M \rrbracket : \text{ln}^* \rightarrow \text{Out}^*$ defined by

$$\text{Next}(q, \varphi) \in M \quad \text{iff there exist} \quad \text{Start} \in \text{Start}, \text{Stop} \in \text{Stop} \text{and } m \in \text{Mem} \quad \text{such that}$$

$$(m^0, \text{Start}, \sigma, \epsilon) \vdash^*(m, \text{Stop}, \epsilon, \gamma).$$

\textbf{(b) Communicating stream X-machine systems}

We introduce, loosely following [25], a simplified definition of communicating stream X-machines (CSXM) and CSXM systems. A CSXM can be thought of as an SXM equipped with one input port ($\text{IN}$) and one output port ($\text{OUT}$). In a standard SXM, the next action of the machine in any given state (i.e. the processing function to be applied) is determined by the current input and current memory value. In a communicating SXM, we also allow the machine to take into account the value, if any, currently present on the input port. The machine can also enter various special \textit{communicating} states, in which it transfers a memory value from its output port to the input port of another machine. This enables the various machines to exchange memory values as and when required, thereby allowing them to coordinate shared computations.

Notice that the input alphabet of a component machine $\Pi_i$ (the values which, together with its current memory, determine its behaviour) is a set of pairs, each describing the current input symbol and input port symbol (i.e. $\text{IN}_i \times \text{IN}_i$).\footnote{These definitions of $\Pi_i$’s input and outputs are technically only valid if each $\text{IN}_i$ and $\text{OUT}_i$ can be assumed finite. While we can rewrite the definition of a CSXM in a more rigorous, but more complicated, form to ensure that all input and output alphabets remain finite without regard to $\text{IN}_i$ and $\text{OUT}_i$, this is unnecessary for our purposes [25].} The result of firing a transition is more complex—in addition to updating local memory the outcome can affect the local output stream, input port and output port, as well as the input port of any other machine in the system. Consequently, we take the output type to be $\text{OUT}_i \times \text{OUT}_i \times \prod_{m=1}^{n} \text{IN}_m$.

\textbf{Definition 2.2.} A \textit{communicating stream X-machine system} (CSXMS) with $n$ components is an $n$-tuple $\mathcal{P}_n = (\Pi_1, \ldots, \Pi_n)$, where each $\Pi_i$ is a communicating SXM (CSXM), i.e. an SXM with input alphabet $\text{IN}_i \times \text{IN}_i$ and output alphabet $\text{OUT}_i \times \text{OUT}_i \times \prod_{m=1}^{n} \text{IN}_m$, where (writing $\text{Mem}_i$ for the memory of $\Pi_i$, and similarly for its other components):

— both $\text{IN}_i$ and $\text{OUT}_i$ are subsets of $(\text{Mem}_i)_{\perp}$.\footnote{These definitions of $\Pi_i$’s input and outputs are technically only valid if each $\text{IN}_i$ and $\text{OUT}_i$ can be assumed finite. While we can rewrite the definition of a CSXM in a more rigorous, but more complicated, form to ensure that all input and output alphabets remain finite without regard to $\text{IN}_i$ and $\text{OUT}_i$, this is unnecessary for our purposes [25].}
— $Q_i$ can be written as a disjoint union $Q_i = Q_i' \cup Q_i''$, where the elements of $Q_i'$ are called ordinary states and those of $Q_i''$ are communicating states;
— $\text{Procs}_i$ can be written as a disjoint union $\text{Procs}_i = \text{Procs}_i' \cup \text{Procs}_i''$, where the elements of $\text{Procs}_i'$ are called ordinary functions and those of $\text{Procs}_i''$ are communicating functions;
— The next-state function, $\text{Next}_i$, is undefined except on $(Q_i' \times \text{Procs}_i') \cup (Q_i'' \times \text{Procs}_i'')$, and $\text{Next}_i(q_i', \psi') \subseteq Q_i'$ for all $q_i' \in Q_i'$, $\psi' \in \text{Procs}_i'$, i.e. ordinary states support ordinary functions, communicating states support communicating function, and the target state of a communicating function is always an ordinary state.

