Visualizing System Factorizations with Behavior Tables*

Alex Tsow and Steven D. Johnson

System Design Methods Laboratory
Computer Science Department
Indiana University
atsow@cs.indiana.edu

Abstract. Behavior tables are a design formalization intended to support interactive design derivation for hardware and embedded systems. It is a reformulation of the DDD transformation system, bridging behavioral and architectural forms of expression. The tabular representations aid in visualizing design aspects that are subject to interactive refinement and optimization. These ideas are illustrated for system factorization, an import class of decompositions used in design derivation. A series of examples shows how features seen in the behavior tables determine the course of a factorization and how the rules of a core behavior table algebra compose into the more large scale transformations done at the interactive level.

1 Introduction

We are developing a formal reasoning system for synchronous system design based on *behavior table* notation and equivalence preserving transformations. This project stems from research embodied in the *DDD transformation system*, an interactive design derivation system based on functional algebra. Behavior tables have been used informally with DDD for a long time. This experience, together with new tools and results in requirements analysis, has motivated us to look at behavior tables as a formal basis for design derivation, that is, as the object of formal manipulation, rather than merely a display notation.

This paper illustrates how behavior tables help in visualizing design transformations. We focus on a particular class of transformations we call *system factorizations*, which are very important in DDD-style derivations. A system factorization is a kind of architectural decomposition, breaking one system description into two or more connected subsystems. The term "factorization" is appropriate because the decomposition involves a distributivity law for conditional expressions [6]. Applications of system factorization range from simple component isolation, analogous to *function allocation* in high level synthesis, to abstract data-type encapsulations, analogous to `architecture` declarations in VHDL.

* This research is supported, in part, by the National Science Foundation under Grant MIP9610358.

In our practice, behavior tables are an intermediate notation. They come into play as the designer is developing an architecture for a behavioral specification. The tables are poor at expressing either algorithmic behavior or architectural organization. Their value lies just in their neutrality between control and structure dominated views of the design.

Interactive visualization is enhanced by the use of color. In the printed version of this paper we use background shading to indicate features that would be colored by a graphical tool. The electronic version [1] includes colorization.

The behavior table for a JK flip-flop with synchronous preset, right, illustrates the most important syntactic features. The primary interpretation is that of a synchronous sequential process, mapping input streams (infinite sequences) to output streams. In this case, for boolean type B, $JK\colon \mathsf{B}^\infty \times \mathsf{B}^\infty \times \mathsf{B}^\infty \to \mathsf{B}^\infty \times \mathsf{B}^\infty$, as indicated in the topmost bar of the table. The three left-hand columns comprise a *decision table* listing all the cases for inputs P, J, and K; $\natural$ is a *don't care* symbol.

JK:(J,K,P) → (Q,$\underline{\mathtt{R}}$)				
P	J	K	Q	$\underline{\mathtt{R}}$
0	$\natural$	$\natural$	1	$\overline{\mathtt{Q}}$
$\natural$	0	0	Q	$\overline{\mathtt{Q}}$
$\natural$	0	1	0	$\overline{\mathtt{Q}}$
$\natural$	1	0	1	$\overline{\mathtt{Q}}$
$\natural$	1	1	$\overline{\mathtt{Q}}$	$\overline{\mathtt{Q}}$

The two right-hand columns comprise an *action table* giving the values for *signals* Q and $\underline{\mathtt{R}}$. Signal names heading the action table columns are of two kinds. Q is *sequential*, denoting a state-holding abstract register; the terms in Q's column specify its next-state values. $\underline{\mathtt{R}}$ is *combinational*, (indicated by the underline); the terms in $\underline{\mathtt{R}}$'s columns specify its current-state values. That is, $\mathtt{Q}_0 = \natural$, $\underline{R}_k = \mathtt{Q}_k$ and

$$\mathtt{Q}_{k+1} = \begin{cases} 1 & \text{if } \mathtt{P}_k = 0\text{; otherwise} \\ \mathtt{Q}_k & \text{if } \mathtt{J}_k = \mathtt{K}_k = 0 \\ 0 & \text{if } \mathtt{J}_k = 0 \text{ and } \mathtt{K}_k = 1 \\ 1 & \text{if } \mathtt{J}_k = 1 \text{ and } \mathtt{K}_k = 0 \\ \overline{\mathtt{Q}_k} & \text{if } \mathtt{J}_k = \mathtt{K}_k = 1 \end{cases}$$

This example also shows that behavior tables are not efficient expressions of architecture. Signal $\underline{\mathtt{R}}$ is invariantly the negation of signal Q, as is seen in the redundant entries in its column of the action table. At the same time, the fact that this pattern is so plainly *seen* is an example of tables' value in design visualization.

After a background review in Sec. 2, Sec. 3 gives a brief syntactic and semantic outline of behavior tables and its core algebra. More details can be found in [7, 9]. In Sec. 4, we demonstrate various factorizations of a "shift-and-add" multiplier, showing how basic visualizations appear in behavior table forms. The section ends with a larger scale example, encapsulating the abstract memory component of a garbage collector. This example shows the scale at which behavior tables have proven effective for design analysis. Section 5 outlines directions for further development.

2 Background

Behavior tables arose as an informal notation used in the DDD transformation system, a formal system which represents designs as functional modeling expressions [8, 2]. In DDD, control oriented behavioral expressions are simultaneous systems of tail-recursive function definitions, each function being a control point. Architectural expressions are recursive systems of streams definitions, representing a network of subsystem connections and their observable values over time. A central step in a design derivation is a construction translating between these two modes of expression. Most transformations performed prior to this step deal with control aspects, while most performed afterward are either architectural manipulations or data refinements.

A higher level design task will involve both control and architecture. For example, in behavioral synthesis, scheduling involves control, allocation involves architecture, and register binding lies in between. Hence, in a typical design derivation, there may be a considerable distance between a behavioral manipulation such as scheduling and the architectural manipulations it *anticipates* [6].

