Applied Mathematics and Computation xxx (2009) xxx–xxx

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc

Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms

B.C. Thompson [a], J.V. Tucker [a], J.I. Zucker [b,*]

[a] *Department of Computer Science, Swansea University, Singleton Park, Swansea SA2 8PP Wales, United Kingdom*
[b] *Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada L8S 4K1*

ARTICLE INFO

Keywords:
Synchronous concurrent algorithm
Dynamical systems
Many sorted algebras
Equational specifications
Streams
Computability on topological algebras
Computable physical systems

ABSTRACT

A *synchronous concurrent algorithm* (SCA) is a parallel deterministic algorithm based on a network of modules and channels, computing and communicating data in parallel, and synchronised by a global clock with discrete time. Many types of algorithms, computer architectures, and mathematical models of physical and biological systems are examples of SCAs. For example, conventional digital hardware is made from components that are SCAs and many computational models possess the essential features of SCAs, including systolic arrays, neural networks, cellular automata and coupled map lattices.

In this paper we formalise the general concept of an SCA equipped with a global clock in order to analyse precisely (i) specifications of their spatio-temporal behaviour; and (ii) the senses in which the algorithms are correct. We start the mathematical study of SCA computation, specification and correctness using methods based on computation on many-sorted topological algebras and equational logic. We show that specifications can be given equationally and, hence, that the correctness of SCAs can be reduced to the validity of equations in certain computable algebras. Since the idea of an SCA is general, our methods and results apply to each of the particular classes of algorithms and dynamical systems above.

1. Introduction

1.1. The concept

A *synchronous concurrent algorithm* (SCA) is an algorithm based on a network of modules and channels, computing and communicating data in parallel, and synchronised by a global clock with discrete time. The etymology of 'synchronous' is Greek: "at the same time". SCAs can process infinite streams of input data and return infinite streams of output data. Most importantly, an SCA is a *parallel deterministic algorithm*.

Many types of algorithms, computer architectures, and mathematical models of physical and biological systems are examples of SCAs. First and foremost, conventional digital hardware, including all forms of serial and parallel computers and digital controllers, are made from components that are SCAs. In many cases, complete specifications of computers at different levels of abstraction are SCAs. Interestingly, the structure of Charles Babbage's Analytical Engine (developed from 1833 onwards) is that of an SCA.

Further, many specialised models of computation possess the essential features of SCAs, including *systolic arrays*, *neural networks*, *cellular automata* and *coupled map lattices*.

* Corresponding author.
E-mail addresses: J.V.Tucker@swansea.ac.uk (J.V. Tucker), zucker@mcmaster.ca (J.I. Zucker).

0096-3003/$ - see front matter © 2009 Published by Elsevier Inc.
doi:10.1016/j.amc.2009.04.058

The parallel algorithms, architectures and dynamical systems that comprise the class of SCAs have many applications, ranging from their use in special purpose devices (for communication and signal processing, graphics and process control) to computational models of biological and physical phenomena.

From the point of view of computing, an SCA can be considered to be a type of *deterministic* data flow network, in which time is explicit and enjoys a primary role. SCAs require a new specialised mathematical theory with applications of its own.

From the point of view of mathematical physics and biology, an SCA can be considered to be a type of spatially extensive discrete space, discrete time, deterministic dynamical system that is studied independently or as an approximation to continuous space, continuous time dynamical systems.

In most cases, SCAs are complicated and require extensive simulation and mathematical analysis to understand their operation, behaviour and verification. In fact, in the independent literatures on the above types of SCAs it is often difficult to formulate precisely

(i) specific SCAs and their operation in time;
(ii) specifications of their spatio-temporal behaviour; and
(iii) the senses in which the algorithms are correct.

In the case of neural networks, correctness is further complicated by the difficulty of writing problem specifications, the existence of a learning phase, and notions of approximate correctness. In the case of non-linear dynamical systems, correctness is concerned with properties such as chaotic, stable, emergent and coherent behaviour over time. Thus, SCAs constitute a wide ranging class of useful algorithms for which many basic questions concerning their structure and design remain unanswered.

In this paper, we formalise the general concept of an SCA equipped with a global clock and analyse precisely ideas about the specification and correctness of SCAs. Our mathematical study of SCA computation, specification and correctness provides a unified theory of deterministic parallel computing systems and deterministic, spatially extensive, non-linear dynamical systems.

The methods are based on abstract computability theory on many-sorted topological algebra and equational logic. We show how to define SCAs by equations over stream algebras in a simple way. We also show that specifications can be given equationally and, hence, that the correctness of SCAs can always be reduced to the validity of equations in certain algebras. Thus, a natural method for verification of SCAs is equational reasoning, although this is incomplete.

Our methods and results apply to each of the classes of algorithms and architectures listed above. In particular, they can be used in case studies and software tools for design and verification of specific classes of SCAs, and as a starting point for a general theoretical analysis of hardware verification.

1.2. The theory

Data is modelled by an algebra

$$A = (A, \mathbb{B}, \mathbb{T};\ F_1, \ldots, F_k)$$

with three carrier sets: the set A of data, $\mathbb{B}$ of Booleans and $\mathbb{T}$ of naturals $\{0, 1, 2, \ldots\}$ (written $\mathbb{T}$ instead of $\mathbb{N}$ because it represents the discrete time on the global clock), and functions $F_1, \ldots, F_k$ which include the standard Boolean operations (with possibly equality on A) and the arithmetic operations of 0 and successor $t + 1$.

The behaviour of SCAs in time is modelled using *streams* of elements of A, which are infinite sequences indexed by (discrete) time. Let $[\mathbb{T} \to A]$ be the set of all streams. The operations on data, time and streams are combined to form a *stream algebra*:

$$\overline{A} = (A, \mathbb{B}, \mathbb{T}, [\mathbb{T} \to A]; F_1, \ldots, F_k, \mathsf{eval}).$$

Typically, in models of hardware systems, SCAs compute with streams of bits, integers or terms. In dynamical systems, SCAs compute with streams of real and complex numbers. To prepare for this mathematical view, we provide some preliminaries on topological algebras in Section 2 and stream algebras and computable algebras in Section 3. We note that all stream algebras are topological algebras and often have certain dense subalgebras that are computable.

In Section 4, we define synchronous concurrent algorithms and architectures and formalise their semantics by means of functions defined by *simultaneous primitive recursion equations* over $\overline{A}$.

More specifically, an SCA based on a network N with m modules and p input streams is specified by a network state function

$$\mathsf{V}^N : A^m \times [\mathbb{T} \to A]^p \times \mathbb{T} \to A^m$$

in which $\mathsf{V}^N(\mathsf{a}, \mathsf{x}, t)$ denotes the state of the SCA on processing p input streams $\mathsf{x} \in [\mathbb{T} \to A]^p$ from initial state $\mathsf{a} \in A^m$ at time $t \in \mathbb{T}$.

In Section 5, we give a sketch of the broad range of types of SCAs (systolic arrays, neural networks, cellular automata and coupled map lattices) with an bibliography.

In Section 6, we consider specifications and correctness criteria for a simple form of the space-time behaviour of SCAs: correctness based on specifications with respect to a single system clock of the SCA. Other forms of correctness are possible, such as correctness based on specifications with respect to a second clock external to the SCA [30].

Please cite this article in press as: B.C. Thompson et al., Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms, Appl. Math. Comput. (2009), doi:10.1016/j.amc.2009.04.058

In Section 7, we consider the SCA equational models from the point of view of computability theory. We define two classes of predicates on $\overline{A}$, broader than the class of PR (primitive recursive) predicates: *equational PR*, which includes the (not necessarily computable) equality relation as primitive, and *equational* λPR, which also includes stream abstraction.

We consider specifications and correctness relations which should be algorithmically testable, e.g., by primitive recursive computations. We prove some results concerning the logical and computational structure of SCA correctness, including results having the following form:

Theorem 1. *The network state function* $\vee^N$ *is PR on the stream algebra* $\overline{A}$.

Theorem 2. *Suppose A is a Hausdorff algebra, and further*

(a) *P, Q and R are equationally λPR on $\overline{A}$,*
(b) *A has a dense computable subalgebra D.*

Then we can effectively construct a computable algebra $C_{V,P,Q,R}$ with signature $\Sigma_{V,P,Q,R}$ that expands by functions the stream subalgebra of eventually constant streams over D, and equations $e_P, e_Q, e_{V,R}$ over $\Sigma_{V,P,Q,R}$ such that the following are equivalent:

(i) *$\vee^N$ is correct w.r.t P, Q and R, i.e., (7.4) holds;*
(ii) $C_{V,P,Q,R} \models e_P \wedge e_Q \to e_{V,R}$.

Thus, *the correctness of the SCA* (as in (i)) *can be reduced to the validity of a conditional equation in a computable algebra* (as in (ii)). Through our definitions, this reduction to conditional equations applies to a wide variety of complex space-time behaviours for a wide variety of computing devices and dynamical systems,

This has several consequences, including the fact that SCA correctness is co-recursively enumerable. This suggests *there are no effectively axiomatisable complete proof systems for SCA verification*. However, we do have the following result in this direction.

Theorem 3. *Given the hypotheses of Theorem 2, we can effectively construct a finite equational specification $(\Sigma_{V,P,Q,R}, E_{V,P,Q,R})$ and equations $e_P, e_Q, e_{V,R}$ over $\Sigma_{V,P,Q,R}$ s.t. the following are equivalent:*

(i) *$\vee^N$ is correct w.r.t P, Q and R, i.e., (7.4) holds;*
(ii) $T(\Sigma_{V,P,Q,R}, E_{V,P,Q,R}) \models e_P \wedge e_Q \to e_{V,R}$.

Section 8 contains some concluding remarks, concerning the issues of (a) a common theoretical framework for SCA networks and analog networks, and (b) generalising the model to allow for partial module functions and streams.

Since the emphasis in this paper is on the a general mathematical model of SCAs, it will be helpful if the reader has some familiarity with theory for algorithmic computability on discrete and continuous data [54,74,65,68,58].

1.3. Origins

The idea of a making a mathematical theory of SCAs that would uncover and analyse common structures and properties between hardware, parallel algorithms, and dynamical systems modelling natural phenomena arises in the work of the second author (JVT) at Leeds University, starting in 1981. Over many years, the SCA notion was developed primarily through studying applications, in work with, for example:

- Harman on hardware design and verification [19,22,21,23–26,17,18,20].
- Holden and Poole on non-linear dynamical systems [32,33,31,52,53].

The first two authors (BCT and JVT) started work on these mathematical foundations for SCA theory in 1987, leading to the report [60]. Although unpublished, it was widely circulated (forming, e.g., part of JVT's lecture notes for the NATO Summer School on *Logic and algebra of specification*, Marktoberdorf, Germany, 1991). There is a full conceptual analysis and extensive reflection on correctness and examples in [60].