**Configurations and configuration changes in a CSXMS.** A configuration of a component CSXM $\Pi_i$ is a tuple $c_i = (in, q_i, \sigma_i, \gamma_i, in, out)$, where $in \in \mathbb{N}_i$, out $\in \mathbb{O}_i$, and the other entries are defined as before. Given a CSXMS $\mathcal{P}_n = (\Pi_1, \ldots, \Pi_n)$, we define a configuration of $\mathcal{P}_n$ to be a tuple $(c_1, \ldots, c_n)$ where each $c_i$ is a configuration of the corresponding $\Pi_i$. A configuration $(c_1, \ldots, c_n)$ is initial provided each $c_i$ is initial (including the requirement that $in = out = \lambda$, so that the first move made by the machine must be ordinary).

There are two ways in which a CSXMS can change its configuration. An ordinary configuration change is one that causes no communication between machines; each machine can either consume and process a symbol present on the input channel, or it can leave the input channel untouched. We model this second case by saying that it consumes the undefined $\lambda$ symbol. A communicating configuration change is one in which a symbol is removed from one machine’s output port and a corresponding symbol is inserted into a second machine’s input port, provided it is currently empty.

**Definition 2.3.** A configuration change $(c_1, \ldots, c_n) \vdash (c_1', \ldots, c_n')$ is ordinary if there is some $i$ such that $c_j' = c_j$ for all $j \neq i$, and some ordinary function $\psi' \in \text{Procs}_i'$ with

- $q_i' \in \text{Next}_i(q_i, i, in)$,
- out$'$ $\in \mathbb{O}_i$, and either
  - $\psi'(i, in, m) = (m', o', out', (in'_m)_{m=1}^n)$, where in $\neq \lambda$ and in$'$ $= \lambda$;
  - $\psi'(i, \lambda, m) = (m', o', out', (in'_m)_{m=1}^n)$ and in$'$ $= in$.

It is communicating if there exists some $i \neq k$ such that

- $c_j' = c_j$ for all $j \notin \{i, k\}$,
- out$_k$ $= out'_i$ $= \lambda$,
- $q_i' \in Q_i'$ (the next state in the sending machine is ordinary),
- $\sigma_i' = \sigma_i$, $\gamma_i' = \gamma_i$, $\sigma_k' = \sigma_k$, $\gamma_k' = \gamma_k$ (all input and output streams are unchanged)
- there exists some communicating function $\psi'' \in \text{Procs}_i''$ which can be applied in the current state, and which generates the symbol that appears in the target machine’s input port, i.e.
  - $q_i' \in \text{Next}_i(q_{ij}, i, in_i)$, and
  - $\psi'(i, in_i, m_i) = (m_i', \text{null}, \lambda, (in'_m)_{m=1}^n)$,

where $c_i = (m_i, q_i, \sigma_i, \gamma_i, in_i, out_i)$, etc.

**Remark 2.4.** A CSXMS, $\mathcal{P}_n = (\Pi_1, \ldots, \Pi_n)$, functions as follows:

(i) each $\Pi_i$ starts with both the input and output ports containing $\lambda$. The only function that can be applied initially should be an ordinary processing function, $\varphi_i \in \text{Procs}_i'$. Hence, the initial state $q_0^i \in \text{Start}_i$ from which $\varphi_i$ emerges must be an ordinary state;
(ii) an ordinary function $\varphi_i$ can process a symbol from $\mathbb{N}_i$ if one is present, or it can proceed by ignoring the input value in which case the content of $\mathbb{N}_i$ remains unchanged. A similar behaviour is expected for the $\mathbb{O}_i$ port; and
(iii) after a communicating function is applied, the machine state will be an ordinary one, and so the next function to be applied (if any) will also be ordinary.

