TECHNICAL REPORT NO. 279

Manipulating Logical Organization with System Factorizations

by

Steven D. Johnson

June 1989

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405–4101

Manipulating Logical Organization with System Factorizations*

Steven D. Johnson
Indiana University

ABSTRACT *Logical* organization *refers to a system's decomposition into functional units, coprocesses, and so on; it is sometimes called the 'structural' aspect of system description. In an approach to digital synthesis based on functional algebra, logical organization is developed using a class of transformations called* system factorizations. *These transformations isolate subsystems and encapsulate them as applicative combinators. Factorizations have a variety of uses, ranging from the refinement of actual architecture to the synthesis of certain kinds of verification conditions. This paper outlines the foundations for this algebra and presents several examples.*

1. Introduction

It is conventional wisdom in hardware design—as in every other branch of programming—to defer decisions about architecture until the control is understood. While design surely begins with an idea of the building blocks needed, engineering should focus first on an algorithm. It is usually a mistake to center a design around some device that seems to solve the problem. Ideally, the final details of architecture are developed once an acceptable algorithm has been designed; components are chosen to serve the needs of control (See [21], p. 166, for a discussion.).

In an algebraic design paradigm, this aspect of engineering is characterized by a family of transformations called *system factorizations*. Their purpose is to introduce a conceptual organization analogous to that of a functional block diagram, consisting of a hierarchy of communicating modules and coprocesses. This paper defines basic

*Research reported herein was supported, in part, by the National Science Foundation under grants numbered MIP8707067 and DCR8521497.

The final article will appear in the proceedings of the Mathematical Sciences Institute work shop entitled, "Hardware Specification, Verification and Synthesis," held at Cornell University, Itaca, NY, July, 1989.

laws from which factorizations are composed and illustrates tactical issues with a series of examples.

The design formalism is founded on functional algebra. A sequence of transformations is applied to an initial expression, resulting in a target expression closer to a physical realization. The list of transformations is called a *derivation* to emphasize that source and target expressions are really dialects of a single modeling language of higher order functions. Following current terminology [3], source expressions correspond to 'specifications' and target expressions to 'implementations.' However, it is more accurate to say that an implementation is specified by the source expression *together with* a derivation sequence, since the latter expresses much of the design intent.

One purpose here is to consolidate previous observations about, and experiences with, the structural aspect of digital design derivation. Earlier work establishes a connection between functional descriptions of control flow and synchronous network descriptions at a level of description reflected by an arbitrarily chosen vocabulary of operations and tests [12, 10, 11]. *Abstract component factorizations* introduce a treatment for encapsulation and information hiding. These transformations isolate data abstractions, such as stack and memory, from the relatively concrete types that can be more directly implemented, such as item and address. Subsequent experimentation with the derivation method revealed that essentially the same transformations are useful in developing other aspects of architecture. Slightly generalized, the factorizations are used to assign operations to units; and they provide an *ad hoc* treatment for communication ports [13].

Ideally, individual transformations preserve a behavioral interpretation of notation. Derived implementations are, therefore, 'correct' under that notion of equivalence—derivation is a form of proof. In practice, however, correctness is relative to side conditions, such as the obvious assumption that that electronics used models logic. In addition, transformations may synthesize verification conditions, which become antecedents to correctness claims (See [20, 4, 22] for related discussions.). In a preliminary exploration of interplay between mechanical derivation and mechanical verification, factorizations play a central role in isolating such conditions [14].

However, the importance of factorization is partly a result of the approach taken to description. A strength of functional modeling is the use of abstraction to gain representation independence. Although types are now common in applicative languages, there may be little enforcement of modularity. Strong encapsulation qualities are gaining prevalence in hardware applications (e.g. [16] and [6]), but a more open view of type abstraction may be justified. As proposed in the opening paragraph, it might be better to specify in terms of arithmetic rather than in terms of ALUs; and it might be better to reason in terms of storage abstractions than in terms of RAM devices. Under this thesis it becomes necessary to have algebra for imposing modularity; and this is the basic effect of a system factorization.

Section 2 outlines a first order framework for *system descriptions*, which express synchronized networks in a simple language of applicative terms. The presentation follows the semantic treatment in [12], at a purely syntactic level, with refinements for multioutput operations. Multiple output combinations play so prominent a role in the description of hardware that it is better to account for them in the metasyntax. Sheeran works builds foundation for μFP [19]. Boute proposes an 'open semantics' approach n the same notational framework. Harmond and Tucker adapt the formal-algebraic theory to the model of sequential behavior used here [8]. Some of the vocabulary used here comes from a text book by Loeckx and Sieber [15].

The restriction to first order expression, though intuitive, is temporary. Higher order techniques certainly have a place in hardware description. The most successful exploitation has been Sheeran's algebra for structural manipulation [18]. O'Donnell develops several useful interpretations for functionals [17]. In both cases, there is a well determined structural meaning to work from, which is not necessarily present in this application. However, this work will be extended as the details are worked out.

Section 3 states the fundamental transformation laws for system descriptions. It is claimed that the coarser grained factorizations illustrated in Section 4 are reducible to combinations of the fundamental laws. In other words, the fundamental laws correspond to the basic inference mechanisms of a theorem prover, and factorizations correspond to tactical lemmas. This relationship is discussed further in Section 5, where experiences with the use of automated factorizations is briefly reviewed.

2. Terms, Combinations and System Descriptions

In the definitions that follow, the language of terms for a many sorted algebra of multioutput functions is extended by a delay operator to form a language of *signal expressions*. A *system description* is a simultaneous system of signal expressions.

The central fact established in this section is that the use of closed *combinations*, or parameterized expressions, commutes with a behavioral interpretation of the language of signals.

