1992

S.M. Eker

Verification of parameterised synchronous concurrent algorithms with OBJ3: The pixel planes architecture revisited

Computer Science/Department of Software Technology   Report CS-R9225 July

Verification of Parameterised Synchronous Concurrent Algorithms with OBJ3: The Pixel Planes Architecture Revisited

S. M. Eker[0]

CWI, P.O.Box 4079, 1009 AB Amsterdam.

Abstract

We consider the verification of parmeterised synchronous concurrent algorithms using OBJ3. Our case study is the linear expression evaluator from the Pixel Planes graphics architecture which has previously been manually verified and has also been verified using OBJ3 in the fixed size non-parameterised case.

Key Words & Phrases: Synchronous Concurrent Algorithms, Algebraic Specification, Hardware Verification, Term Rewriting, OBJ3.
1991 Mathematics Subject Classification: 68Q42 **[Theory of Computing]:** Rewriting Systems; 68Q60 **[Theory of Computing]:** Specification and Verification of Programs; 68Q65 **[Theory of Computing]:** Abstract Data Types, Algebraic Specification.
1991 CR Categories: B.2.2 **[Arithmetic and Logic Structures]:** Performance Analysis and Design Aids;
Note: This work was done while the author was an ERCIM Research Fellow.

1 Introduction

A synchronous concurrent algorithm (SCA) consists of a network of modules and channels which are synchronised by a global clock and compute and communicate data in parallel. Many algorithms and architectures can be formalised as SCAs including clocked hardware. The mathematical theory of SCAs is based on the computable functions over many sorted algebras and is studied in [18, 15, 11, 14, 19, 13, 10, 16, 1, 12].

The Pixel Planes architecture [6, 7] is a high performance graphics system that contains a novel tree structured computation unit called a *Linear Expression Evaluator* (LEE). This LEE was formalised as an SCA and manually verified in [3]. In [2] the equations defining a LEE tree of fixed height were translated in to OBJ3 and a correctness proof obtained by term rewriting. In this paper we extend this work to the case where the height of the LEE tree is a parameter n and we verify it for all n by using induction over tree paths.

2 Synchronous concurrent algorithms

We consider a synchronous concurrent algorithm (SCA) as a network of *sources*, *modules* and *channels* computing and communicating data, as depicted in Figure 1. For simplicity we will assume that all the data is drawn from a single set A. The network has a global discrete clock $\mathbf{T} = \{0, 1, \ldots\}$ to synchronise computations and the flow of data between modules. There are c *sources* labelled $s_1, \ldots, s_c$, and k *modules* labelled $m_1, \ldots, m_k$. The sources perform no computation; they are simply input ports where fresh data arrives at each clock cycle. Each module m_i has $p(i)$ inputs and a single output as depicted in Figure 2. The action of a module m_i is specified by a function

[0]From 15th July 1992 the author's address will be INRIA-LORRAINE, Campus Scientifique, 615 rue du Jardin Botanique - BP 101, F-54602 VILLERS-LES-NANCY CEDEX, France.

Report CS-R9225
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

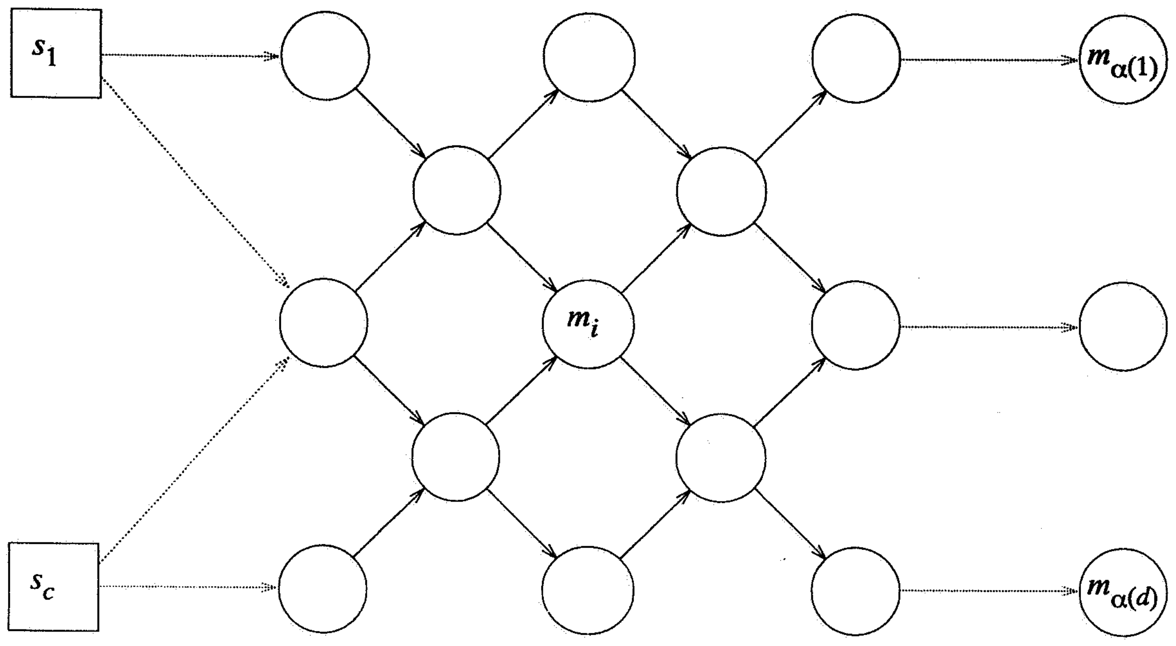

Figure 1: A synchronous concurrent algorithm as a network.

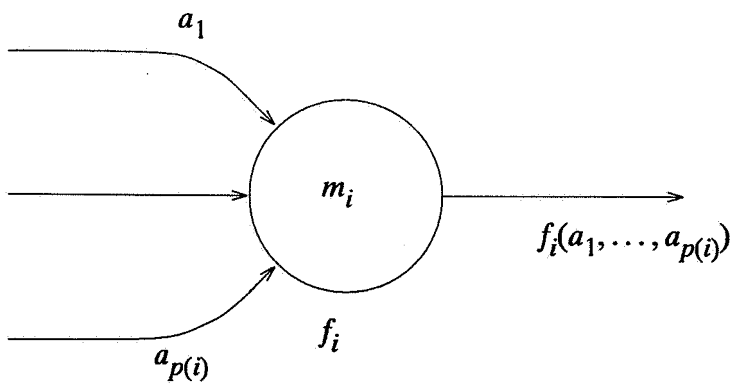

Figure 2: Module m_i.

$$f_i : A^{p(i)} \to A.$$

If the diagram in Figure 1 is to be a *graph* then edges may not branch. Thus where the output of a module is required as an input to several other modules, extra output channels that copy the value must be added to the module (a number of conventions handling this point are discused in §2.2). Results are read out from a subset of the modules $m_{\alpha(1)}, \ldots, m_{\alpha(d)}$ which may be termed output modules or *sinks*.