\[ \text{http://rsta.royalsocietypublishing.org/} \]
(c) X-machine testing

The fact that an SXM can be regarded as an augmented version of its associated automaton means that well-established automated FSM test-set generation strategies (e.g. based on Chow’s W-method [26]) can be ‘lifted’ to provide SXM testing strategies. The goal of SXM testing is to establish whether two SXMs, \(S\) (the specification) and \(I\) (the implementation under test, or IUT) compute the same behaviour. We assume that the complete structure of \(S\) is known and that \(S\) has been minimized, that \(S\) and \(I\) use the same set \(\text{Procs}\) of processing functions (if not, we define \(\text{Procs}\) to be the union of their respective process sets), and attempt to find a finite test set, \(\text{Tests} \subseteq \text{In}^*\), with the property that, if \([S](t) = [I](t)\) for every \(t \in \text{Tests}\), then \(S\) and \(I\) must necessarily compute the same relation. In general, the ability to store data in memory during a computation means that this problem is well known to be uncomputable; it is therefore necessary to impose certain constraints, called design for test (DFT) conditions, as to which implementations \(I\) are considered valid candidates for testing. In particular, we generally assume that some estimate is available as to how many extra states \(I\) has relative to \(S\).

DFT conditions for SXMs are well known, and an adequate set of conditions to ensure testability is [21]

- **deterministic specification**: the behaviours of \(I\) and its associated automaton \(A\) should both be deterministic, i.e. given any state and any two processing functions, \(\varphi_1\) and \(\varphi_2\), applicable in that state, we require \(\text{dom}(\varphi_1) \cap \text{dom}(\varphi_2) = \emptyset\);
- **Procs-completeness**: given any \(\varphi \in \text{Procs}\) and \(m \in \text{Mem}\), there exists some \(i \in \text{In}\) such that \(\varphi(m, i)\) is defined; and
- **Procs-output distinguishability**: examining the output of a processing function should tell us which function it is, i.e. given any \(\varphi_1, \varphi_2 \in \text{Procs}\), if there exist \(m, m_1, m_2 \in \text{Mem}\), \(i \in \text{In}\) and \(o \in \text{Out}\) such that \(\varphi_1(m, i) = (o, m_1)\) and \(\varphi_2(m, i) = (o, m_2)\), then \(\varphi_1 = \varphi_2\).

Since the SXM testing methodology requires us to examine the outputs that are produced when certain test inputs are processed, extending the technique to include CSXMS systems requires the designer to ensure that every function application consumes an input and produces an output. As the communicating functions act only on memory symbols, these must therefore be extended to handle input and output symbols. To do this, an additional input symbol \(a \notin \bigcup \text{In}_i\) is introduced and for each communicating function \(\varphi''_j \in \text{Procs}'_j\) an output symbol \([i, j]\) is added. We now formally redefine \(\varphi''_j\) to take the input symbol \(a\) (this is a communication event) and generate the output symbol \([i, j]\) (I have just applied \(\Pi'_j\)’s communication function, \(\varphi''_j\)). As before, each component CSXM, \(\Pi'_n\), should be deterministic, Procs-complete and Procs-output distinguishable (the extensions applied to the communicating functions mean that these automatically satisfy the last two conditions). The entire CSXMS, \(\mathcal{P}_n\), is then converted into a single SXM, \(\mathcal{P}^T\), and standard SXM testing is applied; however, although the CSXM components are deterministic, the resulting SXM need not be and consequently a testing approach for non-deterministic SXMs is used [19].

The SXM, \(\mathcal{P}^T = (\text{In}, \text{Out}, Q, \text{Mem}, \text{Procs}, \text{Next}, \text{Start}, \text{Stop}, n^P)\), is obtained from the CSXMS, \(\mathcal{P}_n\), with the additional extensions mentioned above, as follows [27]:

