TECHNICAL REPORT NO. 317

An Algebraic Framework for Data Abstraction in Hardware Description

by

Zheng Zhu and Steven D. Johnson

November 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

An Algebraic Framework for Data Abstraction in Hardware Description[1]

Zheng Zhu, Steven D. Johnson
Computer Science Department
Indiana University
Bloomington, Indiana USA

Abstract

The aim of this work is to extend a standard treatment of data types to a foundation for hardware synthesis. Hardware synthesis exposes several problems not typically considered in software oriented theories. These include architectural constraints in the use of type instances, parallelism in the use of multiple instances, and consolidation of distinct types in a common process. Since we are concerned with the question of incorporating (more) concrete implementations of (more) abstract types found in higher levels of specification, our foundation must address these aspects of description. The paper begins with a condensed example of a stack based processor operating on a standard memory. This specification is compared to a target description in which both the stack and the memory are implemented by a single memory process. The remainder of the paper formalizes issues raised in the example.

1 Introduction

If one tries to apply standard treatments of data abstraction to hardware design, a number of formal problems are exposed. For example, there is a need to describe architecture in a type formalization. There is a need to reformulate notions of implementation in the context of parallel control. There is a need to reflect a process oriented view of behavior. While these issues are obviously present in software, one can go a good distance without confronting them directly. However, they are the essence of many hardware design problems.

Our approach to formalizing design is based on function algebra. An abstract behavioral description is transformed to an architecture description by the function algebra [8, 9]. An implementation of this algebra has been used successfully to derived several working designs [2, 11]. However, this algebra manipulates descriptions at a fixed level of data abstraction. Consequently, specifications must be given in more detail than we would like.

We now turn to the question of translating specifications expressed over a higher level of data representation to implementations expressed at a lower level. Our goal is to lay a foundation for a general treatment of data abstraction in hardware description. We develop extensions to the algebraic characterization of data types which permit us to specify architectures. We give a corresponding characterization of control as recursive functions on the resulting structures. In this framework, we consider the question of implementing relatively abstract architectures by relatively concrete ones.

---

[1] Paper to appear in the *Proceedings of Workshop of Designing Correct Circuits*, September, 1990. Published by Verlag-Springer

The algebra of translating specifications expressed over a higher level of data abstraction to implementations expressed at a lower level has much in common with process formalisms, such as those of Milne [16], and Gopalkrishnan [6]. A difference in emphasis is that this work is directed toward *derivation* (i.e. synthesis in an algebraic framework) rather than *verification*. In addition, we focus on the complex interaction between *functional* and *physical* decomposition under the thesis that a logical hierarchy does not always dictate the physical modularity of a design.

In high-level synthesis, methods exist to explore the balance of architecture and control—the problems of *scheduling, allocation, module matching,* and *control synthesis* [14]. However, these methods are typically restricted to some fixed ground type, and they are often tied to specific architecture classes. Our motive is to develop a treatment for these kinds of problems at arbitrary levels of data abstraction. Although the most urgent practical needs remain at lowest levels of representation, such a foundation is required if synthesis is to address system designs.

Mahmood, Mavaddat, Elmasry and Cheng propose using a DT0L language model to specify hardware architectures [13]. The data paths of an architecture are defined by a set of homomorphic functions and the control of hardware is defined by a sequence of applications of those functions. An algorithm for synthesizing control (microcodes) is presented. Although our framework is similar to theirs, there exist two basic differences between the two: First, our framework is designed for derivation of architecture as well as synthesis of hardware control. It addresses issues such as the incorporation of correct data representation. Second, the foundation described takes into account the underlying algebraic properties of the abstract data type structures involved. However, this degree of generality also leads to undecidability of problems.

In Section 2, we present a condensed example to motivate the formal development. The example illustrates issues that we have been forced to deal with in previous design exercises. These issues are neither "solved" by the formal development nor currently handled by our mechanized subset of it. However, we believe that they can be characterized in the framework developed here. Sections 3 reviews fundamentals of the theory of abstract data types. In Sections 4 and 5, we extend the standard definitions to account for hardware architecture. In Section 6, we consider the central question of implementing one extended abstract type by another. Section 7 discusses the issue of modeling hardware control and incorporating architecture implementation into hardware control.

We develop three results in this paper. Theorem 1 (Section 3) states a condition under which one equational theory is said to implement another. This theorem characterizes implementation in terms of equational rewriting which implies that the validity of implementation can be established with the help of a computer. Theorem 2 (Section 5.1) states conditions under which the extended notion of data type preserves the implementation relationship. Finally, Theorem 3 (Section 7) incorporates the functional representation of control into the implementation relationship.

2 Motivation

Consider the simple machine specification shown in Figure 1 using a PureLisp-like notation. The machine has arithmetic instructions, such as *add* and *sub*, a load-constant instruction *ldc*, and a branch instruction *goto*. The machine state has an instruction memory (*M-i*) and an operand stack (S). Each arithmetic operation takes two operands from the stack and returns its result to the stack. The *ldc* operation copies a constant from the instruction memory to the stack; and *goto* sets the memory address register (*a-i*) to the content of the next instruction cell. The standard abstract memory is used, with *wr* (write), *rd* (read) operations. The *dcr*, *inc* operations respectively decrement and increment memory addresses. A standard abstract stack is used with *push*, *pop*, and *top* operations.

One task in implementing this specification is to develop data paths and a finite-state control for it. The intermediate description in Figure 2 associates a hardware control-state with each function definition. Details of algebra to obtain this description from that in Figure 1 are omitted [2]. In this description, each state is constrained to operate at most once on the memory and at most once on the stack; binary arithmetic operations are encapsulated by an arithmetic unit (arith); and three registers, *op1*, *op2* and *I*, are introduced to hold intermediate results.

The target description in Figure 3 further constrains the design in two ways. The stack abstraction S is implemented by a pair consisting of a memory and an address register, (*M-s*,*a-s*); and two memory objects, *M-i* and *M-s*, are merged into a single memory object M. The resulting architecture operates at most once on M in any control state.

Ultimately, it is this merging of distinct functional entities "stack" and "memory" into a single object that interests us. However, in order to reach this problem, we must also address the following issues:

1. What constitutes a correct implementation of hardware components? In the example, we need to know that how the memory *can* be used to implement the stack.

2. How are *correct* implementations *correctly* incorporated in hardware descriptions. The transformations that do this must satisfy not only the functional properties of the implementation, but also the architecture constraints surrounding its use.

We regard (1) as an issue for verification and (2) as an issue for synthesis. Although the correctness of implementations is not the immediate subject of this research, it is necessary to adopt a criterion of correctness before accounting for architectures. Thus, the immediate concern of this paper is a characterization of what one does *with* a correct implementation; that is, how specifications are transformed to incorporate them.

