Microprocessing and Microprogramming 38 (1993) 403-410
North-Holland

Relational Algebra as Formalism for Hardware Design

A.J.W.M. ten Berg, C. Huijs, Th. Krol

Department of Computer Science
University of Twente
P.O.Box 217, 7500 AE Enschede, The Netherlands

This paper introduces relational algebra as an elegant formalism to describe hardware behaviour. Hardware behaviour is modelled by functions that are represented by sets of tables. Relational algebra, developed for designing large and consistent databases is capable to operate on sets of tables and hence on sets of hardware behaviour functions. It pairs the advantages of formal design, such as verification and provable correct designs, to relative ease and simplicity of description. Descriptions tend to be directly mappable to hardware components such as PLA's. This in contrast to other, most predicate based, formal methods that create long and complex descriptions of hardware which make automated theorem provers a necessity for design tasks of practical sizes. Relational algebra can be applied for both combinatorial as sequential designs and also for transformations between designs. We demonstrate the power of this formalism by means of the Mealy to Moore transformation and show that it takes only a few operations.

1. INTRODUCTION

A paradigm for hardware design that becomes more and more important is that of formal design. Formal design, by means of algebra or calculus, makes it possible to prove the behavioural equivalence of different hardware functional decompositions, which is the base for transformational design. Or, in the verification sense, to prove that one set of functions is behaviourally equivalent to a second set of functions. We introduce a new formalism suitable for both combinatorial and sequential hardware design, that originates from the database design field [1,2]. Relational algebra operates on sets of tables that represent relations or functions and allows not only to extract parts of tables but also to compose or "join" tables to larger tables. Relational algebra treats the contents of tables as arbitrary objects, elements of sets. Hence, it can handle behavioural descriptions at both the symbolic level, where inputs, outputs and states are defined as sets of symbols, and the implementation level, where Boolean codes replace those symbols.

Other formalisms [5,6,7] describe hardware functions by predicates and are directed towards hardware verification. These systems require theorem provers for feasible applications, because they lack the powerful operations available in relational algebra. Furthermore, the relation to hardware is quite direct for a table representation, which is in general not the case for predicate based representations. In the latter case, the task to find a sequence of transformations that leads to optimal hardware becomes more difficult.

Among the more pragmatic approaches, we find hardware description languages such as VHDL [8]. VHDL allows to check the functionality of implementations only by simulation, which is also a very computation intensive check. Hence, for practical design complexities, a simulation must be necessarily limited and does not cover the complete behaviour. Furthermore, transformations to more optimal designs cannot be described in the language VHDL itself.

From the digital hardware behaviour point of view, function tables exist for a long time. For example, the truth-tables in use for the basic logic functions as AND, OR and NOT, or tables for Finite State Machine functions [3]. Such table representations become more and more feasible, because a complex multi-input truth-table can be mapped directly onto a single programmed logic array (PLA). But, it can also be decomposed into a

set of smaller functions [4] such that the structure found in the total function is used to obtain more optimal hardware. Such decompositions can be realized elegantly by operations from relational algebra. This paper does not address the optimality aspect primarily, but shows how descriptions of hardware can be formulated with relational algebra.

2. BASIC MATHEMATICS

This section introduces some basic definitions. Relational algebra is build on set theory. Let A be a set, then the number of elements in A is denoted by $\#A$. The fact that an object is a set is denoted by $set(A)$. Let A and B be sets, then the cartesian product $A \times B$ is the set of ordered pairs defined by:

$$(\forall z)[z \in A \times B \Leftrightarrow (\exists x,y)[z=<x,y> \wedge x \in A \wedge x \in B]] \quad (1)$$

Let A and B be sets. A *function* F from A into B , denoted as $function(F,A,B)$, is a subset of $A \times B$ such that for every $a \in A$, there exists one and only one $b \in B$ such that $<a,b> \in F$. If $<a,b> \in F$, then we write $F(a)=b$. The set A is called the *domain* of F and B is the called the *codomain* of F. The *image* of F is then the subset of B which follows from the tuples in F. Functions can also be defined with an implicit domain and codomain; $function(F, A) \Leftrightarrow (\exists B)[function(F, A, B)]$. Because functions are sets, the set operations are valid on functions, but the union of two functions is not by definition a function. A function of which all elements of its codomain are sets is called a *setfunction*.

