A Compositional Circuit Model and Verification by Composition [*]

Zheng Zhu

Integrated System Design Laboratory
Department of Computer Science
The University of British Columbia
Vancouver, B.C. Canada V6T 1Z4
(zhu@cs.ubc.ca)

Abstract. To embrace the fast growth of circuit complexity, verification researchers are probing new verification methods. Verification by composition, among others, is regarded as a promising direction.
Symbolic Trajectory Evaluation (STE) is a theory for digital circuit verification. In the last a few years, STE has been used in proving practical digital circuits and has been proven a practical methodology with a mathematical foundation in circuit verification. However, the circuit model used in the existing STE verification systems is, in general, not compositional.
In this paper, we present a compositional circuit model. This model distinguishes two different types of unknown circuit values, *i.e.* driven undefined value and undriven undefined value. This treatment makes composition of circuit model possible. Major results of the paper are the following:

1. A language for describing finite state machines. This language is used to describe circuits behaviors. Expressions written in the language can be interpreted to the new model in this paper, as well as to the existing model. An operator in the language is designed for finite-state machine composition. The semantics of this operator is consistent to our intuitive understanding of "connecting two circuit node together". The major theorem concerning this operator is that it preserves the properties of the finite-state machines being composed.
2. The finite-state machine description can also be interpreted to the model which is used by the Voss system, a circuit verification system. A theorem shows that under certain conditions, two interpretations

[*] This research was supported, in part, by operating grants OGPO 109688 and OGPO O46196 from the Natural Sciences Research Council of Canada, fellowships from the Province of British Columbia Advanced Systems Institute, and by research contract 92-DJ-295 from the Semiconductor Research Corporation.

of the same finite-state machine description achieve the same verification results. This theorem allows us to perform circuit verification by using the well-developed STE verification system, and then to interpret the verification result in the model presented in this paper.

1 Introduction

To tackle fast growth of circuit complexity, verification researchers are probing new verification methods. Verification by composition, among others, is regarded as a promising and necessary direction.

To perform verification by composition, it is necessary that circuits are modeled in a compositional way. That is, composition of circuit models must preserve properties of each individual components. Therefore, properties which are verified in individual circuits do not need to be verified again in the composite.

The main focus of this paper is a circuit model which is compositional. It is an offspring of the circuit model used by Seger and Bryant [1] which regards a circuit node value as one of $\{0, 1, \perp, \top\}$ where 0, 1 are logical values, and $\perp$, $\top$ are under-defined and over-defined logical values respectively. As illustrated in Section 3 of this paper, an inadequacy of this quadruple circuit model is that it is not compositional.

The organization of this paper is the following: Section 2 briefly introduces the theory of symbolic trajectory evaluation (STE) and the quadruple circuit model used by STE systems. Section 3 first briefly discusses the inadequacy of the quadruple model with respect to model composition, and then introduces a circuit model, which is compositional. A language for describing finite state machines, named Set expressions, and its semantics are introduced. Section 4 discusses composition of finite state machines. Section 5 establishes a relationship between the proposed model and the one used in STE systems. Finally, Section 6 presents applications of this work.

2 STE and the $\mathcal{Q}$ Model

2.1 A Language for Trajectory Evaluation

In symbolic trajectory evaluation [1], system behaviors are given as trajectories over fixed length sequences of states. Each of these trajectories are described by a trajectory formula, called *trajectory formula.* A trajectory formula is in one the following forms:

1. unc. a constant in the language, which represents unconstrained circuit behavior.
2. Node specifications:
   (a) n is 0. The node "n" has value 0
   (b) n is 1. The node "n" has value 1.
3. $F_1 \wedge F_2$. Formulae F_1 and F_2 must both hold;
4. $E \rightarrow F$. The properties represented by the formula F need only hold the boolean expression E is evaluated to true;
5. $\mathbf{N}F$. $\mathbf{N}$ is the only temporal operator used in the language. $\mathbf{N}F$ specifies that F must hold in the following state.