A *basis* describes a family of value sets, $A_1, \ldots, A_n$, together with a collection of total and strict functions, $f_1, \ldots, f_m$, on these sets. These functions are called *operations* in order to distinguish them from defined functions. With each set A_i is associated a type symbol, τ_i, a distinct set of constant symbols, and a distinct set of variable symbols. The notation $v: \tau_i$, sometimes phrased, "v is a τ_i," asserts that the variable v ranges over values in A_i. The *signature*, of an operation describes its domain and range, which in general are nested products. The formula $f: (\tau_1, (\tau_2, \tau_3)) \to (\tau_4, \tau_5, \tau_6)$ asserts that the operation f maps the product $A_1 \times (A_2 \times A_3)$ to the product $A_4 \times A_5 \times A_6$. Since product formation is not considered associative, parentheses are significant in signatures. The signatures $(\tau) \to \tau'$ and $\tau \to \tau'$ are distinct; however, in some metaexpressions, extra mathematical parentheses are used.

Certain properties of the basis are always assumed. There must be a set of truth values, of type *Bool*, which includes constants true and false. Operations with range *Bool* are called *tests*. Each value set has *undetermined element* ("don't care" or "don't know" but *not* "undefined") designated by a '#'. For each type τ, there is a strict *selection operation, sel*: $(\mathit{Bool}, \tau, \tau) \to \tau$.

$$\mathit{sel}(b, v_1, v_2) = \begin{cases} v_1 & \text{if } b \text{ is } \mathsf{true} \\ v_2 & \text{if } b \text{ is } \mathsf{false} \\ \#_\tau & \text{if } b \text{ is } \#_{\mathsf{Bool}} \end{cases}$$

Finite product and projection operations are assumed. Projections are denoted by sans seriff adjectives, 1st, 2nd, 3rd, 4th, 5th ..., ith, ... Finally, it is assumed that arbitrary finite ordered sets of *tokens* can be represented in the basis; furthermore, tokens can be tested for equality. Given that products of *Bool* can be formed, one possible representations for tokens follows.

A language of applicative *terms* is built from the base vocabulary. The usual treatment of terms is extended for multioutput operations.

Definition 1 A *simple term* is either a constant, a (typed) variable, or expresses the application, $f(T_1, \ldots, T_n)$, of an operation f to the terms T_i according to the f's signature. An application without an operation symbol, $(t_1, \ldots, t_n)$, denotes the formation of an n-tuple.

Take the semantics of this language to be an initial algebra, represented by the terms of the language itself [7]. Terms are partitioned into by equivalence relation, written $S \equiv T$, according to equational laws of the basis. The laws of particular bases are not considered in this paper, but certain laws hold in general. Selections reduce according to the definition of *sel* given above. For projections,

$$i\mathsf{th}(T_1, \ldots, T_i, \ldots T_n) \equiv T_i$$

Additional identities are introduced later, in Section 3. The definition of substitution on terms is adapted for multioutput operations by allowing nested lists of variables to serve as substitution patterns. Such a list is called an *identifier*.

Definition 2 An *identifier* is either a variable or a nested list, $(X_1, \ldots, X_n)$, of *distinct* identifiers, meaning that they share no common variables.

Definition 3 The formula $T[R/X]$ denotes a *substitution* of the term R for the identifier X in the term T. The formula $T[R_1/X_1, \ldots, R_n/X_n]$ denotes the simultaneous and respective substitutions of terms R_i for identifiers X_i, $i \in \{1 \,..\, n\}$. Substitution is defined by induction on the language of terms. In the base cases, constants are unchanged and for a variable symbol u,

$$u[R/X] = \begin{cases} R & \text{if } X = u \\ u & \text{if } X \neq v \end{cases}$$

For applications and n-tuples,

$$f(T_1, \ldots, T_n)[R/X] = f(T'_1, \ldots, T'_n)$$

where $T'_i = T_i[R/X]$ for $i \in \{1 \,..\, n\}$ For nested identifiers, a simultaneous substitution is done on the constituents:

$$T[R/(X_1, \ldots, X_n)] = T[1\mathsf{st}(R)/X_1, \ldots n\mathsf{th}(R)/X_n]$$

In the last case, substitution of an n-tuple for an n-element identifier simplifies to

$$T[(R_1, \ldots, R_n)/(X_1, \ldots, X_n)] = T[R_1/X_1, \ldots R_n/X_n]$$

Hierarchic decomposition is expressed by the formation of closed combinations.

Definition 4 A *combination* is a fully parameterized term, written $\lambda X\,.\,T$ where each variable the term T is found in the identifier X. An *instance* of a combination is a term, $T[R/X]$, where R is a term.

Combinations may be nested, but may not be recursive. Where combinations are given names, it is customary write $\mathit{Name}(X) \stackrel{\text{def}}{=} T$ rather than $\mathit{Name} = \lambda X\,.\,T$. The language of terms is extended to include applications of named combinations, such as $\mathit{Name}(R)$.

A *selection combination* encapsulates the decision structure of a nested selection. That is, selection combinations contain only constants, variables, and *sel* operations. A *case expression* is a selection combination associated with a set of tokens. If $A = \{c_1, \ldots, c_n\}$ is a token set, then the form

$$\begin{array}{ll} \mathit{case}\ R\ \mathit{of} & c_1\colon T_1 \\ & \vdots \\ & c_n\colon T_n \end{array}$$

abbreviates a combination which selects T_i where R is (or reduces to) the token c_i.

A *system description* is a collection of equations describing a network. The defining expressions are composed of terms over a given basis plus a notation of delay, expressed by the infix symbol, '!'.

Definition 5 A *signal expression* is either a term or has the form $v\ !\ S$, where v is a simple term—usually a constant or a variable—and S is a signal expression. A *system description* is a family of equations of the form

$$\begin{array}{rcl} X_1 & = & S_1 \\ & \vdots & \\ X_n & = & S_n \end{array}$$

where each X_i is a distinct identifier and each S_i is a signal expression over the variables in $X_1, \ldots, X_n$.