A subset of the tuples of a function is extracted by the *restriction* operation. Let F be a function and C be a set, then the restriction of F to C, denoted $F \upharpoonright C$, is the subset of those tuples $<a,b> \in F$ such that $a \in C$. The complement of $F \upharpoonright C$, denoted as $F \not\upharpoonright C$ is the subset of tuples $<a,b> \in F$ such that $a \notin C$. Formally;

$$(\forall z)[z \in F \upharpoonright C \Leftrightarrow z \in F \wedge (\exists x,y)[z=<x,y> \wedge x \in C]] \quad (2)$$
$$(\forall z)[z \in F \not\upharpoonright C \Leftrightarrow z \in F \wedge (\exists x,y)[z=<x,y> \wedge x \notin C]] \quad (3)$$

For example, let function $F=\{<a,1>, <b,2>, <c,3>\}$ and set $C=\{a,b\}$, then $F \upharpoonright C = \{<a,1>, <b,2>\}$, while $F \not\upharpoonright C = \{<c,3>\}$. The next section defines the operations of relational algebra. The operations in a relational algebra are defined on sets of functions, called tables.

3. RELATIONAL ALGEBRA

In our definitions, we follow to some extent the definitions given by de Brock [2]. In contrast with other descriptions of relational algebra [1] de Brock defines a table as a collection of functions instead of a set of n-tuples. This definition integrates the table heading into the algebra, instead of treating it as a separate object. For hardware behaviour, a table heading represents the identifiers associated with its input, output and internal variables. A table is a set of functions such that all functions share a common domain. This domain is called the *attribute set* of the table. Formally, let A be a set, then;

$$table(T, A) \Leftrightarrow (\forall t)[(t \in T) \Rightarrow function(t, A)] \quad (4)$$

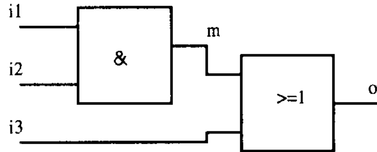
In addition, we introduce two subsets, called *IN* and *OUT* of the attribute set to distinguish the attributes associated with respectively the input and output variables. Notice that *IN* and *OUT* may have a non-empty intersection, as found for example in the definition of VHDL also.

Now we introduce the basic *operations* on tables. Let A be a set and let T and T' be tables over A. Tables are sets of functions and therefore the *intersection* '$\cap$', the *union* '$\cup$' and *difference* '-' operations are defined on them [2]. Clearly, if the attribute sets of both tables are identical then the result of these operations is again a table. The *natural-join operation* combines two tables T and T into a new table T'', such that the attribute set of this result table is the union of the attribute sets of both tables and the result table T'' is a set of functions which are the unions of those function pairs $t \in T$, $t' \in T'$ for which their union is again a function. Formally, let T and T' be tables, then the natural-join operation, denoted $\bowtie$, of T and T' is defined by:

$$(\forall t)[t \in T \bowtie T' \Leftrightarrow (\exists f,f')[(f \in T) \wedge (f' \in T') \wedge (t=f \cup f') \wedge function(f \cup f')]] \quad (5)$$

The next example shows that the AO21 function (AND2-OR2 in cascade), listed in table *T_AO21** and depicted in figure 2 can be considered as the natural-join of a 2-input AND with a 2-input OR

function. The behaviour specification of the 2-input AND function is given by table *T_AND* on {*in1, in2, m*} with *IN*={*in1, in2*} and *OUT*={*m*}. Table *T_AND* consists of four functions with arbitrary names, for example *T_AND*={*row1, row2, row3, row4*}, of which for example function *row2* = {<*in1*,0>, <*in2*,1>, <*m*,0>}.



i1	*i2*	*m*
0	0	0
0	1	0
1	0	0
1	1	1

T_AND ⋈

m	*i3*	*o*
0	0	0
0	1	1
1	0	1
1	1	1

T_OR =

i1	*i2*	*m*	*i3*	*o*
0	0	0	0	0
0	0	0	1	1
0	1	0	0	0
0	1	0	1	1
1	0	0	0	0
1	0	0	1	1
1	1	1	0	1
1	1	1	1	1

*T_AO21**

Figure 1 Tables for AO21 logic function

Due to the natural-join, the table *T_AO21** contains the attribute *m*. Attribute *m* is redundant in this table, because it is no element of either the *IN* or *OUT* set of *T_AO21**. With the *projection* operation such redundant columns can be removed. The projection operation restricts a table to a subset of its attribute set. Let *B* be a set and *T* be a table, then the projection of *T* on *B*, denoted as $T \upharpoonright B$, is defined by:

$$(\forall t)[t \in (T \upharpoonright B) \Leftrightarrow (\exists s)[(s \in T) \wedge (t = s \upharpoonright B)]] \quad (6)$$

