An Introduction to Behavior Tables

Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson

An Introduction to Behavior Tables *

Kamlesh Rath, M. Esen Tuna, Steven D. Johnson
Indiana University Computer Science Department
Bloomington, Indiana.

Abstract

In this paper, we introduce *behavior tables*, an extension of register transfer tables, as a basis for system representation for reasoning about control, datapath, protocol, and data abstraction facets of system synthesis. The novelty in our approach is that it unifies different aspects of system synthesis and alleviates the need to change bases to reason about different facets of a design. Behavior tables can model *indirection* in system specification, by allowing names of registers and states to be treated as values. Behavior tables provide an environment for transformational design to derive a formally "correct" implementation from a specification. The emphasis of our work is on design correctness rather than design automation. Herein, we develop *implementation* relations over different facets of behavior tables. A set of transformations on the different facets of behavior tables, that preserve the implementation relations on the facets, are presented. Behavior tables and the transformations presented in this paper are based on a finite state machine model. Starting from a specification with symbolic data values, transformations are used by the designer to direct the design towards an implementation of interacting behavior tables with boolean data representation and appropriate control and datapath abstractions. To illustrate the use of behavior tables and some of the transformations, we use a behavior table description of the FM9001 processor. The resulting boolean level behavior tables can be synthesized using sequential logic synthesis tools.

1 Introduction

Register transfer level descriptions have been used extensively for datapath specification and design.
With the advent of hardware description languages to model system behavior, register transfer level
descriptions have been reduced to an intermediate form in the datapath synthesis. State machines,
functional languages and petri-nets have been used to model control descriptions. In this paper, we
propose a design representation called *behavior tables* to model the behavior of a system. Behavior
tables are based on a finite-state machine model, and can represent control and datapath facets
of a system, in addition to protocol and data abstraction facets. The feature that distinguishes
behavior tables from other system representations and hardware description languages is that, they
can model *indirection* in system description. We present a set of transformations on different facets
of behavior tables which can be used to derive a suitable implementation from a specification.
Derivation is a formalization of synthesis, with more emphasis on correctness than on automation.
A specification can have many implementations and a particular derivation chooses one. The
transformations add information to the (accumulating) implementation.

Most design automation systems restrict the use of transformations/algorithms to certain phases
of the synthesis path, with different design representations at each step. The separation of control
and datapath in the synthesis path and the use of different modeling environments make it difficult
for a designer to guide the search in the design space. System level transformations that change
both control and datapath are performed before their separation, and structural transformations
are performed after their separation. In our methodology, transformations on all facets of a design
can be reflected in behavior tables. This enables a designer to transform different facets of a design
without having to change design representations.

Along with a global view of a design, views of different facets of a design can also be abstracted
from a behavior table. From our experience, a table form is a useful visual output for a designer to
help navigate the design space. A high-level functional specification of a system is used to construct
the behavior table specification. A sequence of transformations on the different facets of the design
are used to construct a set of interacting boolean level behavior tables which can be realized using
sequential logic synthesis tools (e.g. [1]).

Many design automation systems use directed acyclic graph (DAG) based structures for design
representation. Flamel [2] uses a DAG based representation to model data-flow and control-flow.
Transformations are defined on the DAGs for scheduling and allocation. The System Architect's
Workbench [3] also uses a DAG based internal representation called Value Trace. Behavioral and
structural transformations are defined on the value trace representation. The ADAM synthesis
system [4] also uses data flow graphs for datapath synthesis. Graph based internal representations

are suited for either control-flow or data-flow representation. Representing these and other facets in a single graph form is usually done by annotating the control-flow graph with data-flow and other information, as in HOP [5].

Petri-net based internal representations are also suited for control-flow representation but are not useful for data-flow representation. In the CAMAD [6] system, a petri-net model is used for control representation and a graph representation is used for datapath representation. Semantics preserving structural transformations are used on the petri-net and graph representations to realize a circuit.

Patel [7] proposes a hierarchical system of graph representations to describe concurrency, control-flow and data-flow aspects of a system. This representation addresses the problem of storage optimization for synthesis algorithms, but as Patel acknowledges "one of the first problems ...is the detailed representation of the datapath". The data abstraction aspects of a design are not addressed effectively by this representation. Also, a hierarchical representation indexed only by behavior is not well suited for structural transformations that deal with all data flow through particular functional units. Behavior tables are indexed both by behavior and structure, and are suited for both behavioral and structural transformations.

System-level synthesis in the System Architect's Workbench is accomplished by behavioral transformations [8]. Walker and Thomas show transformations on the controller and selector. Transformations to partition a design into processes are also shown. The processes created using their method have a very simple interaction scheme to transfer data values and control signals using message passing. Their approach can not synthesize components using complex protocols for data transfers and synchronization.

SpecPart [9] partitions algorithm/process grained computations from the SpecChart behavioral specifications. Default protocols are used for interaction between components. The CHOP system-level design partitioner [10] uses task graphs to specify the protocol between every partition. Special purpose hardware units called data-transfer modules are used on both sides of each interaction. Although this method allows for complex protocols and use of off-the-shelf components, the interface has to be designed manually and may be expensive in terms of area and performance because of the special purpose modules.

In our approach, parts of a system at the algorithm/process level or operation level of granularity can be abstracted and the protocol between the components can be incorporated into the components without using any special-purpose modules. We use *Interface Specification Language (ISL)* to specify the protocol between components of a system. Components of a system can be independently synthesized. Alternatively, they can be mapped to off-the-shelf components, such as dynamic RAMs and floating-point units by specifying the interface from timing diagrams. The

3

designer uses transformations to ensure that the implementation is "correct by construction".

Indirection in system description is modeled by allowing names of registers and states to be treated as values in the system. This can significantly reduce the size of the control algorithm that is represented. The ability to change levels of data abstraction is a unique feature of behavior tables. A system can be specified at the symbolic level and later transformed into a boolean system by assigning types to entries in the tables.

The research reported here grew out of our existing design derivation system, which is based on first order functional algebra [11, 12]. As an example of the use of transformations on behavior tables, we sketch parts of the derivation of a boolean level behavior table implementation of Hunt's FM9001 processor.