No interpretation of system descriptions may depend on the textual order of the equations; hence, a system description may be considered as a family of equations $\{X_i = S_i\}_{i \in I}$, for finite index set I. Two interpretations are used in this paper.

Under a *schematic interpretation*, each occurence of a term is associated with a drawing. Operations are shown as boxes connected to their arguments by lines. The signal identifiers name some of these lines, and feedback cycles are allowed. A delay element, sometimes figuratively called a 'register,' is associated with every delay-expression. Figure 1 contains schematics for several of the partial system descriptions discussed in Section 4. Schematics are in a class of interpretations that Boute calls *structural semantics* [1]. Meaning is in close correspondence to syntax and any transformation applied to a description changes its interpretation.

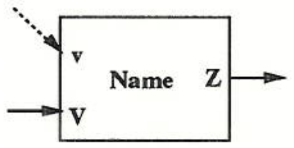
Under a *sequential interpretation*, each signal expression S denotes a uniformly typed sequence of values, $\langle S^0, S^1, \ldots\rangle$. This is a *behavioral semantics* in Boute's terminology. A constant c denotes the constant sequence, $\langle c, c, \ldots\rangle$. The expression $v \; ! \; S$ denotes the sequence $\langle v, S^0, S^1, \ldots\rangle$. Operations are applied element-by-element. If f has signature $f\colon (\sigma_1, \ldots, \sigma_m) \to (\tau_1, \ldots, \tau_n)$, then $f(S_1, \ldots, S_m)$ denotes an n-tuple of sequences, $(T_1, \ldots, T_n)$, such that for all natural numbers i, $f(S_1^i, \ldots, S_m^i) = (T_1^i, \ldots, T_m^i)$. The signal equations of a system description are simultaneously defined; that is, for $k \in \{1 \mathinner{..} n\}$, and for all $i \geq 0$, $X_k^i = S_k^i$ when X_k is a simple variable. A nested identifier is bound to the corresponding projection of its defining expression.

For the sequential interpretation of a system description to be well defined, it is sufficient for each feedback cycle to contain a delay element. For instance, in the system consisting of the single equation, $X = f(X)$, the sequence X is nowhere defined, while in $X = v \; ! \; f(X)$, X^i is, by induction, defined for all i. A system description is said to be *well formed* when it has this no-combinational-feedback property, which is essentially the criterion for clocked digital realizations (e.g. in [9]). The synthesis method developed in [12] guarantees well formedness and the algebra to be discussed preserves it.

The use of combinations freely commutes with the sequential interpretation; hence, invocations of named combinations are allowed in signal definitions. Let $\mathcal{S}[\![T]\!]$ denote the sequential interpretation of a simple term; and let $\mathcal{D}[\![T]\!]$ denote its interpretation as a value. Then for any terms T, R, identifier X, and for all $i \geq 0$, $\mathcal{S}[\![T[R/X]]\!]^i = \mathcal{D}[\![T[\mathcal{S}[\![R]\!]^i/X]]\!]$. In [12] this fact is proved by reducing the language of terms to a set of combining functionals for composition, construction, and projection and showing that each of these functionals commutes the formation of sequences.

With extensions for substitution, subsystems with delay elements may be incorporated in *sequential combinations.* A parameterized system typically has the form

$$\begin{aligned} &Name(v,V) \overset{\text{def}}{=} Z \\ &\quad where \\ &\qquad X_1 = S_1 \\ &\qquad\quad \vdots \\ &\qquad X_n = S_n \end{aligned}$$



The syntax reflects the fact that system descriptions are just a dialect of locally recursive binding expression [10]. The parameter may include an identifier for initial register values (v) and an identifier for input signals (V). In addition there is an output identifier (Z) telling which signals are externally available. Since this is a functional notation, all signals have direction. However, a parameterized system with no delay elements reduces to a multiple output term combination, since there can be no feedback. Both combinations and parameterized systems are depicted in schematics as boxes, containing the name of the combination and labeled input and output ports. Where possible the flow in schematics is left-to-right, top-to-bottom.

3. Elementary Laws of Behavior

The derivation algebra manipulates system descriptions in ways that preserve behavior while improving some structural aspect. It has already been stipulated that the equations of a system may be written down in any order. Hence, the laws are presented as local transformations on the affected signals. The symbol $\overset{C}{\Longleftrightarrow}$ indicates that a transformation is subject to condition C. The first law states that signals can be added or removed provided the well formedness conditions and external functionality are preserved.

> Signal Introduction and Elimination *The signal equation $Y = R$ may be added to (deleted from) a system description under conditions for signal introduction (elimination).*

$$\begin{array}{lcl} SD(v,V) \overset{\text{def}}{=} Z \ where & & SD(v,V) \overset{\text{def}}{=} Z \ where \\ \quad X_1 = S_1 & \overset{(1)}{\Longrightarrow} & \quad X_1 = S_1 \\ \quad\quad \vdots & \overset{(2)}{\Longleftarrow} & \quad\quad \vdots \\ \quad X_n = S_n & & \quad X_n = S_n \\ & & \quad Y = T \end{array}$$

For signal introduction (1) Y must be distinct from identifiers $\{U, X_1, \ldots X_n\}$

and must introduce no combinational feed back. For signal elimination (2) the variables in Y must not occur in any defining expression other than T.

The next law deals with the manipulation of common expressions.

IDENTIFICATION *Signal identifiers can be exchanged with their defining expressions wherever these occur.*

$$\begin{array}{c} \vdots \\ X = S \\ Y = T \\ \vdots \end{array} \quad \Longleftrightarrow \quad \begin{array}{c} \vdots \\ X = S[T/Y] \\ Y = T \\ \vdots \end{array}$$

Using introduction, elimination, and identification, we can replace an expression that occurs several places by a name, whose definition is the common expression. The structural effect is to reduce the number of replicated components. A tactical step in many transformations is to form signals into groups. For example, grouping may be a prelude to simultaneously eliminating a collection of unused signals. Grouping is also a first step toward forming multioutput combinations.