3 Term Algebra and Equational Specification

In this section, we briefly review the concepts of equational specification and implementation in universal algebra. There are mainly two schools in the *algebraic semantics* of abstract data types, *i.e.* *initial* semantics (*e.g.* [5], [4], [15]) and *final* semantics (*e.g* [19], [12]). (See also [1] for a comparative introduction to both approaches.) In our presentation, we attempt to avoid an association with any particular approach to semantics by keeping the framework on syntactic level as much as possible although this preliminary work is influenced by the ADJ group's results [5, 15]. We start with the concepts of Σ-algebra and its special form, Σ term algebra, then the concept of equational specification. At the end, we introduce a definition of implementation for equational specifications and a characterization theorem

```
(define machine (M-s a-s S)
     (case (rd M-s a-s)
        (’add’  (machine M-s (dcr a-s) (push (+ (top S) (top (pop S))) (pop (pop S)))))
        (’sub’  (machine M-s (dcr a-s) (push (- (top S) (top (pop S))) (pop (pop S)))))
        (’mul’  (machine M-s (dcr a-s) (push (* (top S) (top (pop S))) (pop (pop S)))))
        (’div’  (machine M-s (dcr a-s) (push (/ (top S) (top (pop S))) (pop (pop S)))))
        (’eq?’  (machine M-s (dcr a-s) (push (= (top S) (top (pop S))) (pop (pop S)))))
        (’le?’  (machine M-s (dcr a-s) (push (< (top S) (top (pop S))) (pop (pop S)))))
        (’goto’ (machine M-s (rd (M-s (dcr a-s))) S))
        (’ldc’  (machine M-s (dcr (dcr a-s)) (push (rd M-s (dcr a-s)) S)))
```

Figure 1: **Specification of the Machine**

```
(define machine (M-s a-s S op1 op2 I)
     (case I
        (’goto’ (goto     M-s (rd M-s a-s) S op1 op2 I))
        (’ldc’  (ldc1     M-s (dcr a-s) S (rd M-s a-s) op2 I))
        (else   (arithmetic1  M-s a-s (pop S) (top S) op2 I))))

(define goto (M-s a-s S op1 op2 I)
     (machine M-s (dcr a-s) S op1 op2 (rd M-s a-s)))

(define ldc1 (M-s a-s S op1 op2 I)
     (machine M-s (dcr a-s) (push S op1) op1 op2 (rd M-s a-s)))

(define arithmetic1 (M-s a-s S op1 op2 I)
     (arithmetic2 M-s a-s (pop S) op1 (top S) I)))

(define arithmetic2 (M-s a-s S op1 op2 I)
     (machine  M-s (dcr a-s) (push S (arith I op1 op2)) op1 op2 (rd M-s a-s)))
```

Figure 2: **Modified Specification of the Machine**

```
(define machine (M a-s a-i op1 op2 I)
     (case I
        (’goto’ (goto    M (rd M a-s) a-i op1 op2 I))
        (’ldc’  (ldc1    M (dcr2 a-s) a-i (rd M a-s) op2 I))
        (else   (arithmetic1  M a-s (dcr2 a-i) (rd M-s a-i) op2 I))))

(define goto (M a-s a-i op1 op2 I)
     (machine M (dcr2 a-s) a-i op1 op2 (rd M a-s)))

(define ldc1 (M a-s a-i op1 op2 I)
     (ldc2 (wr M (inc2 a-i) op1) a-s (inc2 a-i) op1 op2 I)

(define ldc2 (M a-s a-i op1 op2 I)
     (machine M (dcr2 a-s) a-i op1 op2 (rd M a-s)))

(define arithmetic1 (M a-s a-i op1 op2 I)
     (arithmetic2 M a-s (dcr2 a-i) op1 (rd M a-i) I))

(define arithmetic2 (M a-s a-i op1 op2 I)
     (arithmetic3 (wr M a-i (arith I op1 op2)) a-s (inc2 a-i) op1 op2 I))

(define arithmetic2 (M a-s a-i op1 op2 I)
     (machine M (dcr2 a_s) a_i op1 op2 (rd M a-s)))
```

Figure 3: **Implementation of Machine With One Memory**

for implementation.

Let S be a set of sorts. An S-sorted *signature* Σ is a set $\bigcup_{w \in S^*, s \in S} \Sigma_{w,s}$. Every element σ of set $\Sigma_{w,s}$ is a function symbol of *arity* w and of sort s. The arity of a function symbol expresses what sorts of data it expects to take as its inputs and in what order. The sort of a function symbol expresses the sort of data it returns. Constant symbols are considered as function symbols of *empty* arity. An S-sorted set of variables is $X = \bigcup_{s \in S} X_s$. Each element of the set X_s is called a variable of sort s. Given Σ, the set of terms generated from Σ and X, denoted by $T_\Sigma(X)$ is inductively defined as:

1. If $x_s \in X_s$ for some $s \in S$ then $x_s \in T_{\Sigma_s}(X)$.
2. If $c \in \Sigma_{\lambda,s}$ for some $s \in S$ then $c \in T_{\Sigma_s}(X)$.
3. If $f \in \Sigma_{w,s}$ where $w = s_1 s_2 \ldots s_n$, and $t_i \in T_{\Sigma_{s_i}}(X)$ $1 \leq i \leq n$, then $f(t_1, t_2, \ldots, t_n) \in T_{\Sigma_s}(X)$.
4. $T_\Sigma(X) = \bigcup_{s \in S} T_{\Sigma_s}(X)$.

Traditionally, T_Σ is used to denote $T_\Sigma(\phi)$. Let Y be an S-sorted set of variables which is disjoint from X. A Y-equation of $T_\Sigma(X)$ is a tuple (l, r) where $l, r \in T_\Sigma(X \cup Y)$. We usually write (l, r) as "$l \equiv r$". The set Y is called a set of meta-specification variables. Let E be a set of Y-equations of $T_\Sigma(X)$, then E induces an equivalence relation on the set $T_\Sigma(X)$; hence we can define $T_\Sigma(X)/E$ as the set of equivalence classes of $T_\Sigma(X)$ under E.

Let Σ be an S-sorted signature. A Σ-*algebra* A consists of a family of carriers $< A_s \mid s \in S >$ together with a set of functions $F = \{f_\sigma \mid \sigma \in \Sigma\}$ such that $f_\sigma : A_{s_1} \times A_{s_2} \times \ldots A_{s_n} \rightarrow A_s$ for every $\sigma \in \Sigma_{w,s}$ where $w = s_1 s_2 \ldots s_n$.

A special type of Σ-algebra, which is also our major interest, is $T(\Sigma, X) = \langle T_\Sigma(X), \Sigma \rangle$, called *the term algebra generated from* Σ. If E is a set Y-equations of $T_\Sigma(X)$, then E is a congruence relation on $T(\Sigma, X)$. We use $T(\Sigma, X, E)$ to denote $T(\Sigma, X)$ modulo E. It is important to realize that though X and Y are sets of variables, they are different in a sense that X is a set of *program variables* and Y is a set of *assertion variables*. A detailed discussion of the difference can be found in [18].

An equational specification is $\langle \Sigma, X, E \rangle$. It has been shown (*e.g.* [4, 5]) that equational specifications can be used to specify abstract data types and the algebra $T(\Sigma, X, E)$ can be used as the meaning of the specification. We also show later how an extension of equational specification defines an abstract hardware architecture syntactically.

The following two definitions define the notion of implementing one equational specification by another. These definitions are slight variations of those in [5]. The difference is that they are not based on the initial algebra semantics. The motive for this treatment becomes clear in section 5 where we define architectural implementation independently of semantics.

In the definitions, $T^*_\Sigma(X)$ designates, informally, a set of term-vectors of finite length, and λT designates the set of combinators (*i.e.* closed lambda expressions) involving terms in T.