2 Behavior Tables

Behavior tables are an extension of register transfer tables that can model the control, datapath, protocol, and data abstraction facets of a design. The datapath refers to the functional units and the interconnections between them. The control facet describes the conditions and state that select the functions performed by the datapath. The protocol facet of a design describes the interaction of a system with its environment. The data abstraction facet refers to the representation of data values at symbolic and boolean levels.

Behavior tables are based on a finite state machine model. Each row in a behavior table represents a transition in the machine described by the table. The columns are divided into two sections, the decision section and the action section. The decision section represents the state and conditions that must hold for the transition to be executed. The action section represents the data flow through the functional units and ports and the next state.

The decision columns consist of truth values for internal conditions (predicates), external conditions (control inputs) and the present state. Each column has a *type* associated with it. A *type* can be generic (symbolic values) or specific (e.g. 32-bit boolean). All values in each column must have the same *type* as the column. The action section in a transition is executed if the decision corresponding to the row is *true*. A decision is *true* if the present state, predicates and control inputs equal the values in the corresponding columns in the row. Behavior tables are deterministic, i.e. only one row in the table has a *true* decision at any instant in time. Transitions from any state to a particular state due to conditions such as reset or interrupts can also be modeled by a don't care value in the present state column. The action columns consist of values in registers, input and output ports, and the next state for the system. Values can also be functions on symbolic values, registers, and input ports.

2.1 Finite State Machine Model

A behavior table is a representation for a finite state machine that models system behavior. A machine is defined as $M = \langle T, s, n, P, R, C, S, \hat{I}, @I, V \rangle$, where T is a non-empty set of transitions, s is the present state, n is the next state, P is the set of ports, R is a set of internal registers and combinational signals, C is the set of internal predicates, S is a set of states ($\#$ denotes all states in the machine), $\hat{I}$ is a set of references that includes functions over register names and data input names, $@I$ is a set of dereferences, and V is the domain of values including the don't care value $\#$. The set of ports ($P = CI \cup CO \cup DI \cup DO$) is a union of the sets of control inputs, control outputs, data inputs, and data outputs. We adopt a convention that references to registers and inputs in $\hat{I}$ are written as $\hat{r}$, and dereferences in $@I$ are written as $@r$.

A transition is defined as a function of the form $t = \ if\ [satisfies(t_d, \vec{d})]\ then\ t_a$, where the assignment function $t_d : C \cup CI \mapsto V$ and $t_d : \{s\} \mapsto S$. An assignment function t_d is satisfied with respect to the current conditions $\vec{d}$ if $(\forall c_i \in \{s\} \cup C \cup CI\ .\ t_d(c_i) = \vec{d}_i\ \vee\ t_d(c_i) = \#)$ is $true$.

The action section of each transition is defined by the assignment function, $t_a : R \cup DO \cup CO \mapsto V \cup S \cup I$ and $t_a : \{n\} \mapsto (V \cup S) - \{\#\}$. Each transition in the machine denotes a row in the behavior table. This model can be used to denote transitions from a set of states to a particular state or from a state to one of several states based on register contents. A transition is written as $s_1 \xrightarrow{t} s_2$, if $t(s) = s_1$ and $t(n) = s_2$.

Definition 2.1: t^+ denotes a finite sequence of transitions, $t_0, t_1, \ldots, t_k$, such that for all $0 < i < k$, $t_i(n) = t_{i+1}(s)$. The sequence of transitions defines the function, $t^+(x) = t_k(t_{k-1}(\ldots(t_0(x))\ldots))$. The set of transitions in t^+ is denoted by $\{t^+\}$.

Our model enables us to define transitions to states based on register or port values. Interrupts and continuations can be modeled as transitions in behavior tables. Indirection allows us to control data flow based on register contents without "controller" intervention.

The $care$ set for a transition denotes the set of registers, ports, predicates, inputs and state that do not have don't care values according to the assignment function t. $care_t = \{p \mid t(p) \neq \#\}$. For a sequence of transitions, the care set is defined as $care_{t^+} = \{p \mid t \in \{t^+\}\ .\ \exists t(p) \neq \#\}$.

We define an evaluation function E, which evaluates a machine with respect to streams of input sequences for each control and data input and generates streams of values for registers, control and data outputs, and the next state of the machine. The evaluation of a machine with streams of don't care values for all inputs is denoted by the function $E_\#$. The don't care values are propagated through the evaluation. This can help identify which parts of a machine are independent of external inputs.

Behavior tables do not subsume any clocking/timing discipline, and can be used to model

synchronous and asynchronous designs. A clocking discipline must be chosen to derive an implementation. In this paper we will develop transformations based on a global synchronous clock.

2.2 Behavior Specification Language

(define $MACHINE$
  (lambda $((ci_1 \ ci_2 \ \ldots \ ci_m) \ (di_1 \ di_2 \ \ldots \ di_n))$
    (letrec
      $((s_0$ (lambda $((r_1 \ r_2 \ \ldots \ r_j) \ (co_1 \ co_2 \ \ldots \ co_k) \ (do_1 \ do_2 \ \ldots \ do_l)) \ Exp_0))$
      $\vdots$
      $(s_q$ (lambda $((r_1 \ r_2 \ \ldots \ r_j) \ (co_1 \ co_2 \ \ldots \ co_k) \ (do_1 \ do_2 \ \ldots \ do_l)) \ Exp_q)))$
      $(s_{\text{init}} \ r_{1\text{init}} \ r_{2\text{init}} \ \ldots \ r_{j\text{init}} \ co_{1\text{init}} \ co_{2\text{init}} \ \ldots \ co_{k\text{init}} \ do_{1\text{init}} \ do_{2\text{init}} \ \ldots \ do_{l\text{init}}))))$

Figure 1: Behavior Specification Language

Figure 1 shows the specification language for the behavior of a machine. It is a subset of the Scheme programming language, which can be translated into a behavior table. The specification is a set of mutually recursive function definitions. Each function definition is a conditional expression representing the possible transitions from the present state of the system. The initial power-on/reset state and register and output values are denoted by the last line of the specification. An expression is one of the following forms:

$(S \ v_1 \ v_2 \ \ldots)$
(if pred $Exp_1 \ Exp_2$)
(case pred $(id_1 \ Exp_1) \ (id_2 \ Exp_2) \ \ldots)$
(let $((id \ v) \ \ldots \ ) \ Exp$)

The expression $(S \ v_1 \ v_2 \ \ldots)$ denotes parallel assignments of the values in the expression to registers and outputs, and transfer of control to the state denoted by the value S. *if* and *case* expressions denote conditional statements. *pred* denotes the conditions in the expression, a boolean function over internal predicates and control inputs. *let* expressions are used to assign values to combinational signals.