However, the subtlety of the connections between the SCA models and abstract and concrete computability theories for continuous data types, such as streams of real numbers, was a problem. Thus, a gap of 17 years is partly excused by the need to master computability theories for topological algebras, to which JVT and the third author (JIZ) have devoted many pages in the period [63–69]. Our current understanding enabled us to look at continuous time, continuous state and discrete space systems in our paper [70], where we were motivated by the idea of models capable of unifying disparate analogue technologies. Clearly, this application to analogue computation was inspired by the earlier unification of models work on SCAs.

Please cite this article in press as: B.C. Thompson et al., Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms, Appl. Math. Comput. (2009), doi:10.1016/j.amc.2009.04.058

2. Topological algebras

We briefly survey the basic concepts of topological and metric many-sorted algebras. More details can be found in [65,64,68].

2.1. Basic algebraic definitions

A *signature* Σ (for a many-sorted algebra) is a pair consisting of (i) a finite set $\mathsf{Sort}(\Sigma)$ of *sorts*, and (ii) a finite set $\mathsf{Func}(\Sigma)$ of *(basic) function symbols*, each symbol F having a *type* $s_1 \times \cdots \times s_m \to s$, where $s_1, \ldots, s_m, s \in \mathsf{Sort}(\Sigma)$; in that case we write $F : s_1 \times \cdots \times s_m \to s$. (The case $m = 0$ corresponds to *constant symbols*.)

A Σ*-product type* has the form $u = s_1 \times \cdots \times s_m$ $(m \geqslant 0)$, where $s_1, \ldots, s_m$ are Σ-sorts.

A Σ*-algebra* A has, for each sort s of Σ, a non-empty *carrier set* A_s of sort s, and for each Σ-function symbol $F : u \to s$, a function $F^A : A^u \to A_s$, where, for the Σ-product type $u = s_1 \times \cdots \times s_m$, we write $A^u =_{df} A_{s_1} \times \cdots \times A_{s_m}$. For $m = 0$, F^A is an element of A_s.

The algebra A is *total* if F^A is total for each Σ-function symbol F.

Remark 2.1.1 (*Assumption of total algebras*). For the purpose of this paper, we work only with total algebras, for the sake of simplicity. The interesting generalisation to the framework of partial algebras (with partial operations and partial streams) is left to a future paper (see Section 8).

Given an algebra A, we write $\Sigma(A)$ for its signature.

Example 2.1.2

(a) The algebra $\mathscr{B}$ of *Booleans* has the carrier $\mathbb{B} = \{\mathbb{t}, \mathbb{f}\}$ of sort bool:

$$\mathscr{B} = (\mathbb{B}; \mathbb{t}, \mathbb{f}, \mathsf{and}, \mathsf{or}, \mathsf{not}).$$

(b) The algebra $\mathscr{T}_0$ of naturals has a carrier $\mathbb{T}$ of sort nat, together with the zero constant and successor function:

$$\mathscr{T}_0 = (\mathbb{T}; 0, \mathsf{S}).$$

Note that here and elsewhere we use the notation

$$\mathbb{T} =_{df} \mathbb{N} = \{0, 1, 2, \ldots\}$$

for the set of *natural numbers* (denoted $t, t', \ldots$), since the interpretation of $\mathbb{N}$ throughout this paper will be almost exclusively as a *discrete global clock*.

(c) The ring $\mathscr{R}_0$ of reals has a carrier $\mathbb{R}$ of sort real:

$$\mathscr{R}_0 = (\mathbb{R}; 0, 1, +, \times, -).$$

We make the following

Instantiation Assumption. *For every Σ-sort s, there is a closed term of that sort, called the default term δ^s of that sort. In any Σ-algebra A, it names an element of A_s, called the default element of A_s.*

2.2. Adding Booleans: standard signatures and algebras

Definition 2.2.1 (*Standard signature*). A signature Σ is *standard* if it includes the signature of Booleans, i.e., $\Sigma(\mathscr{B}) \subseteq \Sigma$.

Given a standard signature Σ, a sort of Σ is called an *equality sort* if Σ includes an *equality operator* $\mathsf{eq}_s : s^2 \to \mathsf{bool}$.

Definition 2.2.2 (*Standard algebra*). Given a standard signature Σ, a Σ-algebra A is *standard* if (i) it is an expansion of $\mathscr{B}$; (ii) the equality operator eq_s is interpreted as *identity* on the carrier of each equality sort s.

An example of an equality sort is the sort nat of naturals, with carrier $\mathbb{T}$. Intuitively, equality is "computable" or "decidable" on $\mathbb{T}$.

A non-equality sort is the sort real of reals. Intuitively, equality is ("co-semi-decidable", but) *not* (totally) decidable on $\mathbb{R}$.

Any many-sorted signature Σ can be *standardised* to a signature $\Sigma^{\mathscr{B}}$ by adjoining the sort bool together with the standard Boolean operations; and, correspondingly, any algebra A can be standardised to an algebra $A^{\mathscr{B}}$ by adjoining the algebra $\mathscr{B}$, together with equality at the equality sorts.

Example 2.2.3

(a) A *standard algebra of naturals* $\mathscr{T}$ is formed by standardising the algebra $\mathscr{T}_0$ (Example 2.1.2(b)), with (total) equality and order operations on $\mathbb{T}$:

$$\mathscr{T} = (\mathscr{T}_0, \mathscr{B}; \mathsf{eq}_{\mathsf{nat}}, \mathsf{less}_{\mathsf{nat}}).$$

Please cite this article in press as: B.C. Thompson et al., Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms, Appl. Math. Comput. (2009), doi:10.1016/j.amc.2009.04.058

(b) The *standardised ring of reals* (*cf.* Example 2.1.2(c)):

$$\mathscr{R} = (\mathscr{R}_0, \mathscr{B}).$$

Note that there is no (total) equality on $\mathbb{R}$, as discussed above.

2.3. Adding the naturals: T-standard signatures and algebras

Definition 2.3.1 (*T-standard signature*). A signature Σ is *T-standard* if (i) it is standard, and (ii) it contains the standard signature of naturals, i.e., $\Sigma(\mathscr{T}) \subseteq \Sigma$.

Definition 2.3.2 (*T-standard algebra*). Given an T-standard signature Σ, a corresponding Σ-algebra A is *T-standard* if it is an expansion of $\mathscr{T}$.

Any standard signature Σ can be *T-standardised* to a signature (Σ, T) by adjoining the sort nat and the operations $0, \mathsf{S}, \mathsf{eq}_{\mathsf{nat}}$ and $\mathsf{less}_{\mathsf{nat}}$. Correspondingly, any standard Σ-algebra A can be *T-standardised* to an algebra A^T by adjoining the carrier $\mathbb{T}$ together with the corresponding standard functions.

Throughout this paper, we will assume:

T-standardness Assumption. *The signature Σ, and the Σ-algebra A, are T-standard.*

Definition 2.3.3

(a) A *topological Σ-algebra* is a Σ-algebra with topologies on the carriers such that each of the basic Σ-functions is continuous.
(a) A (*T-*)*standard topological algebra* is a topological algebra which is also a (*T-*)standard algebra, such that the carriers $\mathbb{B}$ (and $\mathbb{T}$) have the discrete topology.

Example 2.3.4

(a) *Discrete algebras:* The standard algebras $\mathscr{B}$ and $\mathscr{T}$ of Booleans and naturals respectively (Sections 2.1 and 2.2) are topological (total) algebras under the discrete topology. All functions on them are trivially continuous, since the carriers are discrete.
(b) The T-standard *topological total real algebra* $\mathscr{R}^T$ is defined by

$$\mathscr{R}^T = (\mathscr{R}, \mathscr{T}; \mathsf{div}_{\mathsf{nat}}),$$

where $\mathscr{R}$ is the standardised ring of reals (Example 2.2.3(b)), $\mathscr{T}$ is the standard algebra of naturals (Example 2.2.3(a)), and $\mathsf{div}_{\mathsf{nat}} : \mathbb{R} \times \mathbb{T} \to \mathbb{R})$ is the total (continuous!) function defined by

$$\mathsf{div}_{\mathsf{nat}}(x,t) = \begin{cases} x/t & \text{if } t \neq 0, \\ 0 & \text{if } t = 0. \end{cases}$$

Note that $\mathscr{R}^T$ does *not* contain (total) Boolean-valued functions '<' or '=' on the reals, since they are not continuous; nor does it contain division of reals by reals, since that cannot be total and continuous. See [64,68,69] for discussions of these issues.

2.4. Metric algebra

A particular type of topological algebra is a *metric algebra*. This is a many-sorted standard algebra A with an associated metric:

$$A = (A_1, \ldots, A_r, \mathscr{R}; F_1^A, \ldots, F_k^A, \mathsf{d}_1^A, \ldots, \mathsf{d}_r^A),$$

where $\mathscr{R}$ is the standardised ring of reals (Example 2.2.3(b)), the carriers A_i are metric spaces with metrics $\mathsf{d}_i^A : A_i^2 \to \mathbb{R}, (i = 1, \ldots, r), F_1, \ldots, F_k$ are the Σ-function symbols other than $\mathsf{d}_1, \ldots, \mathsf{d}_k$, and the functions F_i^A are all continuous with respect to these metrics. The carriers $\mathbb{B}$ and $\mathbb{T}$ (included among the A_i) are given the *discrete metric*, which induces the discrete topology.

Clearly, metric algebras can be viewed as special cases of topological algebras.

Example 2.4.1. The real algebra $\mathscr{R}^T$ (Example 2.3.4(b)) can be recast as a metric algebra in an obvious way.

3. Stream algebras; Computable algebras

3.1. Adding streams to algebras: algebras $\overline{A}$ of signature $\overline{\Sigma}$

Let Σ be a T-standard signature, and A a T-standard Σ-algebra. We define an extension of Σ and a corresponding expansion of A.

We choose a set $\mathsf{S} \subseteq \mathrm{Sort}(\Sigma))$ of *pre-stream sorts*, and then extend Σ^N to a *stream signature* $\overline{\Sigma}^{\mathsf{S}}$ *relative to* **S**, as follows. With each $s \in \mathsf{S}$, associate a new *stream sort* $\bar{s}$, also written $\mathsf{nat} \to s$. Then

(a) $\mathrm{Sort}(\overline{\Sigma}^{\mathsf{S}}) = \mathrm{Sort}(\Sigma) \cup \{\bar{s} | s \in \mathsf{S}\}$;
(b) $\mathrm{Func}(\overline{\Sigma}^{\mathsf{S}})$ consists of $\mathrm{Func}(\Sigma)$, together with the *evaluation function*

$$\mathrm{eval}_s : (\mathsf{nat} \to s) \times \mathsf{nat} \to s$$

for each $s \in \mathsf{S}$.