As our case studies grew, we began printing our specifications in a tabular form in order to plan our derivation strategies. The methodology progressed, and the tables became integral to the design process, although they were still treated simply as a display format. The underlying formal manipulations were still performed on recursive expressions. In the early 1990s we began to consider basing the DDD formal system on tables rather than expressions [15, 16].

The emergence of table-based tools and notations in requirements analysis, such as *Tablewise* [5], *SCR** [4], *RSML's and-or tables* [11], and *PVS table expressions* [12], provided further encouragement. There is a good deal of evidence for tables' value in visualizing complex control problems and as a means for systematic case analysis, especially when the tool is augmented by automated analyses.

We propose a table based tool and sketch behavior table semantics in [7] and describe its core algebra in [9]. As in many formal reasoning tools, the idea is to secure a sound and near-complete kernel of basic identities from which higher level transformations can be composed. The core algebra for behavior tables corresponds to a set of five kernel structural transformations for DDD, first described by Johnson [6].

These very basic rules are too finely grained to be useful in interactive reasoning, or, for that matter, in automated synthesis. They serve instead as an basis for establishing the soundness of higher level rules. Mathematical soundness is established by the fact that higher transformations can be implemented by compositions of the core rules. Although we have not done it yet, the demonstration of this claim will be in the form of a tool built over a core transformation "engine" implementing the more complex higher level transformations.

Of course, nothing precludes the automated use of behavior tables in a *formal synthesis* fashion [10], in which an optimization algorithm drives the derivation. However, we do not know whether tables are a useful representation for that purpose. We are specifically interested in interactive aspects of design derivation

and in how the tables serve to visualize design goals and automated analyses. We would expect this tool to be used at a relatively high level in synchronous-reactive design, both for hardware and embedded software applications.

3 Behavior Table Expressions

Behavior tables are closed expressions composed of first order terms over a user specified algebraic structure (e.g. a many sorted Σ algebra). Variables come from three disjoint sets: inputs I, sequential or *register* signals S, and combinational signals C. Our notion of term evaluation is standard: the value of a term, t, is written $\sigma[\![t]\!]$, where σ is an *assignment* or association of values to variables. A behavior table has the form:

Name: *Inputs* $\rightarrow$ *Outputs*	
Conditions	*Registers and Signals*
$\vdots$	$\vdots$
Guard	*Computation Step*
$\vdots$	$\vdots$

Inputs is a list of input variables and *Outputs* is a subset of the registered and combinational variables. *Conditions* is a set of terms ranging over finite types, such as truth values, token sets, etc. The *guards* are tuples of constants denoting values that the conditions can take. The conditions together with the guards form a *decision table*. Each guard indexes a *computation step* or *action*. An action is tuple of terms, each corresponding to a combinational or registered variable. The actions and the internal signal names (i.e. non input variables) compose the *action table*.

A *behavior table* denotes a relation between infinite input and output sequences, or *streams*. Suppose we are given a set of initial values for the registers, $\{x_s\}_{s \in S}$ and a stream for each input variable in I. Construct a sequence of assignments, $\langle \sigma_0, \sigma_1 \ldots \rangle$ for ISC as follows:

(a) $\sigma_n(i)$ is given for all $i \in I$ and all n.
(b) For each $s \in S$, $\sigma_0(s) = x_s$.
(c) $\sigma_{n+1}(s) = \sigma_n[\![t_{s,k}]\!]$ if guard g_k holds for σ_n.
(d) For each $c \in C$, $\sigma_n(c) = \sigma_n[\![t_{c,k}]\!]$ if guard g_k holds for σ_n.

The stream associated with each $o \in O$ is $\langle \sigma_0(o), \sigma_1(o), \ldots \rangle$. This semantic relation is well defined in the absence of circular dependencies amongst combinational actions $\{t_{c,k} \mid c \in C,\ g_k \in G\}$. The relation is a function (i.e. deterministic) when the branches of the decision table are exhaustive and exclusive. We shall restrict our attention to behavior tables that are *well formed* in these respects. Well formedness reflects the usual quality required of synchronous digital systems, finite state machines, etc.

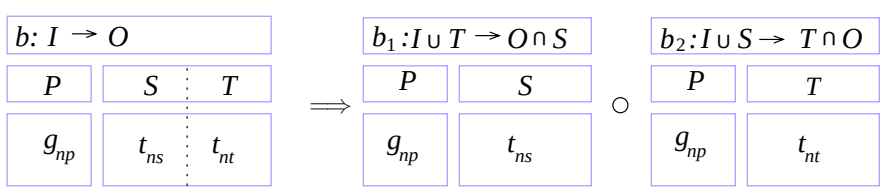
Behavior tables denote persistent, communicating processes, rather than subprocedures. Consequently behavior tables cannot themselves be entries in other behavior tables, but instead are composed by interconnecting their I/O ports. Composition is specified by giving a *connection map* that is faithful to each component's arity. In our function-oriented modeling notation, such compositions are expressed as named recursive systems of equations,

$$\begin{array}{l} \mathsf{S}(U_1,\ldots,U_n) = (V_1,\ldots,V_m) \ \textit{where} \\ \quad (X_{11},\ldots,X_{1q_1}) = \mathcal{T}_1(W_{11},\ldots,W_{1\ell_1}) \\ \qquad\qquad\qquad\vdots \\ \quad (X_{p1},\ldots,X_{pq_p}) = \mathcal{T}_p(W_{p1},\ldots,W_{p\ell_p}) \end{array}$$

in which the defined variables X_{ij} are all distinct, each $\mathcal{T}_k$ is the name of a behavior table or other composition, and the outputs V_k and internal connections W_{ij} are all simple variables coming from the set $\{U_i\} \cup \{X_{jk}\}$.