3 Control and Datapath Facets

The functional units (arithmetic/logic units, registers, switches) in a system and the interconnections between them are called the datapath of a system. The control determines the state, external and internal conditions that select the actions performed by the datapath. Control of a system

consists of the state, external control inputs, and internal predicates in the system. The datapath of a system consists of interconnections between the internal registers/memory, combinational signals, data input ports, data output ports, and the control output ports. We will refer to the decision section of the behavior table as the control facet and the action section as the datapath facet. The present state and next state are considered part of all facets of behavioral tables. Indirection blurs the distinction between control and datapath facets in behavior tables, by providing a decision mechanism in the datapath. Each row in the decision section of the behavioral table represents a satisfiable set of conditions. If the current conditions satisfy decisions in a row, then the actions in corresponding row are executed. Each column in the action section of a behavior table represents the values assigned to a functional unit in different transitions.

3.1 Implementation

The *implementation* relation between control and datapath facets of machines is determined by the present state, the internal predicates, external control inputs, and actions performed on the datapath in sequences of transitions. To define the *implementation* relation we must first define the *inclusion* relation over control decisions and datapath actions. The *inclusion* relation on decisions tests if all conditions tested in one sequence of transitions are tested in the other. In case of datapath actions, all non don't care values at the end of one sequence of actions must have an equivalent value at the end of the other sequence of actions.

Definition 3.2: The *inclusion* relation between control decisions and datapath actions is defined as :

$$t_1^+ \preceq t_2^+ \quad \stackrel{\text{def}}{=} \quad \forall p \in care_{t_2^+} . \forall t_{d_2} \in \{t_{d_2}^+\} . \exists\, t_{d_1} \in \{t_{d_1}^+\} . t_{d_1}(p) \equiv t_{d_2}(p)$$
$$\wedge\ \ t_{a_1}^+(p) \equiv t_{a_2}^+(p)$$

The equivalence of two values can be tested by textual comparison, for simple expressions. In general, this involves verification of equivalence of logical and arithmetic expressions [13], and is therefore a heuristic task. This relation applies only to internal predicates, registers and combinational signals. External input or output columns should not be transformed based on this relation.

The binary relation *simulates* is a maximal relation over states, $\mathcal{S} \subseteq S_1 \times S_2$, where S_1, S_2 are sets of states.

Definition 3.3: A relation over states is a *simulation* relation if $s_1 \sqsubset s_2$ implies :

$$\forall s_2 \xrightarrow{t_2^+} s_2' . \exists s_1 \xrightarrow{t_1^+} s_1' . t_2^+ \preceq t_1^+ \ \wedge\ s_1' \sqsubset s_2'$$

7

If $s_1' \notin S_1$ or $s_2' \notin S_2$, then the side condition $s_1' \sqsubset s_2'$ is generated. Such a condition can be verified for a particular stream of inputs by dynamic evaluation. It may be possible to statically evaluate s_1' and s_2' if the next state is independent of inputs, i.e. $E_\#$ yields a stream of non don't care terms for next state.

Definition 3.4: The control and datapath facets of a machine M_1 is defined to be an *implementation* of the control and datapath of machine M_2 ($M_1 \sqsubseteq M_2$) iff,

$$\forall s_2 \in S_2 \ . \ \exists s_1 \in S_1 \ . \ s_1 \sqsubset s_2$$

where S_1, S_2 are the sets of states in M_1, M_2 respectively. Verifying the *implementation* for two arbitrary machines can be computationally expensive. We will use transformations that preserve the implementation relation to avoid having to verify it.

3.2 Transformations

In this section we define transformations on the control and datapath facets of a behavior table that preserve the implementation relation. Indirection transformations can be used to move parts of the system between control and datapath. The transformations on the datapath have been adapted for use on behavior tables from the algebra on purely functional datapath descriptions reported in [14, 12].

Column Insertion/Deletion/Renaming : A column denoting a new functional unit can be added to a behavior table with any value for any transition. A column can be deleted if all values in the column are don't cares. A column can be renamed and all references to the column in the rest of the table must be replaced by the new name.

Column Merging/Function Identification : Two columns can be merged if for each transition in the behavior table, one of the columns has a don't care value (*conflict-free* - defined in section 5) or both have the same value. A new combinational output column identifying a function in the behavior table is inserted in the datapath. All references to merged columns or identified function in the rest of the table are then replaced by references to the new column.

Generalization : All the values in a column can be normalized as functions with the same number of arguments by padding functions with fewer arguments with don't care arguments. This transformation is used before factorization.

Factorization : This transformation is used to encapsulate a set of functions all with the same number of arguments or a functional unit by generating columns for communicating values to and from a functional unit that can perform all the encapsulated functions. An instruction column with a value corresponding to each encapsulated function is also generated for the new functional unit.

8

Folding/Unfolding : A sequence of transitions can be folded into a single transition, if all transitions out of each intermediate state in the sequence of transitions have a target state within the scope of the folding. A folded transition must *include* the sequence of transitions it replaces. A transition in a specification can be unfolded into a sequence of transitions which satisfies the inclusion relation. Only transitions with don't care values on ports that connect across behavior tables can be folded or unfolded.