Definition 1. Let Σ and Ω be S_1-sorted and S_2-sorted operator domains respectively. A *derivor* $d: \Sigma \rightarrow \Omega$ is a pair of functions $\langle \xi, \kappa \rangle$ where $\xi : S_1 \rightarrow S^*_2$ and $\kappa : \Sigma \rightarrow \lambda(T^*_\Omega(X_2))$, such that for every $\sigma \in \Sigma_{w,s}$, $\kappa(\sigma)$ is a $\xi(s)$-sorted function symbol of arity $\xi(w)$. The d-derived algebra from $T(\Sigma, X_1, E_1)$, $dT(\Omega, X_2, E_2)$, is a Σ-algebra defined as $\langle T_\Omega(X_2)/E_2, d(\Sigma) \rangle$.

Definition 2. Let $P = \langle \Sigma, E_1 \rangle$ be a specification. An *implementation* of P is a triple $<B, d, E>$, where $B = \langle \Omega, E_2 \rangle$ is another equational specification, d is derivor from Σ to Ω, and E is a congruence relation of $T(\Omega, E_2)$, such that $T(\Sigma, E_1)$ is a subalgebra of $(dB)/E$.

Definition 3. Let E_1, E_2 and E_3 be definite sets of equations.

1. $E_1 \vdash E_2$ means that for every $(l, r) \in E_2$, there exist $(l_1, r_1), \ldots, (l_n, r_n) \in E_1$ such that $l_1 \equiv l$, $r_n \equiv r$ and for every $i = 1, \ldots, n-1$, $r_i \equiv l_{i+1}$;
2. $E_1/E_2 \vdash E_3$ means that for every $(l, r) \in E_3$, there exists t such that $E_1 \vdash (l, t)$ and $E_2 \vdash (t, r)$;
3. Let d be a derivor $\Sigma \to \Omega$ and E be a set of equations on T_Σ. Define $d(E) = \{\ (d(l), d(r)) \mid (l, r) \in E\ \}$.

Theorem 1. Let $\langle \Sigma, E_1 \rangle$ and $\langle \Omega, E_2 \rangle$ be two equational specifications. Let $d : \Sigma \to \Omega$ be a 1-1 derivor and E be a set of equations of T_Ω. If $E_2/E \vdash d(E_1)$ then $\langle T_\Omega, d, E \rangle$ is an implementation of T_Σ; that is, $\langle T(\Sigma, E_1), \Sigma \rangle$ is a subalgebra of $\langle T(\Omega, E_2), d(\Sigma) \rangle / E$.

One thing to notice is that $d(E_1) \vdash E_2$ and $d(E_1) \vdash E$ if E can be written as a finite set of equations. The significance of the theorem is that it characterizes implementation in terms of an equational rewriting mechanism.

We simplify the proof by considering the case when $\xi: S_1 \to S_2$. In that case, κ becomes $\Sigma \to \lambda(T_\Omega(X_2))$. The proof for $T^*_\Omega(X_s)$ is a straightforward generalization.

Proof. d is obviously a mapping $T_\Sigma \to T_\Omega$ such that $d(\sigma(t_1, ..., t_n)) = d(\sigma)(d(t_1), ..., d(t_n))$. Since E is a congruence relation on T_{Ω, E_2}, we can assume h is the homomorphism

$$h: T(\Omega, E_2)/E \to T(\Omega, E_2)$$

We prove that $h \circ d$ is a 1-1 homomorphism $T(\Sigma, E_1) \to T(\Omega, E_2)/E$, *i.e.* $T(\Sigma, E_1)$ is a subalgebra of $T(\Omega, E_2)/E$.

1. For every $t_1, t_2 \in T(\Sigma)$, $t_1 \equiv_{E_1} t_2$ implies that $d(t_1) \equiv_{d(E_1)} d(t_2)$, which in turn implies that $d(t_1) \equiv_{E_2/E} d(t_2)$ because $E_2/E \vdash d(E_1)$. That h is a homomorphism defines E, therefore $t_1 \equiv_{E_1} t_2$ implies $h \circ d(t_1) \equiv_{E_2} h \circ d(t_2)$ which means that $h \circ d$ is a well defined.
2. By the definition of d, $d(\sigma(t_1, ..., t_n)) = d(\sigma)(d(t_1), ..., d(t_n))$. Since h is an homomorphism, $h \circ d(\sigma(t_1, ..., t_n)) = h \circ d(\sigma)(h \circ d(t_1), ..., h \circ d(t_n))$.
3. Let $t_1, t_2 \in T_\Omega$ and assume that $h \circ d(t_1) \equiv_{E_2} h \circ d(t_2)$. Then $h \circ d(t_1) \equiv_{E_2/E} h \circ d(t_2)$ which implies $d(t_1) \equiv_{d(E_1)} d(t_2)$. Since d is 1-1, $t_1 \equiv_{E_1} t_2$.

Therefore, the image of $h \circ d$ is a subalgebra of $T(\Omega, E_2)/E$ which is equivalent to saying that $T(\Sigma, E_1)$ is a subalgebra of $(dT(\Omega, E_2))/E$. □

4 Equational Specification of Hardware Architecture

In this section, we use an extended form of equational specification to characterize *register transfer level (RTL)* hardware architecture. A register transfer level hardware description

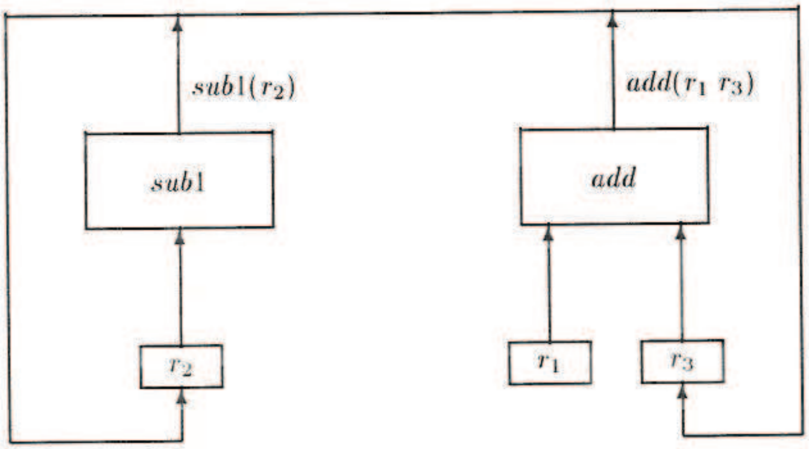

Figure 4: An Architecture

consists of at least 3 parts: an underlying abstract data description; a set of registers; and a set of data paths characterizing interconnections among registers and functional units. Our goal is to extend the definition of underlying abstract data description to include those of registers and data paths. The first observation is the that registers play the role of program variables [18] in a hardware description. Thus, they are treated as an instantiation of the set X mentioned in equational specification.

Let $n \geq 0$ and $R = \{r_1, r_2, ..., r_n\}$ be a set of registers of some hardware architecture. Each r_i is associated with some sort s_i. Intuitively, the action of the circuit is to update each register by a value from the data path of the circuit. Therefore, it is natural to view data paths of the circuit as vectors of n-function (or n-value) where each function (or value) represents a new content for the designate register. In other words, a data path of a circuit with register set R is a function $h: R \rightarrow T_\Sigma(R)$. The interpretation of h has two parts:

1. h specifies n operations which can happen in one cycle;
2. For every $r \in R$, $h(r)$ is the value sent to the register r.

