ALGEBRAIC TOOLS FOR SYSTEM CONSTRUCTION

J.A. Bergstra, J.W. Klop
Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam

J.V. Tucker
Department of Computer Studies, University of Leeds, Leeds LS2 9JT

INTRODUCTION

Computer systems, in all their diversity, often share certain common properties: they are hierarchically organised into levels of abstraction and, at each level, they possess a certain architecture based on their construction from basic subsystems. Of especial interest are the hierarchical and modular structures of concurrent computer systems.

In this paper we consider a variety of computer systems and collect a set of informal principles concerning their hierarchical construction. These ideas are readily transformed into an elementary *formal* account of systems in which levels of abstraction are represented by *algebras* and the relationships between levels are represented by *homomorphisms*.

The algebraic approach to systems is then exemplified in an algebraic theory of concurrent systems based on a set of axioms called ACP - axioms for concurrent processes in part modelled on the calculi of R. Milner.

1. EXAMPLES OF COMPUTER SYSTEMS

1.1. Von Neumann Computer Systems. The term *computer system* usually refers to a configuration of hardware and software such as a computer whose operating system supports a number of languages and software tools. But the idea of the computer system is machine independent and possesses a hierarchical structure organized into levels of abstraction. In the case of the *von Neumann computer system*, this hierarchy is shown in the following figure (adapted from Bell and Newell [10]):

system configuration level
symbolic programming level
register-transfer level
logic level
circuit level
device level

Each level is characterized by a *medium* for processing, *basic components* for this processing, *composition methods* to build systems from the basic components and *rules*

of behaviour to explain the operation of the systems constructed in terms of their components (illustrations may be found in Bell and Newell [10]).

The structure of a system at various levels is consistent and high levels reflect the structure of low levels: it is what we may term a bottom-up system built by a process of abstraction. The von Neumann hierarchy may be termed a bottom-up hierarchy in which data stores and data transfer paths are represented at each level of abstraction: see Backus [6].

1.2. Machines and Languages. A machine defines a programming language in which an instruction represents an operation of the machine. Conversely, the language defines a machine by supposing its instructions to specify certain machine operations to be available. This observation initiates the idea of the *virtual machine* that refines the hierarchy of the von Neumann computer system, matching ideas about machine architecture with ideas about programming languages (see Tanenbaum [25], for instance). Actually, virtual machines serve the following purpose: *a level of abstraction is defined by a programming formalism for which an operational semantics is made based on a model of computation*; that these models of computation are called machines reflects that the von Neumann systems compose a bottom-up hierarchy.

1.3. Other Architectures. The processor-channel-store model underlying von Neumann computer systems may be replaced by other models of computation that give rise to *data flow systems*, *reduction systems*, *systolic systems* and *vector systems*. In each case computers, with architectures derived from the models, are under construction and bottom-up hierarchies for the systems can be expected to evolve.

1.4. Distributed Computer Systems. By a *distributed computer system* we have in mind a system based on a network of computers. The hierarchical structure of such a distributed system is unknown and remains a substantial research problem. A notable attempt at a solution is made in the *ISO-OSI Model* which organises into seven levels of abstraction the interconnections of a distributed computer system (see Tanenbaum [26]); the proper specification of these and other independent layers remains a problem, however, and composition principles are unknown.

1.5. VLSI Systems. A VLSI system is a system specially implemented in silicon using VLSI technology. The need for custom VLSI leads to the problem of programming into silicon wherein system descriptions are compiled into circuits. Thus, the following scientific problem is encountered:

VLSI System Hierarchy Problem. To analyse and structure VLSI computation as a hierarchy of levels of computation; and to develop formal many-level specification languages which have regard for verifying system logic and predicting system performance.

The problem asks for a generalisation of the von Neumann and other machine-language hierarchies; and its answers may be as complex in their organisation. The VLSI Hierarchy Problem is of interest to us: in Dew and Tucker [15] the complexity

theory of the composition principles is investigated experimentally.