GROUPING *A subsystem is grouped by placing its signal definitions in a nested identifier and their corresponding equations respectively in a list.*

$$\begin{array}{rcl} & \vdots & \\ X_i & = & S_i \\ X_{i+1} & = & S_{i+1} \\ & \vdots & \\ X_{i+m} & = & Si + m \\ & \vdots & \end{array} \quad \Longleftrightarrow \quad \begin{array}{c} \vdots \\ (X_i, \ldots, X_{i+m}) = (S_i, \ldots, S_{i+m}) \\ \vdots \end{array}$$

In applying this transformation from right to left, if the defining expression is not an n-tuple, it may be rewritten as a product of projections, $(\mathsf{1st}(S), \ldots, m\mathsf{th}(S))$, under replacement law:

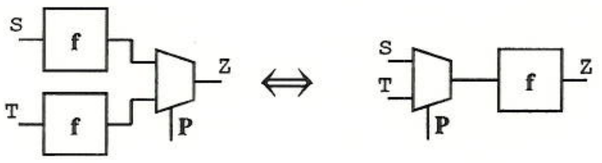
REPLACEMENT *Any term may be replaced by an equivalent one.*

$$\begin{array}{c} \vdots \\ X = R[S/X] \\ \vdots \end{array} \quad \overset{S \equiv T}{\Longleftrightarrow} \quad \begin{array}{c} \vdots \\ X = R[T/X] \\ \vdots \end{array}$$

Two signal expressions are equivalent if they are composed of equivalent terms. Therefore, replacement applies to signal expressions as well as terms. Of central interest are rewritings under the universal base laws, and in particular, manipulation of selection terms. First, the *sel* operation obeys a distributive law for strict operations:

$$sel(P, f(S), f(T)) \quad \equiv \quad f(sel(P, S, T))$$

Expressed with schematics, decisions "push through" operations:



Since *sel* also distributes over products, this equation is sufficient to characterize distribution over complicated signatures. This form of optimization is commonly used in in silicon compilation [5]. However, it is an uncommon variation that is fundamental to factorization. In functional algebra, distribution applies to operations and combinations as well to arguments:

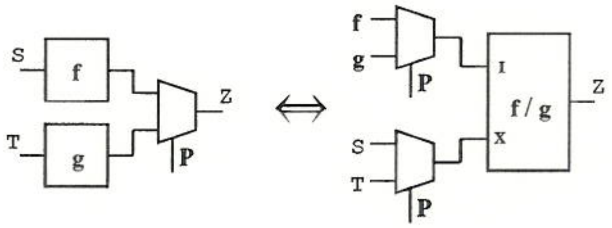
$$sel(P, f(S), g(T)) \cong (sel(P, f, g))(sel(P, S, T))$$

Under first-order restrictions, "$sel(P, f, g)$" is not permitted; however, an implementation of this expression is possible. Define a combination that is capable of doing either f or g, depending on an *instruction token*, $c \in \{\mathsf{f}, \mathsf{g}\}$:

$$\begin{aligned} FG(i, x) \overset{\text{def}}{=} \textit{case } i \textit{ of } & \mathsf{f}\text{: } f(x) \\ & \mathsf{g}\text{: } g(x) \end{aligned}$$

The second order identity can now be expressed as,

$$sel(P, f(S), g(T)) \quad \equiv \quad FG(sel(P, \mathsf{f}, \mathsf{g}), sel(P, S, T))$$



The aim is to encapsulate a collection of operations in a single module for which instructions are generated. This form of distribution also applies to operations of different sorts. For example, suppose that $f\colon(\alpha,\beta) \rightarrow \delta$ and $g\colon(\alpha,\gamma) \rightarrow \delta$. A combination must be formed with enough arguments of enough types to simulate both f and g. The specific form of the combination will depend on implementation factors, but here is one possibility: with parameters $a\colon\alpha$, $b\colon\beta$, and $c\colon\gamma$,

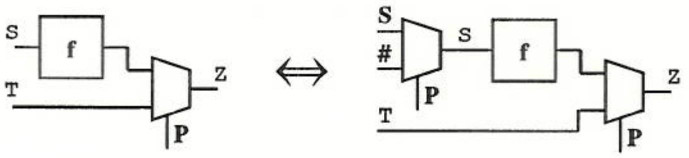
$$FG(i,a,b,c) \overset{\text{def}}{=} \mathit{case}\ i\ \mathit{of}\ \begin{array}{l}\mathsf{f}\colon f(a,b)\\ \mathsf{g}\colon g(a,c)\end{array}$$

As before, *sel* distributes to get

$$\begin{array}{rcl} \mathit{sel}(P, f(A,B), g(A',C)) & \equiv & FG(\mathit{sel}(P,\mathsf{f},\mathsf{g}), \\ & & \quad \mathit{sel}(P,A,A'), \\ & & \quad \mathit{sel}(P,B,\#), \\ & & \quad \mathit{sel}(P,\#,C)) \end{array}$$

This is one way in which don't-care values (#s) are safely introduced to system descriptions. Under the condition P the c input to FG does not matter, so long as it is a γ. Another way of introducing #s comes from replicating selections, according to rewriting rules like

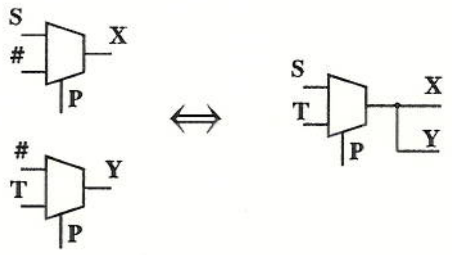
$$\mathit{sel}(P, f(S), T) \quad \equiv \quad \mathit{sel}(P, f(\mathit{sel}(P,S,\#)), T)$$



So long as it is assured that # really means "don't care," and not "don't know," selections simplify according to

$$\begin{array}{rcl} \mathit{sel}(P,S,\#) & \rightarrow & S \\ \mathit{sel}(P,\#,T) & \rightarrow & T \end{array}$$

However, there are many occasions when it is preferable to instantiate #s as suggested by



This kind of transformation is the subject of the final law for rewriting system descriptions.