The interconnections between the sources and modules are represented by a pair of partial functions

$$\gamma : \mathbf{N}_k \times \mathbf{N} \to \{S, M\}$$

and

$$\beta : \mathbf{N}_k \times \mathbf{N} \to \mathbf{N}$$

where $\mathbf{N}_k = \{1, 2, \ldots, k\}$. Intuitively $\gamma(i,j) = S$ indicates that the jth input to module m_i comes from a source whereas $\gamma(i,j) = M$ indicates it comes from another module. The index of the source or module in question is given by $\beta(i,j)$.

To ensure that our networks are well defined we impose the following conditions on γ and β:

$$\forall i,j.[(i \in \{1,\ldots,k\}) \wedge (j \in \{1,\ldots,p(i)\}) \Rightarrow \beta(i,j)\downarrow] \quad (1)$$

$$\forall i,j.[(i \in \{1,\ldots,k\}) \wedge (j \in \{1,\ldots,p(i)\}) \Rightarrow \gamma(i,j)\downarrow] \quad (2)$$

$$\forall i,j.[(\gamma(i,j) = S) \Rightarrow (\beta(i,j) \in \{1,\ldots,c\})] \quad (3)$$

$$\forall i,j.[(\gamma(i,j) = M) \Rightarrow (\beta(i,j) \in \{1,\ldots,k\})] \quad (4)$$

Here the notation $\beta(i,j)\downarrow$ means that $\beta(i,j)$ is defined on the specified arguments i,j given in the expression. The first two conditions ensure that β and γ are defined for every input of every module. The latter two conditions ensure that the sources or modules named by β and γ actually exist.

We will assume that each module is initialised with a well defined output assigned to its output channel. Thus the initial state of the network is an element $b \in A^k$ and we will use the notation b_i to denote the initial value of the output of module m_i for $i = 1, \ldots, k$.

In terms of our intuitive picture, new data are available at each source, and new results at each sink, at every tick of the global clock $\mathbf{T}$. Thus the algorithm processes infinite sequences or streams of data. A stream $\underline{a}(0), \underline{a}(1), \ldots$ of data from A is represented by a map $\underline{a} : \mathbf{T} \to A$ and the set of streams of data is represented by the set $[\mathbf{T} \to A]$ of all such maps. Thus we will specify the input/output behaviour of an architecture, initialised by b, by a mapping V_b from source streams into sink streams:

$$V_b : [\mathbf{T} \to A]^c \to [\mathbf{T} \to A]^d$$

We call V_b a *stream transformer* and the *i/o specification* of the algorithm.

2.1 Representing stream algorithms

We represent each module m_i by a *value function* $v_i : \mathbf{T} \times [\mathbf{T} \to A^c] \times A^k \to A$. Intuitively $v_i(t, \underline{a}, b)$ represents the output of module m_i at time t, when the network is initialised with values b and is computing on the input stream $\underline{a}$. The function v_i is specified as follows

$$v_i(0, \underline{a}, b) = b_i$$

$$v_i(t+1, \underline{a}, b) = f_i(A_1, \ldots, A_{p(i)})$$

where for $j = 1, \ldots, p(i)$:

$$A_j = \begin{cases} \underline{a}_{\beta(i,j)}(t) & \text{if } \gamma(i,j) = S \\ v_{\beta(i,j)}(t, \underline{a}, b) & \text{if } \gamma(i,j) = M \end{cases}$$

We can now write down the i/o specification of the algorithm V_b when initialised with values b and executed on input stream $\underline{a}$ as

$$V_b(\underline{a})(t) = (v_{\alpha(1)}(t, \underline{a}, b), \ldots, v_{\alpha(d)}(t, \underline{a}, b)).$$

where $\alpha(1), \ldots, \alpha(d)$ are the labels of the sinks. We often work with V_b in the following form:

$$V_b : [\mathbf{T} \to A]^c \times \mathbf{T} \to A^d$$

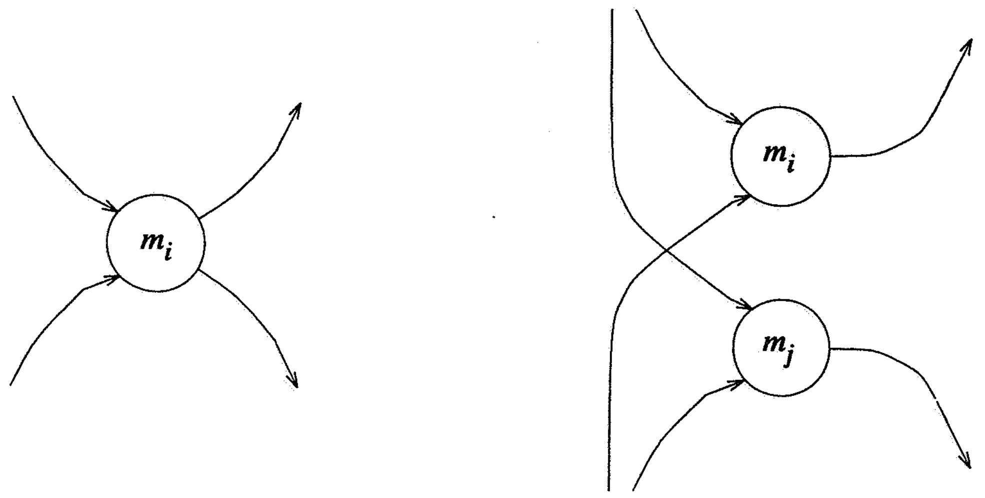

Figure 3: Splitting modules with more than one output.

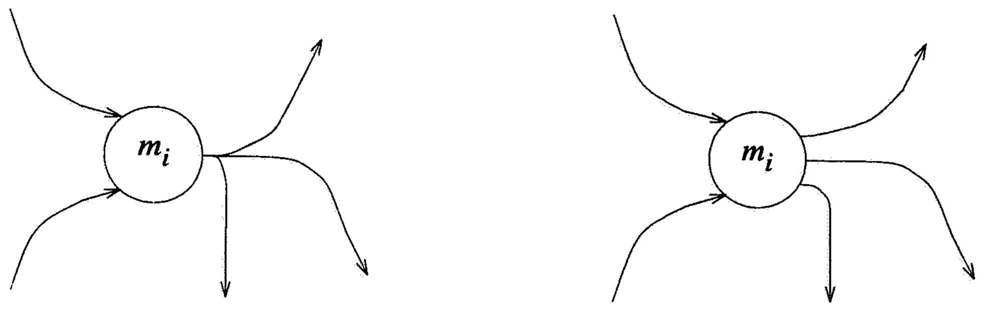

Figure 4: Replacing branching channels.

2.2 Conventions on the network model

Looking at the synchronous network depicted in Figure 1 a number of conventions are involved in this informal notation that are motivated by the formalisation by means of the value functions above. In particular channels do not branch and modules only have a single distinct output value. Where the output of a module is required in several places we need separate channels which carry duplicate outputs of the module in question. In the Pixel Planes example later on, we will find it convenient to draw networks with more than one distinct output value and branching channels (such networks are no longer graphs). When we formalise these networks as shown above we deal with this pictorial convention in the following way. Modules with more than one distinct output are split into separate single output modules, one for each output and each module takes all the inputs of the original; see the translation in Figure 3. Branching channels are replaced by a set of non-branching channels, one for each of the branches of the original; see the translation in Figure 4.