1.6. Software Systems. Within the symbolic level of the von Neumann hierarchy of 1.1 reside *software systems* of considerable complexity. Ideas that determine a hierarchical structure for software systems are emerging from researches into programming languages that support the hierarchical and modular decomposition of programs: the subjects of stepwise-refinement and top-down design of programs, data type specification, data type modules, generic data types immediately come to mind through their popularisation in ADA. (A very useful survey is Wulf [27] and many early research articles are collected in Gries [17].)

Of particular importance is the concept of the *algebraic data abstraction* with which programs may assume a hierarchical structure the levels of which are defined in terms of *the operations allowed on data*. There is a mathematical theory for algebraic data types that programmers may *refer* to and that contains many satisfying results: see ADJ[2,4], Goguen and Meseguer [16].

The algorithmic theory at a fixed level of data abstraction is well understood in conventional terms. But the ideas of a *data type module as a system component* and appropriate *module composition tools* are underdeveloped (Back [5]). And the relation between levels is not well understood: for example, the implementation of a data type specification by a data type module is plagued by problems to do with partial functions.

However in the case of specification languages the study of hierarchies and composition tools is very well advanced as a result of researches of Burstall and Goguen related to CLEAR: see Burstall and Goguen [14].

Since an algebraic structure in the syntax and denotational semantics of programming languages was observed (in ADJ [1]), further research has shown that conventional programming language definitions can be recast in an algebraic (and therefore compositional) mould. The hierarchical implications of this are seen in compilation: see ADJ [3].

1.7. Concurrent Systems. The parallel execution of tasks has led to special developments in software for concurrent systems; for example OCCAM [23] which is based on CSP, first defined in Hoare [19]. In later studies of CSP the modular structure of programs is established by means of composition operators including sequencing, nondeterministic composition and merge; see Hoare, Brookes and Roscoe [20].

Composition operators for concurrent processes have been the subject of long standing research by R. Milner, an introduction to which is Milner [21]. An important idea is that of a *calculus*, called CCS, for the composition operators which describes their effects by means of *laws*.

Hierarchical aspects of system construction are treated only through the simple algebraic idea of encapsulation where an interconnected set of processes is regarded as a single process.

1.8. Functional Programming. In his critique of von Neumann computer systems, J. Backus

identified the need for composition tools for program construction and the requirement that such tools constitute a rich set of program forming operators about which a satisfactory *algebraic* theory can be made. Backus' theory of *functional programming* is intended to satisfy this requirement. In particular, it uses operator laws to support algebraic proofs of program equivalence and correctness: see Backus [6,7].

1.9. Knowledge-based Systems. Finally, we notice A. Newell's thesis concerning a *knowledge level* above the symbolic level of the von Neumann hierarchy in 1.1. The new level of abstraction achieves a separation of knowledge and knowledge representation: the latter belonging to the symbolic level. The knowledge level and its structure is described in Newell [22].

2. PRINCIPLES FOR SYSTEM ORGANISATION

From the examples of hierarchically organised systems the following ideas, by no means exemplified in all the systems, can be collected:

2.1. Levels. A *system* belongs to a well-defined category of systems having an autonomous specification; this category we refer to as a *level of abstraction.*

2.2. Composition. Each level of abstraction is characterised by a collection of *basic systems* and a collection of *composition tools* for system construction. The systems of the level are all manufactured by applying the composition tools to basic systems.

2.3. Architecture. The *structure* or *architecture* of each system is defined by the way the system is configured from the basic system components by the composition tools.

2.4. Hierarchy. Two levels of abstraction L_1 and L_2 may be hierarchically ordered: in symbols, $L_1 < L_2$ meaning that L_1 is *below* L_2 and that L_2 is *above* L_1. The relationship between the systems of levels L_1 and L_2 is expressed as follows:

(i) The view from below: the systems of L_2 are *abstractions* or *modularisations* of the systems of L_1.