Now we can expand A^T to a $(\overline{\Sigma}^{\mathsf{S}})$-*stream algebra* $\overline{A}^{\mathsf{S}}$ by adding for each $s \in \mathsf{S}$:

(i) the carrier for $\bar{s}$, which is the set

$$A_{\bar{s}} = \overline{A}_s = [\mathbb{T} \to A_s]$$

of all *streams on* A_s i.e., functions $u : \mathbb{T} \to A_s$;
(ii) the interpretation of eval_s on A as the function $\mathrm{eval}_s^A : [\mathbb{T} \to A_s] \times \mathbb{T} \to A_s$ which evaluates a stream at a time instant: $\mathrm{eval}_s^A(u, t) = u(t)$.

The algebra $\overline{A}^{\mathsf{S}}$ is the *(full) stream algebra over A with respect to* **S**. (We will usually omit explicit reference to the set **S**.) Note that the Instantiation Assumption does not hold (in general) for the signature of a stream algebra.

3.2. Expanding topological algebras to stream algebras

The algebraic expansion of an algebra A to a stream algebra $\overline{A}$ induces a corresponding topological expansion:

(a) The topological T-standardisation A^T, of signature (Σ, T), is constructed from A by giving the new carrier $\mathbb{T}$ the discrete topology.
(b) Next, a topology on A^T can be extended to one on $\overline{A}$ by giving the stream carriers $[\mathbb{T} \to A_s]$ the *product topology* based on A_s, where the *basic open sets* have the form

$$U = \{u \in \overline{A}_s | u(t_i) \in U_i \text{ for } i = 1, \ldots, n\} \tag{3.1}$$

for some $n > 0, t_1, \ldots, t_n \in \mathbb{T}$ and $U_1, \ldots, U_n$ open subsets of A_s.

With this topology, the operator eval_s^A is continuous.

Remark 3.2.1

(a) This topology is the same as the *inverse limit topology* on $[\mathbb{T} \to A_s]$ [71, Section 2.1].
(b) If A_s is *metrisable* by the metric d_s, then so is $[\mathbb{T} \to A_s]$ [71, Section 3.1], by the metric

$$\mathrm{d}_{\bar{s}}(u, v) =_{df} \sum_{t=0}^{\infty} \min\left(\mathrm{d}_s(u(t), v(t)), 2^{-t}\right).$$

3.3. Regular streams

Let B be a Σ-subalgebra of A. Then the stream algebra $\overline{B}$ over B is a $\overline{\Sigma}$-subalgebra of the stream algebra $\overline{A}$. Further, for any stream sort s, if we replace $[\mathbb{T} \to B_s]$ by *any* non-empty subset of it in the definition of $\overline{B}$, then we again obtain a "stream subalgebra" of $\overline{A}$. All subalgebras of $\overline{A}$ are obtained in this way.

Of special interest is the following subset of the set $\overline{A}_s$ of all streams in $\overline{A}$ of sort s. Define the set of *regular streams of A of sort s* by

$$(\overline{A}_s)_{\mathrm{reg}} = [\mathbb{T} \to A_s]_{\mathrm{reg}} = \{u \in [\mathbb{T} \to A_s] | \exists t_0 \forall t \geqslant t_0 (u(t) = \delta^s)\},$$

where δ^s is the *default element* of A_s (Section 2.1).

Further, for each T-standard Σ-algebra A we define $(\overline{A})_{\mathrm{reg}}$, the *regular stream algebra over A, to be the* $\overline{\Sigma}$-*subalgebra* of the stream algebra $\overline{A}$ obtained by *restricting*, at each stream sort $s, \overline{A}_s$ to the set $(\overline{A}_s)_{\mathrm{reg}}$ of *regular streams* of sort $\bar{s}$.

Lemma 3.3.1. *If B is a Σ-subalgebra of A then the regular stream algebra $(\overline{B})_{\mathrm{reg}}$ over B is a $\overline{\Sigma}$-subalgebra of the stream algebras $\overline{B}, (\overline{A})_{\mathrm{reg}}$, and $\overline{A}$.*

3.4. Dense regular subalgebras

We need the following general topological result.

Please cite this article in press as: B.C. Thompson et al., Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms, Appl. Math. Comput. (2009), doi:10.1016/j.amc.2009.04.058

Lemma 3.4.1. *If X is a topological space and Y a Hausdorff space, and $f : X \to Y$ and $g : X \to Y$ are both continuous, with $f \restriction D = g \restriction D$ for some dense subset D of X, then $f = g$.*

Let A be a Σ-algebra.

Definition 3.4.2. Dense subsetA $\mathrm{Sort}(\Sigma)$-indexed subset D is *dense* in A if for all Σ-sorts s, D_s is dense in A_s.

Lemma 3.4.3. *Let A be a T-standard topological Σ-algebra. Then*

(a) *if A is Hausdorff then so is $\overline{A}$;*
(b) *if D is a dense Σ-subalgebra of A then $\overline{D}$ and $\overline{D}_{\mathrm{reg}}$ are dense $\overline{\Sigma}$-subalgebras of $\overline{A}$.*

Proof. We prove the second part of (ii). Note first that $D_{\mathrm{bool}} = A_{\mathrm{bool}} = \mathbb{B}$ and $D_{\mathrm{nats}} = A_{\mathrm{nats}} = \mathbb{N}$. Now, for any stream sort s, by assumption D_s is dense in A_s. It remains to show that $(\overline{D}_s)_{\mathrm{reg}}$ is dense in $\overline{A}_s = [\mathbb{T} \to A_s]$. Choose any basic open set U in $[\mathbb{T} \to A_s]$, as in (3.1). Since D_s is dense in A_s, we can find $d_i \in U_i \cap D_s$ for $i = 1, \ldots, n$. Now define a stream u by

$$u_i(t) = \begin{cases} d_i & \text{if } t = t_i \quad \text{for } i = 1, \ldots, n, \\ \delta^s & \text{otherwise.} \end{cases}$$

Then $u \in U \cap (\overline{D}_s)_{\mathrm{reg}}$. □

From now on, we will assume that all our topological algebras satisfy the
Hausdorff Assumption. *A is a Hausdorff topological algebra.*

3.5. Computable algebras; Computable stream algebras

In order to investigate effective aspects of correctness specification of SCAs (Section 8), we need the concept of a *computable algebra* [4].

Definition 3.5.1. Recursive number algebraA *recursive number Σ-algebra* Ω is a Σ-algebra in which for each Σ-sort s, Ω_s is a recursive subset of $\mathbb{N}$ and for each Σ-function symbol $F : u \to s$,

$$F^{\Omega} : \Omega^u \to \Omega_s$$

is a total recursive function.

Let A be a T-standard Σ-algebra.

Definition 3.5.2. Effectively presented algebraAn *effective presentation* (α, Ω) for A consists of a recursive number Σ-algebra Ω and a Σ-epimorphism $\alpha : \Omega \to\ A$.

We assume that $\Omega_{\mathrm{nats}} = \mathbb{N}$ and $\alpha_{\mathrm{nats}} = \mathrm{id}_{\mathbb{N}}$.

A is said to be *effectively presented* by (α, Ω).

Next we define the $\mathrm{Sort}(\Sigma)$-*sorted congruence relation*

$$\equiv_\alpha = \langle \equiv_{\alpha,s} | s \in \mathrm{Sort}(\Sigma) \rangle$$

induced by α on Ω:

$$x \equiv_{\alpha,s} y \iff \alpha_s(x) = \alpha_s(y)$$

for all $x, y \in \Omega_s$. Note also that $A \cong \Omega / \equiv_\alpha$.

Definition 3.5.3. Computable algebraA is *computable* if it has an effective presentation (α, Ω) in which $\equiv_\alpha$ is decidable on Ω; that is, for each $s \in S, \equiv_{\alpha,s}$ is decidable.

Note, next, that the *stream algebra* $\overline{A}$ has uncountable carrier sets $\overline{A}_s$ and so it cannot be effectively presented. We therefore work with a regular subalgebra of $\overline{A}$.

Lemma 3.5.4. *Let D be a computable dense Σ-subalgebra of A. Then $\overline{D}_{\mathrm{reg}}$ is a computable dense Σ-subalgebra of $\overline{A}$.*

Proof. It is easy to extend an effective presentation for A with decidable equality to one for $\overline{A}$. The denseness of $\overline{D}_{\mathrm{reg}}$ in $\overline{A}$ follows from Lemma 3.4.3. □

Remark 3.5.5. An example of a computable dense subalgebra of an algebra, satisfying the assumptions of Lemma 3.5.4, is in the real algebra $\mathscr{R}^T$ (Example 2.3.4(b)), in which the rationals $\mathbb{Q}$ form a dense subset of $\mathbb{R}$.

4. Synchronous concurrent algorithms

4.1. Introduction to SCAs

An SCA is an algorithm given by a *network N* of *modules*, *channels*, *sources* and *sinks*. The modules compute and communicate in parallel; computation and data flow between modules is synchronised by a single *global clock* measuring discrete time, with values in $\mathbb{T}$.

For simplicity, assume that our T-standard Σ-algebra A contains only one carrier (apart from $\mathbb{B}$ and $\mathbb{T}$), also called A, of sort data. The data flowing between modules are taken from this set.

The SCA processes *streams* or infinite sequences $u(0), u(1), u(2), \ldots$ of data from A, clocked by $\mathbb{T}$. Such a stream is represented as a function $u : \mathbb{T} \to A$. Let $[\mathbb{T} \to A]$ be the set of all streams over A.

The network N in Fig. 1 is made from a sequence $M_1, \ldots, M_m$ of modules, a set $\mathbf{I}_{\mathsf{in}}$ of p sources and a set $\mathbf{I}_{\mathsf{out}}$ of q sinks. For simplicity we represent the modules, sources and sinks as natural numbers: $I = \{1, \ldots, m\}, \mathbf{I}_{\mathsf{in}} = \{1, \ldots, p\}$ and $\mathbf{I}_{\mathsf{out}} = \{1, \ldots, q\}$.

Communication between modules occurs by means of the *channels*. These have unit bandwidth and are unidirectional; that is, they can transmit only a single datum $a \in A$ at any one time in one direction. Channels may branch with the intention that the datum transmitted along the channel is "copied" and transmitted along each branch. However, channels may not merge.