2.3 User specifications

In order to be able to prove a given algorithm correct we need a formal specification of the task which the algorithm is supposed to perform i.e. a *user specification* or *behavioural specification* which defines the task in terms of outputs required at particular times. Formally a user specification is a stream transformer of the form

$$S : [\mathbf{T} \to A]^c \to [\mathbf{T} \to A \cup \{u\}]^d$$

where u is a special undefined element. An i/o specification $I : [\mathbf{T} \to A]^c \to [\mathbf{T} \to A]^d$ of an algorithm is said to *directly satisfy* or *directly meet* a user specification $S : [\mathbf{T} \to A]^c \to [\mathbf{T} \to A \cup \{u\}]^d$, if and only if, for all input streams $\underline{x} \in [\mathbf{T} \to A]^c$, for each $i \in \{1, \ldots, d\}$ and $t \in \mathbf{T}$, either

$$S_i(\underline{x})(t) = u \text{ or } I_i(\underline{x})(t) = S_i(\underline{x})(t)$$

where S has coordinate functions S_i and I has coordinate functions I_i. There are occasions when the definition of correctness of an algorithm with respect to a specification is more complicated. For example, correctness can involve the scheduling or translation of input-output streams: see [13, 15].

2.4 Component, program and task algebras

We note that in formulating a SCA, the functions f_i that specify the actions of the modules together with the constant 0 and operations $t+1$ and *eval* form a stream algebra with carriers $\mathbf{T}$, A and $\mathbf{T} \to A$. We call this the *component algebra.* The value functions v_i that defined the SCA similarly give rise to a stream algebra which we call the *program algebra.* Finally if we just consider the value functions $v_{\alpha(i)}$ of the output modules we get a third stream algebra which we call the *task algebra.*

The sort A and its operations in the component algebra may be defined abstractly by a set of equations in which case we have a class of component algebras which are *standard algebras* in that time $\mathbf{T}$ has its standard interpretation. In this case the algorithm determines classes of program and task algebras which are also standard algebras.

3 Informal description of Pixel Planes' LEE

In this section we discuss an example of a synchronous concurrent algorithm which we will verify later using OBJ3. This algorithm is the *Linear Expression Evaluator* (LEE) section of the Pixel Planes architecture [6, 7]. Further extensions are described in [5] and implementation details are given in [17]. A review of current experimental and commercial VLSI graphics systems, including Pixel Planes, is given by Fuchs [4]. We shall examine the algorithm in its 'pure' form without the modifications (such as super trees) required to implement it in current VLSI technology.

The Pixel Planes LEE has the task of taking an input stream

$$(a(0), b(0), c(0)), (a(1), b(1), c(1)), \ldots$$

of triples of numbers, and generating in parallel for each point (x, y) on a discrete $n \times m$ grid, the stream

$$a(0)x + b(0)y + c(0), a(1)x + b(1)y + c(1), \ldots$$

of values of the linear expression $ax+by+c$. This is shown diagramatically in Figure 5. At this top level of description it suffices to assume that input and output data are numbers (such as integers or reals); in more specific accounts, including the original paper, the input and output data are bit representations.

The architecture that implements this specification consists of 1-dimensional LEE modules which evaluate $w+vz$ on an input (v, w) for each value of z. Note that the 2-dimensional expression $ax+by+c$ is evaluated as $(c+ax)+by$. The way the 1-dimensional LEE modules are connected together to perform this is shown in Figure 6 with the 1-dimensional LEE modules represented by triangles. Each output of the first 1-dimensional LEE module, which computes the $ax+c$ terms in parallel for each value of x, is connected to a 1-dimensional LEE module, which adds the by term in parallel for each value of y. It is these 1-dimensional LEE's that we will specify and mechanically verify in the following sections.

The internal structure of the 1-dimensional LEE module is shown in Figure 7. It is composed of a tree of smaller modules. The structure of the tree nodes is shown in Figure 8. The left output is simply the top input delayed by one clock cycle. The right output is formed by adding the two inputs, together with the carry bit saved from the previous addition.

4 Specification of the 1-dimensional LEE

Rather than specify and verify the Pixel Planes 1D LEE at the bit level where numbers are represented by a finite sequence of bits and are processed bit serially we choose a higher level model where each

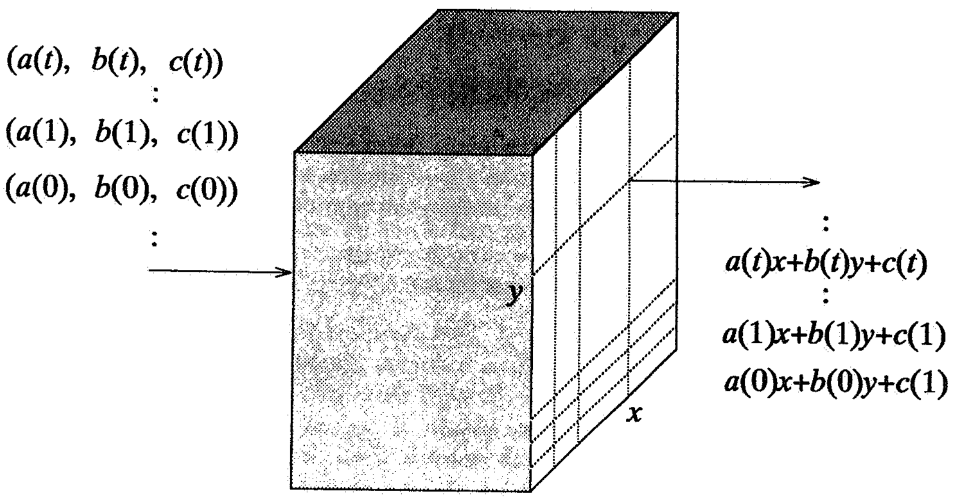

Figure 5: The LEE section of Pixel Planes.

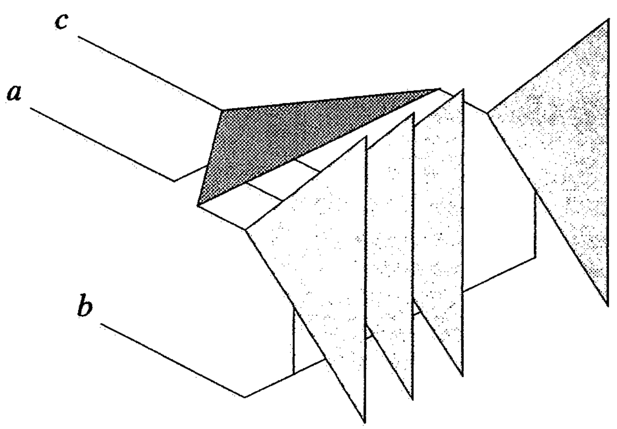

Figure 6: Inside the LEE section.