A verification procedure checks *assertions* in the form of an implication $A \Rightarrow C$; the formula A (the antecedent) gives the stimulus and current state, and the formula C (the consequent) gives the desired response and state transition. Although the language has limited expressive power due to it's lack of such operators as disjunction and negation, along with temporal operators expressing properties of unbounded state sequences, it is designed as a compromise between expressive power and ease of evaluation. In practice, it is proven to be powerful enough to express timing and state transition behavior of circuits, while allowing assertions being verified efficiently.

2.2 Domain of Discourse

In symbolic trajectory evaluation, a circuit is modeled as operating over logic levels $0, 1$, a third value $\bot$ representing an indeterminate or unknown level, and a fourth value $\top$, representing an overly defined value (such as asserting value 0 and 1 to a circuit node at the same time). Let $\mathcal{Q} = \{0, 1, \bot, \top\}$. $\mathcal{Q}$ is partially ordered as shown in Figure 1.

$$\begin{tikzcd}
& \top \arrow[dl] \arrow[dr] & \\
0 \arrow[dr] & & 1 \arrow[dl] \\
& \bot &
\end{tikzcd}$$

The ordering relation is: $\bot \sqsubseteq 0, 1 \sqsubseteq \top$

Fig. 1. Partial Order of $\mathcal{Q}$

Intuitively, $\sqsubseteq$ orders the elements of Q according to the amount of information they carry: $\perp$ carries no information; 0, 1 carry fully defined circuit node values, and $\top$ is the overly defined, thus inconsistent value (too much information).

In lattice theory, a finite set S is a complete lattice under the partial ordering $\sqsubseteq$ if for every $a, b \in S$, there exist a unique smallest $c \in S$ (under the partial order $\sqsubseteq$) such that $a \sqsubseteq c$, $b \sqsubseteq c$, and a unique greatest $d \in S$ (under $\sqsubseteq$) such that $d \sqsubseteq a$, $d \sqsubseteq b$. Given $a, b \in S$, such unique c and d are denoted by $a \sqcup b$ (the least upper bound) and $a \sqcap b$ (the greatest lower bound) respectively. A finite set S has a least upper bound (greatest lower bound) under $\sqsubseteq$ if there exists a unique $l \in S$ ($g \in S$) such that for every $s \in S$, $l \sqsubseteq s$ ($s \sqsubseteq g$). By this definition, Q is a complete lattice under the partial ordering $\sqsubseteq$, where $\top$ and $\perp$ are the least upper bound and greatest lower bound of the lattice respectively. Furthermore, let $Q^n = Q \times Q \times \cdots \times Q$ be the cartesian product of n Qs. We can extend the relation $\sqsubseteq$ (of Q) to a relation of Q^n pair-wisely: for every $a, b \in Q^n$, $a \sqsubseteq b$ if and only if for every $i : 1 \leq i \leq n$, $a_i \sqsubseteq b_i$. It is easy to show that Q^n is a complete lattice under the extended relation $\sqsubseteq$, and for every $a, b \in Q^n$ and every $i : 1 \leq i \leq n$, $(a \sqcup b)_i = a_i \sqcup b_i$, $(a \sqcap b)_i = a_i \sqcap b_i$.

Our intention is to use Q^n as the set of all possible states of an n-node circuit. In practice, a circuit node is usually referenced by its name, or a character string, rather than by a natural number (as a subscript of a product). Therefore, it is often convenient to regard Q^n as a set of functions $N \rightarrow Q$ where N is a set of n node names. In particular, we use $\vec{\perp}$ to denote the function $N \rightarrow \{\perp\}$. That is, $\vec{\perp}(a) = \perp$ for every a.