A *module* is an atomic computing device capable of some specific internal processing. If module M_i has $k_i (> 0)$ input channels and one output channel then we assume the processing of M_i to be specified by a total function $\mathsf{F}_i : A^{k_i} \to A$ with the intention that if $a_1, \ldots, a_{k_i} \in A$ arrive on the module's k_i input channels (one datum per channel) at time t then M_i computes $\mathsf{F}_i(a_1, \ldots, a_{k_i})$, and transmits it at time $t+1$.

A *source* has no input and one output channel (which may branch). A network with p sources will process p *input streams* $x_1, \ldots, x_p \in [\mathbb{T} \to A]$, or, equivalently, a vector-valued input stream $\mathsf{x} \in [\mathbb{T} \to A]^p$ with $\mathsf{x}(t) = (x_1(t), \ldots, x_p(t))$.

The *sinks* each have one input and no output channel. They transmit the q output streams.

An SCA's architecture is given by three *wiring functions*

$$\begin{aligned} &\alpha : \mathbf{I} \times \mathbb{N} \to \mathbf{I}_{\mathsf{in}} \cup \mathbf{I} \\ &\beta : \mathbf{I} \times \mathbb{N} \to \{\mathsf{M}, \mathsf{S}\} \quad \text{(these symbols explained below)}. \\ &\mathsf{out} : \mathbf{I}_{\mathsf{out}} \to \mathbf{I} \end{aligned}$$

The map out is such that for each sink i, $\mathsf{out}(i)$ is the module that supplies i.

The maps α and β are partial functions that enumerate the inputs to a given module in the following way. Given a module $i \in I$ with k_i input channels, for $j = 1, \ldots, k_i$:

- if $\beta(i,j) = \mathsf{M}$ then input channel j of module i is the output channel of *module* $\alpha(i,j)$;
- if $\beta(i,j) = \mathsf{S}$ then input channel j of module i is the output channel of *source* $\alpha(i,j)$.

If $j \notin \{1, \ldots, k_i\}$ then $\alpha(i,j)$ and $\beta(i,j)$ are undefined.

Note that *feedback* is characterised by a module i with input j, where $\beta(i,j) = \mathsf{M}$ and $\alpha(i,j) = i$.

4.2. Informal explanation of operation

Initially, at time $t = 0$, each module i has some initial value $a_i \in A$ on its output channel. The *initial state* of N is specified by the vector $\mathsf{a} = (a_1, \ldots, a_m) \in A^m$. Thus we have:

Initialisation Assumption *At time $t = 0$ there is a single datum on every channel in the network.*

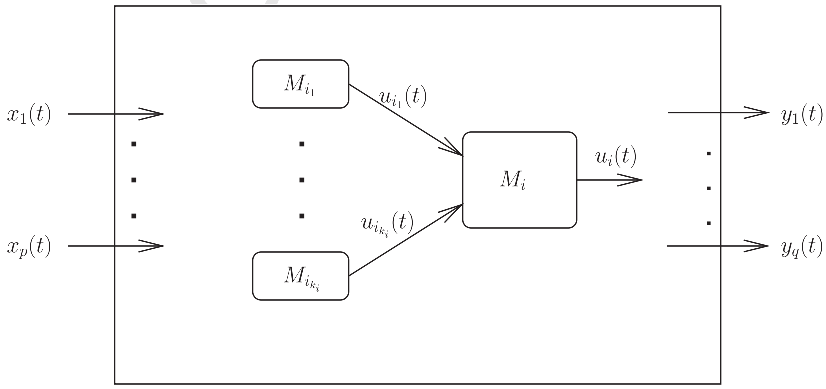

Fig. 1. An SCA network.

Each module i now computes by first reading its input data and then evaluating F_i on these data. The result of this evaluation is stored on the module's output channel.

From the above, we can infer two related assumptions:

4.2.1. Module totality and determinism assumptions

(a) *For each module in N, there is a datum on its output channel at time* $t+1$.
(b) *This item is uniquely determined by the data on its input channels at time t.*

Remark 4.2.1 (*Unit delay assumption*). The module totality and determinism assumptions entail a unit delay assumption: that it takes at most one time cycle for every module to read, evaluate and store in some order, and that any module taking less than one time unit is forced to wait until any slower modules have finished. Hence, as the clock beats $t = 0, 1, 2, \ldots,$ the modules concurrently pass data and compute with each module performing its tth read/evaluate/store sequence starting at time t and ending by time $t+1$. This is a reasonable assumption (assuming module totality!) since, even if we assume that computation time of a module function is (in principle) unbounded for arbitrary inputs, we can always "re-scale" time intervals to bound the computation time by one unit, for any given inputs.

We return to a discussion of the module totality and determinism assumptions in Section 4.6.

4.3. Algebraic formalisation

We start with a T-standard signature (Σ, T) and Σ-algebra A (Section 2.3). As stated above, we assume for convenience that there are only three carriers: A of data, $\mathbb{B}$ of Booleans and $\mathbb{T}$ of naturals (i.e., discrete time instants). Apart from the standard Boolean and arithmetic operations, there may be other functions, including (perhaps) equality on A.

Now we form the *module algebra* A^{F} by adding the module functions to A:

$$A^{\mathsf{F}} = (A; \mathsf{F}_1, \ldots, \mathsf{F}_m).$$

Note that if A is a topological algebra (as we are generally assuming) then in order that A^{F} can also be considered a topological algebra (with the given topology on A), we must assume:

Continuity of Module Functions Assumption. The module functions are all continuous on A.

Next, we extend the algebra A^{F} to the algebra $\overline{A^{\mathsf{F}}}$ of streams over A^{F} (Section 3.1), which we call the *module stream algebra*:

$$\overline{A^{\mathsf{F}}} = (A^{\mathsf{F}}, [\mathbb{T} \to A]; \mathsf{eval}).$$

Recall that the *input* to the network N consists of a tuple of initial values $\mathsf{a} = (a_1, \ldots, a_m) \in A^m$ and a stream tuple $\mathsf{x} = (x_1, \ldots, x_p) \in [\mathbb{T} \to A]^p$.

Lemma 4.3.1 (Network totality and determinism properties). *At each time* $t \in \mathbb{T}$ *there is a value output from each module, which can be determined uniquely from t, u and a.*

Proof. By a simple induction on t, using the initialisation assumption at $t = 0$, and the module totality and determinism assumptions at the induction step. □

For each module $i \in I$ we define its *module value function*

$$\mathsf{V}_i : A^m \times [\mathbb{T} \to A]^p \times \mathbb{T} \to A,$$

where $\mathsf{V}_i(\mathsf{a}, \mathsf{x}, t)$ is the value output by the module i at time t when the network is executed with initial data $\boldsymbol{a}$ and input streams $\boldsymbol{x}$. Note that these functions are total, by the network totality property.

Thus, the state of the network N is given by combining the module value functions $\mathsf{V}_1, \ldots, \mathsf{V}_m$ into the single *network state function*

$$\mathsf{V}^N : A^m \times [\mathbb{T} \to A]^p \times \mathbb{T} \to A^m \tag{4.1a}$$

defined by

$$\mathsf{V}^N(\mathsf{a}, \mathsf{x}, t) = (\mathsf{V}_1(\mathsf{a}, \mathsf{x}, t), \ldots, \mathsf{V}_m(\mathsf{a}, \mathsf{x}, t)). \tag{4.1b}$$

This defines the state of N at each time cycle. (We will sometimes drop the "network superscript" 'N'.)

The concurrent execution of the modules of N is modelled by the parallel evaluation of $\mathsf{V}_1, \ldots, \mathsf{V}_m$. We now develop general formulae for the computation of $\mathsf{V}_1, \ldots, \mathsf{V}_m$ and hence of V^N.

4.4. SCA network equations

We define $\mathsf{V}_1(\mathsf{a}, \mathsf{x}, t), \ldots, \mathsf{V}_m(\mathsf{a}, \mathsf{x}, t)$ for $\mathsf{a} = (a_1, \ldots, a_m) \in A^m, \mathsf{x} = (x_1, \ldots, x_p) \in [\mathbb{T} \to A]^p$, and $t = 0, 1, 2, \ldots,$ by simultaneous recursion on t.

Base case: Initialisation. For $i = 1, \ldots, m$:

$$\mathsf{V}_i(\mathsf{a}, \mathsf{x}, 0) = a_i. \tag{4.2}$$

Recursion step: State transition. Each module i has a functional specification $\mathsf{F}_i : A^{k_i} \to A$, where, if $b_1, \ldots, b_{k_i}$ arrive on i's input channels at time t then the value output by the module at time $t+1$ is $\mathsf{F}_i(b_1, \ldots, b_{k_i})$. Let the SCA have wiring functions α and β as described in Section 4.1. Then for $i = 1, \ldots, m$ and all $t \geqslant 0$

$$\mathsf{V}_i(\mathsf{a}, \mathsf{x}, t+1) = \mathsf{F}_i(b_{i1}, \ldots, b_{ik_i}), \tag{4.3a}$$

where for $j = 1, \ldots, k_i$

$$b_{ij} = \begin{cases} \mathsf{V}_{\alpha(i,j)}(\mathsf{a}, \mathsf{x}, t) & \text{if } \beta(i,j) = \mathsf{M}, \\ x_{\alpha(i,j)}(t) & \text{if } \beta(i,j) = \mathsf{S}. \end{cases} \tag{4.3b}$$

Remark 4.4.1. The Eqs. (4.2) and (4.3) together form a definition by *simultaneous primitive recursion*.

Remark 4.4.2. Stream transformation We can rewrite the network state function V (4.1) as a *stream transformation* by "abstraction" or "currying"; i.e., define

$$\widehat{\mathsf{V}} : A^m \times [\mathbb{T} \to A]^p \to [\mathbb{T} \to A]m, \tag{4.4a}$$

where

$$\widehat{\mathsf{V}}(\mathsf{a}, \mathsf{x})(t) = \mathsf{V}(\mathsf{a}, \mathsf{x}, t). \tag{4.4b}$$

We will reconsider these two forms, from a computational point of view, in Section 7.2.

4.5. Output specification

Note that the network state function V^N gives the values output by *every* module in the network. In many cases we are interested only in the values sent to the network's sinks. When the network has $q > 0$ sinks with $\mathbf{I}_{\mathrm{out}} = \{1, \ldots, q\}$ we use the function $\mathrm{out} : \mathbf{I}_{\mathrm{out}} \to \mathbf{I}$ (Section 4.1). Now define the *network output function*

$$\mathsf{V}_{\mathrm{out}} : A^m \times [\mathbb{T} \to A]^p \times \mathbb{T} \to A^q \tag{4.5a}$$

by

$$\mathsf{V}_{\mathrm{out}}(\mathsf{a}, \mathsf{x}, t) = (\mathsf{V}_{\mathrm{out}(1)}(\mathsf{a}, \mathsf{x}, t), \ldots, \mathsf{V}_{\mathrm{out}(q)}(\mathsf{a}, \mathsf{x}, t)), \tag{4.5b}$$

so that $\mathsf{V}_{\mathrm{out}}(\mathsf{a}, \mathsf{x}, t)$ is the vector of q values at the sinks of N at time t.