(ii) The view from above: the systems of L_1 are *specialisations*, *refinements*, or *implementations* of the systems of L_2

We imagine mappings

ab: $L_1 \rightarrow L_2$ for abstraction

sp: $L_2 \rightarrow L_1$ for specialisation.

Abstraction and specialisation mechanisms are inverse to one another if ab∘sp = sp∘ab = identity.

2.5. Hierarchy and System Architecture. A method of abstraction or specialisation must respect system structure or architecture at both levels of abstraction.

2.6. Bottom-up and Top-down Hierarchies. A *bottom-up hierarchy* is a collection of levels of abstraction $L_1,\ldots,L_n$ and structure preserving abstraction operations $ab_1,\ldots,ab_{n-1}$;

in symbols:

$$L_1 \xrightarrow{ab_1} L_2 \longrightarrow \ldots \longrightarrow L_{n-1} \xrightarrow{ab_{n-1}} L_n .$$

Thus in a bottom-up hierarchy system architectures at high levels reflect the structure of systems at low levels.

The concept of a *top-down hierarchy* is characterised in a similar way using specialisations.

3. ALGEBRAIC MODEL OF SYSTEM ORGANISATION

The ideas of the last section may be formalised as follows:

3.1. Levels of Abstraction. A level of abstraction is represented by an *algebra* A whose elements are *systems*. The composition tools for system construction at the level A are the *operations* of the algebra A. The basic components for system construction at the level A are collected in a set G of *generators* for the algebra A.

3.2. Architectures. A notation for system architectures for a level of abstraction A is made as follows: the components or generators G of A are named by a set of symbols X and the composition operators are named from the signature Σ of A. The algebra $T(\Sigma,X)$ of all Σ-terms over X is an algebra of *system notations* for A that represents the system configurations possible by means of applying the composition tools to the basic components. The unique semantic homomorphism $v: T(\Sigma,X) \to A$ formalises the concept of *system architecture* in these two definitions: An *architecture for a system* $a \in A$ is a term $t \in T(\Sigma,X)$ such that $v(t) = a$. Two architectures t_1 and t_2 are *equivalent as A-systems* if $v(t_1) = v(t_2)$.

3.3. Realisations. Let A_1 and A_2 be two levels of abstraction. Then the systems of A_1 are *realisable as systems of* A_2 if there is a homomorphism $\phi: A_1 \to A_2$; the map ϕ may be called a *realisation*. Abstractions (modularisations) and specialisations (refinements, implementations) are instances of system realisations. *We do not allow an algebraic status to these ideas of abstraction and specialisation.*

3.4. Hierarchy. A *hierarchy* is a sequence of levels of abstractions and realisations

$$A_1 \xrightarrow{\phi_1} A_2 \xrightarrow{\phi_2} \ldots \longrightarrow A_{n-1} \xrightarrow{\phi_{n-1}} A_n .$$

Again we notice that it is *not* an algebraic matter that A_1 is the top of a top-down hierarchy of specialisations or, alternatively, the bottom of a bottom-up hierarchy of abstractions.

3.5. System Laws. The development of a theory for a class of systems using these ideas requires a set of properties of the system composition tools to serve as algebraic axioms: an algebraic theory *begins* with the choice of its basic laws. These laws are established by investigating the semantics of the systems *and* by evaluating the mathe-

matics the laws support. Ideally, the axioms should be formally elegant and small in number, to be easy to memorise and to aid calculation. And the set of axioms allows the construction of free objects for the category of all its models to service the concept of system architectures.

3.6. Concurrent Systems. In the sequel a set of axioms for concurrent processes (ACP) is presented. Many general laws for concurrent processes have been found in the course of studies by R. Milner and his collaborators and a number of calculi have been formed: Milner [21], Hennessy [18]. A search for laws relevant to CSP and OCCAM has also started as part of the study of semantics of that attractive syntax for concurrency: Hoare, Brookes and Roscoe [20], Olderog and Hoare [24].