To express the behavior of a system over time, we use *sequence* of circuit state, *e.g* sequence of elements in Q^n. Conceptually, these sequences are infinite, although the properties expressible in the language can be determined from some finite prefix of a sequence. Given two sequences (of elements of Q^n) $\sigma = \sigma^0\,\sigma^1\,\cdots$ and $\tau = \tau^0\,\tau^1\,\cdots$ we extend the relation $\sqsubseteq$ to the sequences pointwise: if $\sigma = \sigma^0\sigma^1\cdots$ and $\tau = \tau^0\tau^1\cdots$ are two sequences, then $\sigma \sqsubseteq \tau$ iff $\sigma^i \sqsubseteq \tau^i$ for every $i \geq 0$.

The definition of trajectory formulae can be extended to allow node specifications contain symbolic boolean expressions, rather than just 0 and 1. This extension makes specifications written in trajectory formulae very compact. Symbolic evaluation can be thought of as computing circuit behavior for many different operating conditions simultaneously, with each possible assignment of 0 or 1 to the variables in $\mathcal{V}$ indicating a different condition. Formally, this is expressed by defining an *assignment* φ to be a particular mapping from the elements of $\mathcal{V}$ to binary values. A formula F in the logic expresses some property of the circuit in terms of the symbolic variables. It may hold for only a subset of all possible assignments. Such a subset can be represented by a boolean domain function d over $\mathcal{V}$ yielding 1 for precisedly the assignments in the subset. For example, the constant functions 0 and 1, for example, represent the empty assignment set and the set of all possible assignments, respectively. However, allowing symbolic boolean expressions does not add any expressive power. Therefore, unless indicated explicitly, trajectory formulae in this paper are variable-free.

2.3 Use of Symbols

In this paper, we adopt the following convention of notations: every syntactic entity in a language (tern expressions and Set expressions, see Section 3.2) are in sans serif font, such as And, Not. Function symbols are represented by upper-case words, such as AND, NOT, UNION, plus conventional function symbols such as $\sqcup$ and $\sqcap$.

Uses of function $\sqcup$ and relation $\sqsubseteq$ are quite liberal in this paper. Although both are originally defined in a lattice such as Q, they are also used as binary function and relation on

1. functions such as elements of Q^n with the extension in a pair-wise manner; and
2. sequences of elements in Q^n with the extension: let $\sigma = \sigma^0\,\sigma^1\cdots$ and $\tau = \tau^0\,\tau^1\cdots$ by sequences such that for every $i \geq 0$, $\sigma^i \in Q^n$ and $\tau^i \in Q^n$, $\sigma \sqcup \tau$ is a sequence $\alpha = \alpha^0\,\alpha^1\cdots$ such that for every $i \geq 0$, $\alpha^i = \sigma^i \sqcup \tau^i$. $\sigma \sqsubseteq \tau$ if and only if for every $i \geq 0$, $\sigma^i \sqsubseteq \tau^i$.

2.4 Circuit Model Structures

A *circuit model structure* is $M = [(S, \sqsubseteq), Y]$ where S is the set of all functions from a set of nodes N to $\{0, 1, \bot, \top\}$, $\sqsubseteq$ is the ordering relation on S defined in the previous subsection, and Y is a monotone function $S \rightarrow S$. Let S^ω be the set of all (infinite) sequences of elements of S. In general, we are only interested in those sequences related to the behavior of a circuit model, namely, those sequences constrained by function Y in the model structure. We formalize this by introducing the concept of a trajectory. Given a model $M = [(S, \sqsubseteq), Y]$ and an arbitrary sequence $\sigma = \sigma^0\sigma^1\cdots \in S^\omega$, σ is a *trajectory* of M iff for every $i \geq 0$,

$$Y(\sigma^i) \sqsubseteq \sigma^{i+1}$$

We now assign a meaning to the specification language in terms of defining sequences. Let F be a trajectory formula, its *defining sequence*, denoted by $\delta(F)$, is defined as follows:

1. $\delta(\mathsf{unc}) = \vec{\bot}\,\vec{\bot}\cdots$.