3.3 Indirection

Indirection involves changing behavior of a system based on register values. Essentially, a portion of the control is modeled as part of the datapath. A bounded indirect reference in the datapath can be transformed into a set of transitions, each with a decision for a possible reference. Figure 2 shows the transformations on state and value indirection.
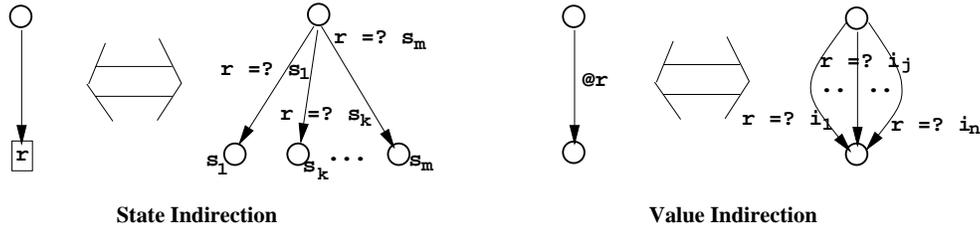

State Indirection   **Value Indirection**

Figure 2: Indirect Transformations

Definition 3.5: A transition with an indirect value reference and a set of transitions without the indirect reference are equivalent if:

$$t_1 \stackrel{I}{\equiv} T_2 \quad \stackrel{\text{def}}{=} \quad t_{a_1}(c) \in @I \;\Rightarrow\; \forall i \in \hat{I} \,.\, \exists t_2 \in T_2 \,.\, t_{d_2}(t_{a_1}) = i$$
$$\wedge \;\; t_1(c) \notin @I \;\Rightarrow \forall t_2 \in T_2 \,.\, t_1(c) \equiv t_2(c)$$

Similarly, a transition with state indirection is equivalent to a set of transitions without state indirection if:

$$t_1 \stackrel{S}{\equiv} T_2 \quad \stackrel{\text{def}}{=} \quad t_1(n) \notin S \;\Rightarrow\; \forall q \in S \,.\, \exists t_2 \in T_2 \,.\, t_2(n) = q$$
$$\wedge \;\; \forall t_2 \in T_2 \,.\, c \neq n \;\Rightarrow\; t_1(c) \equiv t_2(c)$$

4 Protocol Facet

The protocol facet describes the interaction of a machine with its environment using the input/output ports. Communication between machines is modeled as values over connected ports.

9

Other forms of communication, such as using buffers, must be modeled explicitly.

$$
\begin{array}{lll}
\text{Interaction} & A ::= C : D \mid A, A & \text{where} \quad C ::= c_1/v_1, .., c_n/v_n \\
& \mid \ compute \ A_o \mid \ await \ A_i & D ::= d_1/v_1, .., d_n/v_n \\
& \mid A_o \ until \ A_i \mid A_i \ before \ A_o & c_j \in CI \cup CO, \ d_k \in DI \cup DO \\
& & v_j, v_k \in V \\
\text{Expression} & E ::= \ ; A \mid E ; A \mid E [\!] E \mid E^* & \\
\end{array}
$$

$$
\text{Definition} \quad M ::= \ \text{Process} \, (CI, CO, DI, DO) \ \stackrel{\triangle}{=} \ E
$$

Figure 3: ISL Syntax

Interface specification language (ISL) provides a front-end for specification of the protocol facet of a behavior table. We briefly sketch this language here - more details can be found in [15]. ISL can not be used to specify the internal behavior of a system as internal register transfers and internal conditions. A definition in ISL is used to construct a machine that describes the control synchronization and data transfer protocol with its environment. The machine model for the protocol specification is same as the one described in Section 2.1, with designated start and final states. The *complement* of the protocol specifies the environment machine. Our idea of *complement* is similar to Dill's idea of an *environment* of a trace structure [16].

The goal of the transformations on the protocol facet is to decompose a behavior table into interacting behavior tables using the specified protocol definition. We can specify the protocol of a component and incorporate an implementation of an interaction path of its *complement* into the other component. Internal state changes in a machine are not specified in this language to allow for partial specifications of decomposed components. In case of off-the-shelf components, the protocol definition can be written from their timing diagrams.

The protocol specification of a machine has two components, data interaction and control interaction. Data interactions occur over input/output data ports and control interactions occur over input/output control ports. The protocol is defined over input and output control ports (CI, CO), input and output data ports (DI, DO), and data values (V). We use an extension of the language presented in [15], that allows for symbolic values on control and data ports.

The syntax description of ISL is given in Figure 3. An *interaction* consists of a set of values on data ports guarded by certain truth values on control ports. Input interactions A_i denote a set of control inputs guarding data interactions. Similarly output interactions A_o denote a set of control outputs guarding data interactions.

4.1 Complement

The environment of a machine can be constructed using the complement operation. The complement machine can be viewed as the protocol specification of decomposed components in the environment. The machine constructed by a protocol definition is embedded as a "stub" into one side of the protocol interface, and an implementation of an interaction path of its complement is emdedded as the "stub" on the other side of the protocol interface.

For every output(input) port in a machine, a new input(output) port is created in its complement using a "rename" function to generate port names. The internal registers and internal predicates in a machine do not play any role in the complement operation. A machine has the same set of states as its complement. In every transition, the complement machine has the same value on the corresponding renamed ports.

The set of transitions in the complement are:

$$\overline{T} = \{s_1 \xrightarrow{\overline{t}} s_2 \mid s_1 \xrightarrow{t} s_2 \in T \text{ and } \overline{t} = \text{Rename}(t)\}$$

where $\text{Rename}(t)(p') = t(p)$ if $p' = \text{rename}(p)$.

4.2 Path Implementation

The complement of the protocol facet of a behavior table describes its environment. Each path from the start state to the final state in the complement machine represents a valid sequence of interactions to complete a protocol. A machine can interact with any implementation of any path of its complement machine.

To define the *path implementation* relation we must first define the *inclusion* relation over protocol interactions. An interaction t_1 includes an interaction t_2 if all the ports which have non don't care values in t_2 also have equivalent values in t_1.