In contrast, the laws of ACP are made to support an exclusively algebraic study of concurrency. The axioms are used as a kernel of properties of composition operators with which further laws may be employed on occasion to prove a result (see 6.1). It is hoped that the theory of ACP will be an interesting instrument to analyse concurrency and will be of use in the analysis of other calculi.

Semantically, the models of ACP represent distinct levels of abstraction of concurrent systems implemented in agreement with the specifications represented by the axioms of ACP. The homomorphisms of ACP algebras represent the abstractions and specialisations between such levels of abstraction for ACP systems.

4. ALGEBRA OF COMMUNICATING PROCESSES

We will introduce an algebraic system for the analysis of communicating systems made from given systems by means of the following composition tools: for communicating systems x and y we allow the operations of

sequential composition	$x \cdot y$
alternative composition	$x + y$
parallel composition	$x \parallel y$
encapsulation	$\partial_H(x)$

Our analysis will also involve a finite collection of atomic systems and their pattern of communication.

4.1. ACP-algebras. Let A be a finite set, called the set of *atomic actions*. An ACP-*algebra* over A consists of a set P equipped with operators

$$\cdot,\ +,\ \parallel,\ \between,\ |,\ \delta,\ \partial_H.$$

All operators are binary, except the constant δ, a distinguished atomic action and the unary H-projection $(H \subseteq A)$ ∂_H. The set P contains A as a subset on which communication '|' restricts as a map $|: A \times A \to A$.

These operations satisfy the following equational axioms, where a,b,c vary over A and x,y,z over P. Often we will write instead of $x \cdot y$ just xy.

$x + y = y + x$	A1
$x + (y + z) = (x + y) + z$	A2
$x + x = x$	A3
$(x + y)\cdot z = x\cdot z + y\cdot z$	A4
$(x\cdot y)\cdot z = x\cdot(y\cdot z)$	A5
$x + \delta = x$	A6
$\delta\cdot x = \delta$	A7
$a\|b = b\|a$	C1
$(a\|b)\|c = a\|(b\|c)$	C2
$\delta\|a = \delta$	C3
$x\|\|y = x\between y + y\between x + x\|y$	CM1
$a\between x = a\cdot x$	CM2
$(ax)\between y = a(x\|\|y)$	CM3
$(x + y)\between z = x\between z + y\between z$	CM4
$(ax)\|b = (a\|b)\cdot x$	CM5
$a\|(bx) = (a\|b)\cdot x$	CM6
$(ax)\|(by) = (a\|b)\cdot(x\|\|y)$	CM7
$(x + y)\|z = x\|z + y\|z$	CM8
$x\|(y + z) = x\|y + x\|z$	CM9
$\partial_H(a) = a$ if $a \notin H$	D1
$\partial_H(a) = \delta$ if $a \in H$	D2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\partial_H(x\cdot y) = \partial_H(x)\cdot\partial_H(y)$	D4

Axioms of ACP

4.2. Commentary. On intuitive grounds $x\cdot(y + z)$ and $x\cdot y + x\cdot z$ present different mechanisms and an axiom $x\cdot(y + z) = x\cdot y + x\cdot z$ is not included in ACP.

The constant δ is to be interpreted as an action which cannot be performed, hence $\delta x = \delta$; the law $x + \delta = x$ postulates that in the context of an alternative it will never be chosen.

The source of intuition for the $\|$-operation axioms is the arbitrary interleaving semantics of parallelism. The operations $\lfloor\!\lfloor$, left-merge, and $|$, communication merge, are auxiliary operations helpful in obtaining a finitary specification of $\|$. The essential algebraic properties of $\lfloor\!\lfloor$ and $|$ are the *linearity laws* CM4, CM8, CM9. Intuitively, $x\lfloor\!\lfloor y$ is $x\|y$ but takes its initial step from x; and $x|y$ is $x\|y$ but takes its initial step as a communication of an initial action of x and an initial action of y.

4.3. Generators. Let P be a process algebra over A. A *subalgebra* Q *of* P is a subset Q of P containing A and closed under all the operations.