COLLATION *If signal expressions S and T are of the same type, then*

$$\begin{array}{lcl} \vdots & & \vdots \\ X = sel(P, S, \#) & \overset{S,T:\tau}{\Longleftrightarrow} & X = sel(P, S, T) \\ Y = sel(P, \#, T) & & Y = X \\ \vdots & & \vdots \end{array}$$

In the next section, collating is done by introducing a distinct new identifier, $XY = sel(P, S, T)$ and using it in place of X and Y throughout the system.

4. System Factorizations – Four Examples

In essence, the result of a factorization is simply the introduction of a combinator. The transformation problem is to isolate the expression(s) to be combined. Four examples illustrate the tactical issues. The first three deal with purely combinational terms; the last encapsulates a sequential combination. Figure 1 shows schematics for selected system descriptions in this section.

Derivation of hardware centers on the dependent tasks of control synthesis, refinement of architecture, and structuring for physical realization. System factorization addresses the second of these tasks in interaction with the others. Often, it is not possible to obtain a desired architecture without refinement to control; sometimes, the intended physical organization, the 'target technology', guides the imposition of a conceptual hierarchy. The examples are intended to develop a sense of this interplay.

4.1. Example One

The names of the base operations do not matter but are chosen to be evocative. Assume a single type int, with operations $add\colon (\mathsf{int}, \mathsf{int}) \to \mathsf{int}$ and $dn, up\colon \mathsf{int} \to \mathsf{int}$, and consider the system description fragment

$$\begin{array}{ll} & \vdots \\ U & = sel(P, add(A, dn(B)), up(C)) \\ X & = sel(P, dn(D), add(E, F)) \\ & \vdots \end{array} \qquad (1.0)$$

The first derivation goal is to combine all applications of *add*. This is specified by listing a set of combinations to serve as patterns. The instances of these abstractions are called the *subject terms* of the factorization. In this case, the specification set is $\{\lambda uv \,.\, add(u,v)\}$ and the subject terms are $add(E,F)$ and $add(A,w)$, where $w \equiv dn(B)$. A subtask is to isolate the subject terms from the surrounding equations. After distributing selection in signals U and X, the isolated terms can be identified as signals V and Y.

$$\begin{aligned} U &= sel(P, V, up(C)) \\ W &= sel(P, dn(B), \#) \\ X &= sel(P, dn(D), Y) \\ V &= sel(P, add(A, W), \#) \\ Y &= sel(P, \#, add(E, F)) \end{aligned} \tag{1.2}$$

It is important to maintain the subject terms in the context of their use, as is done above. Signals V and Y are now collated into a single signal, VY.

$$\begin{aligned} U &= sel(P, VY, up(C)) \\ W &= sel(P, dn(B), \#) \\ X &= sel(P, dn(D), VY) \\ VY &= sel(P, add(A, W), add(E, F)) \end{aligned} \tag{1.3}$$

Using distribution again, the defining expression for VY is reduced to obtain a single *add* component.

$$\begin{aligned} U &= sel(P, VY, up(C)) \\ W &= sel(P, dn(B), \#) && (1.4) \\ X &= sel(P, dn(D), VY) && (2.0) \\ VY &= add(sel(P, A, E), sel(P, W, F)) \end{aligned}$$

The schematic for (1.4) in Figure 1 simplifies the signal W to $dn(B)$. However, W is left as shown for the resumption of this derivation in Example Two.

As in this example, the factorization process has three phases. First, subject terms are isolated and, if necessary, identified. Next, the subject terms are collated into a reduced collection of expressions. Last, tests are distributed to arguments. The next example shows one way that things can go wrong.

4.2. Example Two

System (1.4) is used as a starting point for this example. Since the form of selection expressions does not change, it makes things a bit clearer to use an abbreviation. Henceforth, the term $sel(P, S, T)$ is written instead as

$$sel_P(S, T)$$

Let us now attempt to factor applications of *dn* and *up* from system (2.0). The subject terms are instances of the combinations $\{\lambda x\,.\,dn(x), \lambda x\,.\,up(x)\}$. As before, they are isolated in context as signals Z and T:

$$\begin{aligned} U &= sel_P(VY, T) \\ X &= sel_P(Z, VY) \\ W &= sel_P(dn(B), \#) \\ Z &= sel_P(dn(D), \#) \\ T &= sel_P(\#, up(C)) \\ VY &= add(sel_P(A, E), sel_P(W, F)) \end{aligned} \tag{2.1}$$

The intended factorization fails because it is impossible to collate the equations for W, Z, and T. There is a *clash* of $dn(B)$ and $dn(D)$. However, these signals can be partitioned into groups that can be collated. With the partition $\{W, T\}$ and $\{Z\}$ we get

$$\begin{aligned} U &= sel_P(VY, WT) \\ X &= sel_P(Z, VY) \\ Z &= sel_P(dn(D), \#) \\ WT &= sel_P(dn(B), up(C)) \\ VY &= add(sel_P(A, E), sel_P(WT, F)) \end{aligned} \tag{2.2}$$

After introducing instruction tokens $\{\mathsf{dn}, \mathsf{up}\}$, selection distributes over the combination of *dn* and *up*. Define the combination *UPDN* to be

$$\begin{aligned} UPDN(i, v) \overset{\text{def}}{=} case\ i\ of\ &\mathsf{dn}\text{: } dn(v) \\ &\mathsf{up}\text{: } up(v) \end{aligned}$$

Replace the defining equation for WT by an instance of $UPDN$. In (2.3), signal Z is also simplified.

$$\begin{aligned} U &= sel_P(VY, WT) \\ X &= sel_P(Z, VY) \\ Z &= dn(D) \\ I &= sel_P(\mathsf{dn}, \mathsf{up}) \\ WT &= UPDN(I, sel_P(B, C)) \\ VY &= add(sel_P(A, E), sel_P(WT, F)) \end{aligned} \tag{2.3}$$

In the combination *UPDN*, the decisions of the surrounding system, based on the test P, are encoded by the instruction signal I. This is the essence of factorization—it is usually a goal to target for problem independent combinations.