The core behavior table algebra presented in [9] has eleven rules. We shall discuss three of them here, to give a sense of what they are like. The core algebra for structural manipulation in DDD had only five rules [6], one of which was later shown by Miner to be unnecessary. There are more rules for behavior tables for several reasons. Behavior tables have decomposition and hiding rules whose counterparts in DDD are subsumed by general laws functional algebra. Behavior tables have rules that affect decision tables and action tables, while no such distinction exists in DDD's system expressions. Behavior table rules apply to isolated rows as well as columns, while the corresponding DDD rules, in affect, apply only to columns. Finally, the purpose of the rules is to be a basis for automation, not mathematical minimality.

The *Decomposition Rule* is central to system factorization. It splits one table into two, both having the same decision part, by introducing the free variables thus created to the I/O signatures.



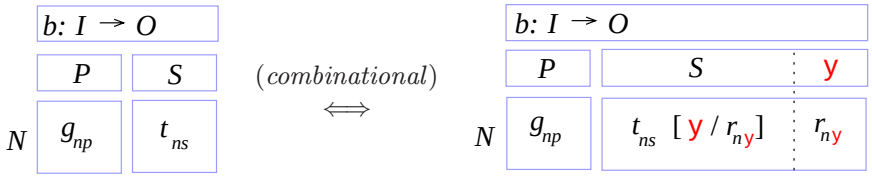
The composition operator, '$\circ$' represents a connection combinator,

$$b(I) \equiv O \ \textit{where} \quad \begin{array}{l} O_1 = b_1(I_1) \\ O_2 = b_2(I_2) \end{array}$$

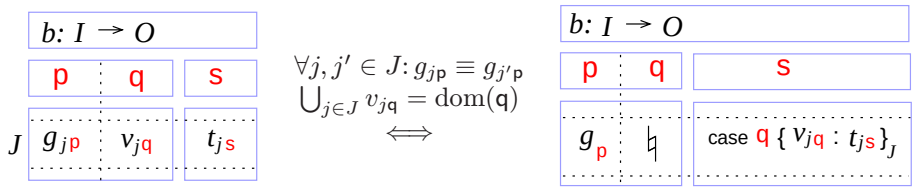
in which $I_1 = I \cup T$, $I_2 = I \cup S$, $O_1 = O \cap S$, and $O_2 = O \cap T$.

The *Identification Rule*, below, introduces and uses a name for a subexpression. In the rule scheme below, guards g_{np} and terms t_{ns} on the left-hand side are indexed by the sets of row numbers N, predicates P, and signals S. On the right, a new signal, y, is introduced. with action terms $\{r_{n\mathsf{y}}\}_{n \in N}$. In each cell

of the action table, should y's defining term occur in t_{ns}, it can be replaced by y. Conversely, a column of an action table can be eliminated, applying the rule from right to left, by substituting y's defining terms for y throughout the rest of the table.



Our third example is the *Conversion Rule*, whose effect is to coalesce rows in the decision table part. This is done by incorporating a selection expression in the action table that chooses the appropriate signal value. The side conditions say that all of the other guarding values in the decision table must be identical, and each possibility for the condition q must be accounted for.



As these examples illustrate, we have developed a notation of *table schemes* to express the core identities. In these schemes, the boxes play the role of quantifiers. Capitalized variables are index sets, lower case variables range over sets of the same name, and sans serif letters are dummy constants.

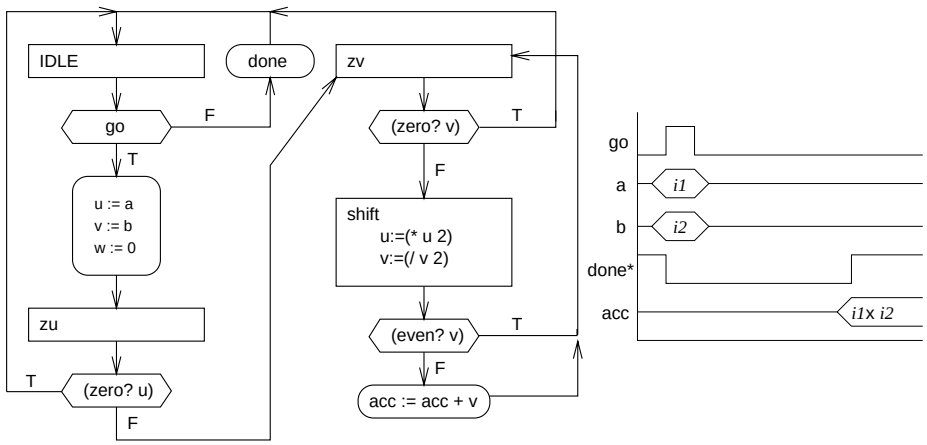
Thus, the form $R \overset{S}{\boxed{x_{rs}}}$ represents $\{g_{r,s} \mid r \in R \textit{ and } s \in S\}$.

4 System Factorization and Decomposition

System factorization is the organization of systems into architectural components. In the context of behavior tables this corresponds to their decomposition into a connection hierarchy in which behavior tables are the leaves. The motivations for factorization include: isolating functions in need of further specification, encapsulating abstract data types, grouping complementary functionality, targeting known components, and separating system control from system data

A designer has three decisions to make when factoring a given system component: which functions and signals to factor, how to group functionality in the presence scheduling conflicts, and assigning usage of communication lines between components. Behavior tables facilitate the this process by coloring terms relevant to each choice. Some of these decisions are characterized as optimization problems; we can automate more of the process by solving these problems.

The first three examples below decompose a behavior table describing a "shift-and-add" multiplication algorithm. The ASM chart [13] and timing diagram from Fig. 1 specifies its algorithm and interface respectively, while DDD automatically constructs the behavior table `MULT`. We introduce function and signal factorization with this example. The last example combines these techniques to isolate an abstract memory from a "stop and copy" garbage collector.