- \(\text{In} = ((\text{In}_1 \cup \{a, \text{null}\}) \times \cdots \times (\text{In}_n \cup \{a, \text{null}\})) \setminus \{(\text{null}, \ldots, \text{null})\}\)
- \(\text{Out} = ((\text{Out}_1 \cup \{1, j\} \mid j \neq 1) \cup \{\text{null}\}) \times \cdots \times (\text{Out}_n \cup \{n, j\} \mid j \neq n) \cup \{\text{null}\}) \setminus \{(\text{null}, \ldots, \text{null})\}\)
- \(Q = Q_1 \times \cdots \times Q_n, \quad \text{Start} = I_1 \times \cdots \times I_n, \quad \text{Stop} = T_1 \times \cdots \times T_n\)
- \(m = (\text{IN}_1 \times \text{Mem} \times \text{OUT}_1) \times \cdots \times (\text{IN}_n \times \text{Mem} \times \text{OUT}_n)\).
- \(n^P = (\lambda, n^P_1, \lambda), \ldots , (\lambda, n^P_n, \lambda)\).
- \(\text{Procs} = \{(\varphi_1, \ldots, \varphi_n) \mid (\forall i)(\varphi_i \in \text{Procs}_i \cup \{i\text{null}_{i}\})\}\).

A similar testing approach is proposed in [28] for a slightly different CSXMS concept.
The SXM $\mathcal{P}^T$ is the product of the CSXMS components. A processing function, $\varphi$, describes a set of functions that are simultaneously applied in the CSXMS components. However, some components might not execute any processing functions during a particular computation step; in this case $\bar{\varphi}_i = e$.

The associated test set consists of input sequences obtained by applying a so-called fundamental test function, $t: \text{Procs}^* \rightarrow \text{In}^*$, to a sequence of processing functions derived from the associated automaton by applying one of the many known state machine-based testing methods [29]. Formally, a test set for an SXM is a finite set of input sequences

$$\text{Tests} = \{ \iota_1 \cdots \iota_p \in \text{In}^* \mid \exists \varphi_1 \cdots \varphi_p \in \text{Procs} \text{ s.t. } t(\varphi_1 \cdots \varphi_p) = \iota_1 \cdots \iota_p \},$$

where we require, for each processing function $f_i = (\bar{\varphi}_{i,1}, \ldots, \bar{\varphi}_{i,n})$ and each associated input element, $\iota_i = (\bar{\iota}_{i,1}, \ldots, \bar{\iota}_{i,n})$, that

- when $\bar{\varphi}_{i,j}$ is either an ordinary or communicating function, then $\bar{\iota}_{i,j} \in (\text{In}_j \cup \{a\})$;
- otherwise, when $\bar{\varphi}_{i,j} = \text{id}_j$ then $\bar{\iota}_{i,j} = (\text{i.e. when the current configuration of the } j\text{th component remains unchanged, then there is no input to this machine component})$.

According to the testing strategy devised for SXMs [16,19,21], such a test set can always be constructed for any SXM—and hence, by extension, for any CSXMS—that satisfies the relevant DFT conditions.

3. P system models

The P system (membrane system) [17] is a model of computation based on eukaryotic cell structures in biology, and the mechanisms used within and between cells to enable communication between their various sub-parts. Since its introduction in [30], the model has diverged into a number of different variants, each modelling a different combination of biologically inspired computational mechanisms. In this section, we describe a basic variant of the model and provide a simple example to illustrate its use for computational purposes. We then show how a testing strategy for a system comprising a P system $\text{Base}$ and an SXM $\text{Control}$ can be defined, corresponding to the basic heterotic framework discussed in §1.

(a) Cell-like P systems

Eukaryotic cells are characterized by the presence of membranes, which separate different regions of the cell into a hierarchically organized system of distinct nested compartments. At any given time, each compartment will contain a mixture of biochemicals, and this mixture changes over time as a result of the coordinated exchange of biochemicals across membrane boundaries. This basic structure is captured by one of the best known and most used types of P system, the cell-like P system, using non-cooperative evolution rules and communication rules [31]. In the sequel, we call these models simply P systems.

**Definition 3.1.** A P system with $n$ compartments is a tuple

$$\text{PS}_n = (V, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n),$$

where

- $V$ is a finite alphabet;
- $\mu$ defines the membrane structure, a hierarchical arrangement of $n$ compartments, identified by integers 1 to $n$;
- for each $i = 1, \ldots, n$, $w_i$ represents the initial multiset in compartment $i$; and
- for each $i = 1, \ldots, n$, $R_i$ represents the set of rules used in compartment $i$.  