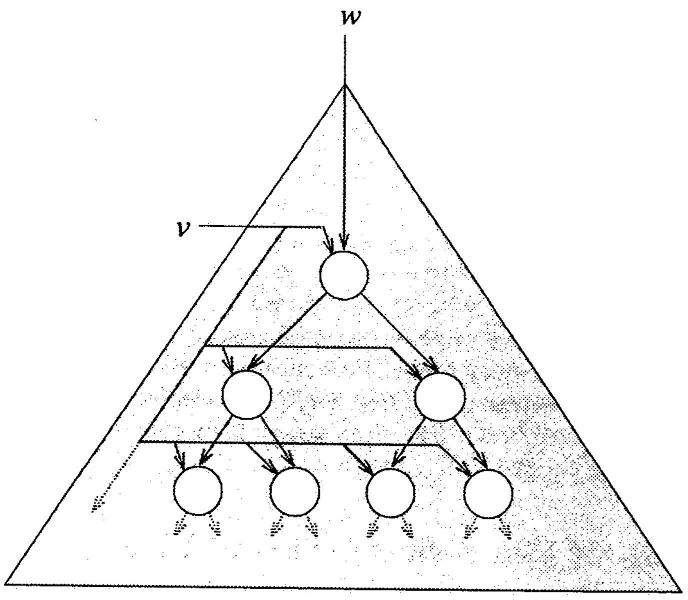

Figure 7: Inside a 1-dimensional LEE module.

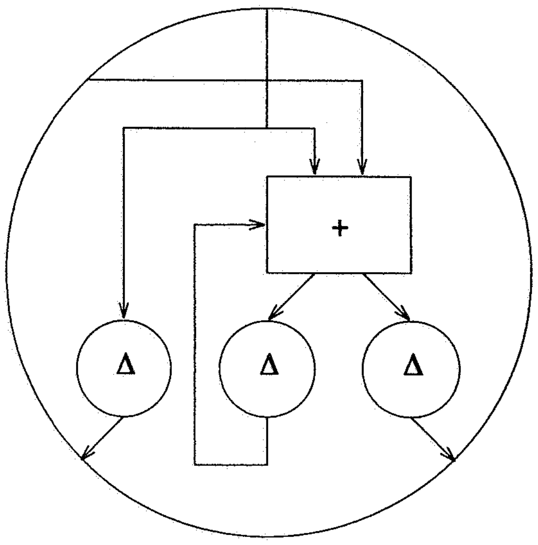

Figure 8: Inside a tree module.

number is considered to be an element of an abstract data type, whose equational axioms are chosen to be just strong enough so that the design can be verified. This approach considerably simplifies the verification process but there is a price to be paid.

In the real bit level Pixel Planes 1D LEE numbers are represented by sequences of bits, the least significant bit first (earliest). Thus multiplication by two can be done cheaply by a one cycle delay. Similarly division by two can be done by letting the bit sequence representing the number to be divided 'get ahead' by one cycle of the bit sequences representing the other numbers being pumped around the architecture. In the real Pixel Planes 1D LEE each number fed in to the v input is multiplied by 2^{n-1} by proceeding it with $n-1$ zeros and is effectively divided by two at each level in the tree as it gets progressively 'out of step' with the numbers generated by each tree processor.

In our abstract model of the Pixel Planes 1D LEE each clock cycle represents a whole new element being fed into each input, not just the current bit of a number and therefore streams of elements cannot be allowed to 'get out step'. Thus we have to add explicit one cycle delays and multipliers to keep the elements 'in step' and to handle the implicit computations on the elements sent to the v input.

4.1 Abstract data types

The data type we choose to specify Pixel Planes 1D LEE is built from three smaller data types. Time is represented by a copy $\mathbf{T} = \langle \mathbf{T} \mid 0, t+1 \rangle$ of the natural numbers without the addition and multiplication operations. The natural numbers $\mathbf{N} = \langle \mathbf{N} \mid 0, n+1, +, \times \rangle$ are used to generate particular abstract numbers via a mapping which we define below. In order to generate abstract numbers to play the role of powers of two in the multiplier modules, it is convenient to add the operation 2^n to the natural numbers. We also need the abstract numbers on which we actually do the computations. We need two operations to represent addition and multiplication and a constant to represent the zero element. We thus have an algebra $\langle R \mid 0, +, \times \rangle$. It was shown in [3] that this algebra needs to satisfy just four equational axioms.

$$(a+b)+c = a+(b+c) \tag{5}$$
$$a+0 = a \tag{6}$$
$$0 \times a = 0 \tag{7}$$
$$(a+b) \times c = (a \times c) + (b \times c) \tag{8}$$

We can thus consider the abstract data type R to be the class of all algebras which satisfy these axioms.

In order to generate abstract numbers to represent particular natural numbers we define the following function. Let α be any element of R. Then we define the function $r_\alpha : \mathbf{N} \to R$ inductively.

$$r_\alpha(0) = 0$$

$$r_\alpha(n+1) = r_\alpha(n) + \alpha$$

Thus $r_\alpha(0) = 0$, $r_\alpha(1) = 0 + \alpha$, $r_\alpha(2) = 0 + \alpha + \alpha$ and so on.

Lemma 1 *The mapping r_α is a homomorphism w.r.t. the the '+' operations in* $\mathbf{N}$ *and R; i.e.* $r_\alpha(a+b) = r_\alpha(a) + r_\alpha(b)$.

This fact is easily proved by induction and we shall prove it in the §5.2 as an illustration of how to do inductive proofs with OBJ3. We note that every ring satisfies these axioms. Particularly important is the case where $R = \mathbf{Z}$ and $\alpha = 1$.

4.2 Implementation specification

This specification is for a generic 1D LEE of height n with 2^n outputs. Since the number of modules (and hence functions and equations defining them) is dependent on n we cannot write a fixed list of equations. Instead, we subscript the module and function names and write *equation templates.*

The ease of the specification and verification of a family of SCAs defined by equation templates is very sensitive to the subscripting scheme used. For linear columns of modules the obvious subscripts to use are the natural numbers. For trees of modules the choice of subscripts is not so clear. On possibility is to subscript each module on a tree of modules by a pair of natural numbers where the first denotes the module's level in the tree and the second denotes the module's position among the modules at that level. This scheme was used in [3, 2]. This has the disadvantage that is is possible to refer to modules that do not exist and thus each equation template which refers to a tree of modules in this way will have conditions on the subscripts to explicitly exclude this possibility.

An alternative subscripting scheme which we use here is to subscript each module in a (binary) tree by its *tree path.* A module's tree path is the list of elements from $\{l, r\}$ which describe the path to the module as a sequence of left and right branches, starting at the root module. Under this scheme the root module of the tree is subscripted by the empty list *nil* and its left and right sons are subscripted by l and r respectively. Going one level down the tree left and right sons of the root module's left son get the subscripts $l.r$ and $r.r$ respectively[1].

The disadvantage of using tree paths is the need to convert them to and from pairs of natural numbers in order to specify input/output connections to the tree. For this purpose we define the following operations on tree paths inductively[2]:

$$|\ | : \{l, r\}^* \to \mathbf{N}$$

$$|nil| = 0$$

$$|l.\phi| = 1 + |\phi|$$

$$|r.\phi| = 1 + |\phi|$$

$$\# : \{l, r\}^* \to \mathbf{N}$$

$$\#(nil) = 0$$

$$\#(l.\phi) = 2 * \#(\phi)$$

$$\#(r.\phi) = 2 * \#(\phi) + 1$$

Intuitively, for a module subscripted by ϕ, $|\phi|$ is its level in the tree and $\#(\phi)$ is its position within that level where both levels and positions start at 0. We also need the following partial operation

$$@ : \mathbf{N} \times \mathbf{N} \to \{l, r\}^*$$

[1]Note that our lists should be read right to left—this somewhat counterintuitive order is chosen to correspond to the right to left order in which lists are traditionally constructed in functional languages.