The projection; $T_AO21^* \upharpoonright \{i1, i2, i3, o\}$ removes the *m* attribute. For interconnections attributes of tables may have to change. For this we provide the *rename* operation. Let *T* be a table and *r* a function, then the *rename* operation on *T*, denoted ∞, is defined by:

$$(\forall z)[z \in (T \infty r) \Leftrightarrow (\exists t)[\ (t \in T) \wedge (z = t \circ r)] \quad (7)$$

Where '$\circ$' is the function composition. Let *F* be a setfunction, then the *general product* of *F*, denoted as $\Pi(F)$, is a table on the attribute set that is also the domain of *F*, such that it contains in its rows the generalized cartesian product of the image of *F*. Hence, Π is a function. So formally:

$$(\forall t)[t \in \Pi(F) \Leftrightarrow (\exists A)[setfunction(F,A) \wedge function(t,A) \wedge (\forall a)[a \in A \Rightarrow t(a) \in F(a)]]] \quad (8)$$

The general product is strongly related to the cartesian product. For example, let *I,S* and *O* be sets and consider the cartesian product $I \times S \times O$ and a setfunction *F*, defined by $F=\{<a,I>, <b,S>, <c,O>\}$. The cartesian product describes the set of all 3-tuples $<x,y,z>$ such that $x \in I$, $y \in S$ and $z \in O$. However, a tuple $<x,y,z>$ can also be considered as the image of a function *f* on domain $\{a,b,c\}$ that is defined by $f=\{<a,x>, <b,y>, <c,z>\}$. The general product Π is then the set of all possible functions *f*, such that $x \in I$, $y \in S$ and $z \in O$. Relations and tables associate in a similar way. Consider a 3-ary relation $R \subseteq I \times S \times O$ and again the setfunction $F=\{<a,I>,<b,S>,<c,O>\}$, then *R* is associated to a table *T* such that $T \subseteq \Pi(F)$.

The *type* of a variable is usually a set of symbols or values from which the value of the variable is an element. Analoque to this, the type of an attribute is a set of symbols or values. The *type function of a table* is then a setfunction on the attribute set of the table. Let *Y* be the type function of *T*, then the type-check of *T* is performed by the predicate *type*(*Y, T*);

$$type(\ Y, T) \Leftrightarrow T \subseteq \Pi(Y) \quad (9)$$

For example, the type of the table *T_AND* in figure 1 is be a setfunction $Y=\ \{<in1,\{0,1\}>, <in2,\{0,1\}>, <m, \{0,1\}>\}$. Consider the type *Y* of a table *T* and its general product $\Pi(Y)$. Due to several reasons, an specification table *T* may be incomplete, in the sense that it does not specify the output

variable values for all combinations of the input variable values, thus $(T \Vdash IN) \subset (\Pi(Y) \Vdash IN)$. Incomplete specifications make it impossible to determine if some other table is an implementation. For example: let A be a table that specifies behaviour and let Y be its type; $Y=\{<in1,\{0,1\}>, <in2,\{0,1\}>, <out,\{0,1\}>\}$ with $IN=\{in1,in2\}$ and let B be another table with type Y and identical IN set.

in1	*in2*	*out*
0	0	0
0	1	1
1	0	1

table *A* (specification)

in1	*in2*	*out*
0	0	0
0	1	1
1	0	1
1	1	0

table *B* (implementation?)

Figure 2 Example of an incomplete specification

If $(\forall t)[t \in B \Rightarrow t \in A]$, then table B is called an implementation of table A. Unfortunately, this is false for the tables in figure 2, because the function $\{<in1,1>, <in2,1>, <out,0>\}$ occurs in B but not in A. Hence, we require that tables cover the *complete* general product of their type function projected on their IN attribute subset. Let T be a table and let $type(Y,T)$ and IN be a subset of the attribute set of T, then;

$$complete(T, Y, IN) \Leftrightarrow (T \Vdash IN = (\Pi(Y) \Vdash IN)) \quad (10)$$