Definition 4.6:

$$t_1 \overset{\text{ptl}}{\preceq} t_2 \overset{\text{def}}{=} \forall p \in care_{t_2}. \, t_1(p) \equiv t_2(p)$$

The binary relation *path simulates* is a maximal relation over states, $\mathcal{S}_p \subseteq S_1 \times S_2$, where S_1, S_2 are the sets of states in the two machines. $s_1 \sqsubseteq_p s_2$ implies that there are possible interactions from s_1 and s_2, such that the interaction from s_1 includes the interaction from s_2, and they lead to states which satisfy the same relation. It also implies that, if s_2 is a wait state for a control input, then s_1 must also be a wait state for the control input, or s_1 must lead to a wait state for the control input.

Definition 4.7: A relation over states is a *path simulation* relation if $s_1 \sqsubseteq_p s_2$ implies :

1. $\exists s_1 \xrightarrow{t_1} s_1', s_2 \xrightarrow{t_2} s_2' \;.\; t_1 \overset{ptl}{\preceq} t_2 \;\wedge\; s_1' \sqsubseteq_p s_2'$

2. $\exists s_2 \xrightarrow{t_{2_1}} s_2, s_2 \xrightarrow{t_{2_2}} s_2' \;.\; s_2 \neq s_2' \;\wedge\; care_{t_{2_2}} \cap CI \neq \phi \;\Rightarrow$

   $$\left(\exists s_1 \xrightarrow{t_{1_2}} s_1' \;.\; s_1 \neq s_1' \;\wedge\; t_{1_2} \overset{ptl}{\preceq} t_{2_2} \;\wedge\; s_1' \sqsubseteq_p s_2' \right) \wedge$$

   $$\left(\left(\exists s_1 \xrightarrow{t_{1_1}} s_1 \;\wedge\; t_{1_1} \overset{ptl}{\preceq} t_{2_1} \right) \vee \right.$$

   $$\left. \left(\exists s_1 \xrightarrow{t_{1_3}} s_k \;\wedge\; t_{1_3} \overset{ptl}{\preceq} t_{2_1} \;\wedge\; s_k \sqsubseteq_p s_2 \right) \right)$$

Definition 4.8: A machine M_1 *path implements* machine M_2 ($M_1 \sqsubseteq_p M_2$) if :

1. The start state of M_1 path simulates the start state of M_2.

2. The final state of M_1 path simulates the final state of M_2.

Intuitively, a machine can interact with any path implementation of its complement.

4.3 Sequential Decomposition

In this section we introduce the *sequential decomposition* transformation on the protocol facet of behavior tables. Sequential decomposition is a generalization of the *factorization* transformation on the datapath facet [14] to include protocols in factoring procedures from a system. This transformation is used to extract parts of a system at the operation, algorithm, or procedure level of granularity into a co-process, with non-trivial control synchronization and data transfer protocols. Sequential decomposition is different from classical FSM decomposition (e.g. [17]), which assumes tightly-coupled sub-machines that can share state and input information.

A group of functions in a behavior table can be abstracted to a behavior table that performs the function. The protocol for communication with the abstract component is specified in ISL. A path implementation of the complement of the protocol facet of the abstract component is embedded into the original behavior table. Embedding is done by merging the start state and final states of the path implementation into the source and target states of the transition that contained the abstracted function. The other operations in the replaced transition are scheduled in the earliest possible transition in the embedded path implementation, such that the data dependencies in the original behavior table are preserved. A set of connections between ports in the machines are also generated by this transformation.

5 Data Abstraction Facet

The Data Abstraction facet of the behavior table describes the representation of values in the system as symbols or values represented in other domains (e.g. boolean). The organization of the

12

behavior tables helps in visualizing data abstraction in a design. Transformations on the values in the behavior table can be used to "correctly" map the values to a different domain. A *type* is defined as a domain for representation of values of a specified width. Changes in data abstraction are made on columns of behavior tables by specifying the types for the columns. The name of a column and all values in the column are changed to a vector in the new type. Values that are outputs of functions over other registers and input ports must also be of the same type as the column. Data representation changes on input/output ports produce side-conditions for corresponding changes on all ports that are connected together on other machines in the system.

Changing the level of data abstraction of a machine $M = \langle T, s, n, P, R, C, S, \hat{I}, @I, V \rangle$ creates a new machine $M' = \langle T', \vec{s}, \vec{n}, P', R', C', S', V' \rangle$. We define the representation function, that maps columns in M to a partition of columns in M'. $\mathcal{R} : \{s\} \cup \{n\} \cup P \cup R \cup C \times \ type \ \mapsto \{\vec{s}\} \cup \{\vec{n}\} \cup [P'] \cup [R'] \cup [C']$. We also define a function that maps values in one data type to vectors of values in another: $\mathcal{V} : V \cup S \cup \hat{I} \times \ type \ \mapsto V'^* \cup S'^* \cup \hat{I}^*$.

For the data representation changes to be valid, the representation function $\mathcal{R}$ must be onto. It must also be one-to-one or it must be *conflict-free*. We will assume that $\mathcal{R}$ is onto, and we define the *conflict-free* condition. If after data abstraction changes, two or more columns are mapped on to the same column, then they must be *conflict-free*.

Definition 5.9: A data representation function $\mathcal{R}$ is *conflict-free* iff :

$$\forall p' \in P' \cup R' \cup C' \ . \ \exists p_1, p_2 \in P \cup R \cup C \ . \ p' \in \mathcal{R}(p_1) \cap \mathcal{R}(p_2) \Rightarrow \forall t \in T \ . \ t(p_1) = \# \vee t(p_2) = \#$$

The above conditions are *sufficient* to state that $\mathcal{R}$ is valid and can be used to transform the machine M. The conditions for validity of the data representation function have been adapted from our previous work on abstract data types [18].

5.1 Transformations

In this section we define the *represents* relation on the data abstraction facets of machines and the transformations to change levels of data abstraction. A *type* must be chosen for each column in the behavior table. A machine *represents* another ($M' \sqsubseteq_\mathcal{R} M$) if each column in M is changed using $\mathcal{R}, \mathcal{V}$ and a *type*, to a vector of columns in M'.