[2]Note that we consider '.' to be the list concatenation operation and *nil* to be the left and right identity of this operation.

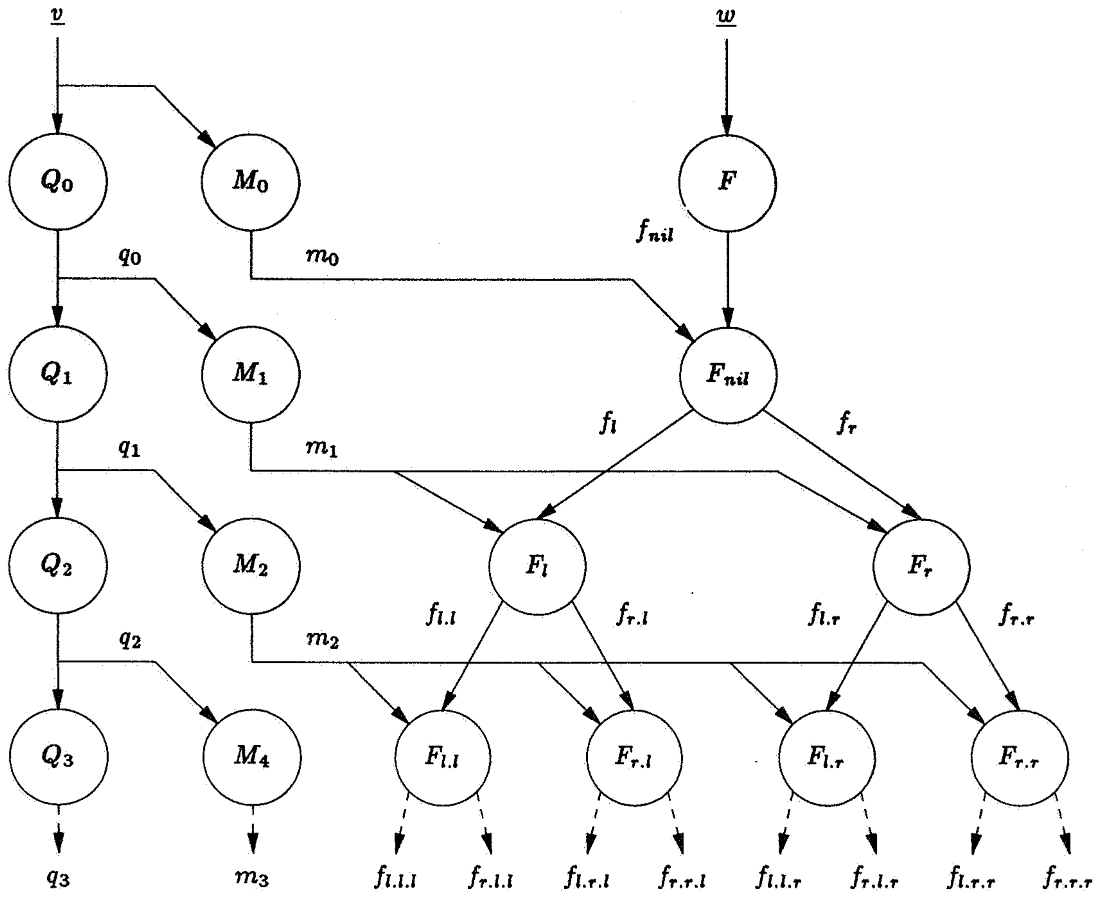

Figure 9: The abstract version of the Pixel Planes 1-dimensional LEE.

$$@(0,0) = nil$$

$$@(i+1,j) = \begin{cases} l.@(i,j/2) & \text{if } j \text{ is even} \\ r.@(i,(j-1)/2) & \text{if } j \text{ is odd} \end{cases}$$

where $@(i,j)$ is only defined for $j \in \{0,\ldots,2^i-1\}$. Intuitively $@(i,j)$ is the tree path for the jth module at level i.

In writing down our equation templates we will use natural numbers and tree paths as subscripts to the modules and function symbols and make use of various operations on them. It must be stressed that that this notation is simply a mechanism for writing down a specification of a parameterised family of SCAs. The extra copy of the natural numbers and the set of tree paths together with their operations are not part of the SCA formalism, take no part in the actual computations performed by the architecture and can be eliminated whenever the parameter n is instantiated to a particular natural number. However as we shall see in §5 this mechanism can be formalised and used to prove theorems about a parameterised family of SCAs. The abstract version of the Pixel Planes 1D LEE which we specify is shown in Figure 9 (compare it with Figure 7).

The process of writing down the implementation specification is straightforward—for each module shown in Figure 9 we assign a value function. (Note that under our convention, modules have more than one distinct output are split and each output gives rise to a value function.) The value functions are defined using the scheme in §2.1. Here all modules are initialised to $0 \in R$ at time 0.

We start with the one cycle delays, $Q_0,\ldots,Q_{n-2}$. Module Q_0 takes its input from the $\underline{v}$ input stream; the others each take their input from their immediate predecessor. The value functions for the delay modules are q_i for $i \in \{0, \ldots, n-2\}$:

$$q_i : \mathbf{T} \times [\mathbf{T} \to R]^2 \to R$$

$$q_i(0, \underline{v}, \underline{w}) = 0$$

$$q_0(t+1, \underline{v}, \underline{w}) = \underline{v}(t)$$

$$q_{i+1}(t+1, \underline{v}, \underline{w}) = q_i(t, \underline{v}, \underline{w})$$

Next we have the multiplication processors $M_0, \ldots, M_{n-1}$. Module M_0 takes its input directly from the $\underline{v}$ input stream; the others take their inputs from the preceeding delay module. Each multiplication processor M_i multiplies its input by the element $r_\alpha(2^{n-1-i})$. The value functions for the multiplication processors are m_i for $i \in \{0, \ldots, n-1\}$:

$$m_i : \mathbf{T} \times [\mathbf{T} \to R]^2 \to R$$

$$m_i(0, \underline{v}, \underline{w}) = 0$$

$$m_0(t+1, \underline{v}, \underline{w}) = r_\alpha(2^{n-1}) \times \underline{v}(t)$$

$$m_{i+1}(t+1, \underline{v}, \underline{w}) = r_\alpha(2^{n-(i+2)}) \times q_{i-1}(t, \underline{v}, \underline{w})$$