The rules capture the way that a chemical species in one cell compartment can be used to trigger the production of new chemical species in both that compartment and others. A typical rule has the form \(a \rightarrow (a_1, t_1) \cdots (a_m, t_m)\), where \(a, a_1, \ldots, a_n \in V\) and \(t_1, \ldots, t_m \in \text{[here]} \cup \{1, \ldots, n\}\). When this rule is applied in a compartment to the symbol \(a\), it is replaced in that compartment by the collection of symbols \(a_i\) for which \(t_i = \text{here}\) (by convention, symbols of the form \((a_i, \text{here})\) are often written \(a_i\), with the destination being understood). Those symbols \(a_i\) for which \(t_i = k\) are added to the compartment labelled \(k\), provided this is either a parent or a child of the current one. The rules are applied in \textit{maximally parallel mode} in each compartment; for example, if a compartment contains two copies of the symbol \(a\), then the rule above will be fired twice (simultaneously), once for each occurrence.

A \textit{configuration} of the P system, \(PS_n\), is a tuple \(c = (u_1, \ldots, u_n)\), where \(u_i \in V^*\) for each \(i = 1, \ldots, n\), which represents the instantaneous disposition of chemical species within the cell’s compartments. A \textit{computation} from a configuration \(c_1\), using maximally parallel mode, leads to a new configuration \(c_2\); this process is denoted \(c_1 \Rightarrow c_2\).

We now discuss, following [32], a testing strategy for P systems which is inspired by the testing principles developed for context-free grammars [33], called \textit{rule-coverage}. Other methods for testing P systems also exist, for example mutation-based testing [34]; some are inspired by finite state machine testing [31], others by X-machine testing [35]. Approaches combining verification and testing have also been investigated [36,37].

\[\text{(b) Rule coverage testing in P systems}\]

We introduce first some new concepts.

\textbf{Definition 3.2.} A configuration \(c = (u_1, \ldots, u_n)\) covers a rule

\[r_i : a_i \rightarrow (a_{i_1}, j_1) \cdots (a_{i_m}, j_m)\]

if there is a computation path starting from the initial configuration and resulting in configuration \(c\), during which rule \(r_i\) is used. Formally,

\[c_0 = (w_1, \ldots, w_n) \Rightarrow^* (x_1, \ldots, x_j, \ldots, x_h, \ldots, x_1a_{i_1}, \ldots, x_{j_1}, \ldots, x_h, \ldots, x_1) \Rightarrow^* (x'_{i_1}, x'_{j_1}, \ldots, x'_{h_1}, \ldots, x'_{i_2}, x'_{j_2}, \ldots, x'_{h_2}, \ldots, x'_{i_n}, x'_{j_n}) \Rightarrow^* c = (u_1, \ldots, u_n)\].

\textbf{Definition 3.3.} A \textit{test set}, in accordance to the rule coverage principle, is a set \(\text{Tests}^{rc} \subseteq (V^*)^n\), such that for each rule \(r \in R_i, 1 \leq i \leq n\), there is \(c \in \text{Tests}^{rc}\) which covers \(r\).

The strategy involved here is to find a test set \(\text{Tests}^{rc}\) which unavoidably covers every rule used in the P system, so that when we observe one of the configurations in \(\text{Tests}^{rc}\) we can safely deduce that the rule must have been fired during the computation. For example, let us consider the P system with two compartments, \(PS_2 = (V, [[1], 2], s, t, R_1, R_2)\). This has compartment 2 inside compartment 1, and the alphabet \(V\) is the set of symbols that appear in the rules of \(R_1\) and \(R_2\). Compartment 1 initially contains symbol \(s\), compartment 2 contains \(t\) and the rules associated with each compartment are

\[R_1 = \{r_{11} : s \rightarrow abc; r_{12} : a \rightarrow d; r_{13} : a \rightarrow c(a, 2); r_{14} : bc \rightarrow cc; r_{15} : c \rightarrow f\}\]

and

\[R_2 = \{r_{21} : t \rightarrow b; r_{22} : ab \rightarrow c\}.
\]