Thus, a set of data paths of a circuit architecture H specifies a set of *possible* basic operations the circuit can have. Call an element of H an R-datapath. For example, let $R = \{x, y\}$ and $c \in H$ be the function $c(x) = add(x, y)$ and $c(y) = y$. This c "connects" the output of an adder to the register x. Data transfers not explicitly mentioned in H are prohibited. As shown in the next section, H also characterizes all the possible computations in one cycle. It plays the role of Σ in equational specifications.

Let us consider an example (Figure 4) of specifying a hardware architecture over natural numbers, a system in which the basic data type is *natural-number*, possibly augmented with metatypes such as *boolean*. Assume R consists of three registers r_1, r_2, r_3 of type natural number, an *add* and a *sub*1 which perform addition and decrement, respectively. Suppose the architecture can either add or decrement in one cycle, but not both. These requirements can be specified by a set of three functions $\{h_1, h_2, h_3\} \subseteq H$ where

$$h_1 = \{(r_1, r_1), (r_2, sub1(r_2)), (r_3, r_3)\}$$
$$h_2 = \{(r_1, r_1), (r_2, r_2), (r_3, add(r_1, r_3))\}$$
$$h_3 = \{(r_1, r_1), (r_2, r_2), (r_3, r_3)\}\}$$

the meaning of h_3 is "nothing happens in one cycle."

Given a set H, we would like to mimic the definition of $T_\Sigma(R)$ to define all the computations generated from H. Let $\mathcal{F}_{\Sigma,R}$ be the set of all functions $R \to T_\Sigma(R)$. The following two definitions classify two sets of possible computations derived from H.

Definition 4.

1. Let $f, g \in \mathcal{F}_{\Sigma,R}$. $h = f \circ g$ is an element of $\mathcal{F}_{\Sigma,R}$ such that for every $i : 1 \leq i \leq n$, $h(r_i) = f(r_i)[r_j \leftarrow g(r_j) : 1 \leq j \leq n] \in T_\Sigma(R)$ where $f(r_i)[r_j \leftarrow g(r_j) : 1 \leq j \leq n]$ denotes substituting every occurrence of r_j in $f(r_i)$ with $g(r_j)$ for all $j : 1 \leq j \leq n$. Define B_H be the transitive closure of H under $\circ$.
2. The set of *computation sequences derived from H*, denoted by D_H, is the transitive closure of H under concatenation. Concatenation of h and g is denoted by $h\ g$.

Each member of D_H is a sequence of computations allowed by the architecture while each element of B_H is a outcome of a computation sequence of the architecture. This suggests that every member of D_H designates an element of B_H as the *result or interpretation* of the computation sequence and every element of B_H is designated by an element of D_H. This observation is defined by a function $\mathcal{I}$:

Definition 5. Define function $\mathcal{I}: D_H \to \mathcal{F}_{\Sigma,R}$ as

1. $\mathcal{I}(f) = f$ if $f \in H$, and
2. $\mathcal{I}(f\ g) = f \circ g$ if $f, g \in \mathcal{F}_{\Sigma,R}$.

Since $\circ$ is associative, $\mathcal{I}$ is well defined. It is easy to show that $B_H = \mathcal{I}(D_H)$. As a matter of fact, B_H and D_H define different meanings of an architecture specification.

In equational specification, E is used to define corresponding properties of the function symbols in Σ. We would also like to have a set of equations to define properties of elements in H. Since elements of H are merely vectors of elements of Σ, the properties defined by E can be extended to relations on B_H and D_H, denoted by E_B and E_D and called R-extensions of E, respectively. Let $f, g \in B_H$ and $u_p \ldots u_2\ u_1 \in D_H$, $v_q \ldots v_2\ v_1 \in D_H$. $(f, g) \in E_B$ if and only if $(f(r_i), g(r_i)) \in E$ for all $1 \leq i \leq n$ and $(s, v) \in E_D$ if and only if $p = q$ and $(u_i, v_i) \in E$ for every $1 \leq i \leq p$. For simplicity, we usually drop the subscripts B and D when they are fixed by the context.

Furthermore, properties of H may need more equations than those extended from the equations of the underlying abstract data types as above. For example, in an implementation of stack by memory-address pair, $[m_1, 0] = [m_2, 0]$ is an equation but it can not be obtained from the extension of the equations concerning memory and address data types. However, the set of equations defining properties of H should contain the R-extension of E.

Definition 6. A specification of architecture is a quadruple $\langle A, R, \overline{E}, H \rangle$ where $A = \langle \Sigma, E \rangle$ is an equational specification, called *the underlying abstract data type specification.* $\overline{E}$ is a set of equations on B_H (D_H) which contains the R-extension of E. $H \subseteq \mathcal{F}_{\Sigma,R}$, which contains the identity function $id: r \to r$ for every $r \in R$.

5 Algebras for Hardware Architecture Specifications

In the last section, we defined hardware architecture as an extended form of equational specification. In this section, we define an H-algebra for the architecture specification $\langle A, R, \overline{E}, H \rangle$. This H-algebra serves as the meaning of the specification.

Architecture Algebra

Let $\langle A, R, \overline{E}, H \rangle$ be a specification of some architecture. The following lemmas state properties of D_H and B_H. The proofs are simple.

Lemma 1. Let $b \in B_H$, $d \in D_H$ and $f \in H$. If we define $f(b) = f \circ b$ and $f(d) = f\ d$ then both $\langle B_H, H \rangle$ and $\langle D_H, H \rangle$ are H-algebras.

Lemma 2. E_B and E_D are equivalence relations on B_H and D_H respectively.

Lemma 3. E_B and E_D are congruence relations on $\langle B_H, H \rangle$ and $\langle B_H, H \rangle$ respectively. Therefore, $\langle B_H, H \rangle / E_B$ and $\langle D_H, H \rangle / E_D$ are H-algebras.

We use $B(H, E_B)$ and $D(H, E_D)$ to denote $\langle B_H, H \rangle / E_B$ and $\langle D_H, H \rangle / E_D$ respectively. Both $B(H, E_B)$ and $D(H, E_D)$ are important in observing the behavior of a hardware architecture $P = \langle A, R, E, H \rangle$. The elements of $B(H, E_B)$ represent all the possible results which can be computed by P; this is P's behavioral aspect. The elements of $D(H, E)$ represent not only what results the architecture can obtain (through the interpretation $\mathcal{I}$), but also describe how those computations are performed. Therefore, $D(H, E_D)$ characterizes the operational aspect as well as behavioral aspect of P. It is a matter of emphasis to decide which aspect of P is used to characterize it.

Definition 7. Given an architecture specification $P = \langle A, R, E, H \rangle$. The behavior of P is defined as $B(H, E_B)$ and the computation of P is defined as $D(H, E_D)$.

Evaluation of Terms

We now discuss the evaluation of elements of $T_\Sigma(R)$. As mentioned previously, functions of D_H and B_H represent possible computations by the architecture. However, we did not mention the values of those computations.

Let S be a set of sorts and $\mathcal{A}$ be a S-sorted Σ-algebra. Let μ be a special *undefined* (or *unknown*) value present in every carrier. For every sort s, we extend A_s of some Σ-algebra to contain the value μ: $\mathcal{A}_{s,\mu} \overset{\text{def}}{=} A_s \cup \{\mu\}$. An *interpretation* $\mathcal{V}$ of a specification $\langle A, R, E, H \rangle$ is a triple $\mathcal{V} = \langle \mathcal{A}, \mathcal{H}, \mathcal{O} \rangle$ where

$\mathcal{H}$: $\cup_{s \in S} \Sigma_{\lambda,s} \to \mathcal{A}$.