Finally we have the tree processors. The topmost tree processor P is simply a one cycle delay with a single value function f_{nil}. The processors P_ϕ for $\phi \in \{l, r\}^*$, $|\phi| < n$ each have two outputs; a left one with value function $f_{l.\phi}$ and a right one with value function $f_{r.\phi}$. We define these value functions as follows, for $\phi \in \{l, r\}^*$, $|\phi| \leq n$:

$$f_\phi : \mathbf{T} \times [\mathbf{T} \to R]^2 \to R$$

$$f_\phi(0, \underline{v}, \underline{w}) = 0$$

$$f_{nil}(0, \underline{v}, \underline{w}) = 0$$

For $\phi \in \{l, r\}^*$, $|\phi| < n$:

$$f_{l.\phi}(t+1, \underline{v}, \underline{w}) = f_\phi(t, \underline{v}, \underline{w})$$

$$f_{r.\phi}(t+1, \underline{v}, \underline{w}) = f_\phi(t, \underline{v}, \underline{w}) + m_{|\phi|}(t, \underline{v}, \underline{w})$$

The stream transformer defined by this implementation specification is

$$LEE^n : [\mathbf{T} \to R]^2 \to [\mathbf{T} \to R]^{2^n}$$

$$LEE^n(\underline{v}, \underline{w})(t) = (f_{@(n,0)}(t, \underline{v}, \underline{w}), \ldots, f_{@(n,2^n-1)}(t, \underline{v}, \underline{w}))$$

4.3 User specification

Let R be any structure satisfying equations (5)-(8). Our user specification is

$$U^n : [\mathbf{T} \to R]^2 \to [\mathbf{T} \to R \cup \{u\}]^{2^n}$$

defined by its coordinate functions; for $j \in \{1, \ldots, 2^{n-1}\}$:

$$U_j^n(\underline{v}, \underline{w})(t) = \begin{cases} u & \text{if } t < n+1 \\ \underline{w}(t-(n+1)) + r_\alpha(j) \times \underline{v}(t-(n+1)) & \text{otherwise} \end{cases}$$

Here $(n+1)$ is the time delay between the data arriving at the sources and valid results being available at the sinks.

Now for verification we want the i/o specification to satisfy the user specification. We derive the following equivalent condition on value functions; for $j \in \{0, \ldots, 2^n - 1\}$:

$$\forall t \in \mathbf{T}.[t \geq n+1 \Rightarrow f_{@(n,j)}(t, \underline{v}, \underline{w}) = \underline{w}(t-(n+1)) + r_\alpha(j) \times \underline{v}(t-(n+1))]$$

For automatic verification it is convenient to eliminate the subtraction operations and replace the partial operation @ with the fully defined operations # and | |. An equivalent condition is given in the following theorem.

Theorem 1 *If R satisfies equations (5)-(8) then for $\phi \in \{l, r\}^*$ and $\in \mathbf{T}$,*

$$|\phi| = n \Rightarrow (f_\phi(t+n+1, \underline{v}, \underline{w}) = \underline{w}(t) + r_\alpha(\#(\phi)) \times \underline{v}(t))$$

It is this condition which expresses the correctness of the algorithm for any structure R that satisfies our abstract arithmetic axioms.

5 Verification using OBJ3

OBJ3 [9] is an algebraic specification language. There are two major syntactic entities in OBJ3—objects and theories—which are known collectively as modules. Both consist of a many sorted signature (a declaration of sorts, constants and operations) and a set of (possibly conditional) equations. The difference between objects and theories is mostly a semantic one—objects denote the initial model of the signature and equations while theories denote the variety of all models. Operationally the OBJ3 interpreter provides a term rewriting engine where conditional equations are handled by backtracking on failure.

Sometimes an equation $t_1 = t_2$ can be proved from a set of hypotheses H by rewriting t_1 and t_2 to normal forms in a module containing H and checking these normal forms for syntactic equality. However we will use two further principles in the verification process.

The first is the 'Constants Lemma' which allows us to replace universally quantified variables with unconstrained variables. We state this theorem informally—its formal statement and proof is given in [8].

Lemma 2 *Let Σ, X be disjoint signatures, let E be a set of Σ-equations and let A be a ground $(\Sigma \cup X)$-equation. We can consider A as a Σ-equation, $\forall X.A(X)$ by universally quantifying all constant and function symbols drawn from X. Then $\forall X.A(X)$ is true in every $\langle \Sigma, E\rangle$ model if and only if A is true in every $\langle \Sigma \cup X, E\rangle$ model.*

The second principle is proof by induction. The general structural induction scheme for initial models is stated and proved in [8]. We however require induction for a class of models, called *standard models* in which certain sorts (including natural numbers and tree paths) have their usual interpretations but other sorts may not.

5.1 Data types in OBJ3

Since the statement of correctness requires expressions that combine module indices and time we will use a single copy of the natural numbers for both purposes. Notice that we also need subtraction on the natural numbers. Since subtraction is only partially defined on the natural numbers we have two options. Firstly we can alter the semantics of subtraction so that $i - j = 0$ for $i < j$. This is called cut-off arithmetic. The second approach is to use the integers with natural numbers as a subsort. Although this latter approach is conceptually cleaner it leads to difficulties when we need to use an expression which syntactically denotes an integer (but which is known to always evaluate to a natural number because of conditions on some of its subexpressions) as an argument to a operation which is only defined on natural numbers. Thus we take the first option. We specify the natural numbers with cut off subtraction by the following OBJ3 module.

```
obj NAT is sort Nat .
        ops 0 1 2 : -> Nat .
        op _+_ : Nat Nat -> Nat [assoc comm prec 3] .
        op _*_ : Nat Nat -> Nat [assoc comm prec 2] .
        op _-_ : Nat Nat -> Nat [prec 3] .
        op 2^_ : Nat -> Nat [prec 1] .
        op _>_ : Nat Nat -> Bool .

vars I J K : Nat .
```

```
        eq I + 0 = I .
        eq I - 0 = I .
        eq 0 - I = 0 .
        eq (I + K) - (J + K) = I - J .
        eq I * 0 = 0 .
        eq I * 1 = I .
        eq I * (J + K) = I * J + I * K .
        eq 2^ 0 = 1 .
        eq 2^ (I + 1) = 2^ I * 2 .
        eq 0 > I = false .
        eq I + 1 > 0 = true .
        eq I + K > J + K = I > J .
        eq 2 = 1 + 1 .
*** Cut off arithmetic lemmas:
        cq (I - J) + K = (I + K) - J if I + 1 > J .
        eq (2^ I) * J  + (2^ I) * J = 2^ (I + 1) * J .
        cq I + K > J = true if I > J .
endo
```

We also include three trivial lemmas that hold in the initial model of this module. Notice that the '+' and '×' operations are associative and commutative and we specify this in OBJ3 by means of the 'assoc' and 'comm' attributes. Technically these associativity and commutativity properties are also lemmas and the initial model would be unchanged if they were left out. Putting commutativity in as an equation (with the operational semantics being *left→right* rewrite rules) would lead to non-termination. Similarly including the associativity law both ways around would also lead to non-termination. The 'prec' attribute is used to specify the precedences of the operations for the OBJ3 mix-fix parser. We also need to add equations to define the other properties of the operations.