A test set for \(PS_2\) is \(\text{Tests}^{rc} = \{(dbe, b), (ccf, c)\}\), as can be seen from the following two computations:

\[c_0 = (s, t) \Rightarrow (r_{11}, r_{21}) (abe, b) \Rightarrow (r_{12}, \text{null}) (dbe, b)\]
and
\[ c_0 = (s, t) \xrightarrow{(r_{11}, r_{21})} (abc, b) \xrightarrow{(r_{13}, r_{23}, \text{null})} (cbf, ab) \xrightarrow{(r_{14}, r_{22})} (ccf, c). \]

One can easily observe that \( \text{Tests}^{RC} \) is a test set. All of the rules in both \( R_1 \) and \( R_2 \) are covered by at least one element of \( \text{Tests}^{RC} \), and there is no way to obtain these configurations without firing each of them at least once.

(c) Testing a heterotic P system/stream X-machine system

We turn now to our first example of heterotic testing. We will assume for this example that \( \text{Base} \) is the P system representation of a biocomputational process, while \( \text{Control} \) is an SXM representation of a classical digital computer. As prescribed in [13], we assume that the biosystem generates an output which the computer inspects; the computer then provides the biosystem with new initial configuration, and the process repeats.

The example above shows that \( \text{Base} \) can be tested in isolation, and we saw in §2 that general test strategies also exist for testing \( \text{Control} \) (subject, in both cases, to certain DFT conditions being satisfied). From a testing point of view, this means that unit testing can be assumed to have taken place before the components are combined to form the overall system. The question we now address is how to devise an integration test strategy for the combined system.

The simplest approach is to show that \( \text{Base} \) and \( \text{Control} \) can be represented as components of a CSXMS which models their full combined behaviour. Since this CSXMS is testable, it follows that the \( \text{Base} + \text{Control} \) heterosystem is also testable. Recall that for integration testing purposes, our goal is to test the system generated by composing the P system component (\( \text{Base} \)) with the controller (\( \text{Control} \)). However, we can easily build communicating SXMs to stepwise-simulate these two agents (figure 1). The \( \text{Base} \) simulation holds and manipulates the P system’s configurations in memory using an ordinary function that simulates rule execution. Once the computation has run to completion, a second function moves the current memory value (i.e. the final configuration) to the output port, and a communicating function then sends the configuration to \( \text{Control} \). This uses an ordinary function to examine the input port, decides how \( \text{Base} \) should be re-initialized and sends the relevant configuration to its output port. A communicating function then transfers this back to \( \text{Base} \), which uses it as its new initial configuration and the whole process repeats.

Since the simulation of \( \text{Base} \) is now part of the CSXMS construction, and we require this to satisfy the SXM DFT conditions, the same should also be true of the P system, and hence of any
biological system it may describe. While this may potentially be experimentally unreasonable, we should note that the simulation is doing more work than is required, since we do not need to check, for example, that the P system has computed its terminal configuration correctly (this has already been addressed at the unit testing stage). For integration testing purposes, we can instead regard Base and Control as ‘black boxes’, and focus simply on their mutual interactions. In general, abstracting away the components’ detailed internal behaviours in this way will considerably simplify the task of ensuring the DFT conditions are satisfied.

4. Shortcomings and ongoing research

The construction outlined in figure 1 is entirely general, provided both Base and Control can be stepwise-simulated as components of a CSXMS. This is generally possible, because the underlying SXM model is Turing-complete. However, it is not enough that the components’ behaviours can be simulated; it is also important that the simulations are efficient; it would rarely be reasonable, for example, to require companies to build SXM simulations of quantum components in order to test their behaviours as part of a larger system. Apart from the intractability problems that would probably arise, this would introduce a new layer of processing (construction of the simulation itself), which would itself require verification.