Note (*cf.* Remark 4.4.2) that we can also reformulate $\mathsf{V}_{\mathrm{out}}$ as a stream transformation by abstraction:

$$\widehat{\mathsf{V}}_{\mathrm{out}} : A^m \times [\mathbb{T} \to A]^p \to [\mathbb{T} \to A]q,$$

where

$$\widehat{\mathsf{V}}_{\mathrm{out}}(\mathsf{a}, \mathsf{x})(t) = \mathsf{V}_{\mathrm{out}}(\mathsf{a}, \mathsf{x}, t).$$

4.6. Generalisation of the model

There are many fruitful generalisations of our mathematical model, defined by weakening or generalising some of the conditions in our definition. We mention four here, of which the first two have already been studied, and the last two are suitable for future investigation.

(i) *Infinite SCAs*. These consist of infinitely many modules, each of which has only finitely many input and output channels, but each output channel may branch infinitely, copying data to infinitely many modules. There are many interesting examples, including infinite hardware systolic arrays [41,57] and infinite cellular automata. Infinite SCAs are useful for modelling parameterised families of finite SCAs.

(ii) *Non-unit delays*. One can generalise the timing properties of SCAs by relaxing the unit delay assumption (Section 4.2). Many interesting algorithms have this property. Note that the network totality and determinism properties still hold. Generalisation of the theory to such a network requires course-of-values recursive functions, and course-of-values inductive proofs [29], but is otherwise straightforward.

(iii) *Partial algebras of data*. This is a particularly interesting – and theoretically non-trivial – generalisation. Here we drop the module totality assumption, and (more generally) the assumption that the algebra A is total. This is of practical importance, in the case, for example, that A is an algebra of reals, that includes the operation of *real division*, and the Boolean operations of *equality* and *order*. In order that these operations be *continuous*, we must make them *partial*, as discussed in Example 3(b) and [64,68,69]. In such a framework, the module functions will also be partial, as will the network state function. We will also have to work with *partial streams*. We discuss this further in Section 8.2(i).

(iv) *Nondeterministic SCAs*. This is a closely related to the previous generalisation. (The connection between partiality and nondeterminism and continuity is discussed in [68].) Here we drop the Module Determinism Assumption (Section 4.2).

5. Examples of synchronous concurrent algorithms

Before developing our theory, and to illustrate the breadth of the concept of an SCA, we give, very briefly, five types of SCA, to which our theory has been applied. For all these examples (and especially neural networks) correctness is treated poorly in the existing literature. A number of examples are worked out in detail in [60].

5.1. Clocked digital systems

Here we have in mind electronic circuits made from Boolean logic, a global clock, and clocked storage elements such that every closed signal path passes through at least one such storage element [44]. Useful references on the specification and verification of such hardware systems are [6,8,38,34,28,49,56]. Case studies on modelling hardware with SCAs have been made in connection with

(i) *components*: in particular, the modelling of fixed length buffers and RS flip-flops as SCAs over bit strings [75,29,11];
(ii) *computers*: *cf*. our work with Harman cited in Section 1.3; and
(iii) *graphics processors*: *cf*. our work with Eker [12–15].

5.2. Systolic arrays

This notion was developed by Kung and others to isolate a class of algorithms particularly well-suited to avoiding the von Neumann bottleneck and to special-purpose implementation in VLSI circuits. As explained informally in [37], a systolic array is a (synchronous, concurrent) network of processing elements with the following properties:

(i) the network comprises a small number of different types of simple processor;
(ii) the network data and control flows have a regular and modular structure;
(iii) the array is such that each piece of input data is used many times, and
(iv) the algorithm employs much parallelism through pipelining and multiprocessing.

As an example, the buffer mentioned in the previous subsection has all these properties. Further examples and discussion can be found in [37,44,72,45,50,16,51,46,43]. We have applied our tools to the specification and verification of systolic arrays of many types [59,39,30,29,41,47,48,57].

5.3. Neural networks

The notion of an (artificial) neural network is due to McCulloch and Pitts [42]. These networks were first defined in order to provide a mathematical characterisation of logical aspects of activity levels in nervous systems in living organisms. Since then they have become of interest to researchers in mathematics, physics and engineering sciences, artificial intelligence and cognitive science. As witnessed by the many publications in this field, *neurocomputation* is a very active subject area [27,40,1].

Formalisation of the models as SCAs leads to clarification of the models' operation and specification [32,61].

5.4. Cellular automata

The notion of a cellular automaton was invented by von Neumann [73] in order to study evolution and self-reproduction in biological systems. Recently, many disparate applications of cellular automata have been discovered in mathematics, physics, chemistry and biology [7,76,55,77]. In general a cellular automaton can be described as a finite or infinite two-dimensional array of cells. Our tools are currently limited to algorithms with finitely many cells, so we can interpret finite cellular automata as SCAs.

5.5. Coupled map lattices

A coupled-map lattice (or CML) is a dynamical system based on discrete space, discrete time and continuous state. It is a generalisation of iterated map dynamical systems [10]. It can also be considered as a generalisation of a cellular automaton (which has a discrete state). CMLs are surveyed in [9,35]. They can also be interpreted as SCAs [33,31].

Please cite this article in press as: B.C. Thompson et al., Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms, Appl. Math. Comput. (2009), doi:10.1016/j.amc.2009.04.058

6. Specifications and correctness

First, we define the concept of S-indexed sets and mappings.

Let S be a finite non-empty set. An *S-indexed set* A is a family $A = \langle A_s | s \in S \rangle$.

Given two S-indexed sets $A = \langle A_s | s \in S \rangle$ and $B = \langle B_s | s \in S \rangle$, an *$S$ -indexed mapping* from A to B is a family $f = \langle \langle f_s | s \in S \rangle$ where $f_s : A_s \to B_s$ for each $s \in S$. In symbols we write $f : A \to B$.

6.1. Syntax: terms and conditional equations

(a) $\mathsf{T}(\Sigma)$ is the $\mathsf{Sort}(\Sigma)$-indexed set of Σ-terms (denoted $t, \ldots$), where the set $\mathsf{T}_s(\Sigma)$ of such terms of sort s (denoted $t^s, \ldots$) is defined (simultaneously over S) by

$$t^s ::= \mathrm{x}^s | c | F(t_1^{s_1}, \ldots, t_m^{s_m}),$$

where x^s is a variable of sort s, c is a constant symbol of sort s, and F is a Σ-function symbol of type $s_1 \times \cdots \times s_m \to s (m > 0)$.

(b) $\mathsf{Eq}(\Sigma)$ is the set of *Σ-equations* $(t_1^s = t_2^s)$ between Σ-terms of the same Σ-sort. We also write equations as $e, e', \ldots$.

(d) $\mathsf{CondEq}(\Sigma)$ is the set of *Σ-conditional equations*

$$e_1 \wedge \cdots \wedge e_n \to e \quad (n \geqslant 0).$$

6.2. Semantics: satisfaction

A Σ-conditional equational specification is a pair (Σ, E) where $E \subseteq \mathsf{CondEq}(\Sigma)$.

Let A be a Σ-algebra. The concepts:

(a) *A satisfies the Σ-conditional equation e*, written $A \models e$, and

(b) *A satisfies the conditional equational specification* (Σ, E), written $A \models E$, are defined in the standard way.

6.3. Correctness of an SCA

We introduce the concept of *relational correctness* of an SCA.

Suppose that a computational task or behaviour is specified by a relation of the form

$$R \subseteq A^m \times [\mathbb{T} \to A]^p \times \mathbb{T} \times A^q \tag{6.1}$$

such that for each $\mathsf{a} \in A^m, \mathsf{x} \in [\mathbb{T} \to A]^p$, $\mathsf{t} \in \mathbb{T}$ and $\mathsf{y} \in A^q$,

$$R(\mathsf{a}, \mathsf{x}, t, \mathsf{y})$$

means that $\boldsymbol{y}$ is acceptable as an output for an initial state $\boldsymbol{a}$ and input stream $\boldsymbol{x}$ at time t. We call R the *specifying relation*.

There are various ways of formulating correctness w.r.t. a specifying relation R, depending on how we treat *initialisations* and *inputs*: We can consider a *particular initialisation*, or *all initialisations from some subset of A^m* (possibly all of A^m). Similarly, we can consider a *particular input stream*, or *all inputs from some subset of* $[\mathbb{T} \to A]^p$ (possibly all of $[\mathbb{T} \to A]^p$). To take a typical (and useful) case:

Definition 6.3.1. Correctness for initialisations and inputs from some set For any sets $P \subseteq A^m$ of initialisations and $Q \subseteq [\mathbb{T} \to A]^p$ of inputs, the SCA is *correct w.r.t. P, Q and R* if

$$(\forall \mathsf{a} \in P)(\forall \mathsf{x} \in Q)(\forall t \in \mathbb{T})\ R(\mathsf{a}, \mathsf{x}, t, \mathsf{V}_{\mathrm{out}}(\mathsf{a}, \mathsf{x}, t)). \tag{6.2}$$

Here the output value function $\mathsf{V}_{\mathrm{out}} : A^m \times [\mathbb{T} \to A]^p \times \mathbb{T} \to A^q$ (4.5) is a *selection function* for the relation R, relative to P and Q.

Note that if we want to specify the behaviour of the whole state of the SCA, we can simply modify the above definition by replacing $\mathsf{V}_{\mathrm{out}}$ by $\boldsymbol{V}$.

7. Primitive recursive computability on stream algebras

7.1. Simultaneous primitive recursion on abstract algebras

In [62], we developed a theory of *abstract computability on standard abstract many-sorted algebras*. We formulated a *generalised Church–Turing thesis*, which identifies a certain class of functions (namely, 'μPR' or '***While***' computable) with functions algorithmically computable on such structures.

We also developed a theory of *generalised primitive recursion* over T-standard algebras A. These generalise Kleene's primitive recursion functions on $\mathbb{N}$ [36], and form a proper subclass of the class μPR.