MULT: (go,a,b) → (done,acc)						
go	state	state	u	v	acc	done
0	idle	idle	♮	♮	acc	1
1	idle	ztest	a	b	0	0
♮	zu	(if (zero? u) idle zv)	u	v	acc	0
♮	zv	(if (zero? v) idle shift)	u	v	acc	0
♮	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc (+ acc v))	0

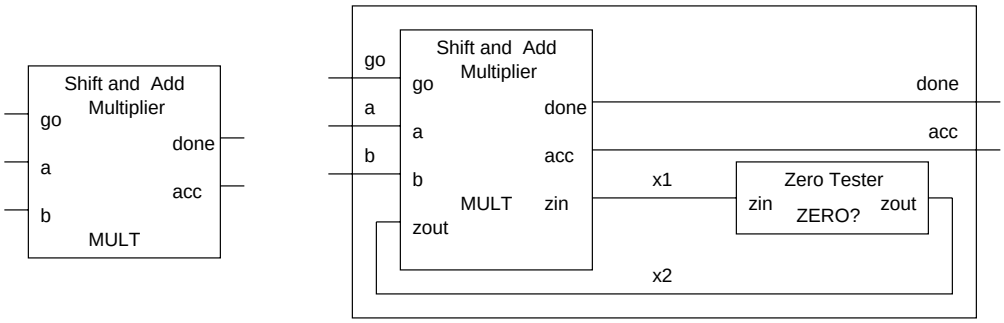
Fig. 1. Algorithmic, protocol, and behavior table specification of a shift-and-add multiplier

4.1 Single Function Factorization

The simplest variety of system factorization separates instances of one particular function f from a sequential system S. The decomposition produces two tables: a trivial table F that unconditionally applies f to its inputs, and a modified system S' where instances of f are replaced with the F's output and where new combinational signals have been created to carry the arguments of f to F.

We demonstrate this type of factorization by isolating the function `zero?` from the multiplier in Fig. 1. The block diagram view summarizes this archi-

tectural refinement. The connections hierarchy is maintained using the lambda expression below:



$$\begin{aligned}&\lambda(go, a, b).(done, acc)\ \textit{where}\\ &\quad(done, acc, x1) = MULT(go, a, b, x2)\\ &\quad(x2) \qquad\qquad\quad = ZERO?(x1)\end{aligned}$$

For the remainder of the examples, we assign I/O signals of separate blocks the same name to imply connection.

The *specification set* defines the function(s) to be factored; it is $\{zero?\}$ for this illustration. Guiding selection of *subject terms*, the instances of functions from the specification set the user intends to factor, the table below highlights all applications of `zero?`. We make no refinement to this initial selection of subject terms.

MULT: (go,a,b) → (done,acc)						
go	state	state	u	v	acc	done
0	idle	idle	♮	♮	acc	1
1	idle	ztest	a	b	0	0
♮	zu	(if (zero? u) idle zv)	u	v	acc	0
♮	zv	(if (zero? v) idle shift)	u	v	acc	0
♮	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc (+ acc v))	0

Tabular notation conveys scheduling information because the actions of each row occur in the same semantic step. The color scheme identifies scheduling collisions (i.e. rows with multiple subject term) by using a different color to render the conflicting terms. This example has consistent scheduling because each use of `zero?` occurs in a different row; we display these terms with a black background.

The transformation augments the original behavior table `MULT` with a new combinational signal `zin` to carry the actual arguments to `zero?`. Signal `zin` becomes an output of `MULT` and is connected to the input channel of `ZERO?`, the table encapsulating the function `zero?`. `ZERO?` is a trivial behavior table because it has no decision table; it simply places the value of `(zero? zin)` on its output

zout. The decomposition routes zout back to MULT and replaces the subject terms with the signal zout. The changes to MULT are highlighted with dark grey in the result of our decomposition below.

MULT: (go,a,b,zout) → (done,acc,zin)							
go	state	state	u	v	acc	done	zin
0	idle	idle	♮	♮	acc	1	♮
1	idle	ztest	a	b	0	0	♮
♮	zu	(if zout idle zv)	u	v	acc	0	u
♮	zv	(if zout idle shift)	u	v	acc	0	v
♮	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc (+ acc v))	0	♮

ZERO?:(zin)→(zout)
So*
(zero? zin)

4.2 Factoring Multiple Functions

We can generalize factorization to permit the encapsulation of several functions. This transformation follows the single function form, but also introduces a communication line to transmit control. Given a stream system S and a specification set $\{\mathtt{f_1}, \ldots, \mathtt{f_n}\}$, the transformation produces the following:

- a modified system S' containing combinational output signals for each argument of every f_i and a combinational output *inst* to select the operation; the subject terms are replaced with the appropriate input channels
- a table F using the *inst* signal to select the required function from the specification set

The initial form of the transformation makes inefficient use of communication lines by assigning dedicated input and output lines to each f_i. The following example illustrates how to remedy this using behavior tables to illuminate optimization opportunities.

The system in Fig. 1 is the starting point for this transformation, and the block diagram in Fig. 2 shows the connections between the new MULT (block defined by the dotted boundary) and the isolated component, ALU. We now use the expanded specification set $\{\mathtt{zero?}, \mathtt{even?}, \mathtt{add}\}$. To assist with selecting subject terms, the instances of function application from the specification set are highlighted according to their scheduling: a light grey background with a black foreground indicates parallel applications, while a black background with a white foreground indicates a single application.