2. Let $b \in \{0,1\}$, $\delta(n \text{ is } b)$ is a sequence $\sigma \vec{\bot} \cdots \vec{\bot} \cdots$ where σ a function $N \to \{0,1,\bot\}$ defined by: for every $x \in N$,

$$\sigma(x) = \begin{cases} b & x = n \\ \bot & x \neq n \end{cases}$$

3. $\delta(F_1 \wedge F_2) = \delta(F_1) \sqcup \delta(F_2)$.
4. $\delta(E \to F) = \begin{cases} \delta(F) & \text{E is evaluated to 1} \\ \vec{\bot}\,\vec{\bot} \cdots & \text{Otherwise} \end{cases}$
5. $\delta(\mathbf{N}F) = \vec{\bot}\,\delta(F)$.

Assume that $\delta(F) = \delta^0 \delta^1 \cdots$ is the defining sequence of formula F, define the *defining trajectory* of F constrained by Y, denoted by $\tau_Y(F) = \tau^0 \tau^1 \cdots$, as follows:

$$\tau^i = \begin{cases} \delta_F^0 & i = 0 \\ \delta_F^i \sqcup Y(\tau^{i-1}) & i > 0 \end{cases}$$

2.5 Specification and Verification

The truth semantics of trajectory formulae is defined relative to circuit model and its defining trajectories. In particular, given a circuit model M and a trajectory σ, a trajectory formula F is true on the trajectory σ, written $\sigma \models F$, is defined as follows:

1. $\sigma \models \text{unc}$ for all σ.
2. $\sigma^0 \sigma^1 \cdots \models n \text{ is } b$ iff $b \sqsubseteq \sigma^0(n)$.
3. $\sigma \models F_1 \wedge F_2$ iff $\sigma \models F_1$ and $\sigma \models F_2$.
4. (a) $\sigma \models E \to F$ if $\sigma \models F$ and E is evaluated to 1.
   (b) $\sigma \models E \to F$ for every σ if E is evaluated to 0.
5. $\sigma^0 \sigma^1 \cdots \models \mathbf{N}F$ iff $\sigma^1 \cdots \models F$.

A specification of a circuit is a pair of trajectory formulae A and C, denoted by $A \Rightarrow C$, where A and C are called *antecedent* and *consequent* respectively. $A \Rightarrow C$ is a specification of a circuit model $\mathcal{M}$ if for every trajectory σ of $\mathcal{M}$, $\sigma \models A$ implies $\sigma \models C$.

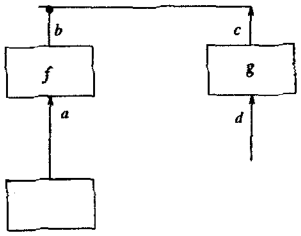
A major theorem proved in [1] is the following:

Theorem 1. *Let A and C be two trajectory formulae and $\mathcal{M} = [(S, \sqsubseteq), Y]$ be a circuit model. $\delta(C) \sqsubseteq \tau_Y(A)$, if and only if for every trajectory σ of $\mathcal{M}$, $\sigma \models A$ implies $\sigma \models C$, namely, $A \Rightarrow C$ is a specification of $\mathcal{M}$.*

Informally, this theorem can be interpreted as the following: the next-state function Y is a function from circuit states to circuit states. $\tau_Y(A)$ is the sequence of states when the input to the circuit is what specified by the formula A. Each state includes the values of input/output circuit nodes as well as internal state nodes. $\delta(C) \sqsubseteq \tau_Y(A)$ means that in every state, a circuit node value is either equal to what specified in the formula C, or its value is not mentioned in the formula C *i.e.* unrelated to the assertion $A \Rightarrow C$. By this theorem, in order to show that a circuit model has a property $A \Rightarrow C$, it is sufficient to show that $\delta(C) \sqsubseteq \tau_Y(A)$.