We next define the abstract arithmetic data type R. Recall that this has one constant, '0' and two operations, '+' and '×' and satisfies four equational axioms. Since operationally these are considered as rewrite rules which way around they are declared is important. For instance if the associativity axiom is written as $(A+B)+C = A+(B+C)$ some of our proofs would fail. The initial model of this signature and set of equations is in fact the unit algebra. Our intention is to verify the LEE over the class of all algebras which satisfy these equations and thus we want variety semantics and so we use an OBJ3 theory.

```
th ABS is sort Abs .
        op 0 : -> Abs .
        op _+_ : Abs Abs -> Abs [prec 3] .
        op _*_ : Abs Abs -> Abs [prec 2] .

vars A B C : Abs .
        eq A + (B + C) = (A + B) + C .
        eq A + 0 = A .
        eq 0 * A = 0 .
        eq (A + B) * C = (A * C) + (B * C) .
endth
```

We also need to formalise the tree paths and their operations. This is done using two sorts. The sort `Branch` just has two constants `l` and `r`. The sort `Path` is intended to be a list of elements from `Branch` and has a constant `nil`. These lists are constructed by an invisible[3] operation which takes an element from `Branch` and a list and returns a new list. The operations $|\;|$ and $\#$ which map tree paths into natural numbers are defined in the same recursive manner as we defined them in §4.2.

```
obj TREE is protecting NAT .
```

[3] The reason the invisible operations is used rather than a "." is that the latter tends to confuse the OBJ3 mixfix parser

```
        sorts Branch Path .
        ops l r : -> Branch .
        op nil : -> Path .
        op __ : Branch Path -> Path .
        op |_| : Path -> Nat .
        op # : Path -> Nat .

var B : Branch .
var P : Path .
        eq | nil | = 0 .
        eq | B P | = 1 + | P | .
        eq #(nil) = 0 .
        eq #(l P) = 2 * #(P) .
        eq #(r P) = 1 + 2 * #(P) .
endo
```

We now need the r_α operation to map natural numbers into our abstract numbers. We introduce this as another module defined in terms of the NAT and ABS modules above.

```
th RMAP is protecting NAT + ABS .
        op alpha : -> Abs .
        op r_ : Nat -> Abs [prec 1] .

var N : Nat .
        eq r(0) = 0 .
        eq r(s N) = r(N) + alpha .
endth
```

Here 'protecting' is one of OBJ3's several module importation modes and the expression 'NAT+ABS' forms the module containing all the sorts, operations and equations of both NAT and ABS (module union). The idea of the protecting mode is that the semantics of the imported modules is protected—no new elements are created and no previously unequal elements are equated.

5.2 Proving the homomorphism property of r_α

Recall from §4.1 that we stated that Lemma 1 could be easily proved by induction. We now do this proof as an example of how to do inductive proofs with OBJ3. We do induction on the second argument of r_α.

We start with the basis case: $r_\alpha(a, 0) = r_\alpha(a) + r_\alpha(0)$. To prove this in OBJ3 we first create a new module that includes an 'unconstrained' constant to represent the variable a (making use of the 'Constants Lemma').

```
th BASIS is protecting RMAP .
        op a : -> Nat .
endth
```

We then ask OBJ3 to do the proof by rewriting the terms on either side of the equality symbol to a normal form and then checking for syntactic equivalence.

```
reduce in BASIS : r(a + 0) == r(a) + r(0) .
```

We then do the induction step; assume that $r_\alpha(a + b) = r_\alpha(a) + r_\alpha(b)$ and prove $r_\alpha(a + b + 1) = r_\alpha(a) + r_\alpha(b + 1)$. We write the induction hypothesis as another OBJ3 module:

```
th INDUCT is protecting BASIS .
        op b : -> Nat .
        eq r(a + b) = r(a) + r(b) .
endth
```

And do the proof by rewriting as before:

```
reduce in INDUCT : r(a + s b) == r(a) + r(s b) .
```

Now that the lemma is proven we create a new module LEMMA1 containing it so we can use it later.

```
th LEMMA1 is protecting RMAP .
vars A B : Nat .
        eq r(A + B) = r(A) + r(B) .
endth
```

5.3 Converting the implementation specification to OBJ3

Recall that our original specification consisted of equation templates which we considered to be a specification of an infinite family of SCAs for $n = 2, 3, \ldots$. Thus each subscripted value function in our specification actually refers to an infinite family of value functions. In order to formalise the equation templates in OBJ3 we use the convention that the subscript on a value function becomes a new argument to the function.

Since OBJ3 does not support higher order functions we must remove the stream arguments $\underline{v}, \underline{w}$ from the value functions in our original specification. We simply declare $\underline{v}$ and $\underline{w}$ to be functions from the naturals to our abstract arithmetic data type without giving any equations to define them. They are thus 'unconstrained' functions and behave like constant function symbols.

Using these conventions our implementation specification may be written as the following OBJ3 module.

```
obj LEE is protecting LEMMA1 + TREE .
        op n : -> Nat .
        op v : Nat -> Pix .
        op w : Nat -> Pix .
        op q : Nat Nat -> Pix .
        op m : Nat Nat -> Pix .
        op f : Path Nat -> Pix .

var I T : Nat .
var P : Path .
        cq q(I, 0) = 0                  if n > I + 1 .
        eq q(0, T + 1) = v(T) .
        cq q(I + 1, T + 1) = q(I, T)    if n > I + 2 .

        cq m(I, 0) = 0                                          if n > I .
        eq m(0, T + 1) = r(2^ (n - 1)) * v(T) .
        cq m(I + 1, T + 1) = r(2^ (n - (I + 2))) * q(I, T)      if n > I + 1 .

        cq f(P, 0) = 0                                  if n + 1 > | P | .
        eq f(nil, T + 1) = w(T) .
        cq f((l P), T + 1) = f(P, T)                    if n + 1 > | l P | .
        cq f((r P), T + 1) = f(P, T) + m(| P |, T)      if n + 1 > | r P | .
endo
```

5.4 Verification of the LEE

The general idea is to prove a closed form for each of the (parameterised) value functions using induction on natural numbers, induction on tree paths and case analysis.

We start with the delay modules.

Theorem 2 *For $i \in \mathbf{N}$, $i + 1 < n$*

$$q(i, t + i + 1) = \underline{v}(t)$$

We prove this by induction on i. First we create a module to assert that $i+1 < n$ for the basis case where $i = 0$. We then reduce the equation $q(0, t+0+1) = \underline{v}(t)$ in it.

```
obj T2-BASIS is protecting LEE .
        op t : -> Nat .

        eq n > 1 = true .
endo

reduce q(0, t + 0 + 1) == v(t) .
```

Notice that in the module T2-BASIS we included the equation $n > 1 = tt$ instead of the equation $n > 0+1 = tt$. This simplification by hand is necessary since the left hand side of the latter equation is reducible and any term that might match it could be reduced before a match takes place. In order to prove the induction step we create a module to assert the induction hypothesis together with the condition $(i+1)+1 < n$ and reduce the equation $q((i+1), t+(i+1)+1) = \underline{v}(t)$ in it.

```
obj T2-INDUCT is protecting T2-BASIS .
        op i : -> Nat .

        cq q(i, t + i + 1) = v(t) if n > i + 1 .
        eq n > (i + 1)  + 1 = true .
        eq n > i + 1 = true .                         *** extra property
endo

reduce q(i + 1, t + (i + 1) + 1) == v(t) .
```

Notice that we had to add the extra property that $n > i+1$. Although this follows trivially from $n > (i+1)+1$ it cannot easily be proved by rewriting. Handling inequalities is awkward in a pure term rewriting system—ideally we would like a lemma of the form $i > j+1 \Rightarrow i > j$ but if this were proved and added as a conditional equation we would get nontermination. We create a new module asserting the theorem so that it can be used in later proofs.

```
obj THEOREM2 is protecting LEE .
var I T : Nat .
        cq q(I, T + I + 1) = v(T) if n > I + 1 .
endo
```

Next we consider the multiplier modules.