MULT: (go,a,b) → (done,acc)						
go	state	state	u	v	acc	done
0	idle	idle	♮	♮	acc	1
1	idle	ztest	a	b	0	0
♮	zu	(if (zero? u) idle zv)	u	v	acc	0
♮	zv	(if (zero? v) idle shift)	u	v	acc	0
♮	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc (+ acc v))	0

The table shows that the initial subject terms `(+ acc v)` and `(even? v)` execute simultaneously. The designer may resolve the conflict by refining the selection of subject terms, but may also allow parallel applications; this decision should be guided by a knowledge of complementary functionality. For didactic reasons, we choose to eliminate this scheduling collision by removing the `(even? v)` from the set of subject terms.

Following the decomposition above, combinational output lines `i1`, `i2` and `i3` are added to carry arguments to `zero?` and `+`. Additionally, the table acquires a combinational output `inst` used to select the functionality of the encapsulated component. The new inputs to `MULT`, `out1` and `out2` replace subject terms corresponding to `zero?` and `+` respectively. The new table, `ALU` uses the `inst` input to determine which operation to perform; each function has its own input and output signals. As before modifications to the `MULT` table have a white foreground over a dark grey background.

MULT: (go,a,b,out1,out2) → (done,acc,i1,i2,i3,inst)										
go	state	state	u	v	acc	done	i1	i2	i3	inst
0	idle	idle	♮	♮	acc	1	♮	♮	♮	♮
1	idle	ztest	a	b	0	0	♮	♮	♮	♮
♮	zu	(if out1 idle zv)	u	v	acc	0	u	♮	♮	zero
♮	zv	(if out1 idle shift)	u	v	acc	0	v	♮	♮	zero
♮	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc out2)	0	♮	acc	v	add

ALU: (i1,i2,i3,inst)→(out1,out2)		
inst	out1	out2
zero	(zero? i1)	♮
add	♮	(+ i2 i3)

This initial decomposition is very conservative about the use of input and output lines. An automated minimization algorithm could be run here, but we also want to provide the user with an interface to manually control this process; a designer may want to preserve various inputs and outputs for specific uses such as "address" or "data".

The tables aid the user with the collection of compatible input and output signals. Highlighting the nontrivial terms (values besides ♮) of `i1`, `i2`, and `i3`,

we see that the present scheduling allows the values on `i1` and `i2` to be sent on a single output line.

MULT: (go,a,b,out1,out2) → (done,acc,i1,i2,i3,inst)										
go	state	state	u	v	acc	done	i1	i2	i3	inst
0	idle	idle	♮	♮	acc	1	♮	♮	♮	♮
1	idle	ztest	a	b	0	0	♮	♮	♮	♮
♮	zu	(if out1 idle zv)	u	v	acc	0	u	♮	♮	zero
♮	zv	(if out1 idle shift)	u	v	acc	0	v	♮	♮	zero
♮	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc out2)	0	♮	acc	v	add

MULT: (go,a,b,out1,out2) → (done,acc,i1,i2,inst)									
go	state	state	u	v	acc	done	i1	i2	inst
0	idle	idle	♮	♮	acc	1	♮	♮	♮
1	idle	ztest	a	b	0	0	♮	♮	♮
♮	zu	(if out1 idle zv)	u	v	acc	0	u	♮	zero
♮	zv	(if out1 idle shift)	u	v	acc	0	v	♮	zero
♮	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc out2)	0	acc	v	add

Similarly, coloring the nontrivial terms on `out1` and `out2` suggests that the the results of `(zero? i1)` and `(+ i1 i2)` can be returned on the same output channel. However, these terms have different types preventing this optimization. Later in the design pipeline, reducing integers to boolean vectors enables this optimization.

ALU: (i1,i2,inst)→(out1,out2)		
inst	out1	out2
zero	(zero? i1)	♮
add	♮	(add i1 i2)

4.3 Signal Factorization

The next family of factorizations encapsulates state. Given a stream system S and a specification set of *signals* $\{\mathtt{v_1}, \ldots, \mathtt{v_n}\}$, the transformation removes the columns v_i from S and introduces a control signal *inst* to create S'. To create the isolated component ST, signal factorization constructs a table using the signals v_i as the entries in its action table and *inst* in the decision table to enumerate the possible updates to v_i.

Beginning this example where the one from Sec. 4.2 ends, signals `u` and `v` determine the specification set. Intuitively, factoring these signals yields a pair of shift registers. Figure 2 shows this decomposition within the dotted box. Since the subject terms in a signal factorization are always the columns of the specified signals, we reserve the limited palette for identifying updates to `u` and `v`. Light grey, black, and dark grey correspond to assignment of new variables `a` and `b` to `u` and `v`, the preservation of `u` and `v`, and the binary left and right shift of `u` and `v`. Subject terms are italicized.