$\mathcal{O}$: interprets an $s_1 \ldots s_k$-sorted function symbol of Σ as a function of $A_{s_1,\mu} \times \ldots \times \mathcal{A}_{s_k,\mu} \to \mathcal{A}_{s,\mu}$. For every $\sigma \in \Sigma$, $\mathcal{O}(\sigma)$ is a function: if any one of the arguments of $\mathcal{O}(\sigma)$ is μ, then $\mathcal{O}(\sigma)$ returns μ.

Given $t \in T_\Sigma$, $\mathcal{V}(t)$ is defined as follows:

1. $\mathcal{V}(t) = \mathcal{H}(t)$ if $t \in \Sigma_{\lambda,s}$ for some $s \in S$.
2. $\mathcal{V}(t) = \mu$ if $t = r \in R$.
3. $\mathcal{V}(\mu) = \mu$.
4. $\mathcal{V}(t) = O(f)(\mathcal{V}(t_1), \mathcal{V}(t_2), \ldots, \mathcal{V}(t_n))$ if $t = f(t_1, t_2, \ldots, t_n)$.

This definition of $\mathcal{V}$ implies that all computations start with an undefined state, that is, all registers initially contain μ. However, it is also convenient to use the concept of *evaluation in an environment.* An environment e for an architecture $\langle A, R, E, H \rangle$ is an element of B_H. Let $t \in T_\Sigma$, the evaluation of t in the environment e, denoted as $\mathcal{V}_e(t)$, is defined as: $\mathcal{V}_e(t) = \mathcal{V}(t(e))$ where $t(e)$ is obtained by substituting every occurrence of r in t by $e(r)$ for all $r \in R$.

6 Implementation of Architectures

This section discusses the issue of implementation. The basic idea is to extend implementation of equational specifications to that of architecture specifications. It also explores the relationship between implementation of an architecture and its underlying abstract data type.

Implementation of Architectures

Definition 8 states a notion of architecture implementation. It merely reiterates the intuition stated in Definition 2 in a context of architecture specification.

Definition 8. Let $K_\Sigma = \langle A_\Sigma, R_\Sigma, E_\Sigma, H_\Sigma \rangle$ and $K_\Omega = \langle A_\Omega, R_\Omega, E_\Omega, H_\Omega \rangle$ be two architecture specifications. K_Ω is said to be an implementation (architecture) of K_Σ if there exists a derivor $d\colon H_\Sigma \rightarrow B_{H_\Omega}$ and an congruence relation E on $B(H_\Omega, E_\Omega)$ such that $B(H_\Sigma, E_\Sigma) \subseteq \langle B_{H_\Omega}/E_\Omega, d(H_\Sigma) \rangle / E$.

Let K_Σ be an architecture specification, let A_Σ be K_Σ's underlying type specification, and let $\langle A_\Omega, d, E \rangle$ be an implementation of A_Σ. We are interested in the derivation of an implementation architecture of K_Σ from d and E. Our approach is to extend d to a derivor function $H_\Sigma \rightarrow H_\Omega$ such that the extension is a derivor of the architecture implementation. We also discuss the condition under which we can "naturally extend" the implementation of the underlying type specification to derive an implementation of the corresponding architecture.

Let $d = \langle \xi, \kappa \rangle$ be a derivor from Σ to Ω. A first attempt at extending d to $\overline{d} = \langle \xi, \gamma, h \rangle$ is to let:

1. $\xi : S_\Sigma \rightarrow S_\Omega^*$ be that of d;
2. $\gamma : R_\Sigma \rightarrow [R_\Omega]$ where $[R_\Omega]$ is a partition of R_Ω which satisfies the condition that for every $s \in S_\Sigma$, there exists a $[r] \in [R_\Omega]$ and there exists an vectorization for $[r]$ such that the resulting vector is $\xi(s)$ sorted;

Our goal for h is to extend κ to $h\colon H_\Sigma \rightarrow H_\Omega$ according to γ. Since γ is a function from R_Σ to $[R_\Omega]$, a partition of R_Ω, h should be a function from H_Σ to $[R_\Omega] \rightarrow T_\Omega^*$. Fortunately, this does not cause any serious problem since we can always "flatten" elements of $[R_\Omega] \rightarrow T_\Omega^*$ to simulate functions of $R_\Omega \rightarrow T_\Omega$. Therefore, we ignore the difference between functions $[R_\Omega] \rightarrow T_\Omega^*$ and functions $R_\Omega \rightarrow T_\Omega$.

A natural way to achieve h from κ is that for every $c \in H_\Sigma$, $h(c)$ is a function $[R_\Omega] \rightarrow T_\Omega^*$ defined as: for every $r \in R_\Sigma$,

$$h(c)(\gamma(r)) = c(r)[\forall \sigma \in \Sigma.\ \sigma \leftarrow d(\sigma)]$$

That is, the result of substituting every occurrence of $\sigma \in \Sigma$ in $c(r)$ by $d(\sigma)$.

Unfortunately, this definition does have a problem when γ is not bijection. First, $\hbar(c)$ is *total* only if γ is onto. Second, if γ is not 1-1, that is, there are $r_1, r_2 \in R_\Sigma$ such that $r_1 \neq r_2$ but $\gamma(r_1) = \gamma(r_2)$, then h defined above is not well defined.

To solve the first problem, we add an assumption that γ is onto. It is important to realize that this assumption does not trivialize the result we are about to develop. As a matter of fact, we can define $\hbar(c)$ as: For every $c \in H_\Sigma$, $\hbar(c) \in B_\Omega$ such that for every $r \in R_\Sigma, \hbar(c)(\gamma(r)) = d(c(r))$. Although there are more than one such elements in B_H, it can be shown that the choice among those does not affect our result. To prove this requires additional notation which is largely irrelevant to our major purpose. Thus, we choose to impose the assumption that γ is onto.

We solve the second problem by introducing a *conflict-free* condition on the elements of H_Σ. It is intended to assure for every data transfer $h \in H$, at most one register from each partition of R_Σ by γ is updated. Assume that $R_r = \{\ r' \in R_\Sigma \mid \gamma(r') = r\ \}$ for every $r \in R_\Omega$. Then

Definition 9. Let $r \in R$,

1. K_r is the set of all terms in $T_\Sigma(R)$ which have r as one of their subterms;
2. $B_r = \{\ b(r) \mid b \in B_H\ \}$;
3. $T_r = K_r \cup B_r$;
4. $\Sigma_r = \{\ \sigma \mid \sigma$ appears in an element of $T_r\ \}$.

Lemma 4. If $r \in R$, T_r, Σ_r as defined, then $\langle T_r, \Sigma_r \rangle$ is an algebra. $\langle T_r, \Sigma_r \rangle$ is called *the subalgebra of* $\langle T_\Sigma, \Sigma \rangle$ *associated with* r.

If $r_1 \neq r_2$ but $\gamma(r_1) = \gamma(r_2)$, it is necessary to have $\overline{d}(T_{r_1})$ distinct from $\overline{d}(T_{r_1})$. This consideration motives the following definition.