Briefly, we define a class PR(A) of PR (primitive recursive) functions on A, generated by schemes for (i) the initial functions and constants, i.e., the interpretations on A of the Σ-functions, (ii) projections, (iii) definition by cases, (iv) composition, and (v) simultaneous primitive recursion, where the function

$$\mathsf{f} : A^m \times [\mathbb{T} \to A]^p \times \mathbb{T} \to A^m$$

is defined by

$$\begin{aligned} \mathsf{f}(\mathsf{a},\mathsf{x},0) &= \mathsf{g}(\mathsf{a},\mathsf{x}) \\ \mathsf{f}(\mathsf{a},\mathsf{x},t+1) &= \mathsf{h}(\mathsf{a},\mathsf{x},t,\mathsf{f}(\mathsf{a},\mathsf{x},t)) \end{aligned} \tag{7.1}$$

with

$$\begin{aligned} &\mathsf{g} : A^m \times [\mathbb{T} \to A]^p \to A^m, \\ &\mathsf{h} : A^m \times [\mathbb{T} \to A]^p \times \mathbb{T} \times A^m \to A^m. \end{aligned}$$

This is a simple recursion for an A^m-valued function, equivalent to an m-fold *simultaneous recursion* defining m A-valued functions. Note that the defining Eqs. (4.2) and (4.3) for the network value functions in Section 4.4 are a special case of this.

Note also that the class μPR(A) is formed from PR(A) by adding a scheme for the (constructive) least number operator.

Lemma 7.1.1. *For any topological algebra A, all functions in* PR(A) *are continuous.*

This is proved, in fact for all μPR functions, in [65].

We now consider a class of relations on algebras broader than primitive recursiveness.

Definition 7.1.2 (*Equationally PR definable relations*). A relation $R \subseteq A^u$ on an algebra A is *equationally* PR *definable* on $A(\mathrm{PR} = (A))$ if there are PR(A) functions $\mathsf{f}_R, \mathsf{g}_R : u \to s$ for some Σ-sorts u, s such that for all $\mathsf{a} \in A^u$

$$\mathsf{a} \in R \Longleftrightarrow \mathsf{f}_R(\mathsf{a}) = \mathsf{g}_R(\mathsf{a}). \tag{7.2}$$

We call the r.h.s. of (7.2) a *PR defining equation* for R, and the pair $(\mathsf{f}_R, \mathsf{g}_R)$ PR *defining functions* for R.

Remark 7.1.3 (*Comparison of PR and* PR$^=$ *computability*). Note that PR$^=(A)$ is (in general) a strictly broader concept than PR(A). For on the one hand, any PR(A) relation R is also PR$^=(A)$, since (if χ_R is the characteristic function of R)

$$a \in R \Longleftrightarrow \chi_R(a) = \mathsf{true}$$

(a special case of (7.2)). But on the other hand, the range sort s (in Definition 7.1.2) need not be an equality sort (*cf.* Section 2.2), i.e., equality at sort s is *not* necessarily PR.

7.2. Primitive recursion on stream algebras

Assume for simplicity (as stated in Section 4) that our T-standard Σ-algebra A contains (apart from $\mathbb{B}$ and $\mathbb{T}$) only one carrier A of data.

Consider now PR stream valued functions or *stream transformers* on $\overline{A}$:

$$f : [\mathbb{T} \to A]^m \times A^n \to [\mathbb{T} \to A]. \tag{7.3}$$

It has been shown [63] that all PR stream transformers f of type as in (7.3) have the form

$$f(u_1, \ldots, u_m, a_1, \ldots, a_n) = u_{f_0(u_1,\ldots,u_m,a_1,\ldots,a_n)}$$

for some PR function

$$f_0 : [\mathbb{T} \to A]m \times A^n \to \mathbb{T}.$$

In other words, PR stream transformers are not "interesting": *they only return one of the input streams, the choice of which one depending primitive recursively on the inputs.*

We therefore consider a broader, more interesting class of stream transformers, namely the class λPR($\overline{A}$) formed from PR($\overline{A}$) by adding a scheme for *stream* (λ)-*abstraction*. Note that a function f as in (7.3) will be in λPR($\overline{A}$) if its "cartesian" or "uncurried" form

$$\check{f} : [\mathbb{T} \to A]m \times A^n \times \mathbb{T} \to A$$

is in PR($\overline{A}$), where

$$\check{f}(\mathsf{u},\mathsf{a},t) = f(\mathsf{u},\mathsf{a})(t).$$

Note also that we can define the class λPR$^=(A)$ of *equational* λPR *definable* relations on A, analogously to PR$^=(A)$ (Definition 7.1.2).

Now assume A, and hence $\overline{A}$, are topological algebras.

Lemma 7.2.1. *For f as in (7.3), f is continuous iff $\check{f}$ is continuous.*

Hence, from Lemma 7.1.1:

Lemma 7.2.2. *All functions in $\lambda\mathrm{PR}(\overline{A})$ are continuous.*

Corollary 7.2.3. *Let A be Hausdorff T-standard algebra, and D a dense subalgebra of A. Let f and g be $\lambda\mathrm{PR}$ functions on $\overline{A}$. Then the following are equivalent:*

(i) $f = g$ on $\overline{A}$
(ii) $f = g$ on $\overline{D}$
(iii) $f = g$ on $(\overline{A})_{\mathrm{reg}}$
(iv) $f = g$ on $\overline{D}_{\mathrm{reg}}$.

Proof. From Lemmas 3.4.1, 3.4.3 and 7.2.2. □

7.3. Primitive recursiveness of SCA state function

Recall the module algebra A^{F}, module stream algebra $\overline{A^{\mathrm{F}}}$, module value functions $\mathrm{V}_1, \ldots, \mathrm{V}_m$, network state function $\boldsymbol{V}$ and network output function $\mathrm{V}_{\mathrm{out}}$. (Sections 4.3–4.5).

Theorem 1. *For any SCA over a T-standard algebra A, with module algebra A^{F}:*

(a) *The module value functions $\mathrm{V}_1, \ldots, \mathrm{V}_m$, network state function V and network output function $\mathrm{V}_{\mathrm{out}}$ are in $\mathrm{PR}(\overline{A^{\mathrm{F}}})$.*
(b) *The abstracted forms $\widehat{\mathrm{V}}$ and $\widehat{\mathrm{V}}_{\mathrm{out}}$ are in $\lambda\mathrm{PR}(A^{\mathrm{F}})$.*

Proof. The main step in (a) is to show that V is definable (uniquely) from the module functions by simultaneous primitive recursion (Eqs. (4.2) and (4.3) as special cases of scheme (7.1)), using a simple inductive argument parallelling the PR definition. □

7.4. Computability of relational correctness specification

Recall the Definition 6.3.1 of correctness for a specifying relation R with initialisations and input streams from sets $P \subseteq A^m$ and $Q \subseteq [\mathbb{T} \to A]^p$, respectively:

$$(\forall \mathrm{a} \in P)(\forall \mathrm{x} \in Q)(\forall t \in \mathbb{T})\ R(\mathrm{a}, \mathrm{x}, t, \mathrm{V}_{\mathrm{out}}(\mathrm{a}, \mathrm{x}, t)). \tag{7.4}$$

Theorem 2. *For an SCA over a Hausdorff T-standard algebra A, with continuous module functions, and module algebra A^{F}, suppose*

(a) P, Q and R are $\lambda\mathrm{PR}^{=}$ on $\overline{A^{\mathrm{F}}}$,
(b) A^{F} has a dense computable subalgebra D.

Then we can effectively construct a computable algebra $C_{V,P,Q,R}$ with signature $\Sigma_{V,P,Q,R}$ that expands $\overline{D}_{\mathrm{reg}}$ by functions, and equations $e_P,\ e_Q, e_{V,R}$ over $\Sigma_{V,P,Q,R}$ such that the following are equivalent:

(i) *V is correct w.r.t P, Q and R, i.e., (7.4) holds;*
(ii) *$C_{V,P,Q,R} \models e_P \wedge e_Q \to e_{V,R}$.*

Consequently, correctness in the sense of (i) can be effectively reduced to the validity of conditional equations in a computable algebra and is co-recursively enumerable.

Proof. We prove $(i) \Rightarrow (ii)$. Consider the statement

$$\mathrm{a} \in P \wedge \mathrm{x} \in Q \to R(\mathrm{a}, \mathrm{x}, t, \mathrm{V}_{\mathrm{out}}(\mathrm{a}, \mathrm{x}, t)). \tag{7.5}$$

Let $(\mathrm{f}_P, \mathrm{g}_P), (\mathrm{f}_Q, \mathrm{g}_Q)$ and $(\mathrm{f}_R, \mathrm{g}_R)$ be $\lambda\mathrm{PR}$ defining functions for the sets P, Q and R respectively. By assumption and Theorem 1, these functions, as well as V, are all $\lambda\mathrm{PR}$ on A^{F}. By assumption (i), (7.5) holds on A^{F}, and therefore it holds on $\overline{D}_{\mathrm{reg}}$, by Corollary 7.2.3 (with A replaced by A^{F}). Since D is a computable algebra, so is $\overline{D}_{\mathrm{reg}}$, by Lemma 3.5.4, with effective presentation (α, Ω) say (recall Section 3.5). Now expand $\overline{D}_{\mathrm{reg}}$ to the algebra

$$C_{V,P,Q,R} =_{df} (\overline{D}_{\mathrm{reg}}; \mathrm{V}, \mathrm{f}_P, \mathrm{g}_P, \mathrm{f}_Q, \mathrm{g}_Q, \mathrm{f}_R, \mathrm{g}_R) \tag{7.6}$$

with signature $\Sigma_{V,P,Q,R}$. Since the seven functions shown in (7.6) are all $\lambda\mathrm{PR}$ over $\overline{D}_{\mathrm{reg}}$, they are "$\alpha$-computable" on $\overline{D}_{\mathrm{reg}}$. (This follows from the soundness theorem for abstract computability [68]). Hence, $C_{V,P,Q,R}$ is also a computable algebra. Moreover, (7.5) has the form of a conditional equation $e_P \wedge e_Q \ \to\ e_{V,R}$ over $C_{V,P,Q,R}$. Hence (ii) follows. □

Please cite this article in press as: B.C. Thompson et al., Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms, Appl. Math. Comput. (2009), doi:10.1016/j.amc.2009.04.058

That the correctness problem is co-r.e. follows from the α-computability of the functions noted above, together with the decidability of $\equiv_\alpha$.

Example 7.4.1. Let A be the T-standard topological algebra $\mathscr{R}^T$ (Example 2.3.4(b)). A has a dense computable subalgebra $D = \mathscr{Q}^T$ consisting of the rationals $\mathbb{Q}$ with the same signature as A. As a very simple example of a specifying relation that is λPR$^=$ over $\overline{A}^F$ (in fact, PR$^=$ over $\overline{A}$), we could take

$$R(a, x_1, x_2, t, y) \iff x_1(t)^2 + x_2(t)^2 = y^2,$$

where x_1 and x_2 are input stream variables and y is an output variable. A more interesting example would be something like

$$R'(a, x_1, x_2, t, y) \iff (0 < y^2) \wedge (y^2 < x_1(t)^2 + x_2(t)^2),$$

i.e., a Boolean combination of equalities and inequalities between λPR terms.