Let $X = \{x_i \mid i \in I\}$ be a subset of a process algebra P over A. The smallest subalgebra of P containing X is denoted by $\langle X\rangle$. The algebra P is said to be *generated* by a subset X if $P = \langle X\rangle$.

4.4. Homomorphisms. Let P and Q be process algebras over A. A *homomorphism* $\phi: P \to Q$ is a map which respects all operations and which leaves atoms invariant. The *image* of a homomorphism $\phi: P \to Q$ is an A-subalgebra of Q, denoted by $\phi(P)$.

4.5. Syntax and Semantics. Let P be an ACP-algebra with atom set A. We define the *communication function* $\gamma_P: A \times A \to A$ for P by:

$$\gamma_P(a,b) = a|b.$$

Given a function $\gamma: A \times A \to A$, one defines ACP_γ as the class of all ACP-algebras P with $\gamma_P = \gamma$. Clearly for ACP_γ to be nonempty, γ must satisfy the requirements C1-3.

Let us now fix a communication function γ that satisfies C1-3. Then ACP_γ contains an initial algebra denoted by A_ω. Let $X_1,\ldots,X_k$ be a set of formal symbols. Now $A_\omega[X_1,\ldots,X_k]$ is the *free* ACP_γ*-algebra over* k *generators*.

According to the thesis of ADJ [1,2] one conceives $A_\omega[X_1,\ldots,X_k]$ as an algebra of *system notations*, finding their semantics as homomorphic images. Indeed, if $p_1,\ldots,p_k$ are processes in P, an ACP_γ-algebra, then there is a unique homomorphism

$$\phi: A_\omega[X_1,\ldots,X_k] \to P$$

mapping X_i to p_i. If we see the p_i as realisations of the X_i, then ϕ extends these realisations to all system notations in $A_\omega[X_1,\ldots,X_k]$.

On A_ω one defines *projection operators* $(.)_n: A_\omega \to A_\omega$ as follows:

$$\begin{aligned}(a)_n &= a\\ (ax)_1 &= a\\ (ax)_{n+1} &= a(x)_n\\ (x + y)_n &= (x)_n + (y)_n.\end{aligned}$$

For each $n \geqslant 1$ a congruence relation $\equiv_n$ on A_ω is obtained by

$$x \equiv_n y \Leftrightarrow (x)_n = (y)_n.$$

The algebras $A_\omega/\equiv_n$ are again ACP_γ-algebras. We write A_n for $A_\omega/\equiv_n$. Clearly, $(.)_n$ induces a homomorphism

$$(.)_n : A_{n+1} \to A_n.$$

The chain $A_1 \xleftarrow{(.)_1} A_2 \xleftarrow{(.)_2} A_3 \longleftarrow \ldots$ determines a *projective limit* which we denote by A^∞.

5. SOLVING RECURSION EQUATIONS

Consider the algebra $A_\omega[X_1,\ldots,X_k]$ and let E be a system of equations:

$$\begin{cases} X_1 = t_1(X_1,\ldots,X_k) \\ \vdots \\ X_k = t_k(X_1,\ldots,X_k) \end{cases}$$

where the terms t_i are built from the constants and operations of ACP and the variables from $X = (X_1,\ldots,X_k)$. This system generates a congruence $\equiv_E$ on $A_\omega[X_1,\ldots,X_k]$. We will consider the quotient algebra $A_\omega[X_1,\ldots,X_k]/\equiv_E$, also denoted as

$$A_\omega[X]/X=t(X), \text{ or } A_\omega(X,E)$$

where $X = t(X)$ is short for the system of equations E. This algebra is also in ACP_γ. In $A_\omega(X,E)$ the X_i are solutions of E. In this way we have a purely algebraic method for solving fixed point equations.

It can be shown that for each system of equations E, there exists a homomorphism

$$\phi : A_\omega(X,E) \to A^\infty.$$

In the case of a single equation this was shown in Bergstra and Klop [11]. The homomorphism need not be unique.