3 The Model $\mathcal{F}$

In this section, we introduce a slightly different lattice than $\mathcal{Q}$ for circuit modeling. The motivation of $\mathcal{F}$ can be illustrated by the example in Figure 2.



a, d	0	1	$\perp$
$b = f(a)$	0	1	$\perp$
$c = g(d)$	1	0	$\perp$

Fig. 2. Example: Composition of Circuits

In this example, functions f and g are defined in the truth table. An interesting situation is when the value on 'a' is $\perp$, and the value on 'd' is 0, which lead to $f(\perp) = \perp$ and $g(0) = 1$. Since the outputs of f and g are connected, how do we reconcile two different values $\perp$ and 1? There are two different interpretations to the fact $f(\perp) = \perp$, which lead to different answers to the question:

- The value on 'b' and 'c' can be $g(0) = 1$ if the node 'b' is not driven by any value, *i.e.* $\perp$ is interpreted as an undriven "unknown" value.
- Alternatively, the value on 'b' and 'c' can be an unknown value ($\top$) if 'b' is driven by some unknown but valid logical value, possibly be 0. *i.e.*, $\perp$ is interpreted as a driven but unknown value.

This example illustrate a need of differentiating two different type of $\perp$s, *i.e.* driven unknown value and undriven unknown value, when we consider composition of circuits. However, when modeling circuits by $\mathcal{Q}$, these two different unknown values are treated equally. Therefore, circuits modeled in $\mathcal{Q}$ are, in general, not compositional. This motivates our attempt to enrich $\mathcal{Q}$ in order to obtain a compositional model of circuits.

3.1 The Model $\mathcal{F}$

The example in Figure 2 reveals that, in order to be compositional, it is essential for a circuit model to distinguish two different types of unknown values. For this reason, we extend the lattice $(\mathcal{Q}, \sqsubseteq)$ to $(\mathcal{F}, \sqsubseteq)$ where $\mathcal{F} = \{0, 1, X, \bot, \top\}$ and the partial order $\sqsubseteq$ is shown in the following diagram.

$$\begin{tikzcd}
 & \top \arrow[dl] \arrow[d] \arrow[dr] & \\
0 \arrow[dr] & X \arrow[d] & 1 \arrow[dl] \\
 & \bot &
\end{tikzcd}$$

The ordering relation is: $\bot \sqsubseteq 0, X, 1 \sqsubseteq \top$

Apparently, $\mathcal{F}$ is a complete lattice under the ordering relation indicated by arrows in the picture. In the context of circuit modeling, the intuition behind these 5 values is the following: 0 and 1 have their conventional meaning. $\top$ is an over-constrained value. $\bot$ represents an unconstrained value. It could be used to model a don't-care input, or an undriven, unknown output value, such as the high-impedance state of a tri-state output. X is also an unconstrained value. The difference between X and $\bot$ is that X represents a driven unknown output value.

Conventional boolean functions, and $\sqcup$, $\sqcap$ can be extended to the values in $\mathcal{F}$. The following "truth-tables" are those for AND, NOT, $\sqcup$, and $\sqcap$:

	$\bot$	0	X	1	$\top$
$\bot$	X	0	X	X	$\top$
0	0	0	0	0	$\top$
X	X	0	X	X	$\top$
1	X	0	X	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

AND

	$\bot$	0	X	1	$\top$
$\bot$	$\bot$	0	X	1	$\top$
0	0	0	$\top$	$\top$	$\top$
X	X	$\top$	X	$\top$	$\top$
1	1	$\top$	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$\sqcup$

	$\bot$	0	X	1	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$	$\bot$	0
X	$\bot$	$\bot$	X	$\bot$	X
1	$\bot$	$\bot$	$\bot$	1	1
$\top$	$\bot$	0	X	1	$\top$

$\sqcap$

$\bot$	0	X	1	$\top$
X	1	X	0	$\top$

NOT

Similar to $\mathcal{Q}$-model structures, a $\mathcal{F}$-model structure is $((S, \sqsubseteq), Y)$ where S is the set of all functions from a set of nodes (names) to $\mathcal{F}$, and Y, the next-state function of the model structure, is a function $S \rightarrow S$.