Some operations of the relational algebra may result in tables that are not complete, hence a *completion* operation is part of the algebra. Let $table(T,A)$ and $type(Y,T)$ be true and IN be the input attribute subset, $IN \subseteq A$, then the completion $\uparrow$ of T with respect to $Y \vdash IN$ is defined by:

$$(\forall t)[t \in (T \uparrow (Y \vdash IN)) \Leftrightarrow (t \in T \vee (t \in \Pi(Y) \wedge (t \vdash IN \notin T \Vdash IN)))] \quad (11)$$

The concepts discussed until now can be assembled to a component specification, that is defined by the predicate *comp_spec*:

$$comp_spec(T, A, Y, IN, OUT) \Leftrightarrow table(T, A) \wedge type(Y, T) \wedge (IN \subset A) \wedge (OUT \subset A) \wedge complete(T, Y, IN) \quad (12)$$

Notice, that this specification of a component can model deterministic behaviour (a function) as well as non-deterministic behaviour (a relation). To distinguish between both, we introduce the concept of *functional dependency* [1]. A functional dependency exists if the contents of some pair of column subsets can be considered as a function. Let A, B and C be sets and T be a table on A and $B \subseteq A$ and $C \subseteq A$ and let the setfunction Y be the type of T, then a functional dependency M is a function that maps a restriction to the subset B of each function $t \in T$ on a restriction to the subset C of the same function t. Hence, the function M is a set of tuples $<t \vdash B, t \vdash C>$, where $t \in T$. The set B is called the *dependency-domain* of M and the set C is called the *dependency-image* of M.

$$fun_dep(M, B, C, T, Y) \Leftrightarrow ((T \Vdash B) = (\Pi(Y) \Vdash B)) \wedge (\forall t,t')[((t \in T) \wedge (t' \in T) \wedge (<t \vdash B, t \vdash C> \in M) \wedge (<t' \vdash B, t' \vdash C> \in M) \wedge (t \vdash B = t' \vdash B)) \Rightarrow (t \vdash C = t' \vdash C)] \quad (13)$$

Let M be a functional dependency defined by $fun_dep(M, B, C, T, Y)$, then $fun_dep(M, B, C, T, Y) \Rightarrow function(M, \Pi(Y) \Vdash B, T \Vdash C)$. Notice that in general, the result table of a completion operation contains no functional dependencies. For the properties of functional dependencies we refer to de Brock [2].

Let T and L be tables with the same cardinality on respectively attribute sets A and B, with $A \cap B = \varnothing$, then the *extension* operation on T and L, denoted by $T \bowtie L$, is a table on $A \cup B$ such that each function in T is unified with one and only one function in L. This operation makes it possible to 'zip' tables with equal numbers of rows. The next section considers sets of component specifications.

3.1 Extensions to relational algebra

The previous sections considered only the specification of behaviour for a single component. This section considers the behaviour specified by sets of component specifications, called *structures*. Such a set can be obtained through the concept of *decomposition*, that can be implemented by the operations defined in section 3. The relation with the environment of a structure is expressed also in terms of components. These are respectively for the inputs

and outputs: *comp_spec*($\Pi(Y)$, A, Y, $\varnothing$, A) and *comp_spec*($\Pi(Y)$, A, Y, A, $\varnothing$). The table of these components is the general product of their type function by which they have no impact on the behaviour of a structure. Their contribution is essentially needed for network representations of a structure. A structure represents a set of *comp_spec* objects by means of a set of functions defined on a common domain that consists of identifiers. This simplifies the identification of the parts of the component specifications contained in the structure.

Let D be a set and v, g, t, *in*,*out* be functions on domain D, then a structure v is a function that maps each identifier in D on a table, g is a function that maps each element of D on a set of attributes, t maps each element of D on a type function and the *in* and *out* functions map each element of D on respectively the *IN* and *OUT* attribute subsets. Formally;

$$\begin{aligned} & structure(v,g,t,D,in,out) \Leftrightarrow set(D) \wedge function(v,D) \\ & \quad \wedge setfunction(in,D) \wedge setfunction(out,D) \\ & \quad \wedge setfunction(g,D) \wedge setfunction(t,D) \wedge \\ & \quad (\forall e)[e \in D \Rightarrow \\ & \qquad comp_spec(v(e), g(e), t(e), in(e), out(e))] \\ & \qquad \wedge (\exists f)[f \in D \wedge (v(f)=\Pi(t(f))) \wedge (g(f)=out(f)) \\ & \qquad \wedge (in(f)=\varnothing)] \wedge \\ & \qquad (\exists h)[h \in D \wedge (v(h)=\Pi(t(h))) \wedge (g(h)=in(h)) \\ & \qquad \wedge (out(h)=\varnothing)] \end{aligned} \tag{14}$$