An improvement in the specification of subject terms is needed, in order to avoid clashes. A convention that almost always works is to use match free variables in the pattern abstractions to identifiers in the given network. For example, the specification set $\{dn(B), \lambda x \,.\, up(x)\}$ determines the factorization in (2.3). This technique is used in the remaining examples.

4.3. Example Three

Let us follow an alternative derivation from system (1.0), which is repeated below.

$$\begin{aligned} U &= sel_P(add(A, dn(B)), up(C)) \\ X &= sel_P(dn(D), add(E, F)) \end{aligned} \tag{3.0}$$

The first subgoal is to factor $\{up(C), \lambda u.add(A, u)\}$. Isolating the instances yields

$$\begin{aligned} W &= sel_P(dn(B), \#) \\ X &= sel_P(dn(D), add(E, F)) \\ U &= sel_P(add(A, W), up(C)) \end{aligned} \tag{3.1}$$

In order to distribute selection in U, we must first determine how to superimpose the arguments (A, W) and (C). A, C, and W are all ints, so there are many possibilities. Suppose that the following combination is given:

$$XALU(i, a, b) \overset{\text{def}}{=} case\ i\ of\ \begin{aligned}[t] \mathsf{add}&: add(a, b) \\ \mathsf{dn}&: dn(a) \\ \mathsf{up}&: up(b) \end{aligned}$$

With this component as a target, selection distributes in signal U as shown below.

$$\begin{aligned} U &= XALU(U_I, U_A, U_B) \\ U_I &= sel_P(\mathsf{add}, \mathsf{up}) \\ U_A &= sel_P(A, \#) \\ U_B &= sel_P(W, C) \\ W &= sel_P(dn(B), \#) \\ X &= sel_P(dn(D), add(E, F)) \end{aligned} \tag{3.2}$$

Similarly, the set $\{dn(B), add(E,F)\}$ factors as signal Z in system (3.3):

$$\begin{aligned}
U &= XALU(U_I, U_A, U_B)\\
U_I &= sel_P(\mathsf{add}, \mathsf{up})\\
U_A &= sel_P(A, \#)\\
U_B &= sel_P(Z, C)\\
X &= sel_P(dn(D), Z)\\
Z &= XALU(Z_I, Z_A, Z_B)\\
Z_I &= sel_P(\mathsf{dn}, \mathsf{add})\\
Z_A &= sel_P(\#, E)\\
Z_B &= sel_P(B, F)
\end{aligned} \tag{3.3}$$

Depending on the surrounding circumstances, signals U_A and Z_A might be collated or simplified to A and E (as they are in Figure 1). As a general stratagem, it is better to defer simplification to the last stages of derivation. Wolf has suggested leaving this kind of optimization entirely to automatic logic synthesizers [23], and our experience bears this out.

4.3. Example Four

This example illustrates encapsulating sequential components. Again the vocabulary is suggestive but arbitrary. Assume three types, mem, addr, and data; with operations $rd : (\mathsf{mem}, \mathsf{addr}) \rightarrow \mathsf{data}$ and $wt : (\mathsf{mem}, \mathsf{addr}, \mathsf{data}) \rightarrow \mathsf{mem}$; and consider the system

$$\begin{aligned}
&\vdots\\
M &= m^0 \;!\; sel_P(M, wt(M,X,Y), M)\\
N &= sel_P(rd(M,U), V, W)\\
&\vdots
\end{aligned} \tag{4.0}$$

The form $sel_P(\star,\star,\star)$ now stands for a three-alternative selection combination. Since the goal is to hide the abstract signal M: mem from the surrounding system, this kind of transformation has been called an 'abstract component' factorization in [12, 10], and later a 'signal factorization' in [13]. The specification set is $\{M, \lambda u \,.\, rd(M,u), \lambda u\, v.wt(M,u,v)\}$. With knowledge of how arguments to rd and

wt should be superimposed, the subject terms in (4.0) are isolated as follows:

$$\begin{aligned} M &= m^0 \,!\, sel_P(M, wt(M, M_A, M_D), M) \\ M_R &= sel_P(rd(M, M_A), \#, \#) \\ M_A &= sel_P(U, X, W) \\ M_D &= sel_P(\#, X, \#) \\ N &= sel_P(M_R, V, W) \end{aligned} \tag{4.1}$$

With instruction tokens $i \in \{\mathsf{id}, \mathsf{rd}, \mathsf{wt}\}$ a *MEMORY* combination is formed, with definition

$$\begin{aligned} &MEMORY(m, I, A, D) \overset{\text{def}}{=} R \\ &\qquad where \\ &\qquad (R, M) = (\#, m^0) \,!\, case\ i\ of\ \mathsf{id} : (\#, M) \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{rd} : (rd(M, A), M) \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{wt} : (\#, wt(M, A, D)) \end{aligned}$$

With the *MEMORY* combination, System description (4.1) becomes

$$\begin{aligned} M &= MEMORY(m^0, M_I, M_A, M_D) \\ M_I &= sel_P(\mathsf{rd}, \mathsf{wt}, \mathsf{id}) \\ M_A &= sel_P(U, X, W) \\ M_D &= sel_P(\#, X, \#) \\ N &= sel_P(M, V, W) \end{aligned} \tag{4.2}$$

According to the specification for *MEMORY*, it would be all right in a tightly coupled implementation to use a conventional read/write memory for M, but the id instruction must be eliminated. One approach is rewriting the signal M_R in system (4.1) to

$$M_R = sel_P(rd(M, M_A), \#, rd(M, M_A))$$

It follows that only two distinct instructions are needed. An alternative is to deduce from the *MEMORY* combination that id = rd in representation. In either case, the derivation that encapsulates M must reflect the implementation intent.