3.2 Representation of Next-state Functions

We now introduce a notation for describing next-state functions of finite state machines: tern and Set expressions. A tern expression can be regarded as an extension of boolean expressions in $\mathcal{F}$, and a Set expression can be regarded as a description of next-state function of a finite-state machine.

3.2.1 tern Expressions and Set Expressions

A tern expression is defined as:

tern ::= One | Zero | X | Val str | And tern tern | Not tern

where One, Zero, X, And, and Not are the syntactical representations of 0, 1, X, and functions AND, and NOT respectively, and

Val str

is used to refer to the value on the node which is named by the string str. It plays a role similar to that of boolean variables in boolean expressions. As a concrete example, the following tern expression

(Not (And (Val 'in1') (Val 'in2')))

describes the output of an NAND gate whose input nodes are 'in1' and 'in2' respectively.

In the definition of tern expression, we carefully excluded symbols which correspond to $\perp$ and $\top$ in $\mathcal{F}$. This choice will be justified when the semantics of tern and Set expressions is presented later.

A Set expression is defined as:

Set ::= Empty | Element str, Driver | Union Set Set | Join Set Set
Driver ::= (tern, tern)

The constant Empty corresponds to an empty finite state machine (which has neither internal state nor output). Element is the constructor that actually introduces a new node which is named by the string str and defines driver functions for the node. A driver function is given in Driver which is a pair of tern expressions. the first expression is a guard and the second expression is the value being driven when the guard is evaluated to 1. For example,

Element 'n' (One, Zero)

creates a circuit node whose value is always 0.

The constructor Union is used to create a collection of Element definitions.

$$\text{Union } S_1 \ S_2$$

is a collection of node drivers which contains all the node drivers in either S_1 or S_2. If there are both S_1 and S_2 contain a driver for the same node, then the Union constructor will use the greatest lower bound of the values being driven at the same time when both guards are evaluated to 1 (See more in Section 3.2.2).

The expression

$$\text{Join } S_1 \ S_2$$

is similar to Union S_1 S_2, except when both S_1 and S_2 contain a driver for the same node, then the Join operator will use the least upper bound of the values being driven at the same time when both guards are evaluated to 1.

3.2.2 Interpreting Set Expressions to $\mathcal{F}$

The semantics of Set expressions includes evaluation of tern expressions, and an interpretation of Set expressions. This interpretation effectively translates a Set expression to a next-state function (of a finite-state machine).

Given a tern expression t, t is a constant if it does not contain any Val (such as Val 'a') subexpressions. The evaluation of a constant tern expression t, denoted as $\mathcal{E}(t)$, maps t to an element in $\mathcal{F}$:

1. $\mathcal{E}(\text{Zero}) = 0$, $\mathcal{E}(\text{One}) = 1$, and $\mathcal{E}(\text{X}) = X$;
2. $\mathcal{E}(\text{And } t_1 \ t_2) = \text{AND } \mathcal{E}(t_1) \ \mathcal{E}(t_2)$;
3. $\mathcal{E}(\text{Not } t) = \text{NOT } \mathcal{E}(t)$

Given a Set expression S, *a node of* S is a circuit node name (str) such that either

$$\text{Val str}$$

appears in S, or

$$\text{Element str driver}$$

appears in S. A node space of S is a set of nodes such that every node of S belongs to the set. Apparently, there are more than one node space of a given S. Without losing generality, we assume that there exists a universal set of circuit nodes (names), denoted by $\mathcal{U}$, which includes all the circuit nodes we are interested in. We use $\mathcal{U}$ as the node space of any given Set expression, unless explicitly indicated otherwise.