Definition 5.10: The *represents* relation over machines is defined as :

$$\forall p \in \{s\} \cup \{n\} \cup P \cup R \cup C \ . \ \mathcal{R}(p, \ type) = \vec{p} \Rightarrow \forall t \in T \ . \ \exists t' \in T' \ . \ t'(\vec{p}) \equiv \mathcal{V}(t(p), \ type)$$

Transformations on the data abstraction facet of behavior tables involve specification of a data representation function $\mathcal{R}$ and a function $\mathcal{V}$ that maps symbolic values in the behavior table to values

	present state	(nlist-p oracle)	(a-imm-p ins)	(reg-dir-p (mode-a ins))	(pre-dec-p (mode-a ins))	(post-inc-p (mode-a ins))	(reg-dir-p (mode-b ins))	(pre-dec-p (mode-b ins))	(post-inc-p (mode-b ins))	(store -resultp)
0	intr	1	#	#	#	#	#	#	#	#
1	intr	0	#	#	#	#	#	#	#	#
2	fetch	#	#	#	#	#	#	#	#	#
3	op-a	#	1	0	0	0	#	#	#	#
4	op-a	#	0	1	0	0	#	#	#	#
5	op-a	#	0	0	1	0	#	#	#	#
6	op-a	#	0	0	0	1	#	#	#	#
7	op-a	#	0	0	0	0	#	#	#	#
8	op-b	#	#	#	#	#	1	0	0	#
9	op-b	#	#	#	#	#	0	1	0	#
10	op-b	#	#	#	#	#	0	0	1	#
11	op-b	#	#	#	#	#	0	0	0	#
12	alu-op	#	#	#	#	#	1	#	#	1
13	alu-op	#	#	#	#	#	0	#	#	1
14	alu-op	#	#	#	#	#	#	#	#	0

Table 1: Behavior Table Specification - Decision Section

in some type. Each column in a behavior table is transformed using $\mathcal{R}$ to a vector of columns. The values in the transformed column are coerced to new values in the type using $\mathcal{V}$. Changing the level of data abstraction of functions involves coercing the function to the new *type*. Changing the data representation of the present and next state columns is analogous to state assignment.

In case of boolean types with different widths, the coercion functions can be as simple as extracting or padding bits on values. For functions, such as "+", defined over symbols, the new representation would be a defined over bit vectors. It should be kept in mind while specifying the data representation and coercion functions that some values (or set of values) may be impossible to represent in certain types. Ports in a behavior table that are connected in a net with ports in other behavior tables must have the same type. Changing the level of data abstraction of a port generates side-conditions for all ports in the net.

6 Example - Deriving an FM9001 Implementation

The FM9001 [19] is a 32-bit microprocessor, the third generation processor description defined by Hunt and mechanically verified at the gate level using the Nqthm theorem prover. Bose has derived an Actel FPGA implementation of the FM9001 using our digital design derivation tool [20], which is adept at handling datapath oriented designs with monolithic control. Here, we give a sketch some of the transformations on a behavior table specification of the FM9001. We start from a functional high-level specification with indirection, for a compact specification, with the goal of deriving a boolean behavior table implementation, interacting with a memory sub-system.

	next-state	regs	flags	mem	pc	ins	opa	opb	b-addr	oracle
0	intr	regs	flags	mem	#	#	#	#	#	oracle
1	fetch	regs	flags	mem	(car oracle)	#	#	#	#	(cdr oracle)
2	op-a	(ur regs pc (inc (@pc)))	flags	mem	pc	(read (@pc) mem)	#	#	#	oracle
3	op-b	regs	flags	mem	pc	ins	(extend (a-imm ins) 32)	#	#	oracle
4	op-b	regs	flags	mem	pc	ins	(@(rn-a ins))	#	#	oracle
5	op-b	(ur regs (rn-a ins) (dec (@ (rn-a ins))))	flags	mem	pc	ins	(read (dec (@(rn-a ins))) mem)	#	#	oracle
6	op-b	(ur regs (rn-a ins) (inc (@ (rn-a ins))))	flags	mem	pc	ins	(read (inc (@(rn-a ins))) mem)	#	#	oracle
7	op-b	regs	flags	mem	pc	ins	(read (@(rn-a ins)) mem)	#	#	oracle
8	alu-op	regs	flags	mem	pc	ins	opa	(@(rn-b ins))	#	oracle
9	alu-op	(ur regs (rn-b ins) (dec (@(rn-b ins))))	flags	mem	pc	ins	opa	(read (dec (@(rn-b ins))) mem)	(dec @(rn-b ins))	oracle
10	alu-op	(ur regs (rn-b ins) (inc (@ (rn-b ins))))	flags	mem	pc	ins	opa	(read (inc (@(rn-b ins))) mem)	(@(rn-b ins))	oracle
11	alu-op	regs	flags	mem	pc	ins	opa	(read (@(rn-b ins)) mem)	(@(rn-b ins))	oracle
12	intr	(ur regs (rn-b ins) (do-op . . .))	(uf flags ins opa opb)	mem	pc	ins	opa	opb	b-addr	oracle
13	intr	regs	(uf flags ins opa opb)	(write b-addr mem (do-op flags opa opb ins))	pc	ins	opa	opb	b-addr	oracle
14	intr	regs	(uf flags ins opa opb)	mem	pc	ins	opa	opb	b-addr	oracle

Table 2: Behavior Table Specification - Action Section

6.1 Control and Datapath Transformations

We start with a behavior table with five states, fifteen transitions, ten internal predicates and ten functional objects. The decision section of the table is shown in Table 1. The first transformation we perform is to *generalize* the functions (mode-a ins) and (mode-b ins) to (mode op ins) in the decision section, by adding a register op in the action section with the value a, when the next state is op-a, and b when the next state is op-b. We can now merge both the predicate columns with name (reg-direct-p (mode op ins)), because they are *conflict-free*. Similarly, we can merge both the (pre-dec-p ...) columns and both the (post-inc ...) columns. These transformations reduce the decision columns from ten to seven. We can also *generalize* the functions (rn-a ins) and (rn-b ins) in the action section, to a function (rn op ins) using the op register.
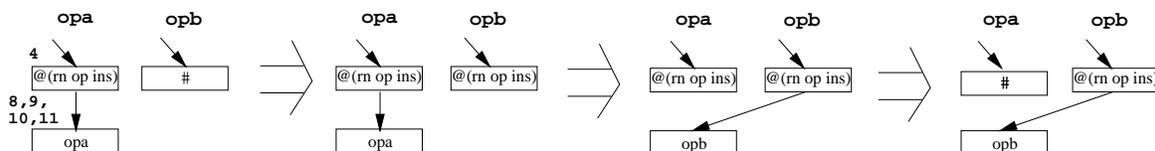