Definition 10. Let $\langle A, R, E, H \rangle$ be a specification of architecture and $\overline{d} = \langle \xi, \gamma, h \rangle$. For $r \in R$, let $[r]_\gamma = \{\ q \in R \mid \gamma(q) = \gamma(r)\ \}$. H is said *conflict-free with respect to* $\overline{d}$ if

1. For every $f \in H$ and every $r \in R$, there is at most one $q \in [r]_\gamma$ such that $f(q) \neq q$;
2. For every $r \in R_\Omega$, all $r' \in [r]_\gamma$, $\overline{d}(T_{r'})$s are distinct from each other and are subalgebras of $T(\Omega, E_\Omega)$.

Now we can give a definition of the extension of d to $\overline{d}$: $H_\Sigma \to H_\Omega$ as follows:

Definition 11. Let $d = \langle \xi, \kappa \rangle$ be a derivor $\Sigma \to \Omega$, the extension of d to $H_\Sigma \to H_\Omega$ is $\overline{d} = \langle \xi, \gamma, h \rangle$:

1. $\xi : S_\Sigma \to S_\Omega$ is ξ of d;
2. $\gamma : R_\Sigma \to [R_\Omega]$ is onto where $[R_\Omega]$ is a partition of R_Ω defined by γ;
3. h is a 1-1 function defined as: for every $c \in H_\Sigma$, $\hbar(c)$ is a function $\gamma(R_\Sigma) \to T^*_\Omega(R_\Omega)$ such that for $q \in [R_\Omega]$, $\hbar(f)(q) = d(f(r))$ such that $r \in R_q$ and $f(r) \neq r$.

$\overline{d}$ is said to be well defined if ξ, γ, and h are all functions.

The *conflict-free* condition is a sufficient condition under which the extension of d to $\overline{d}$ in Definition 11 preserves the implementation relationship:

Theorem 2. Let $K_\Sigma = \langle A_\Sigma, R_\Sigma, E_\Sigma, H_\Sigma \rangle$ be an architecture specification, $A_\Sigma = \langle A, E_\Sigma \rangle$, $A_\Omega = \langle \Omega, E_\Omega \rangle$, and $\langle A_\Omega, d, E \rangle$ be an implementation of A_Σ. Let $\overline{d} = \langle \xi, \gamma, h \rangle$ be as defined by Definition 11. If H_Σ is conflict-free with respect to $\overline{d}$ then $\langle K_\Omega, \overline{d}, E \rangle$ is an implementation of K_Σ where $K_\Omega = \langle \Omega, \gamma(R_\Sigma), E_\Omega, h(H_\Sigma) \rangle$ is called $\overline{d}$-generated architecture.

Proof. The only thing we need to prove is that there exists a 1-1 homomorphism $\overline{h}$ which maps $B(H_\Sigma, E_\Sigma)$ to $B(h(H_\Sigma), E_\Omega)/\overline{E}$.

Let h be the 1-1 homomorphism which maps $T(\Sigma, E_\Sigma)$ to some subalgebra of $(dA_\Omega)/E$. We extend h to $\overline{h}: B(H_\Sigma, E_\Sigma) \rightarrow B(h(H_\Omega), E_\Omega)/\overline{E}$ in the following way: for every $c \in H_\Sigma$, $f = \overline{h}(c)$ is a function such that for every $r \in \gamma(R_\Sigma)$, $f(r) = h(c(r'))$ where $r' \in R_r$ and $c(r') \neq r'$ or $f(r) = r$ if every $c(r') = r'$. Since there is only one such $r' \in R_r$ by the non-conflict condition and $R_r \neq \phi$ for every r, $f(r)$ is well defined thus so is $f = \overline{h}(c)$.
Claim: $\overline{h}$ is a 1-1 homomorphism. We justify the claim by:

1. Let $b_1, b_2 \in B_{H_\Sigma}$,

   $$b_1 \equiv_{E_\Sigma} b_2$$

   is equivalent to

   $$\forall r \in R_\Sigma.\ b_1(r) \equiv_{E_\Sigma} b_2(r)$$

   implies

   $$\forall r \in R_\Sigma.\ h(b_1)\gamma(r)) \equiv_{d(E_\Sigma)} h(b_2)\gamma(r))$$

   implies

   $$\forall r \in R_\Sigma.\ h(b_1)\gamma(r)) \equiv_{E_\Omega/E} h(b_2)\gamma(r))$$

   is equivalent to

   $$\overline{h}(b_1) \equiv_{E_\Omega/E} \overline{h}(b_2)$$

   Therefore, $\overline{h}$ is well defined;

2. Let $b_1, b_2 \in B_{H_\Sigma}$, We consider the image of $\overline{h}(b_1 b_2)$. $b_1 b_2 = b_1[\forall r_i \in \gamma(R_\Sigma).\ r_i \leftarrow b_2(r_i)]$, therefore, for every $r \in \gamma(R_\Sigma)$

   $$\begin{aligned} \overline{h}(b_1 b_2)(r) &= \overline{h}(b_1[\forall r_i \in \gamma(R_\Sigma).\ r_i \leftarrow b_2(r_i)])(r) \\ &= h(b_1(r')[\forall r_i \in R_\Sigma.\ r_i \leftarrow b_2(r_i)]) && \text{The definition of } \overline{h} \text{ and } r' \in R_r \\ &= h(b_1(r')[\forall r_i \in R_\Sigma.\ r_i \leftarrow h(b_2(r_i))]) && h \text{ is homomorphism} \\ &= h(b_1(r'))\overline{h}(b_2) && \text{The definition of } \overline{h} \\ &= \overline{h}(b_1)\overline{h}(b_2)(r) \end{aligned}$$

   Therefore, $\overline{h}(b_1 b_2) \equiv \overline{h}(b_1)\overline{h}(b_2)$ thus $\overline{h}$ is a homomorphism.

3. Let $b_1, b_2 \in B_{H_\Sigma}$ and assume that

$$\overline{h}(b_1) \equiv_{E_\Omega/E} \overline{h}(b_2)$$

which is equivalent to

$$\forall r \in \gamma(R_\Sigma).\ (\overline{h}(b_1))(r) \equiv_{E_\Omega/E} (\overline{h}(b_2))(r)$$

By the *conflict-free* condition of the theorem and the definition of $\overline{h}$, the above equation is equivalent to

$$\forall r \in \gamma(R_\Sigma).\ \forall r' \in R_r.\ h(b_1(r')) \equiv_{E_\Omega/E} h(b_2(r'))$$

which is equivalent to

$$\forall r \in R_\Sigma.\ h(b_1(r)) \equiv_{E_\Omega/E} h(b_2(r))$$

And since h is a 1-1 homomorphism thus

$$\forall r \in R_\Sigma.\ b_1(r) \equiv_{E_\Sigma} b_2(r)$$

and thus

$$b_1 \equiv_{E_\Sigma} b_2$$

which proves that $\overline{h}$ is 1-1.

This justifies the claim and concludes the proof. □

Serialization

In the previous discussion, the *conflict-free* condition of data paths is imposed as a necessary condition of implementation. The part of the problem to make a given set of data paths conflict-free with respect to γ, without changing the meaning of architecture, is called *serialization.* It is analogous to *allocation* and *scheduling* in high level synthesis (*e.g.* [3]), *register allocation* and *code generation* in program compilation, and *microcode generation* (*e.g.* [13]). What follows is a brief review of the material presented in [20].