A state is a function $\varphi : \mathcal{U} \rightarrow \mathcal{F}$. An example of such a function is $\vec{\bot} : \mathcal{U} \rightarrow \{\bot\}$. That is, for every $a \in \mathcal{U}$, $\vec{\bot}(a) = \bot$.

We now extend the evaluation of a constant tern expression to arbitrary tern expressions: given a state φ, the evaluation of a tern expression t in the state φ, denoted as $\mathcal{E}(t, \varphi)$, is defined recursively as:

1. $\mathcal{E}(\mathsf{Zero}, \varphi) = 0$, $\mathcal{E}(\mathsf{One}, \varphi) = 1$, and $\mathcal{E}(\mathsf{X}, \varphi) = X$;
2. $\mathcal{E}(\mathsf{Val}\ str, \varphi) = \varphi(str)$;
3. $\mathcal{E}(\mathsf{And}\ t_1\ t_2,\ \varphi) = \text{AND}\ \mathcal{E}(t_1, \varphi)\ \mathcal{E}(t_2, \varphi)$;
4. $\mathcal{E}(\mathsf{Not}\ t,\ \varphi) = \text{NOT}\ \mathcal{E}(t, \varphi)$

The interpretation (semantics) of a Set expression S, denoted by $[\![S]\!]$, is a function which maps a state to a state: for any state s,

1. $[\![\mathsf{Empty}]\!](s) = \vec{\bot}$.
2. $[\![\mathsf{Element}\ n, (g, v)]\!](s)$ is $\psi : \mathcal{U} \rightarrow \mathcal{F}$, such that for every $a \in \mathcal{U}$,

$$\psi(a) = \begin{cases} \mathcal{E}(v, s) & \text{if } \mathcal{E}(g, s) = 1 \text{ and } n = a \\ \bot & \text{Otherwise} \end{cases}$$

3. $[\![\mathsf{Union}\ S_1\ S_2]\!](s) = \text{UNION}\ [\![S_1]\!](s)\ [\![S_2]\!](s)$.
4. $[\![\mathsf{Join}\ S_1\ S_2]\!](s) = [\![S_1]\!](s) \sqcup [\![S_2]\!](s)$.

where the function UNION (in $\mathcal{F}$) is defined by the following "truth-table":

	$\bot$	0	X	1	$\top$
$\bot$	$\bot$	0	X	1	$\bot$
0	0	0	$\bot$	$\bot$	0
X	X	$\bot$	X	$\bot$	X
1	1	$\bot$	$\bot$	1	1
$\top$	$\bot$	0	X	1	$\top$

The truth-table of the function UNION

4 Finite State Machine Compositions

The purpose of $\mathcal{F}$ is to provide the capability of composing next state functions of finite state machines. In this section, we show that the Join constructor of Set expression realizes circuit composition. The relationship between the Join and Union operators will also be discussed.

4.1 Circuit Model Compositions

In circuit designs, composition of two (or more) physical circuits means connecting (wiring) nodes with the same name (in different circuits) together to form a new circuit. The following "truth table" gives our understanding of "connecting two node values" in $\mathcal{F}$:

	$\bot$	0	X	1	$\top$
$\bot$	$\bot$	0	X	1	$\top$
0	0	0	$\top$	$\top$	$\top$
X	X	$\top$	X	$\top$	$\top$
1	1	$\top$	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

The truth table of "connecting" values in the 5-element domain. Note that the truth table is identical to that of the $\sqcup$ operator in the same domain.