Figure 4: Datapath Transformations

Figure 4 shows the next sequence of transformations on data-flow diagrams for registers opa and opb after transition 4. The first transformation copies the values from the column opa to opb over the don't care values for transitions 4, 5, 6, and 7. We then change the values in column opa for transitions 8, 9, 10 and 11, from opa to opb. This is a valid transformation because the values in opa and opb are always the same in the state op-b, the source state for transitions 8, 9, 10 and 11. With this transformation it becomes apparent that the values in opa in the state op-b are not used in any transition out of the state, and are overwritten in every outgoing transition from that state. These "dead" values are replaced by don't cares in transitions 3, 4, 5, and 6, preserving the *implementation* relation over all sequences of transitions from state op-a to alu-op.
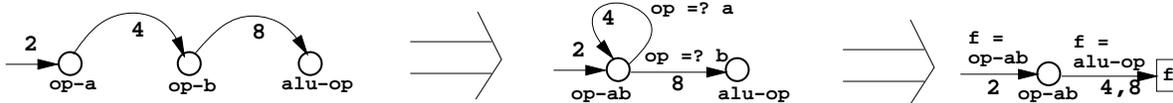

Figure 5: Control Transformations

The next set of transformations are illustrated in Figure 5. We merge the states op-a and op-b into a single state op-ab, and use the value in op (derived earlier) to decide on which transition to take. We then merge the transitions 4 and 8, which differ only in the next-state column. A register future-state ([f] in Figure 5) is added, which precomputes the next state in all transitions leading

16

to the source state of transition 4, and gets the value alu-op in transitions 4 and 8. The next-state values for transitions 4 and 8 are then replaced by register future-state for state indirection. Now, these two transitions are exactly the same, and can be merged. Similarly, transitions 5 and 9, 6 and 10, and 7 and 11 can also be merged resulting in reduction of one state and four transitions.

As we have seen here, control and datapath transformations go hand in hand. This reinforces our main thesis that, a design representation which unifies different facets of a system in a single formal model can be used to address unexplored aspects of system design. We will not elaborate on the other transformations on the control and datapath facets due to space constraints.

6.2 Protocol Transformations

We will now derive an implementation of the processor, by *sequential decomposition* of the memory object. Decomposition of the memory is based on the protocol specification of a memory sub-system, written below in ISL syntax:

$$\mathsf{Mem}(\mathrm{strobe}, \mathrm{RW}, \mathrm{ADDR}, \overline{\mathrm{dtack}}, \overline{\mathrm{DIN}}, \mathrm{DOUT}) \triangleq$$

$$[\; ; \; \mathrm{strobe/T}, \mathrm{RW/T} : \mathrm{ADDR}/V_{\mathrm{addr}} \; until \; \overline{\mathrm{dtack}} : \overline{\mathrm{DIN}}/V_{\mathrm{read}}$$

$$[\!| \quad ; \; \mathrm{strobe/T}, \mathrm{RW/F} : \mathrm{ADDR}/V_{\mathrm{addr}}, \mathrm{DOUT}/V_{\mathrm{write}} \; until \; \overline{\mathrm{dtack}} \;]^*$$

The state diagram for the memory is shown in Figure 6. The *complement* of Mem, $\overline{\mathsf{Mem}}$ is the description of its environment. Read and Write memory operations in the processor correspond to a path in $\overline{\mathsf{Mem}}$. These *path implementations* of $\overline{\mathsf{Mem}}$ are embedded into the processor description in place of the transitions with read and write operations (Figure 7). This assures us that the processor can interact properly with the memory.
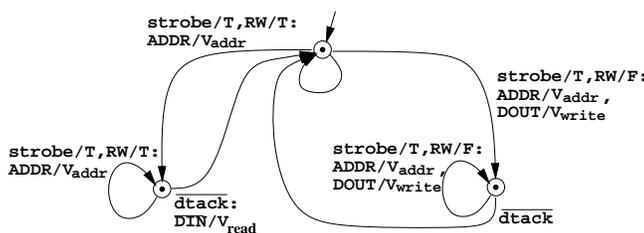

Figure 6: Memory Sub-system Protocol Specification

The column mem in Table 2 is replaced by the ports in $\overline{\mathsf{Mem}}$ as columns in the behavior table. The control input dtack is added as a column in the decision section. All read operations in the table are replaced by the port DIN, with appropriate values on $\overline{\mathrm{ADDR}}$ and $\overline{\mathrm{strobe}}$ in the embedded sequence of transitions. Similarly, write operations are replaced by appropriate values on $\overline{\mathrm{ADDR}}$,
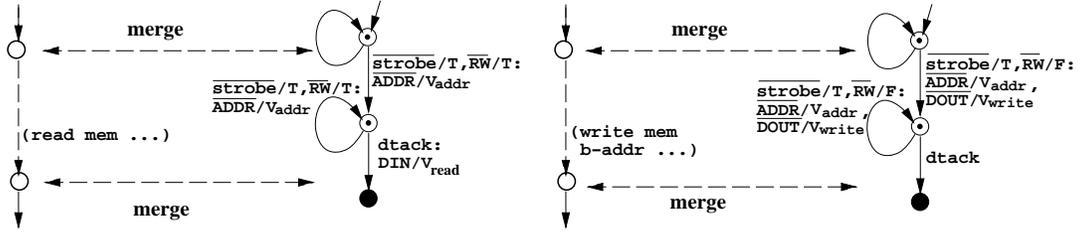
17

Figure 7: Embedding Path Implementations for Read and Write

$\overline{\text{strobe}}$ and $\overline{\text{DOUT}}$ in the embedded sequence of transitions. Other operations in the replaced transitions are scheduled in the embedded sequence of transitions such that the *implementation* relation on the control and datapath are preserved.