Definition 12. Let$\langle \Sigma, R, E, H \rangle$ be a specification and $G \subset H$ be set of conflict-free data paths with respect to some γ. A *serialization problem* is: given a $h \in H - G$, find $c_n, c_{n-1}, \ldots c_1 \in G$ $(n \geq 2)$ such that $h \equiv_E c_n \circ c_{n-1} \circ \ldots \circ c_1$. Such an h is said to be *serializable* by G.

If all elements of H are serializable by G, H can be replaced by G and $\langle A, R, E, G \rangle$ is an implementation of $\langle A, R, E, H \rangle$. Unfortunately, it is readily proved that the serialization problem is undecidable, in general [20]. Nevertheless, heuristic algorithms have been found to cope with similar problems (such as register allocation or microcode generation). [13] presents an algorithm to cope with microprogram generation which we believe can be employed to solve serialization problems where E is empty.

Research Direction

We discussed the implementation issue of architecture specification. We presented a derivational approach to construct an implementation of architecture from an implementation of an underlying type specification and an architecture specification. Although the method provides an insight to the relationship between implementation of architecture and its underlying type specification, it has the following limitations:

1. We are not able to find an effective decision procedure to test the *conflict-free* condition of the theorem and we suspect that it is not recursively decidable. This severely limits our chance of applying this theorem to any automated derivation system, which is one of the major goals of our research. Thus, a stronger characterization of implementation may be needed.
2. Theorem 2 only discusses behavioral implementation but does not consider timing issues. We would like to consider implementations which take multiple cycles where the more abstract type is intuitively combinational. That is, we would like to extend $\hbar$ to a function $H_\Sigma \rightarrow D_{R_\Omega, H_\Omega}$.
3. The theorem only addresses derivations whose γ function is onto, *i.e.* the derived architectures have register sets $\gamma(R_\Sigma)$ although it is not necessary by the definition. When a derived architecture's register set R is strictly contained in $\gamma(R_\Sigma)$, the registers in $R - \gamma(R_\Sigma)$ can be used as temporary registers. Because using temporary registers usually results in introducing new states of control, the framework has to have capability to address sequential properties of circuits.

7 Recursive Functions as Hardware Controls

The standard definition of recursive functions over the natural numbers can be generalized as that of recursive functions on abstract data types [7, 9, 18]. Tucker and Zucker present a theoretical treatment of recursive function over abstract data types [18]. Johnson shows that a recursive function of an abstract data type in certain form can be systematically transformed to a sequential circuit representation which segregates the control of circuit from its abstract basis [9], and examples of modeling digital circuits by recursive functions on abstract bases are shown in [7, 9]. We take a similar approach to hardware control here. As we have already shown that a hardware architecture can be modeled by extended abstract data types, we view a hardware system as a pair, consisting of an extended abstract data type and a recursive function on the extended abstract data type; the latter serves to control the circuit. This section also gives the meaning of such a circuit specification and discusses the concept of implementation of circuit specifications.

The set of recursive functions defined below directly corresponds to the primitive (course-of-value) recursive functions in the conventional theory of recursive functions, e.g. [17]. It is called *control functions* for circuit.

Definition 13. Let $A = \langle \Sigma, R, E, H \rangle$ be a specification of an architecture, e be a variable ranging over $B_{H,E}$, called an environment variable, and $\mathcal{V}$ be the evaluation function defined in Section 4. The set of control functions over A, $F(A)$, is defined as follows:

1. (Primitive Operators). If for every $f \in H$, $h(e) = f \circ e$, then $h \in F(A)$;

2. (Function Composition). If $f, g \in F(A)$, then $Com_{f,g} = f \cdot g(e) \in F(A)$ where $\cdot$ is the standard function composition;

3. (Definition by Cases). An n-case definition is $IF_{h,p,n}$ where $n > 0$ is a natural number, $h : \langle h_i \in F(A) \mid 0 \leq i < n \rangle$ and $p : \langle p_i \in T_\Sigma \mid 0 \leq i < n \rangle$.

$$IF_{h,p,n}(e) = \begin{cases} h_i(e) & \text{if } \mathcal{V}_e(p_i) = 1 \text{ for } 0 \leq i \leq n-1 \\ \perp & \text{Otherwise} \end{cases}$$

$IF_{h,p,n} \in F(A)$.

4. (Primitive Recursion). If $h_1, h_2, h_3 \in F(A)$, $p \in T_\Sigma$ and $PR_{h_1,h_2,h_3,p}$ is defined as:

$$PR_{h_1,h_2,h_3,p}(e) = \begin{cases} h_1(e) & \text{if } \mathcal{V}_e(p) = 0 \\ h_3 \cdot PR_{h_1,h_2,h_3,p} \cdot h_2(e) & \text{if } \mathcal{V}_e(p) > 1 \\ \perp & \text{Otherwise} \end{cases}$$

where $\mathcal{V}_{h_2(e)}(p) < \mathcal{V}_e(p)$ then $PR_{h_1,h_2,h_3,p} \in F(A)$.

Now, we can define a circuit specification as the combination of an architecture specification and a recursive function.

Definition 14. A specification of a circuit is a tuple $\langle A,\ f \rangle$ where A is a specification of an architecture and $f \in F(A)$.

In hardware designs, two hardware systems are said to be equivalent if these two systems produce the same outputs for the same inputs and the same initial states [3]. However, other criteria can be added. For example, a more restrictive condition is that two systems not only produce the same results but also take the same time (measured by cycles, for example) to produce the same results. We consider two hardware systems to be *equivalent* in the former sense. In our framework, input to a hardware system is included in those functions $f \in H$ such as $f(r) = c$ for some $r \in R$ and $c \in \Sigma_\lambda$[2]. In real designs, *equivalent values* does not necessarily mean identical values but values which are *equal with respect to some equivalence relation* or *equal with respect to some mapping*. For example, in a (natural number) modulo 2^n system, an element k may be thought of being equal to an n-bit binary string $b_{n-1}...b_1b_0$ if $k = h_n(b_{n-1}...b_1b_0)$ where $h_n(b_{n-1}...b_1b_0) = \Sigma_{0 \leq i \leq n-1}\ 2^i \times b_i$. Combining the discussion above with our definition of implementation of architecture, we have the following definition.

Definition 15. Let $\langle B_\Omega, d, E \rangle$ be an implementation of an architecture specification $B_\Sigma = \langle A, R, E_\Sigma, H_\Sigma \rangle$ and $f \in F(B_\Sigma)$. $d(f)$ is recursively defined as:

1. If $f = c \in H_\Sigma$ then $d(f) = \hbar(c)$;

2. If $f = Com_{f_1,f_2}$ then $d(f) = Com_{d(f_1),d(f_2)}$;

3. If $f = IF_{h,p,n}$, then

$$d(f)(e) = \begin{cases} d(h_i)(d(e)) & \text{if } \mathcal{V}_{d(e)}(d(p_i)) \equiv_E 1 \text{ for } 0 \leq i \leq n-1 \\ \perp & \text{Otherwise} \end{cases}$$

---

[2] We may extend this by allowing c be either some element of Σ_λ or some function which represents (part) of the environment in which the circuit is operating. Although this requires some minor changes in our definition of the constraint set H, it does not complicate the problem.

$$\begin{tikzcd}
K_\Sigma \arrow[r, "f_\Sigma"'] \arrow[d, "h"] & K_\Sigma \\
K_\Omega \arrow[r, "h(f_\Sigma)"] & K_\Omega \arrow[u, "h^{-1}"']
\end{tikzcd}$$

Figure 5: $f_\Sigma = h^{-1}\ h(f_\Sigma)\ h$

4. If $f = PR_{h_1,h_2,h_3,p}$ then

$$d(PR_{h_1,h_2,h_3,p})(e) = \begin{cases} d(h_1)(d(e)) & \text{if } \mathcal{V}_{d(e)}(d(p)) \equiv_E 0 \\ PR' & \text{if } \mathcal{V}_{d(e)}(d(p)) > 1 \\ \perp & \text{Otherwise} \end{cases}$$

where $PR' = d(h_3) \cdot PR_{d(h_1),d(h_2),(h_3),d(p)} \cdot d(h_2)(d(e))$.