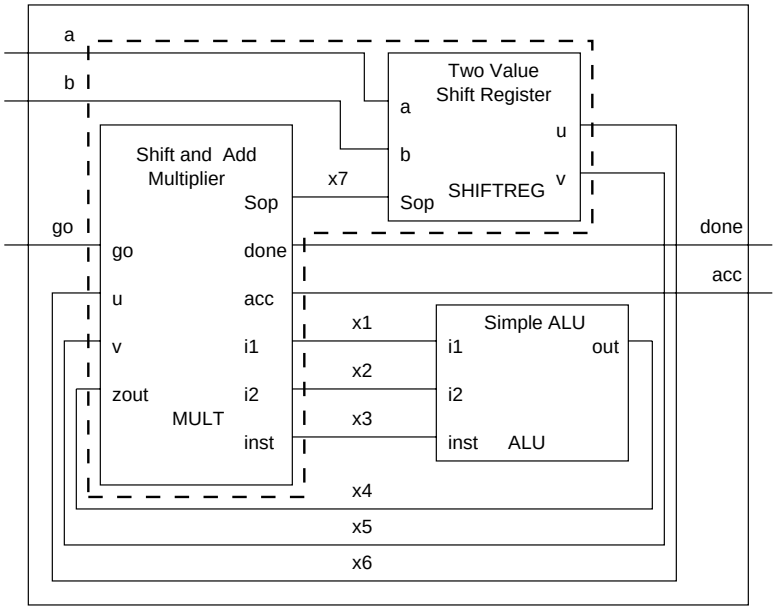


Fig. 2. Result of successive factorizations in Sec. 4.2 and Sec. 4.3

MULT: (go,a,b,out) → (done,acc,i1,i2,inst)									
go	state	state	*u*	*v*	acc	done	i1	i2	inst
0	idle	idle	♮	♮	acc	1	♮	♮	♮
1	idle	ztest	*a*	*b*	0	0	♮	♮	♮
♮	zu	(if out1 idle zv)	*u*	*v*	acc	0	u	♮	zero
♮	zv	(if out1 idle shift)	*u*	*v*	acc	0	v	♮	zero
♮	shift	zv	*(* u 2)*	*(/ v 2)*	(if (even? v) acc out2)	0	acc	v	add

The new signal in MULT, Sop, carries the instruction tokens init, hold, and shft corresponding to the signal updates above. The isolated component SHIFTREG acquires its action table from the columns u and v in MULT. Apart from Sop, the only inputs to SHIFTREG are a and b –the unresolved variables appearing in the terms of its action table. Since MULT no longer requires a and b, they are eliminated from its input signature. Changes to MULT are italicized.

MULT: (go,out,u,v) → (done,acc,i1,i2,inst,*Sop*)

go	state	state	*Sop*	acc	done	i1	i2	inst
0	idle	idle	♮	acc	1	♮	♮	♮
1	idle	ztest	*init*	0	0	♮	♮	♮
♮	zu	(if out1 idle zv)	*hold*	acc	0	u	♮	zero
♮	zv	(if out1 idle shift)	*hold*	acc	0	v	♮	zero
♮	shift	zv	*shft*	(if (even? v) acc out2)	0	acc	v	add

SHIFTREG: (a,b,Sop) → (u,v)

Sop	u	v
init	a	b
hold	u	v
shft	(* u 2)	(/ v 2)

4.4 The Stop and Copy Garbage Collector

Figure 3 is the behavior table for a garbage collector. Its "stop and copy" algorithm uses two memory half-spaces represented by abstract registers OLD and NEW. The goal of this factorization is to hide the memory data abstraction in a separate component.

OLD and NEW, together with their methods, rd and wt, define the specification set. The corresponding subject terms are highlighted in Fig. 3 with either black or light grey. As a combination of signal and function factorization, the instruction tokens select both state update and return value. References to OLD and NEW cannot appear in the resulting version of GC; consequently instruction naming subsumes their role as parameters to rd and wt. To illustrate this, we have shaded the instances of the "write NEW; read OLD" instruction (Mwnro) with light grey.

Like the previous examples, the initial decomposition supplies dedicated input and output signals for each operation of the factored component. Figure 4 displays the results, and indicates altered or new terms with dark and light grey. The dark grey shows the replacement of terms calling rd and also guides the optimization of new signals, while the light grey shows instances of the Mwnro.

MEM: (Mwoi1, Mwoi2, Mwni1, Mwni2, Mroi, Mrni, Mop) → (MROout, MRNout)

Mop	OLD	NEW	MROout	MRNout
Mnop	OLD	NEW	♮	♮
Mswap	NEW	OLD	♮	♮
Mwold	(wt OLD Mwoi1 Mwoi2)	NEW	♮	♮
Mwnew	OLD	(wt NEW Mwni1 Mwni2)	♮	♮
Mrold	OLD	NEW	(rd OLD Mroi)	♮
Mrnew	OLD	NEW	♮	(rd NEW Mrni)
Mwnro	OLD	(wt NEW Mwni1 Mwni2)	(rd OLD Mroi)	♮

We complete the factorization by collecting compatibly typed inputs and outputs when possible. Figure 5 shows these final input signals, and the table below displays the condensed output signals.

GC:(RQ) ⟶ (AK)

	NOW	RQ	(= U A)	(pointer? H)	(bvec-h? H)	(eq? 'forward (tag d))	(tag H)	(= C 0)	(= C 1)	NOW	OLD	NEW	H	D	C	U	A	AK
1	idle	1	↯	↯	↯	↯	↯	↯	↯	driver	OLD	NEW	H	D	C	(rd OLD H)	0	0
2	"	0	↯	↯	↯	↯	↯	↯	↯	idle	OLD	NEW	H	↯	↯	↯	↯	1
3	driver	↯	1	↯	↯	↯	↯	↯	↯	idle	NEW	OLD	0	D	C	U	A	1
4	"	↯	0	↯	↯	↯	↯	↯	↯	nextobj	OLD	NEW	(rd NEW U)	D	C	U	A	0
5	nextobj	↯	↯	1	↯	↯	↯	↯	↯	objtype	OLD	(wt NEW U (cell H A))	H	(rd OLD H)	C	U	A	0
6	"	↯	↯	0	1	↯	↯	↯	↯	driver	OLD	NEW	H	(rd OLD H)	C	(+ U (btow-u (pr-pt H)) (cin 1))	A	0
7	"	↯	↯	0	0	↯	↯	↯	↯	driver	OLD	NEW	H	(rd OLD H)	C	(+ U (const 0) (cin 1))	A	0
8	objtype	↯	↯	↯	↯	1	↯	↯	↯	driver	OLD	(wt NEW U (cell H d))	H	D	C	(+ U (const 0) (cin 1))	A	0
9	"	↯	↯	↯	↯	0	fixed	↯	↯	copy	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(add1-2 (fixed-size H))	U	A	0
10	"	↯	↯	↯	↯	0	vec	↯	↯	vec	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0
11	"	↯	↯	↯	↯	0	bvec	↯	↯	vec	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0
12	vec	↯	↯	↯	↯	↯	↯	1	↯	driver	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0
13	"	↯	↯	↯	↯	↯	↯	1	↯	copy	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	(rd OLD H)	C	U	(add1-a A)	0
14	copy	↯	↯	↯	↯	↯	↯	↯	1	driver	OLD	(wt NEW A d)	H	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0
15	"	↯	↯	↯	↯	↯	↯	↯	0	copy	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	(rd OLD H)	(sub1 C)	U	(add1-a A)	0