Our methodology is powerful enough to derive the memory controller for a DRAM memory system, with a protocol specification from DRAM timing diagrams, (reported in [21]).

6.3 Data Abstraction Transformations

Data abstraction is an important part of the design process, because it enables the designer to reason at the abstract symbolic level before assigning representations to functional units and values in a system. Consider the change of representation for the column ins from symbolic to 32-bit boolean. The transformation is shown in Figure 8.
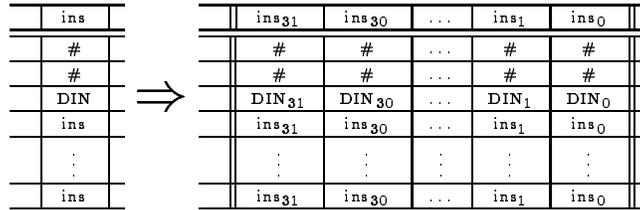

Figure 8: Data Abstraction Transformation

The only non don't care values in the column in Table 2, are ins and (read (@pc) mem). Let us assume that the read has been factored to the memory sub-system using protocol transformations, and replaced by DIN. Changing the representation of ins, will introduce a coercion on DIN from symbolic to 32-bit boolean. This in turn will generate a side-condition that the port $\overline{\text{DIN}}$ in Mem, which is connected to DIN, is also of type 32-bit boolean.

The other columns in the behavior table can similarly be changed to boolean values using the data representation function $\mathcal{R}$, and the value mapping function $\mathcal{V}$. The present, next, and future state columns are represented as boolean values after state assignment, which is also a part of $\mathcal{V}$.

18

7 Conclusion

In this paper, we presented behavior tables as a model for system representation. We described a transformational design environment based on behavior tables, that can be used to derive a correct implementation from a specification. The significance of our work is to enable a system designer to specify a system at a high-level of abstraction, and provide a design environment that can be used to transform the specification into an implementation at a lower level of abstraction. We bring together different aspects of system design in a unified reasoning framework. Transformations on control and datapath facets, and protocol and data abstraction facets of behavior tables, lend the foundation for a system design tool. Behavior tables also provide a useful visual interface to the designer to look at different facets of a design.

The research reported here is work-in-progress. A design environment based on behavior tables is being implemented using Motif widgets in X-windows, Scheme and C. Heuristic optimizations can be built on top of the transformations described here for automation of the design process.

In this paper, we introduced indirection in hardware description. We have used indirection to reference functional units within a behavior table. This can be extended to include references to components, ports, and nets outside a behavior table. We would like to explore other uses of indirection, especially for hardware-software co-design problems.

References

[1] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," in *Proceedings of International Conference on Computer Design*, pp. 328–333, IEEE, Oct. 1992.

[2] H. Trickey, "Flamel: A high-level hardware compiler," in *Transactions on Computer-Aided Design 1987*, pp. 259–269, IEEE, Mar. 1987.

[3] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn, "The system architect's workbench," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 337–343, 1988.

[4] R. Jain, K. Küçükçakar, M. J. Mlinar, and A. C. Parker, "Experience with the ADAM synthesis system," in *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 56–61, June 1989.

[5] G. C. Gopalakrishnan, R. M. Fujimoto, V. Akella, and N. S. Mani, "HOP: A process model for synchronous hardware; semantics and experiments in process composition," *Integration, the VLSI journal*, vol. 8, pp. 209–247, 1989.

[6] Z. Peng, *A Formal Methodology for Automated Synthesis of VLSI Systems*. PhD thesis, Linköping University, Sweden, 1987.

[7] M. R. K. Patel, "A design representation for high level synthesis," in *Proceedings of EDAC*, pp. 374–379, 1990.

[8] R. A. Walker and D. E. Thomas, "Behavioral transformation for algorithmic level IC design," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 10, pp. 1115–1128, 1989.

[9] F. Vahid and D. D. Gajski, "Specification partitioning for system design," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 219–224, 1992.

[10] K. Küçükçakar and A. C. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 514–519, 1991.

[11] S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations*. Cambridge: MIT Press, 1984. ACM Distinguished Dissertation 1984.

[12] B. Bose, "DDD - A Transformation system for Digital Design Derivation," Tech. Rep. 331, Department of Computer Science, Indiana University, May 1991.

[13] S. Devadas and K. Keutzer, "An Automata-Theoretic Approach to Behavioral Equivalence," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 30–33, November 1990.

[14] S. D. Johnson, "Manipulating logical organization with system factorizations," in *Hardware Specification, Verification and Synthesis: Mathematical Aspects* (Leeser and Brown, eds.), vol. 408 of *LNCS*, pp. 260–281, Springer, July 1989. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, 1989.

[15] K. Rath and S. D. Johnson, "Toward a basis for protocol specification and process decomposition," in *Proceedings of IFIP Conference on Hardware Description Languages and their Applications* (D. Agnew, L. Claesen, and R. Camposano, eds.), pp. 157–174, Elsevier, Apr. 1993. Also published as Technical Report No. 375, Dept. of Computer Science, Indiana University.

[16] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1988.

[17] S. Devadas and A. R. Newton, "Decomposition and factorization of sequential finite state machines," *Transactions on Computer-Aided Design 1989*, vol. 8, pp. 1206–1217, Nov. 1989.

[18] Z. Zhu and S. D. Johnson, "An algebraic framework for data abstraction in hardware description," in *Proceedings of The Oxford Workshop on Designing Correct Circuits* (Jones and Sheeran, eds.), Springer, 1990.

[19] W. A. Hunt, "A formal HDL and its use in the FM9001 verification," in *Mechanized Reasoning in Hardware Design* (C. Hoare and M. Gordon, eds.), Prentice-Hall, 1992.

[20] B. Bose and S. D. Johnson, "DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation," in *Proceedings of IFIP Conference on Correct Hardware Design and Verification Methods*, Springer, 1993.

[21] K. Rath, B. Bose, and S. D. Johnson, "Derivation of a DRAM memory interface by sequential decomposition," in *Proceedings of the International Conference on Computer Design*, pp. 438–441, IEEE, Oct. 1993.