Theorem 3 *For* $i \in \mathbf{N}$, $n > i$

$$m(i, t+i+1) = r(2^{n-(i+1)}) \times v(t)$$

We prove this by case analysis on i. To prove the case $i = 0$ we create a module to assert that $n > 0$ and reduce $m(0, t+0+1) = r(2^{n-(0+1)}) \times \underline{v}(t)$ in it.

```
obj T3-CASE1 is protecting THEOREM2 .
        op t : -> Nat .

        eq n > 0 = true .
endo

reduce m(0, t + 0 + 1) == r(2^ (n - (0 + 1))) * v(t) .
```

The other case where $i > 0$ is done by proving the theorem for $i+1$ where i ranges over all natural numbers. In fact this technique is really induction on natural numbers except we do not use the induction hypothesis.

```
obj T3-CASE2 is protecting THEOREM2 .
        ops i t : -> Nat .

        eq n > i + 1 = true .
endo

reduce m(i + 1, t + (i + 1) + 1) == r(2^ (n - ((i + 1) + 1))) * v(t) .
```

Again we create a new module to assert the theorem for latter use.

```
obj THEOREM3 is protecting THEOREM2 .
var I T : Nat .
        cq m(I, T + I + 1) = r(2^ (n - (I + 1))) * v(T) if n > I .
endo
```

Finally we consider the tree processors.

Theorem 4 *For* $p \in \{l, r\}^*$, $|p| < n + 1$

$$f(p, t + |p| + 1) = w(t) + r(2^{n-|p|} \times \#(p)) \times \underline{v}(t)$$

Note that the correctness theorem is just a special case of this theorem when $|p| = n$. We prove this theorem by induction on the tree path p. The basis case where $p = nil$ is straightforward.

```
obj T4-BASIS is protecting THEOREM3 .
        op t : -> Nat .
endo

reduce f(nil, t + | nil | + 1) == w(t) + r(2^ (n - | nil |) * #(nil)) * v(t) .
```

In the induction step we need to consider both left and right branchs so we need to reduce two equations in the module containing the induction hypothesis.

```
obj T4-INDUCT is protecting T4-BASIS .
        op p : -> Path .
vars I J K : Nat .

        cq f(p, t + | p | + 1) =
                 w(t) + r(2^ (n - | p |) * #(p)) * v(t) if n + 1 > | p | .
        eq n > | p | = true .
endo

reduce f((l p), t + | l p | + 1) ==
           w(t) + r(2^ (n - | l p |) * #(l p)) * v(t) .

reduce f((r p), t + | r p | + 1) ==
           w(t) + r(2^ (n - | r p |) * #(r p)) * v(t) .
```

This completes the verification.

References

[1] S. M. Eker. *Formal Foundations for the Design of Rasterization Algorithms and Architectures.* PhD thesis, School of Computer Studies, University of Leeds, 1991.

[2] S. M. Eker, V. Stavridou, and J. V. Tucker. Verification of Synchronous Concurrent Algorithms Using OBJ3: A Case Study of the Pixel Planes Architecture. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 231–252. Springer-Verlag, 1991.

[3] S. M. Eker and J. V. Tucker. Specification and verification of synchronous concurrent algorithms: A case study of the Pixel-Planes architecture. In P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, pages 16–49, Wokingham, England, 1989. Addison Wesley.

[4] H. Fuchs. An introduction to Pixel-Planes and other VLSI intensive graphics systems. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 675–688, Berlin, 1988. Springer-Verlag.

[5] H. Fuchs, J. Goldfeather, J. P. Hultquist, S. Spach, Jr. F. P. Brooks, J. G. Eyles, and J. Poulton. Fast spheres, shadows, textures, transparencies and image enhancements in Pixel-Planes. *Computer Graphics*, 19(3):169–187, July 1985.

[6] H. Fuchs and J. Poulton. Pixel-Planes: A VLSI-orientated design for a raster graphics engine. *VLSI Design*, 2(3):20–28, 1981.

[7] H. Fuchs, J. Poulton, A. Paeth, and A. Bell. Developing Pixel-Planes, a smart memory-based raster graphics system. In *Proceedings of the 1982 MIT Conference on Advanced Research in VLSI*, pages 137–146, Dedham MA, 1982. Artech House.

[8] J. A. Goguen. OBJ as a Theorem Prover with Applications to Hardware Verification. In *Proceedings of 2nd Banff Workshop on Hardware Verification*, Banff, Canada, June 1988.

[9] Joseph A. Goguen, Timothy Winkler, Josè Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ3. In J. A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ*. Cambridge University Press, (to appear).

[10] H. A. Harman and J. V. Tucker. Clocks, retiming and the formal specification of a UART. In G. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 375–396. North-Holland, 1988.

[11] N. A. Harman. *Formal Specification of Digital Systems*. PhD thesis, School of Computer Studies, University of Leeds, 1989.

[12] K. M. Hobley. *Specification and Verification of Synchronous Concurrent Algorithms*. PhD thesis, School of Computer Studies, University of Leeds, 1991.

[13] K. M. Hobley, B. C. Thompson, and J. V. Tucker. Specification and verification of synchronous concurrent algorithms: A case study of a convolution algorithm. In G. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 347–374. North-Holland, 1988.

[14] A. R. Martin. *Specification and Simulation of Synchronous Concurrent Algorithms*. PhD thesis, School of Computer Studies, University of Leeds, 1989.

[15] K. Meinke. *A Graph Theoretic Model of Synchronous Concurrent Algorithms*. PhD thesis, Department of Computer Studies, University of Leeds, 1988.

[16] K. Meinke and J. V. Tucker. Specification and representation of synchronous concurrent algorithms. In F. H. Vogt, editor, *Proceedings of Concurrency '88*, pages 163–180. Springer-Verlag LNCS 335, 1988.

[17] John Poulton, Henry Fuchs, John D. Austin, John G. Eyles, Justin Heinecke, Cheng-Hong Hsieh, Jack Goldfeather, Jeff P. Hultquist, and Susan Spach. PIXEL-PLANES: Building a VLSI-based graphic system. In H. Fuchs, editor, *Proceedings of the Chapel Hill Conference on VLSI*, pages 35–60, Rockville MA, 1985. Computer Science Press.

[18] B. C. Thompson. *A Mathematical Theory of Synchronous Concurrent Algorithms*. PhD thesis, Department of Computer Studies, University of Leeds, 1987.

[19] B. C. Thompson and J. V. Tucker. Synchronous concurrent algorithms. Technical report, Department of Mathematics and Computer Science, University College Swansea, 1989.