The problem here is that equality and order, as *total predicates* on $\mathbb{R}$, are not computable [64,68,69]. In this paper, we have solved this problem for equality by using the *computable subalgebra* $\mathscr{Q}^T$ of $\mathscr{R}^T$, together with the concept of *equational PR definability* (Definition 7.1.2).

To handle '<', we can proceed similarly, extending the model of PR computability on stream algebras PR$(\overline{A})$ to a model PR$^{=,<}(\overline{A})$, in which '<', as well as '=', is allowed as an extra basic predicate. And so on, for other non-computable predicates used in specifications.

We could ask if condition (ii) in Theorem 2 could be replaced by a statement that the conditional equation is a *valid consequence* of a certain set of axioms, i.e., a completeness result. However the correctness problem for conditional equations in stream algebras is complete Π^0_1 [4] and so completeness fails.

In this direction, however, we can prove the following, using results of Bergstra and Tucker on initial algebra semantics [2–5].

Theorem 3. *With the hypotheses of Theorem 2, we can effectively construct a finite equational specification* $(\Sigma_{V,P,Q,R}, E_{V,P,Q,R})$ *and equations* $e_P,\ e_Q, e_{V,R}$ *over* $\Sigma_{V,P,Q,R}$ *such that the following are equivalent:*

(i) V is correct w.r.t. P, Q and R, i.e., (7.4) holds;
(ii) $T(\Sigma_{V,P,Q,R}, E_{V,P,Q,R}) \models\ e_P \wedge e_Q \to e_{V,R}$,

where $T(\Sigma_{V,P,Q,R}, E_{V,P,Q,R})$ is the $\Sigma_{V,P,Q,R}$-term model generated by $E_{V,P,Q,R}$.

Other work on the use of higher order equational methods in hardware verification is presented in [47,48,57].

8. Concluding remarks

Since the concept of an SCA is quite general, our methods and results provide a *unified model* for the various classes of algorithms, architectures and physical models mentioned in the introduction, as well as for several others.

We can also construct a *unified model* for SCA networks and analog networks. This is done in [71], and summarised in the following subsection.

8.1. Comparison with continuous-time analog networks

In [70] we develop a theory of analog networks. There are some striking resemblances – and differences – between that theory and the theory of SCAs developed here.

Both models have global clocks. Whereas the SCA model has *discrete time*, modelled by the naturals, the analog model has *continuous time*, modelled by the set $\mathbb{T} = \mathbb{R}^{\geqslant 0}$ of non-negative reals. Now streams on A are taken to be *continuous* functions from $\mathbb{R}^{\geqslant 0}$ to A, and the set of all such streams is denoted $\mathscr{C}[\mathbb{T}, A]$. Nevertheless, there are formal resemblances in the networks of modules: compare Fig. 1 in this paper and Fig. 2 in [70]. The main difference is this (writing F_i for the module function for M_i). In SCAs (*cf.* Fig. 1) if the input channels to module M_i carry streams $u_{i_1}, \ldots, u_{i_{k_i}}$ and the output channel carries the stream u_i, then for all $t \in \mathbb{T}$

$$\mathsf{F}_i(u_{i_1}(t), \ldots, u_{i_{k_i}}(t)) = u_i(t+1), \tag{8.1}$$

i.e., F_i acts on *input data* $u_1(t), \ldots, u_{k_i}(t)$ to produce an *output datum* $u_i(t+1)$.

In analog networks, by contrast, the module functions (which we now write as $\widehat{\mathsf{F}}_i$) act on *input streams* to produce *output stream*:

$$\widehat{\mathsf{F}}_i(u_{i_1}, \ldots, u_{i_{k_i}}) = u_i. \tag{8.2}$$

The main consequence of this is that whereas with SCAs, it is very simple to find (or construct) the network state function, by a simultaneous primitive recursion (Section 4.4); for analog networks a much more sophisticated approach is required. To make any progress, we must first assume that F_i is *causal*, where $\mathsf{F} : \mathscr{C}[\mathbb{T}, A]^k \to \mathscr{C}[\mathbb{T}, A]$ is said to be causal if for all $\mathsf{u}, \mathsf{v} \in \mathscr{C}[\mathbb{T}, A]^k$ and $t > 0$,

$$\mathsf{u}\upharpoonright_{[0,t)} = \mathsf{v}\upharpoonright_{[0,t)} \Rightarrow \mathsf{F}(\mathsf{u})(t) = \mathsf{F}(\mathsf{v})(t).$$

In such a case we can find the network state function as the *fixed point* of a contracting functional [70].

In order to provide a *unified model* for these two types of networks, we first define, for an SCA, an "abstracted" version of the network state function

$$\widehat{\mathsf{F}}_i(u_{i_1},\ldots,u_{i_{k_i}})(t) =_{df} \begin{cases} a_i & \text{if } t = 0, \\ \mathsf{F}_i(u_{i_1}(t-1),\ldots,u_{i_{k_i}}(t-1)) & \text{if } t > 0, \end{cases} \tag{8.3}$$

(where a_i is the output of F_i at $t = 0$) to mimic the analog stream transformer (8.2).

Note now that in the case of SCAs,

(1) streams on $\mathbb{T}$ are automatically continuous, since $\mathbb{T}$ is a discrete set;
(2) from (8.3) it can easily be seen that the module functions, and hence the network state function $\boldsymbol{V}$ (or $\widehat{\mathsf{V}}$; *cf.* 4.4), are automatically causal – something that can by no means be assumed for analog networks.These points explain the comparative simplicity of construction of network state functions for SCAs, compared to analog networks, as noted above. But note two further points:
(3) The SCA state function $\boldsymbol{V}$ can *also* be constructed as the fixed point of a contracting functional, thus providing a unified model for these two types of networks. Details are given in [71].
(4) The construction in (3) is along the lines of Kleene's proof of his first recursion theorem [36, Thm XXVI]. However the fixed point in Kleene's construction is obtained as a limit of a sequence of *partial streams*, starting with the empty stream, whereas the fixed point in [71] is obtained as a limit of a sequence of *total streams*, starting with an arbitrary stream. (At stage n, the approximations by these two methods give identical values at the first n places.) Thus, Kleene's framework involves *partial* functions, unlike the framework here and in [70,71]. See, however, Section 8.2(1) below.

8.2. Proposed generalisations of the theory

(1) *Partial module functions*. We want to investigate the theory of some of the generalisations of SCAs listed in Section 4.6, particularly the last two, where, from considerations of continuity, we may have to drop the module totality and determinism assumptions, and (hence also) the unit delay assumption, (Section 4.2), and deal with models based on partial data algebras [68], with partial (and nondeterministic) module and network functions, and partial (and non-deterministic) streams. We will also have to replace our global clock model with a system of *local clocks*. We conjecture that this will be equivalent to the global clock model, with the totality, determinism and unit delay assumptions, in the special case that the algebra A, and the function modules, are total.
(2) *Specifiability based on* μPR *(semi-)computability*. In Section 7, we investigated computability of specifications based on $\mathrm{PR}(\overline{A})$ computable relations. It would be worth investigating the same problem for $\mu\mathrm{PR}(\overline{A})$ computable – or semicomputable – relations. In this way, we could get *non-total* relational specifications, which might fit in well with a partial function/ partial stream model (see point (1) above).

Acknowledgements

We thank the following colleagues for many useful and stimulating discussions on the subject: J.A. Bergstra, B.R.J. McConnell, M.J. Poole, R. Stephens, W.B. Yates, S.M. Eker, K. Hobley, A.R. Martin, and A.V. Holden. We also thank two anonymous referees for helpful comments. The research of the second and third authors was supported in part by a grant from EPSRC (Engineering and Physical Sciences Research Council, UK). The research of the third author was supported in part by a grant from NSERC (Natural Sciences and Engineering Research Council, Canada).

References

[1] J. Anderson, E. Rosenfeld (Eds.), Neurocomputing: Foundations of Research, MIT Press, 1988.
[2] J. Bergstra, J.V. Tucker, A characterisation of computable data types by means of a finite equational specification method, in: J. de Bakker, J. van Leeuwen (Eds.), Seventh International Colloquium on Automata, Languages and Programming, Noordwijkerhout, Lecture Notes in Computer Science, vol. 85, Springer-Verlag, The Netherlands, July 1980, pp. 76–90.
[3] J. Bergstra, J.V. Tucker, The completeness of the algebraic specification methods for data types, Information and Control 54 (1982) 186–200.
[4] J. Bergstra, J.V. Tucker, Algebraic specifications of computable and semicomputable data types, Theoretical Computer Science 50 (1987) 137–181.
[5] J. Bergstra, J.V. Tucker, Equational specifications, complete term rewriting systems and computable and semicomputable algebras, Technical Report CS-20-92, Department of Computer Science, Swansea University, Swansea, Wales, 1992.
[6] G. Birtwhistle, P. Subrahmanyam (Eds.), VLSI Specification, Verification and Synthesis, Kluwer, 1988.
[7] C. Choffrut (Ed.), Automata networks: LITP Spring School on Theoretical Computer Science, Lecture Notes in Computer Science, vol. 316, Springer-Verlag, 1986.
[8] L. Claesen (Ed.), Proceedings of the IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design, Elsevier, 1989.
[9] J. Crutchfield, K. Kaneko, Phenomenology of spatio-temporal chaos, in: H. Bai-lin (Ed.), Directions in Chaos, University of Illinois Press, 1987.
[10] R. Devaney, An Introduction to Chaotic Dynamical Systems, Addison-Wesley, 1989.
[11] A. Dew, A. King, J.V. Tucker, A. Williams, The prioritiser experiment: estimation and measurement of computation time in VLSI, in: K. McEvoy, J.V. Tucker (Eds.), Theoretical Foundations of VLSI Design, Cambridge Tracts in Theoretical Computer Science, vol. 10, Cambridge University Press, 1990, pp. 347–401.