Notice that this definition leaves it free to have one input *comp_spec* for all input variables, or one for each variable. Hence, we need two functions *IN_comp* and *OUT_comp* to determine input- and output component specifications. Let v be a structure described by *structure*(v,g,t,D,in,out), then:

$$\begin{aligned} & (\forall e)[e \in IN_comp(v) \Leftrightarrow e \in D \wedge (v(e)=\Pi(t(e))) \wedge \\ & \qquad (g(e)=out(e)) \wedge (in(e)=\varnothing)] \end{aligned} \tag{15}$$

$$\begin{aligned} & (\forall e)[e \in OUT_comp(v) \Leftrightarrow e \in D \wedge (v(e)=\Pi(t(e))) \wedge \\ & \qquad (g(e)=in(e)) \wedge (out(e)=\varnothing)] \end{aligned} \tag{16}$$

Furthermore, to retrieve the variables and attribute sets involved with the input and output *comp_spec* objects we define two additional functions:

$$(\forall x)[x \in input_attr(v) \Leftrightarrow x \in \bigcup_{e \in IN_comp(v)} g(e)] \tag{17}$$

$$(\forall x)[x \in output_attr(v) \Leftrightarrow x \in \bigcup_{e \in OUT_comp(v)} g(e)] \tag{18}$$

As example of a structure we show the structure *vAO21* that describes the tables in figure 1 of function AO21.

structure(*vAO21*, *g*, *t*, *D*, *in*, *out*), where:
D={AND,OR, IN_env, OUT_env}
vAO21={<AND, *T_AND*>, <OR, *T_OR*>,
<IN_env,Π({<*i1*,{0,1}>, <*i2*,{0,1}>,<*i3*,{0,1}>}) >,
<OUT_env,Π({<*o*,{0,1}>})> }
g={<AND, {*i1*, *i2*, *m*}>, <OR, {*m*, *i3*, *o* }>,
<IN_env,{*i1*,*i2*,*i3*}>,<OUT_env,{*o*}>}
t={<AND, {<*i1*,{0,1}>, <*i2*,{0,1}>, <*m*,{0,1}>}>,
<OR, {<*m*,{0,1}>, <*i3*,{0,1}>, <*o*,{0,1}>} >
<IN_env,{<*i1*,{0,1}>, <*i2*,{0,1}>, <*i3*,{0,1}>}>,
<OUT_env,{<*o*,{0,1}>}> }
in={<AND,{*i1*, *i2*}>, <OR,{*m*, *i3*}>,
<IN_env,{∅}>,<OUT_env,{*i1*,*i2*,*i3*}>}
out={<AND,{*m*}>, <OR,{*o*}>,
<IN_env,{*i1*,*i2*,*i3*}>,<OUT_env,{∅}>}

The behaviour of a structure is defined as the table that results from the natural join [1] over all tables found in the structure. Let D be a set and v be a structure on D, and B be a table, then B is called the *behaviour* of v iff B is the natural join of all tables in v. Formally;

$$behaviour(B, v) \Leftrightarrow \quad B = \bowtie_{e \in D} v(e) \tag{19}$$

Let T_{parent} be a table on A, then a pair of tables $\{T_{son1}, T_{son2}\}$ can be considered as a decomposition of T_{parent} if: $T_{parent}=(T_{son1} \bowtie T_{son2}) \upharpoonright A$. This can be rewritten to; $T_{parent}= B_{sons} \upharpoonright A$, where $B_{sons}= T_{son1} \bowtie T_{son2}$ and denotes the behaviour of both son tables. Hence, the pair of tables $\{T_{son1}, T_{son2}\}$ can be considered as part of a structure, without its input and output component specifications.

4. SEQUENTIAL BEHAVIOUR

This section introduces the concept of time in the description of the behaviour of components and structures. We presume the existence of an infinitely running clock. Time is then modelled by clockcycle indices. A signal is then a variable with time behaviour. In tables, we replace the attributes of a

table that were variable identifiers, by tuples that contain a signal identifier and a clockcycle index and denote them instead of the tuple <a, c> by a_c. Causal behaviour requires that the tuples of the *IN* attribute subset of a table have lower clock-cycle indices than the tuples of the *OUT* attribute subset.