As we have noted above, however, the simulations are doing more work than is actually required for integration testing purposes. Indeed, the use of simulations in figure 1 was only introduced for theoretical reasons, to allow us to establish that testability is indeed possible. For practical purposes, it would be more sensible to use physical implementations of Base and Control as experimental oracles. Instead of simulating a P system, for example, we could instruct Control to pass details of the next initial configuration to an automated biochemical assembly, causing it to run a physical instantiation of the P system. Having run the experiment, automated machinery could be used to determine the concentrations of relevant chemicals in the resulting mix, and use these to determine the next signal to be transmitted to Control’s input port (in terms of the formal model, we would modify the example above so that configurations are passed to Control as elements of In rather than via the communications port). While this narrative glosses over numerous important problems relating to the physical management of biochemical computing systems (can we be certain, for example, that the systems in question can be manipulated as required? Are the ‘outputs’ we seek easily observable?), the approach has the obvious advantage that each component can be implemented in the form in which it was originally manufactured for unit testing, so we can be confident both that the integration and unit test methodologies are consistent with one another, and also that no additional testing is required due to the introduction of an additional simulation stage.

Nonetheless, there are situations in which the CSXMS approach proposed above cannot easily be applied in its current form, even ignoring intractability problems that are likely to arise in systems which combine simulable systems to generate non-simulable ones. Following [8,13], we have so far assumed the simplest possible design of heterotic system, in which a single Control unit repeatedly coordinates the configuration of a single Base unit, and where each unit’s computation is allowed to run to completion before control passes to the other. In such a system, it is always possible to say which component is running ‘now’, and which will be running ‘next’. But one can easily envisage situations in which the concept of a ‘next’ computation step is essentially meaningless. For example, consider a future nano-bot system designed to deliver medication to a specific site in a patient’s body. One can envisage a scenario in which the bots (Base) form a swarm of independent magnetically detectable agents, which continually adjust their motion by interacting with the ambient electromagnetic field in their vicinity. To make the system work, an external apparatus monitors the bots’ positions in real time, and uses this information to make continuous adjustments to the electromagnetic field surrounding the patient. In such a system, the continuity of interaction is an intrinsic part of the specification, and it would not be appropriate to simplify the system by assuming alternate executions of Base and Control. Doing so might well
allow us to generate a CSXMS-based test set, but it would not allow us to test the intricacies of the system’s underlying real-time functionality.

In situations like this, where the concept of a ‘next computation step’ has no meaning, it is not possible to model system changes using the kind of next-state relation associated with automata or SXMs. Instead, we need a model in which mutually interacting processes can be defined and combined, no matter whether their operation assumes discrete time, analogue time, linear time, branching time or even some combination of temporal structures. Our research in this direction is ongoing and involves the construction of a generalized X-machine model which preserves the essential features of the SXM model, while allowing computations to be defined over arbitrary temporal structures.

Since the relevance of applying a processing function \( \varphi \) in a state \( q \) is determined solely by the configuration change induced once traversal of the associated transition has completed, we can describe the transition by a relation \( \text{Trans} : 2 \rightarrow (\mathcal{C} \leftrightarrow \mathcal{G}) \), where \( \mathcal{G} \) is the set of possible configurations for the SXM in question, and

\[
\text{Trans}(0) = \text{id}_{\mathcal{G}}, \quad \text{Trans}(1) = \hat{\varphi}, \tag{4.1}
\]

where \( \text{id}_{\mathcal{G}} \) is the identity relation on \( \mathcal{G} \) (we assume \( \text{id}_{\mathcal{G}} \in \text{Procs} \)), and \( \hat{\varphi}(c) = \{ c' \mid c \vdash_{\varphi} c' \} \).

Writing the transition in this way highlights the role of the timing structure, in this case \( 2 = \{ 0, 1 \} \), in determining the effect of firing the transition. Firing a transition changes the configuration from \( c \in \text{Trans}(0)(c) \) to some \( c' \in \text{Trans}(1)(c) \). If we wish to include instead a continuously evolving analogue procedure for computing \( \varphi \), we can do so formally by replacing the existing transition with any continuous function \( \text{Trans}^\prime : [0, 1] \rightarrow (\mathcal{G} \leftrightarrow \mathcal{G}) \) that also satisfies (4.1). Similarly, transfinite models of computation can be instantiated using \( \text{Time} = \beta + 1 \) for suitable limit ordinals \( \beta \) (where we regard \( \beta \), the maximal value in \( \beta + 1 \), as the value ‘1’ in (4.1)).