[12] S. Eker, Foundations for the design of rasterisation algorithms and architectures, Ph.D. Thesis, School of Computer Studies, University of Leeds, 1990.
[13] S. Eker, V. Stavridou, J.V. Tucker, Verification of synchronous concurrent algorithms using obj3*. A case study of the pixel planes architecture, in: G. Jones, M. Sheeran (Eds.), Designing Correct Circuits, Springer-Verlag, 1991, pp. 231–252.
[14] S. Eker, J.V. Tucker, Specification, derivation and verification of concurrent line drawing algorithms and architectures, in: R. Earnshaw (Ed.), Theoretical Foundations of Computer Graphics and CAD, Springer-Verlag, 1988, pp. 449–516.
[15] S. Eker, J.V. Tucker, Specification and verification of synchronous concurrent algorithms: a case study of the pixel planes architecture, in: P. Dew, R. Earnshaw, T. Heywood (Eds.), Parallel Processing for Computer Vision and Display, Addison-Wesley, 1989, pp. 16–49.
[16] D. Evans (Ed.), Systolic Algorithms, Gordon and Breach, 1992.
[17] A. Fox, N. Harman, Algebraic models of correctness for microprocessors, Formal Aspects of Computer Science 12 (2000) 298–312.
[18] A. Fox, N. Harman, Algebraic models of correctness for abstract pipelines, Journal of Logic and Algebraic Programming 57 (2003) 71–107.
[19] N. Harman, Formal specifications for digital systems, Ph.D. Thesis, School of Computer Studies, University of Leeds, 1989.
[20] N. Harman, Algebraic models of behaviour and correctness of smt and cmt processors, Journal of Logic and Algebraic Programming 74 (2007) 32–56.
[21] N. Harman, J.V. Tucker, Clocks, retimings, and the formal specification of a UART, in: G. Milne (Ed.), The Fusion of Hardware Design and Verification (Proceedings of the IFIP Working Group 10.2 Working Conference), North-Holland, 1988, pp. 375–396.
[22] N. Harman, J.V. Tucker, Formal specifications and the design of verifiable computers, in: Proceedings of 1988 UK IT Conference, Held Under the Auspices of the Information Engineering Directorate of the Department of Trade and Industry, Institute of Electrical Engineers, 1988, pp. 500–503.
[23] N. Harman, J.V. Tucker, The formal specification of a digital correlator, I: user specification process, in: K. McEvoy, J.V. Tucker (Eds.), Theoretical Foundations of VLSI Design, Cambridge University Press, 1990, pp. 161–262.
[24] N. Harman, J.V. Tucker, Consistent refinements of specifications for digital systems, in: P. Prinetto (Ed.), Correct Hardware Design Methodologies (Proceedings ESPRIT BRA 3216 Workshop), Elsevier, 1991, pp. 281–304.
[25] N. Harman, J.V. Tucker, Algebraic methods and the correctness of microprocessors, in: G. Milne, L. Pierre (Eds.), Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science, vol. 683, Springer-Verlag, 1993, pp. 92–108.
[26] N. Harman, J.V. Tucker, Algebraic models of microprocessors: architecture and organisation, Acta Informatica 33 (1996) 421–456.
[27] R. Hecht-Nielson, Neurocomputation, Addison-Wesley, 1990.
[28] C. Hoare, M. Gordon (Eds.), Mechanical Reasoning and Hardware Design, Prentice-Hall, 1992.
[29] K. Hobley, The specification and verification of synchronous concurrent algorithms, Ph.D. Thesis, School of Computer Studies, University of Leeds, 1990.
[30] K. Hobley, B. Thompson, J.V. Tucker, Specification and verification of synchronous concurrent algorithms: a case study of a convoluted algorithm, in: G. Milne (Ed.), The Fusion of Hardware Design and Verification (Proceedings of IFIP Working Group 10.2 Working Conference), North-Holland, 1988, pp. 347–374.
[31] A. Holden, M. Poole, J.V. Tucker, H. Zhang, Coupled map lattices as computational systems, Chaos 2 (1992) 367–376.
[32] A. Holden, B. Thompson, J.V. Tucker, The computational structure of neural systems, in: A. Holden, V. Kryukov (Eds.), Neurocomputers and Attention I: Neurobiology, Synchronisation and Chaos, Manchester University Press, 1990, pp. 223–240.
[33] A. Holden, B. Thompson, J.V. Tucker, Can excitable media be considered as computational systems?, Physica D 49 (1991) 240–246
[34] G. Jones, M. Sheeran (Eds.), Designing Correct Circuits, Springer-Verlag, 1991.
[35] K. Kaneko (Ed.), Coupled Map Lattices: Theory and Applications, John Wiley & Sons, 1993.
[36] S. Kleene, Introduction to Metamathematics, North-Holland, 1952.
[37] H.-T. Kung, Why systolic architectures?, Computer 15 (1982) 37–47
[38] A. Leeser, G. Brown (Eds.), Hardware Specification, Verification and Synthesis: Mathematical Aspects, Lecture Notes in Computer Science, vol. 408, Springer-Verlag, 1989.
[39] A. Martin, J.V. Tucker, The concurrent assignment representation of synchronous systems, Parallel Computing 9 (1988) 227–256.
[40] J. McClelland, D. Rumelhart, Parallel Distributed Processing, Bradford Books, vol. 1, MIT Press, 1986.
[41] B. McConnell, J.V. Tucker, Infinite synchronous concurrent algorithms: the specification and verification of a hardware stack, in: H. Schwichtenberg (Ed.), Logic and Algebra for Specification, Springer-Verlag, 1993.
[42] W. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, Bulletin of Mathematical Biophysics, Series 2 (5) (1942) 115–133.
[43] K. McEvoy, J.V. Tucker (Eds.), Theoretical Foundations of VLSI Design, Cambridge Tracts in Theoretical Computer Science, vol. 10, Cambridge University Press, 1990.
[44] C. Mead, L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.
[45] A. Megson, An Introduction to Systolic Algorithm Design, Oxford University Press, 1992.
[46] A. Megson (Ed.), Transformational Approaches to Systolic Design, Chapman and Hall, 1993.
[47] K. Meinke, L. Steggles, Specification and verification in higher order algebra: a case study of convolution, in: J. Heering, K. Meinke, B. Möller, T. Nipkow (Eds.), Higher Order Algebra, Logic and Term Rewriting, Lecture Notes in Computer Science, vol. 816, Springer-Verlag, 1994, pp. 189–222.
[48] K. Meinke, L. Steggles, Correctness of dataflow and systolic algorithms using algebras of streams, Acta Informatica 38 (2001) 45–88.
[49] G. Milne, L. Pierre, in: Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science, vol. 683, Springer-Verlag, 1993.
[50] L. Moore, Systolic Arrays, Oxford University Press, 1988.
[51] N. Petkov, Systolic Parallel Processing, Elsevier, 1993.
[52] M. Poole, J.V. Tucker, A. Holden, Hierarchies of spatially extended systems and synchronous concurrent algorithms, in: B. Möller, J.V. Tucker (Eds.), Prospects for hardware foundations, Lecture Notes in Computer Science, vol. 1546, Springer-Verlag, 1998, pp. 184–235.
[53] M. Poole, J.V. Tucker, A. Holden, Hierarchical reconstructions of cardiac tissue, Chaos, Solitons & Fractals 13 (2002) 1581–1612.
[54] M. Pour-El, J. Richards, Computability in Analysis and Physics, Springer-Verlag, 1989.
[55] F.F. Soulié, Y. Robert, M. Tchuente (Eds.), Automata Networks in Computer Science, Manchester University Press, 1986.
[56] V. Stavridou, in: Formal Specifications for Digital Design, Cambridge Tracts in Theoretical Computer Science, vol. 37, Cambridge University Press, 1993.
[57] L. Steggles, Verifying an infinite systolic algorithm using third-order equational methods, Journal of Logic and Algebraic Programming 69 (2006) 75–92.
[58] V. Stoltenberg-Hansen, J.V. Tucker, Effective algebras, in: S. Abramsky, D. Gabbay, T. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. 4, Oxford University Press, 1995, pp. 357–526.
[59] B. Thompson, A mathematical theory of synchronous concurrent algorithms, Ph.D. Thesis, School of Computer Studies, University of Leeds, 1987.
[60] B. Thompson, J.V. Tucker, Algebraic specification of synchronous concurrent algorithms and architectures (Revised), Research Report 9-91, Department of Computer Science, Swansea University, Swansea, Wales, 1991.
[61] B. Thompson, J.V. Tucker, W. Yates, Algebraic specification of neural networks and correctness, Technical Report, Department of Computer Science, Swansea University, Swansea, Wales, 1993.
[62] J.V. Tucker, J.I. Zucker, in: Program Correctness over Abstract Data Types, with Error-State Semantics, CWI Monographs, vol. 6, North-Holland, 1988.
[63] J.V. Tucker, J.I. Zucker, Computable functions on stream algebras, in: H. Schwichtenberg (Ed.), Proof and Computation: NATO Advanced Study Institute International Summer School at Marktoberdorf, 1993, Springer-Verlag, 1994, pp. 341–382.
[64] J.V. Tucker, J.I. Zucker, Computation by 'while' programs on topological partial algebras, Theoretical Computer Science 219 (1999) 379–420.
[65] J.V. Tucker, J.I. Zucker, Computable functions and semicomputable sets on many-sorted algebras, in: S. Abramsky, D. Gabbay, T. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. 5, Oxford University Press, 2000, pp. 317–523.
[66] J.V. Tucker, J.I. Zucker, Abstract computability and algebraic specification, ACM Transactions on Computational Logic 3 (2002) 279–333.
[67] J.V. Tucker, J.I. Zucker, Infinitary initial algebra specifications for stream algebras, in: W. Sieg, R. Sommer, C. Talcott (Eds.), Reflections on the Foundations of Mathematics: Essays in honor of Solomon Feferman, Lecture Notes in Logic, Association for Symbolic Logic, vol. 15, 2002, pp. 234–256.

Please cite this article in press as: B.C. Thompson et al., Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms, Appl. Math. Comput. (2009), doi:10.1016/j.amc.2009.04.058

[68] J.V. Tucker, J.I. Zucker, Abstract versus concrete computation on metric partial algebras, ACM Transactions on Computational Logic 5 (2004) 611–668.
[69] J.V. Tucker, J.I. Zucker, Computable total functions, algebraic specifications and dynamical systems, Journal of Logic and Algebraic Programming 62 (2005) 71–108.
[70] J.V. Tucker, J.I. Zucker, Computability of analog networks, Theoretical Computer Science 371 (2007) 115–146.
Q1 [71] J.V. Tucker, J.I. Zucker, Computation on algebras of continuous functions, in preparation.
[72] J. Ullman, Computational Aspects of VLSI, Addison-Wesley, 1984.
[73] J. von Neumann, Theory of self-reproducing automata, in: A. Burks (Ed.), Papers of John von Neumann on Computing and Computing Theory, University of Illinois Press, 1966.
[74] K. Weihrauch, Computable Analysis: An Introduction, Springer-Verlag, 2000.
[75] A. Williams, Theoretical and empirical studies in vlsi complexity theory, Ph.D. Thesis, School of Computer Studies, University of Leeds, 1989.
[76] S. Wolfram (Ed.), Theory and Applications of Cellular Automata, World Scientific, 1986.
[77] S. Wolfram, A New Kind of Science, Wolfram Media, 2002.