We distinguish two types of behavioural relations between signals in hardware. Let v be a structure, then its behaviour is said to be *combinatorial* iff all clockcycle indices in the attribute tuples are identical and the behaviour is said to be *sequential* iff there exists at least one tuple element of the output attribute subset that has a higher clockcycle index than the clockcycle indices of the input attributes. Formally:

$$\mathit{sequential}(v) \Leftrightarrow (\exists a,c,a',c')[(\langle a,c\rangle \in \mathit{input_attr}(v)) \wedge (\langle a',c'\rangle \in \mathit{output_attr}(v)) \wedge (c'>c)]]] \quad (20)$$

Let v be a structure, then the set of state signals is the subset of identifiers occurring in tuples of both the *input* and *output* attribute sets and which tuples have different clockcycle indices as second element. A structure v is said to be a finite state machine if its set of state signals is not empty.

$$(\forall z)[\, z \in \mathit{state_signals}(v) \Leftrightarrow (\exists c,c')[(\langle z,c\rangle \in \mathit{input_attr}(v)) \wedge (\langle z,c'\rangle \in \mathit{output_attr}(v)) \wedge (c'>c)]]] \quad (21)$$

$$(\forall x)[\, x \in \mathit{state_attr}(v) \Leftrightarrow (\exists z,c)[(x=\langle z,c\rangle) \wedge (z \in \mathit{state_signals}(v))]] \quad (22)$$

$$\mathit{fsm}(v) \Leftrightarrow (\exists z)[z \in \mathit{state_signals}(v)] \quad (23)$$

Notice, the difference with a sequential machine, where it is not necessary that a state_signal occurs

5 FINITE STATE MACHINES

Our finite state machine definition is general in the sense that it allows non-determinism. In this section we introduce some restrictions to distinguish between *Mealy* and *Moore* behaviour. Generally, two behaviours are distinguished in a finite state machine, these are the *transition-behaviour* and the *output-behaviour*. The transition behaviour produces the next state symbol from the current state- and input symbols, while the output behaviour produces the output symbol from the input- and current state symbols. Let v be a structure and a finite state machine and let B be its behaviour, then the transition behaviour is the behaviour B of v projected onto the input attribute subset unified with the subset of the state-attributes that are part of the output attributes. The output behaviour is then B projected on the input attributes unified with those output attributes that do not belong to the state attributes. So formally;

$$\mathit{trans_behaviour}(TF, v) \Leftrightarrow (\exists B)[\mathit{behaviour}(B,v) \wedge TF=B \restriction (\mathit{input_attr}(v) \cup (\mathit{output_attr}(v) \cap \mathit{state_attr}(v))] \quad (24)$$

$$\mathit{output_behaviour}(OF, v) \Leftrightarrow (\exists B)[\mathit{behaviour}(B,v) \wedge OF=B \restriction (\mathit{input_attr}(v) \cup (\mathit{output_attr}(v) - \mathit{state_attr}(v))] \quad (25)$$

Non-determinism in the transition behaviour causes no problems if it is certain that those transitions are not used by any input sequence. However, we presume 'random' like input symbol sequences and thus need determinism. Let v be a structure and a finite state machine, then the finite state machine v is said to be deterministic iff there exists a functional dependency between the input attributes of v and the state attributes that are part of the output attributes;

$$\mathit{determ_fsm}(v) \Leftrightarrow \mathit{fsm}(v) \wedge (\exists TF, M, Y)[\, \mathit{trans_behaviour}(TF, v) \wedge \mathit{fun_dep}(M, \mathit{input_attr}(v), \mathit{output_attr}(v) \cap \mathit{state_attr}(v), TF, Y)] \quad (26)$$

For the fsm output behaviour, usually the so-called *Mealy* and *Moore* behaviours are distinguished. In the Moore machine the output symbol is computed from the state symbol only, while in the Mealy machine the output symbol is computed from both the input symbol and state symbol.

$$\mathit{Mealy_fsm}(v) \Leftrightarrow \mathit{fsm}(v) \wedge (\exists OF, M, Y)[\, \mathit{output_behaviour}(OF, v) \wedge \mathit{fun_dep}(M, \mathit{input_attr}(v), \mathit{output_attr}(v) - \mathit{state_attr}(v), OF, Y)] \quad (27)$$

$$Moore_fsm(v) \Leftrightarrow fsm(v) \wedge (\exists OF, M, Y)[\ output_behaviour(OF, v) \wedge fun_dep(M, state_attr(v) - output_attr(v), output_attr(v) - state_attr(v), OF, Y)] \quad (28)$$

Examples of both types of finite state machines are shown in figure 3. Both machines have equivalent behaviour, that is, they produce the same sequence of output symbols on some sequence of input symbols.

i_c	trs_c	trs_{c+1}	o_c
i1	trs1	trs1	o1
i2	trs1	trs2	o1
i1	trs2	trs1	o2
i2	trs2	trs2	o2

i_c	s_c	s_{c+1}	o_c
i1	s1	s1	o1
i2	s1	s1	o2

Figure 3 Examples of respectively a Moore and a Mealy fsm.