In this example, a sequential combination is encapsulated with the effect of factoring off some of the original description's state. The tactics happen to be specialized for a known component, but supposing that the purpose of the factorization was simply to suppress mention of the type mem, how should we regard the combination *MEMORY*? One way of looking at it is to consider *MEMORY* as an expression of correct peripheral behavior. In realization, the *MEMORY* process must respond to instructions rd and wt as *specified* by the combination. Otherwise, (4.2) would not behave correctly. From this point of view, factorization has synthesized a verification condition from a higher level of description.

In any case, the goal of this factorization has been accomplished. System (4.2) makes no mention (other than m^0, which could also be abstracted) of the type mem.

5. Conclusions

A semiautomated transformation system has been implemented to explore digital design derivation. Designs are constructed from a restricted class of functional control specifications. Factorizations are applied to the resulting system descriptions in order to develop a logical organization. Then, binary representations are incorporated for concrete symbolic types, and the description is restructured for logic synthesis. A number of moderate designs have been derived, including several versions of a garbage collector, a prototype computer system based on an SECD processor, and several conventional processor designs. Designs have been realized in MSI and VLSI, using programmable target technologies. The examples in Section 4 are distilled from the design of a garbage collector [13]. In that derivation, a series of eight factorizations was needed to develop a logical orgainization. This is typical of designs we have done. Selection combinations, which model controllers, have between 30 and 100 alternatives in these designs.

A derivation is entered as a sequence of transformation commands, which is applied to the initial design description. The designer looks at intermediate descriptions in order to evaluate the derivation. If things are not going as planned, the derivation script is modified. The factorizations provided by the system are substantially the same as the examples illustrate. The factorization algorithm takes a specification of subject terms, isolates them, and then attempts to form a combination. Intervention is required if there are clashes. Arguments are superimposed according to a convention that is adequate at lower levels of description.

The experimentation shows that modularity can often be established later in the design process. To a degree, it is possible to specify a design without declaring its architecture beforehand. However, at least two kinds of problems arise. The most frequent involves clashes in collation. Methods for resolving clashes take many forms, ranging from trial and error to refinement of the control description

[14, 2]. Another kind of problem arises in information hiding factorizations of the kind illustrated in Example Four. If the intent is to encapsulate an abstract type, inference must be used to determine the subject terms. An example of this problem is discussed in [13].

Derivation is a formal method that can be likened to direct verification. The fundamental laws correspond to the logical machinery of a theorem proving system. It is the task of the designer to engineer an 'equational' proof that the implementation has the behavior of its specification. Factorizations correspond to lemmas, introduced to guide the proof process, and should be reducible to basic laws.

At the same time, factoring is a technique for suppressing facets of design that are not accounted for in a realization. A factored combination represents "the rest of the system," stating how the environment must behave to use the implementation correctly. Thus, factorizations can be viewed as a means of integrating design synthesis and formal verification. It is this aspect that motivates the refinement of their formal foundations.

References

[1] RAYMOND BOUTE, "Systems Semantics: Principles, Applications, and Implementations," *ACM Transactions on Programming Languages and Systems* **10**(1):188–155 (1988).

[2] C. DAVID BOYER AND STEVEN D. JOHNSON, "A Derived Garbage Collector," *Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages* (1989), to appear.

[3] PAOLO CAMUARATI AND PAOLO PRINETTO, "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research," *Computer* **21**(7):8–19 (1988).

[4] A.L. DAVIS, "What Do Computer Architects Design Anyway?" (Preliminary papers of the 1988 Banff Hardware Verification Workshop).

[5] DANIEL D. GAJSKI, *Silicon Compilation,* Addison-Wesley, Reading, 1987.

[6] GANESH C. GOPALAKRISHNAN, RICHARD M. FUJIMOTO, AND KEVIN SMITH, "Specification Driven Design of Custom Architectures in HOP, (Preliminary papers of the 1988 Banff Hardware Verification Workshop).

[7] 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 (ed.) *Current Trends in Programming Methodology* Vol. IV, Prentice Hall, 1978, 80–145.

[8] H.A. Harman and J.V. Tucker, "Clocks Retimings, and the Formal Specification of a UART," in G. J. Milne (ed.), *The Fusion of Hardware Design and Verification,* North-Holland, Amsterdam, 1988, 375–396 (Proceedings of the IFIP WG 10.2 Working Conference, Glasgow, 1988).

[9] Fredrick J. Hill and Gerald R. Peterson, *Introduction to Switching Theory and Logical Design,* (3rd ed.) John Wiley and Sons, New York, 1981.

[10] Steven D. Johnson, "Applicative Programming and Digital Design", *Eleventh Annual ACM Symposium on Principles of Programming Languages* (1984) 218–227.

[11] Steven D. Johnson, "Digital Design in a Functional Calculus," in: G.J. Milne and P.A. Subrahmanyam (eds.), *Formal Aspects of VLSI Design*, Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1986 (Proceedings of the 1985 Workshop on VLSI, Edinburgh).

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

[13] Steven D. Johnson, Bhaskar Bose, and C. David Boyer, "A Tactical Framework for Digital Design," in *VLSI Specification, Verification and Synthesis,* (eds.) Graham Birtwistle and P.A. Subramanyam, Kluwer Academic Publishers, 1987, 349–384 (proceedings of the 1987 Calgary Hardware Verification Workshop).