From this observation one concludes that A^∞ solves all systems of fixed point equations, and that $A_\omega(X,E)$ is guaranteed to have a nontrivial structure. The algebra $A_\omega(X,E)$ is a countable semicomputable structure, whereas A^∞ is uncountable.

Problem: Under which circumstances is $A_\omega(X,E)$ computable?

6. SOME MATHEMATICAL PROPERTIES OF ACP-ALGEBRAS

Consider A_ω, whose domain consists of terms built from A, +, ·, $\|$, $\lfloor\!\lfloor$, $|$, ∂_H. In fact it can be shown [12] that each term t is equivalent to a term t' built using A, + and · only:

NORMAL FORM THEOREM. *For each closed term* t *there is a closed term* t' *not containing* $\|$, $\lfloor\!\lfloor$, $|$, ∂_H *such that* ACP $\vdash$ t = t'.

6.1. Algebras with Standard Concurrency and Handshaking. A useful intuition about communicating processes is to postulate that $\|$ is commutative and associative. This leads to the following requirements for $\|$, $\between$ and $|$, called *axioms of standard concurrency:*

$$(x \between y) \between z = x \between (y \| z)$$

$$(x | y) \between z = x | (y \between z)$$

$$x | y = y | x$$

$$x \| y = y \| x$$

$$x | (y | z) = (x | y) | z$$

$$x \| (y \| z) = (x \| y) \| z$$

These axioms are not independent relative to ACP, for instance commutativity and associativity of $\|$ are derivable from the other axioms.

In [12] it is shown that A_ω and A^∞ satisfy the axioms of standard concurrency.

Moreover, matters are greatly simplified by adopting the *handshaking axiom*:

$$x | y | z = \delta.$$

Both CSP and CCS adopt this axiom. The handshaking axiom implies that all proper communications are binary.

Let P be an ACP-algebra with standard concurrency and handshaking. Let $x_1,\ldots,x_k$ be processes in P. We make the following abbreviations: X_k^i is obtained by merging $x_1,\ldots,x_k$ except x_i, and $X_k^{i,j}$ is obtained by merging $x_1,\ldots,x_k$ except x_i,x_j. Here we suppose $k \geqslant 3$. Then one easily proves the following generalisation of the ACP-axiom CM1:

EXPANSION THEOREM. $x_1 \| \ldots \| x_k = \sum_i x_i \between X_k^i + \sum_{i \neq j} (x_i | x_j) \between X_k^{i,j}$.

6.2. Literature. We will catalogue the principal influences on ACP. In addition to work on calculi for concurrency, from Milner's CCS we have adopted the laws A1-5 and the idea of the expansion theorem; Milner's restriction operator is here called the encapsulation operator. (In [13] Milner's τ-laws have been incorporated in the algebraic framework.) From Hennessy [18] we have adopted laws C1 and C2.

The left-merge $\between$ and projective limit A^∞ first appeared in [11]. The full system ACP, including $|$, was introduced in [12]. Our work on ACP arose from a question in De Bakker and Zucker [8] about the existence of solutions for non-guarded fixed point equations in their topological model of processes (A^∞ is equivalent to their space of uniform processes).

REFERENCES

[1] ADJ (GOGUEN, J.A., J.W. THATCHER, E.G. WAGNER & J.B. WRIGHT), *Initial algebra semantics and continuous algebras*,

[2] ADJ (GOGUEN, J.A., J.W. THATCHER & E.G. WAGNER), *An initial algebra approach to the specification, correctness and implementation of abstract data types* , in R.T. Yeh (ed.): Current trends in programming methodology IV, Data structuring, Prentice Hall, Englewood Cliffs (1978), 80-149.

[3] ADJ (THATCHER, J.W., E.G. WAGNER & J.B. WRIGHT), *More advice on structuring compilers and proving them correct*, in: H.A. Maurer (ed.), Automata, languages and programming, 6th Colloquium, Springer LNCS 71 (1979), 596-615.