5.1 Mealy to Moore transformation

We develop this transformation for component specifications, that can also be viewed as behaviours of structures. Let *TMealy* be a component specification with table *TMealy* on *AMealy* = $\{i_c, s_c, s_{c+1}, o_c\}$ and type *YMealy*=$\{<i_c,I>, <s_c,S>, <s_{c+1},S>, <o_c,O>\}$ and let *TMoore* be defined on attribute set $\{i_c, trs_c, trs_{c+1}, o_c\}$ with type *YMoore*=$\{<i_c,I>, <trs_c,TRS>, <trs_{c+1},TRS>, <o_c,O>\}$. Furthermore we assume that both *TMealy* and *TMoore* are completely deterministic.

Then, the Mealy$^+$ machine is equal to the Mealy machine, except that it produces its output attribute in the next clockcycle. Hence, the Mealy$^+$ machine can be retrieved from the Mealy machine by a simple rename operation with the function called *add_latch_out*. It is produced by the rename operation with the function:

$$add_latch_out = \{<i_c,i_c>, <s_c,s_c>, <s_{c+1},s_{c+1}>, <o_{c+1},o_c>\}$$

Thus $TMealy+ = TMealy \infty add_latch_out$ implies that $table(TMealy^+, \{i_c, s_c, s_{c+1}, o_{c+1}\})$. The functional dependency networks of both machines are shown in figure 4.

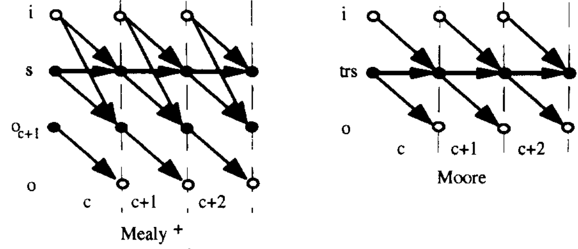


Figure 4 Mealy$^+$ and Moore FSM.

The basic idea of the transformation between Mealy$^+$ and Moore is to consider the $\{s_{c+1}, o_{c+1}\}$ attribute pair as next state information, called trs_{c+1} and the $\{s_c, o_c\}$ attribute pair as state information, called trs_c. With the attributes of TMealy$^+$ not all information at the present clockcycle c is available, because this machine produces only output at clockcycle c+1. Hence, we consider a shift back over one clockcycle of the TMealy$^+$ machine by a rename operation with the function shift^{-1}.

$$shift^{-1} = \{<i_{c-1},i_c>, <s_{c-1},s_c>, <s_c,s_{c+1}>, <o_c,o_{c+1}>\}$$

Then, ($TMealy^{+-1} = TMealy^+ \infty shift^{-1}$) implies that: $table(TMealy^{+-1}, \{i_{c-1}, s_{c-1}, s_c, o_c\})$ The output information in clockcycle c is then added by a natural join of TMealy$^+$ with TMealy^{+-1} to table *T_FS*:

i_{c-1}	s_{c-1}	s_c	o_c	i_c	s_{c+1}	o_{c+1}
i1	s1	s1	o1	i1	s1	o1
i2	s1	s1	o2	i1	s1	o1
i1	s1	s1	o1	i2	s1	o2
i2	s1	s1	o2	i2	s1	o2

$T_FS = TMealy^+ \bowtie TMealy^{+-1}$

This natural-join joins both tables on the s_c attribute. The state set and the next state set are then retrieved by the following projections of *T_FS* :

$$T_states^+ = T_FS \upharpoonright \{s_{c+1}, o_{c+1}\}.$$
$$T_states = T_FS \upharpoonright \{s_c, o_c\}$$