Fig. 3. Behavior Table for a Garbage Collector

GC: (RQ) ⟶ (AK)

	NOW	RQ	(= U A)	(pointer? H)	(bvec-b? H)	(eq? 'forward (tag d))	(tag H)	(= C 0)	(= C 1)	NOW	H	D	C	U	A	AK	Mwoi1	Mwoi2	Mwni1	Mwni2	Mrol	Mrni	Mop
1	idle	1	↯	↯	↯	↯	↯	↯	↯	driver	H	D	C	MROout	0	0	↯	↯	↯	↯	H	↯	Mrold
2	"	0	↯	↯	↯	↯	↯	↯	↯	idle	H	↯	↯	↯	↯	1	↯	↯	↯	↯	↯	↯	Mnop
3	driver	↯	1	↯	↯	↯	↯	↯	↯	idle	0	D	C	U	A	1	↯	↯	↯	↯	↯	↯	Mswap
4	"	↯	0	↯	↯	↯	↯	↯	↯	nextobj	MRNout	D	C	U	A	0	↯	↯	↯	↯	↯	U	Mrnew
5	nextobj	↯	↯	1	↯	↯	↯	↯	↯	objtype	H	MRNout	C	U	A	0	↯	↯	U	(cell H A)	H	↯	Mwnro
6	"	↯	↯	0	1	↯	↯	↯	↯	driver	H	MRNout	C	(+ U (btow-u (pr-pt H)) (cin 1))	A	0	↯	↯	↯	↯	H	↯	Mrold
7	"	↯	↯	0	0	↯	↯	↯	↯	driver	H	MRNout	C	(+ U (const 0) (cin 1))	A	0	↯	↯	↯	↯	H	↯	Mrold
8	objtype	↯	↯	↯	↯	1	↯	↯	↯	driver	H	D	C	(+ U (const 0) (cin 1))	A	0	↯	↯	U	(cell H D)	↯	↯	Mwnew
9	"	↯	↯	↯	↯	0	fixed	↯	↯	copy	(cell H (add1-ptr (pr-pt H)))	D	(add1-2 (fixed-size H))	U	A	0	H	(cell forward A)	↯	↯	↯	↯	Mwold
10	"	↯	↯	↯	↯	0	vec	↯	↯	vec	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0	H	(cell forward A)	↯	↯	↯	↯	Mwold
11	"	↯	↯	↯	↯	0	bvec	↯	↯	vec	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0	H	(cell forward A)	↯	↯	↯	↯	Mwold
12	vec	↯	↯	↯	↯	↯	↯	1	↯	driver	(cell H (add1-ptr (pr-pt H)))	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0	↯	↯	A	D	↯	↯	Mwnew
13	"	↯	↯	↯	↯	↯	↯	1	↯	copy	(cell H (add1-ptr (pr-pt H)))	MROout	C	U	(add1-a A)	0	↯	↯	A	D	H	↯	Mwnro
14	copy	↯	↯	↯	↯	↯	↯	↯	1	driver	H	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0	↯	↯	A	D	↯	↯	Mwnew
15	"	↯	↯	↯	↯	↯	↯	↯	0	copy	(cell H (add1-ptr (pr-pt H)))	MROout	(sub1 C)	U	(add1-a A)	0	↯	↯	A	D	H	↯	Mwnro

Fig. 4. Control Side of Initial Decomposition

GC:(RQ) ⟶ (AK)

	NOW	RQ	(= U A)	(pointer? H)	(bvec-h? H)	(eq? 'forward (tag d))	(tag H)	(= C 0)	(= C 1)	NOW	H	D	C	U	A	AK	MWtAd	MData	MRAd	Mop
1	idle	1	¤	¤	¤	¤	¤	¤	¤	driver	H	D	C	Mout	0	0	¤	¤	H	Mrold
2	"	0	¤	¤	¤	¤	¤	¤	¤	idle	H	¤	¤	¤	¤	1	¤	¤	¤	Mnop
3	driver	¤	1	¤	¤	¤	¤	¤	¤	idle	0	D	C	U	A	1	¤	¤	¤	Mswap
4	"	¤	0	¤	¤	¤	¤	¤	¤	nextobj	Mout	D	C	U	A	0	¤	¤	U	Mrnew
5	nextobj	¤	¤	1	¤	¤	¤	¤	¤	objtype	H	Mout	C	U	A	0	U	(cell H A)	H	Mwnro
6	"	¤	¤	0	1	¤	¤	¤	¤	driver	H	Mout	C	(+ U (btow-u (pr-pt H)) (cin 1))	A	0	¤	¤	H	Mrold
7	"	¤	¤	0	0	¤	¤	¤	¤	driver	H	Mout	C	(+ U (const 0) (cin 1))	A	0	¤	¤	H	Mrold
8	objtype	¤	¤	¤	¤	1	¤	¤	¤	driver	H	D	C	(+ U (const 0) (cin 1))	A	0	U	(cell H D)	¤	Mwnew
9	"	¤	¤	¤	¤	0	fixed	¤	¤	copy	(cell H (add1-ptr (pr-pt H)))	D	(add1-2 (fixed-size H))	U	A	0	H	(cell forward A)	¤	Mwold
10	"	¤	¤	¤	¤	0	vec	¤	¤	vec	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0	H	(cell forward A)	¤	Mwold
11	"	¤	¤	¤	¤	0	bvec	¤	¤	vec	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0	H	(cell forward A)	¤	Mwold
12	vec	¤	¤	¤	¤	¤	¤	1	¤	driver	(cell H (add1-ptr (pr-pt H)))	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0	A	D	¤	Mwnew
13	"	¤	¤	¤	¤	¤	¤	1	¤	copy	(cell H (add1-ptr (pr-pt H)))	Mout	C	U	(add1-a A)	0	A	D	H	Mwnro
14	copy	¤	¤	¤	¤	¤	¤	¤	1	driver	H	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0	A	D	¤	Mwnew
15	"	¤	¤	¤	¤	¤	¤	¤	0	copy	(cell H (add1-ptr (pr-pt H)))	Mout	(sub1 C)	U	(add1-a A)	0	A	D	H	Mwnro