More generally, given any timing structure, \( \text{Time} \), we can replace any transition in an SXM with a function of the form \( \text{Trans}'' : \text{Time} \rightarrow (\mathcal{G} \leftrightarrow \mathcal{G}) \) without changing its overall behaviour, provided \( \text{Trans}'' \) has a minimum element 0 and maximum element 1 and satisfies (4.1). Since we are considering physically realizable computations, we also impose the condition that \( \text{Trans}'' \) should be continuous when regarded as a function on \( \text{Time} \) (we regard this as a defining property of what it means for \( \text{Time} \) to be a sensible model of time for the computation in question, rather than a constraint on \( \text{Trans}'' \)).

Formally, however, the notion that \( \text{Trans}'' \) is continuous presupposes the existence of topologies on both \( \text{Time} \) and \( \mathcal{G} \). For philosophical reasons, we assume that \( \text{Time} \) is partially ordered and assign it the associated compact Hausdorff topology. Similarly, we can define a natural Tychonov topology on \( \mathcal{G} \) [24]. In this way, we postulate, we can instantiate each transition function using whichever paradigm is most appropriate for the function being modelled, thereby allowing truly general heterotic systems to be brought under the SXM testing umbrella [24,38].

Another shortcoming of the descriptions presented above is our focus on deterministic systems. In principle, we can model non-deterministic behaviours using the generalized timing structure approach just described, by regarding the system’s various potential execution traces as occurring along different paths in branching time. However, it is worth noting that the problem of testing non-deterministic systems is itself of considerable current research interest [39], and it is possible that extending newer emerging approaches may prove as promising, or more so, than extending the X-machine test strategy. In addition, research continues to extend the scope of the SXM testing strategy itself, and it is likely that ideas currently being investigated will be of use in modelling and testing heterotic behaviours also. For example, Merayo et al. have recently considered techniques for generating test sets for timed systems modelled using SXMs [40,41]. Their work assumes a more constrained model of time than ours (they assume that time is linearly rather than partially ordered), but nonetheless their general approach offers many useful insights, including techniques for testing systems with timeout capabilities.
5. Summary and conclusion

In this paper, we have considered the problem of constructing test sets for integration-testing a heterotic system $H$, composed of two interacting systems, Base and Control. For Turing-simulable systems, this can be achieved by re-expressing Base and Control as communicating components within a CSXMS model. Since all such models have an associated test-set generation strategy, this allows us to generate adequate test sets for $H$, provided the relevant design-for-test conditions are satisfied. We illustrated our approach by describing how a test set can be generated for a heterotic system combining an automaton-based Control system with a bio-related P system (Base). It remains important that these components can also be tested in isolation, and we have seen how unit testing of a P system can be achieved.

It will also be important to validate our method experimentally, since many of the systems we envisage being included in practical heterotic systems cannot be simulated efficiently using traditional SXM-based models and would be better included as experimental oracles. Such experiments could be conducted both in silico and in the laboratory. For example, we can perform various mutation tests on the combined system $H$, by deliberately seeding Base and Control with faults and testing our method’s ability to detect them.

The technical structure we presented to deduce the existence of a test set is quite general, but while it can easily be generalized to include several interacting components, it cannot cope with situations involving processes whose interactions are sufficiently complicated that the question what communication event comes next? is essentially meaningless. In such cases, it is necessary to devise an extended model of X-machine computation which is sufficiently general to allow computations and communications with any temporal structure. In this context, it is also important to remember that physical systems are invariably noisy, and it will be especially important when devising a fully general testing strategy to ensure that tolerances and thresholds can be specified, and more importantly, tested for. Work on this topic is continuing, and we hope to report positive results in due course.

Acknowledgements. The authors would like to acknowledge the insightful and helpful comments provided by the anonymous referees.

References


23. Stannett M. 2001 Computation over arbitrary models of time. Technical Report CS-01-08, Department of Computer Science, University of Sheffield, Sheffield, UK.