The contents of tables *T_states*$^+$ and *T_states* are identical, only both attribute sets differ. The basic transformation between the *T_FS* and the Moore machine is then done by identifying or labelling all states and next-states in the *T_FS* machine by a symbol set *TRS*. This can be done by extension of

the *T_states*$^+$ and *T_states* tables. Presume two tables *T_TR* and *T_TR*$^+$: $table(T_TR^{+1}, \{trs_{c+1}\})$, $table(T_TR, \{trs_c\})$, where T_TR^{+1} contains a symbol for each pair of symbols in the $\{s_{c+1}, o_{c+1}\}$ columns and *T_TR* is its time shifted version.

$$T_OUTD^+ = T_states^+ \bowtie T_TR^+$$
$$T_OUTD = T_states \bowtie T_TR$$

Hence, T_OUTD^+ is a table on $\{trs_{c+1}, s_{c+1}, o_{c+1}\}$. For our example *Mealy*$^+$ fsm, both tables become:

trs_{c+1}	s_{c+1}	o_{c+1}
trs1	s1	o1
trs2	s1	o2

T_OUTD^+

trs_c	s_c	o_c
trs1	s1	o1
trs2	s1	o2

T_OUTD

The following two natural-joins insert the trs_c and trs_{c+1} attributes labelling all states and next states in the *T_FS* table:

$$T_FSM = T_FS \bowtie T_OUTD^+ \bowtie T_OUTD$$

i_{c-1}	s_{c-1}	s_c	o_c	trs_c	i_c	s_{c+1}	o_{c+1}	trs_{c+1}
i1	s1	s1	o1	trs1	i1	s1	o1	trs1
i2	s1	s1	o2	trs2	i1	s1	o1	trs1
i1	s1	s1	o1	trs1	i2	s1	o2	trs2
i2	s1	s1	o2	trs2	i2	s1	o2	trs2

Then, from the table *T_FSM* both tables *TMealy*$^+$ and *TMoore* can be retrieved by projection.

$$TMealy^+ = T_FSM \upharpoonleft \{i_c, s_c, s_{c+1}, o_{c+1}\}$$
$$TMoore = T_FSM \upharpoonleft \{i_c, trs_c, trs_{c+1}, o_c\}$$

The projection of the table *TMealy*$^+$ is trivial, because *T_FSM* was build from it. The projection on $\{i_c, trs_c, trs_{c+1}, o_c\}$ contains all information of *TMealy*$^+$, because of the definition of the columns of the trs_c and trs_{c+1} attributes. Then summarizing, the transformation between the Mealy FSM and the Moore FSM is:

$$T_Moore = ((T_Mealy^+ \bowtie T_Mealy^{+-1}) \bowtie T_OUTD^+ \bowtie T_OUTD) \upharpoonleft \{i_c, trs_c, trs_{c+1}, o_c\}$$

The number of operations is relatively small, the real complexity however hides inside these operations.

6. CONCLUSIONS

We presented a formalism for the formal description of hardware designs. This formalism is both simple and powerful. As an important result, design transformations can be described in relational algebra, for example the traditionally algorithmically described Mealy to Moore machine transformation is described in only a few operations. A further advantage is that this formalism covers also the common truth-table descriptions, handles don't care values, and reflects closely design styles that make use of programmable logic as PLA's. Further research in this field is dedicated to the transformational design style for which relational algebra is likely to be a good backbone.

REFERENCES

[1] Ullman, J.D., Principles of Database Design and Knowledgebase Systems, Computer Science Press, Rockville, 1989.

[2] de Brock, E.O., Database models and retrieval languages, PhD. Thesis, Technical University Eindhoven, the Netherlands, 1984.

[3] Hennie, F.C., Finite State Models for Logical Machines, Wiley, New-York, 1968.

[4] ten Berg, A.J.W.M., Flexible Controlpath Microarchitecture Synthesis based on Artificial Intelligence, Proceedings Euro-DAC'92 conference, Hamburg, Germany, IEEE, New-York, 1992, pp. 112-117.

[5] Boute, R.T., System Semantics and Formal Circuit Description, IEEE Transactions on circuits and systems, cas-33, december 1986.

[6] Jones, G. and Sheeran, M., Circuit design in Ruby, in: Formal Methods for VLSI design, IFIP WG 10.5 lecture Notes, edited by J. Staunstrup, North Holland, 1990.

[7] Gordon, M., Why higher-order logic is a good formalism for specifying and verifying hardware, in: Formal Aspects of VLSI Design, eds. G. Milne and P.A. Subrahmanyam, Elsevier Science Publ., 1986, pp. 153-177.

[8] IEEE Standard VHDL language reference manual (IEEE std 1076-1987), IEEE, New York, 1988.