[4] ADJ (EHRIG, H.,H.-J. KREOWSKI, J.W. THATCHER, E.G. WAGNER & J.B. WRIGHT), *Parameterized data types in algebraic specification languages*, in: J.W. de Bakker & J. van Leeuwen (eds.), Automata, languages and programming, 7th Colloquium, Springer LNCS (1980), 157-168.

[5] BACK, R.J., *Locality in modular systems*, in: M. Nielsen & E.M. Schmidt (eds.), Automata, languages and Programming, 9th Colloquium, Springer LNCS 140 (1982),1-13.

[6] BACKUS, J., *Can programming be liberated from the von Neumann style? - a functional style and its algebra of programs*, CACM 21 (1978), 613-639.

[7] BACKUS, J., *Is computer science based on the wrong fundamental concept of a 'program'? An extended concept*, in: J.W. de Bakker & J.C. van Vliet (eds.), Algorithmic languages, North-Holland 1981, 133-165.

[8] DE BAKKER, J.W. & J.I. ZUCKER, *Denotational semantics of concurrency*, Proc. 14th ACM Symp. on Theory of Computing (1982), 153-158.

[9] DE BAKKER, J.W. & J.I. ZUCKER, *Processes and the denotational semantics of concurrency*, Report IW 209/82, Mathematisch Centrum, 1982, to appear in Information & Control.

[10] BELL, C.G. & A. NEWELL, *Computer structures: Readings and Examples*, McGraw-Hill 1971.

[11] BERGSTRA, J.A. & J.W. KLOP, *Fixed point semantics in process algebras*, Report IW 206/82, Mathematisch Centrum, 1982.

[12] BERGSTRA, J.A. & J.W. KLOP, *Process algebra for communication and mutual exclusion*, Report IW 218/83, Mathematisch Centrum, 1983.

[13] BERGSTRA, J.A. & J.W. KLOP, *An abstraction mechanism for process algebras*, Report IW 231/83, Mathematisch Centrum, 1983.

[14] BURSTALL, R.M. & J.A. GOGUEN, *The semantics of CLEAR, a specification language*, in: Proc. on Abstract Software specifications, Copenhagen, Springer LNCS 86 (1980), 292-332.

[15] DEW, P.M. & J.V. TUCKER, *An experimental study of a timing assumption in VLSI complexity theory*, Univ. of Leeds, Dept. of Computer Studies, Report 168.

[16] GOGUEN, J.A. & J. MESEGUER, *An initiality primer*, in preparation.

[17] GRIES, D. (ed.), *Programming methodology*, Springer, Berlin 1978.

[18] HENNESSY, M., *A term model for synchronous processes*, Information & Control, Vol. 51, nr.1 (1981), 58-75.

[19] HOARE, C.A.R., *Communicating sequential processes*, C. ACM 21 (1978), 666-677.

[20] HOARE, C.A.R., S.D. BROOKES & A.W. ROSCOE, *A theory of communicating sequential processes*, to appear in JACM.

[21] MILNER, R., *A Calculus for Communicating Systems*, Springer LNCS 92, 1980.

[22] NEWELL, A., *The knowledge level*, Artificial Intelligence 18 (1982), 78-127.

[23] OCCAM, *The OCCAM programming manual*, INMOS, Bristol 1982.

[24] OLDEROG, E.R. & C.A.R. HOARE, *Specification-oriented semantics for communicating processes*, in: J. Díaz (ed.), Automata. languages and programming, 10th Colloquium, Springer LNCS 154 (1983), 561-572.

[25] TANENBAUM, A.S., *Structured computer organisation*, Prentice Hall, Englewood Cliffs, 1976.

[26] TANENBAUM, A.S., *Computer networks*, Prentice Hall, Englewood Cliffs, 1981.

[27] WULF, W.A., *Abstract data types: a retrospective and prospective view*, in: P. Dembinski (ed.), Mathematical Foundations of Computer Science 1980, Springer LNCS (1980), 94-112.