Let $P_\Sigma = \langle K_\Sigma, f_\Sigma\rangle$ and $P_\Omega = \langle K_\Omega, f_\Omega\rangle$ are specifications of hardware systems.

Definition 16. If there exists a function $h : K_\Sigma \to K_\Omega$, a family of functions $h^{-1} = \langle h_r^{-1} : K_\Omega \to T_r\rangle_{r\in R_\Sigma}$ (for the definition of T_r, see Definition 10) and an equivalence relation E of K_Ω such that for every $a \in K_\Sigma$ and $r \in R_\Sigma$, $h_r^{-1}(f_\Omega(h(a))) \equiv_E f_\Sigma(a)(r)$, then $\langle P_\Omega, h, E\rangle$ is called an implementation of P_Σ.

The following theorem associates the implementation of systems to that of the architecture. It is analogous to the corollary of Theorem 2.

Theorem 3. If $P_\Sigma = \langle K_\Sigma, f_\Sigma\rangle$ is a circuit specification and $\langle K_\Omega, d, E\rangle$ is an implementation of K_Σ then $\langle P_\Omega, h, E\rangle$ is an implementation of P_Σ where $P_\Omega = \langle K_\Omega, d(f_\Sigma)\rangle$ and $h : K_\Sigma \to K_\Omega$ is defined as:

$$h(b)(r) = \begin{cases} d(b)(r) & \text{if } r \in \gamma(R_\Sigma) \\ r & \text{if } r \in R_\Omega - \gamma(R_\Sigma) \end{cases}$$

for every $b \in B_{R_\Sigma,H_\Sigma,E_\Sigma}$.

Since $\langle K_\Omega, d, E\rangle$ is an implementation of K_Σ, h^{-1} is actually implicitly defined by d. Figure 5 is a pictorial illustration of Theorem 3.

This theorem gives a derivational method of finding an implementation of a circuit specification through an implementation of its architecture. According to Theorem 2, we can go further to find an implementation through that of the underlying type specification.

8 Summary

This paper gives a formal approach to the description of hardware architecture systems based on the underlying abstract data types' specification. The semantics of architecture is given as universal algebras. We show that the implementation of a hardware system can be obtained from the implementation of the underlying abstract data types. The correctness of such an implementation is thus directly related to the correctness of the underlying abstract data type implementation.

This was done by extending the specification of an abstract data type to include a description of architecture, in the form of a set of permitted data transfers. A criterion for correct implementation of an underlying abstract data type was then adapted to the extension. Finally, a functional characterization of control was added and the notion of implementation was again extended to include control.

Theorem 1 states that the correctness of implementation, as defined in Definition 2, is equivalent to the provability of the implemented type's equations. Theorem 2 states that this form of correctness can be extended to conflict-free specifications of architectures. Theorem 3 states that control of the implementing architecture is derivable from that of the implemented architecture.

Acknowledgment

We are grateful to J. V. Tucker for his thoughtful comments on an early draft of this paper.

We are also grateful to National Science Foundation for its support, in part, of this research under the grants numbered MIP87-07067, DCR85-21947, and MIP89-21842.

References

[1] J. A. Bergstra and J. V. Tucker. Initial and final algebra semantics for data type specification: Two characterization theorems. *SIAM Journal of Computing*, Vol 12(No. 2):366–387, May 1983.

[2] C. David Boyer and Steven D. Johnson. Using the digital design derivation system: Case study of a VLSI garbage collector. In J. Darringer and F. Ramming, editors, *Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages (CHDL)*, Amsterdam, 1989. Elsevier. Also published as Technical Report 274, Computer Science Department, Indiana University, April 1989.

[3] Raul Camposano. Behavior-preserving transformations for high-level synthesis. In M. Leeser and G. Brown, editors, *VLSI Specification, Verification and Synthesis: Mathematical Aspects*, New York, July 1989. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, Springer-Verlag. Lecture Notes in Computer Science Vol-408.

[4] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1; Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[5] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, chapter 5, pages 80–149. Prentice-Hall, Englewood Cliffs, N.J. 07632, 1978.

[6] G. Gopalkrishnan, R. M. Fujimoto, V. Akella, N. S. Mani, and K. N. Smith. Specification-driven design of custom architecture in HOP. In P.A. Subrahmanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 128–170. Springer Verlag, 1989.

[7] 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. J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 347–374. Elsevier Science Publishers B.V., 1988.

[8] Steven D. Johnson. Applicative programming and digital design. In *Proceedings Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 218–227, 1984.

[9] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. The MIT Press, Cambridge, 1984.

[10] Steven D. Johnson. Manipulating logical organization with system factorizations. In G. Brown M. Leeser, editor, *Hardware Specification, Verification and Synthesis: Mathematical Aspects, Mathematical Sciences Institute Workshop*, pages 260–281. Cornell University, Ithaca, NY, USA, Springer Verlag, July, 1989. Lecture Notes in Computer Science Vol 408.

[11] Steven D. Johnson, Bhaskar Bose, and C. David Boyer. A tactical framework for digital design. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer Academic Publishers, Boston, 1988.

[12] S. Kamin. Final data types and their specification. *ACM Transaction of Programming Languages and Systems*, 5(1):97–123, January 1983.

[13] M. Mahmood, F. Mavaddat, M. I. Elmasry, and M. H. M. Cheng. A formal language model of local microcode synthesis. In Luc Claesen, editor, *Proceedings of The International Workshop on The Applied Formal Method for Correct VLSI Designs*, Leuven, Belgium, 1989. Elsevier Science Publishers B.V.

[14] M. C. McFarland, A. C. Parker, and R. Camposano. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 330–336, Anaheim, CA, 1988. ACM/SIGDA.

[15] J. Meseguer and J.A. Goguen. Initiality, induction, and computability. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.

[16] G. J. Milne. CIRCAL and the representation of communication concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2), 1985.

[17] Hartley Rogers. *Theory of Recursive Functions and Effective Computation*. McGraw-Hill Book Company, 1967.

[18] J. V. Tucker and J. I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*. North-Holland, 1988.

[19] M. Wand. Final algebra semantics and data type extensions. *Journal of Computing System Science*, 19:27–44, 1979.

[20] Zheng Zhu and S. D. Johnson. An algebraic characterization of structural synthesis for hardware. In Luc Claesen, editor, *Proceedings of The international Workshop on The Applied Formal Methods for Correct VLSI Designs*. North Holland, 1989.