[14] Steven D. Johnson, Bhaskar Bose and Robert W. Wehrmeister, "On the Interplay of Hardware Derivation and Hardware Verification: Experiments with the FM8501 Microprocessor Description," submitted.

[15] Jacques Loeckx and Kurt Sieber, *The Foundations of Program Verification,* John Wiley and Sons, Chichester, 1984.

[16] George J. Milne, "CIRCAL and the Representation of Communication, Concurrency, and Time," *ACM Transactions on Programming Languages and Systems* **7**:270–298 (1985).

[18] Mary Sheeran, "Retiming and Slowdown in Ruby," in G. J. Milne (ed.), *The Fusion of Hardware Design and Verification,* North-Holland, Amsterdam, 1988, 289–308 (Proceedings of the IFIP WG 10.2 Working Conference, Glasgow, 1988).

[19] Mary Sheeran, *μFP, and Algebraic for VLSI Design,* D. Phil. Thesis, Programming Research Group Monograph PRG–39, Oxford University, 1983.

[20] P.A. Subrahmanyam, "Contextual Constraints, Temporal Abstraction and Observational Equivalence in VLSI Design," in G. J. Milne (ed.), *The Fusion of Hardware Design and Verification,* North-Holland, Amsterdam, 1988, 159–R184 (Proceedings of the IFIP WG 10.2 Working Conference, Glasgow, 1988).

[21] Franklin P. Prosser and David E. Winkel, *The Art of Digital Design* (2nd ed.) Prentice-Hall, Englewood Cliffs, 1987.

[22] GLYNN WINSKEL, "A Compositional Model of MOS Circuits," in *VLSI Specification, Verification and Synthesis,* (eds.) Graham Birtwistle and P.A. Subramanyam, Kluwer Academic Publishers, 1987, 323–347 (proceedings of the 1987 Calgary Hardware Verification Workshop).

[23] WAYNE WOLF, personal communication.

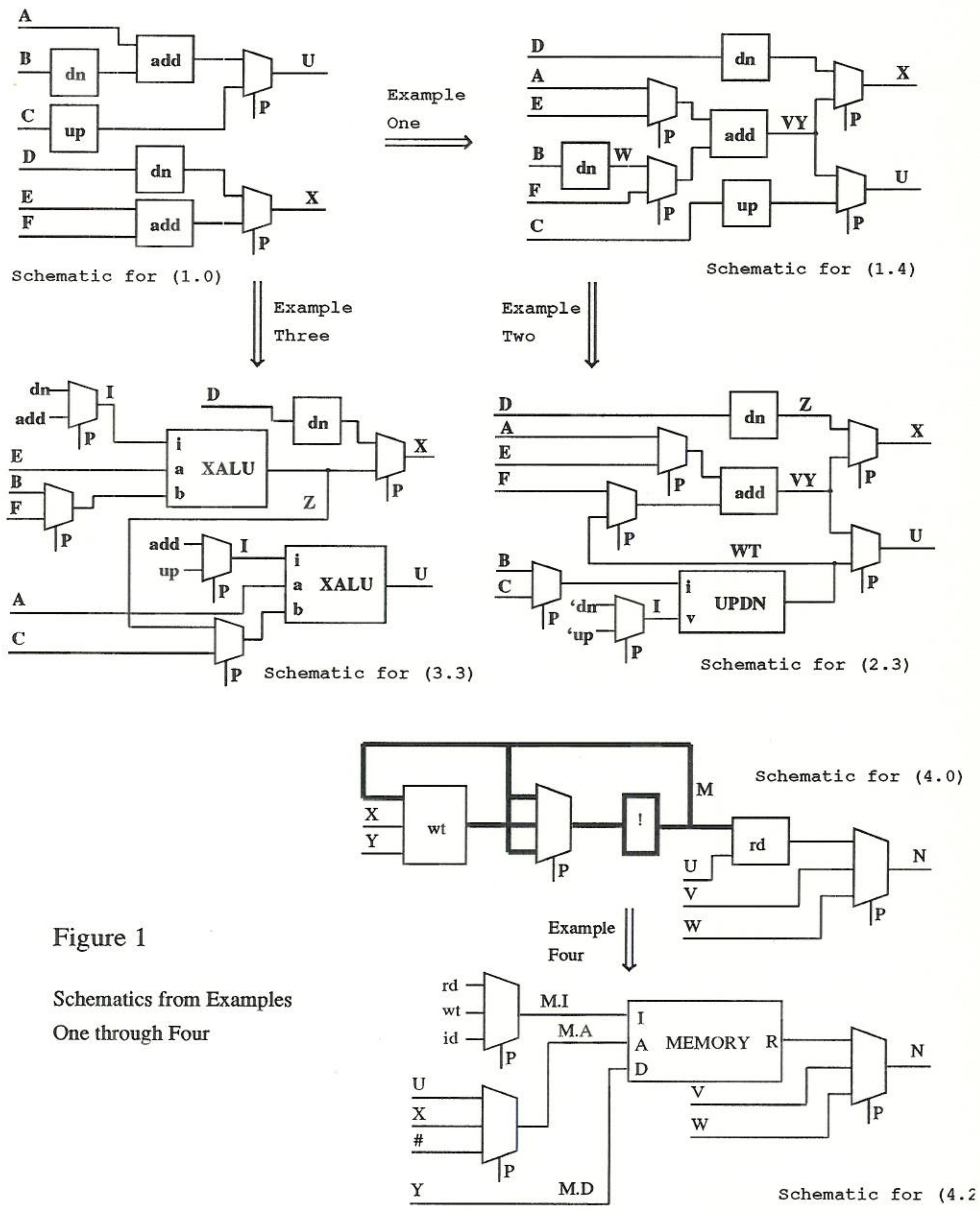


Figure 1

Schematics from Examples One through Four