Fig. 5. End Result of Signal Factorization

MEM: (MWtAd, MData, MRdAd, Mop) → (Mout)			
Mop	OLD	NEW	Mout
Mnop	OLD	NEW	♮
Mswap	NEW	OLD	♮
Mwold	(wt OLD MWtAd MData)	NEW	♮
Mwnew	OLD	(wt NEW MWtAd MData)	♮
Mrold	OLD	NEW	(rd OLD MRdAd)
Mrnew	OLD	NEW	(rd NEW MRdAd)
Mwnro	OLD	(wt NEW MWtAd MData)	(rd OLD MRdAd)

A realization of this design used a dual-ported DRAM capable of performing the `Mwnro` instruction in just a little over one DRAM cycle [3].

5 Further Work

Truth tables, decision tables, register transfer tables, and many similar forms are commonplace in design practice. Their already evident utility suggests that tabular notations have a role in design visualization and should, therefore, be considered for interfacing with formal reasoning tools. In our experience with interactive digital design derivation, tables have already proven helpful in proof planning and design analysis, but our previous automation has involved a shallow embedding of tabular syntax in an expression oriented modeling notation.

We are working toward a graphical interface with highlighting and animation facilities to support interactive visualization and proof management. This paper has illustrated ways in which these facilities are used in decomposition tasks. We demonstrated several kinds of factorizations, ranging from simple function allocation to abstract data type encapsulation. However, this is by no means the full range of decompositions needed.

The core algebra underlying the factorization transformations of this paper is not adequate to implement all the constructions of a high level synthesis system. They are not able to perform scheduling, for example, or even retiming. The DDD system uses fold/unfold transformations on control oriented behavior expressions to perform scheduling refinements, but these remain to be incorporated in the behavior table laws.

The reason is that we are still trying to determine suitable "levels" of equivalence. All the laws of [9] are bisimulation preserving, too strong a relationship for many kinds of design optimizations, such as scheduling. On the other hand, weaker forms of bisimulation may still be too strong to account for transformations that incorporate transaction protocols [14].

Heuristics are another topic of continuing study. Our research emphasizes interaction in order to make design processes observable, but it does not promote manual design over automation. Among the main uses of system factorization are targeting reusable subsystems, IP cores, and off-the-shelf components. These goal directed activities propagate constraints back into the main design. We are interested in exploring how visualization techniques, such as those of Sec. 4 help designers solve such problems.

References

1. FMCAD 2000. http://link.springer.de/series/lncs/.
2. Bhaskar Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Computer Science Department, Indiana University, USA, 1994. Technical Report No. 456, 155 pages, ftp://ftp.cs.indiana.edu/pub/techreports/TR456.ps.Z.
3. Robert G. Burger. The scheme machine. Technical Report 413, Indiana University, Computer Science Department, August 1994. 59 pages.
4. Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: a toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, 1995.
5. D. N. Hoover, David Guaspari, and Polar Humenn. Applications of formal methods to specification and safety of avionics software. Contractor Report 4723, National Aeronautics and Space Administration Langley Research Center (NASA/LRC), Hampton VA 23681-0001, November 1994.
6. Steven D. Johnson. Manipulating logical organization with system factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989.
7. Steven D. Johnson. A tabular language for system design. In C. Michael Holloway and Kelly J. Hayhurst, editors, *Lfm97: Fourth NASA Langley Formal Methods Workshop*, September 1997. NASA Conference Publication 3356, http://atb-www.larc.nasa.gov/Lfm97/.
8. Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In *IFIP and ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, 1991. Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).
9. Steven D. Johnson and Alex Tsow. Algebra of behavior tables. In C. M. Holloway, editor, *Lfm2000*. Langley Formal Methods Group, NASA, 2000. Proceedings of the 5th NASA Langley Formal Methods Workshop, Williamsburg, Virginia, 13-15 June, 2000, http://shemesh.larc.nasa.gov/fm/Lfm2000/.
10. Ramayya Kumar, Christian Blumenröohr, Dirk Eisenbiegler, and Detlef Schmid. Formal synthesis in circuit design – a classification and survey. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer Aided Design*, pages 294–309, Berlin, 1996. Springer LNCS 1166. Proceedings of FMCAD'96.
11. Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specifiation for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.
12. Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Revised May 1996. Available, with specification files, from URL `http://www.csl.sri.com/csl-95-12.html`.
13. Franklin P. Prosser and David E. Winkel. *The Art of Digital Design*. Prentice/Hall International, second edition, 1987.
14. Kamlesh Rath. *Sequential System Decomposition*. PhD thesis, Computer Science Department, Indiana University, USA, 1995. Technical Report No. 457, 90 pages.
15. Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. Behavior tables: A basis for system representation and transformational system synthesis. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 736–740. IEEE, November 1993.

16. M. Esen Tuna, Kamlesh Rath, and Steven D. Johnson. Specification and synthesis of bounded indirection. In *Proceedings of the Fifth Great Lakes Symposium on VLSI (GLSVLSI95)*, pages 86–89. IEEE, March 1995.