Let A, C be any trajectory formulae, S_1, S_2 be any Set expressions. Also let $Y = [\![\text{Join } S_1\ S_2]\!]$, $Y_1 = [\![S_1]\!]_Q$, $Y_2 = [\![S_2]\!]_Q$. If $\tau_Y(A)$ does not have any $\top$ element, (*i.e.* let $\tau_Y(A) = \tau^0 \cdots \tau^n \cdots$. For every $i \geq 0$, $\top$ is not in the range of τ^i.) then

Theorem 2. *$\delta(C) \sqsubseteq \tau_{Y_1}(A)$ implies $\delta(C) \sqsubseteq \tau_Y(A)$, and $\delta(C) \sqsubseteq \tau_{Y_2}(A)$ implies $\delta(C) \sqsubseteq \tau_Y(A)$.*

This theorem shows properties which are held in components are also held in composition, if the composition is modeled by the constructor Join.

4.2 Implementation of Model Compositions

In practice, the function JOIN poses two problems:

1. Potentially, it may create a large number of $\top$ elements. To catch all these $\top$ elements requires extensive computing resource. According to our experience, it is responsible for up to 5% increase of computing time during verifications.
2. The verification may be too pessimistic: it is may be acceptable to have a circuit node have a $\top$ value. In fact, during transient states, occurrences of $\top$ on a circuit node is quite common and does not necessarily mean that the circuit presents undesirable behavior.

To solve these two problems, we use the Union operator in the Set expressions, rather than Join, to compose two Set expressions.

From the truth-table of UNION (Section 3.2.2), we find that it has the following properties:

1. It generates far less $\top$s than $\sqcup$ does;
2. UNION does not generate a $\top$ unless both operands are $\top$.

3. Except the bottom row and the right-most column, the truth-table of UNION corresponds that of $\sqcup$ in the following way: the corresponding entries in the tables are either equal to each other, or the entry in the table of $\sqcup$ is $\top$ and the entry in that of UNION is $\bot$.

The last property is essential: from verification's perspective, if a circuit node has the value $\bot$, then nothing can be concluded about that circuit node's value, except its being $\bot$; If a node value is $\top$, then it implies that the circuit may have had undesirable behavior, such as short-circuits. This implies a possible malfunction of the circuit implementation unless the specification does not explicitly mention the value of this node *i.e.* the node has value $\bot$. Therefore, it is safe to replace $\top$ values in a trajectory (sequence of circuit states) by $\bot$ as far as verification is concerned. This property suggests that circuit composition might be realizeded by the Union operator as well. The following theorem confirms this.

Let S_1 and S_2 be two Set expressions, $Y = [\![\mathsf{Join}\ S_1\ S_2]\!]$, and $Y' = [\![\mathsf{Union}\ S_1\ S_2]\!]$. The following theorem shows the relationship between Y and Y':

Theorem 3. *Given trajectory formulae A and C,*

1. *If $\delta(C) \sqsubseteq \tau_{Y'}(A)$, then $\delta(C) \sqsubseteq \tau_Y(A)$.*
2. *If $\delta(C) \sqsubseteq \tau_Y(A)$ and $\tau_Y(A)$ does not contain any $\top$ value, then $\delta(C) \sqsubseteq \tau_{Y'}(A)$.*

This theorem says that, instead of verifying $\delta(C) \sqsubseteq \tau_Y(A)$, we verify $\delta(C) \sqsubseteq \tau_{Y'}(A)$. If $\delta(C) \sqsubseteq \tau_{Y'}(A)$ holds, then $\delta(C) \sqsubseteq \tau_Y(A)$ holds. Otherwise, either $\delta(C) \sqsubseteq \tau_Y(A)$ fails, or $\tau_Y(A)$ contains $\top$ values. By proving $\delta(C) \sqsubseteq \tau_{Y'}(A)$, we can avoid significant amount $\top$-checking, and avoid to be too pessimistic in interpreting verification results.

5 Relationship Between the $\mathcal{Q}$-Model and the $\mathcal{F}$-Model

This section discusses the relationship between the two different circuit models. The purpose of establishing such a relationship is practical: Voss, a circuit verification system which uses the $\mathcal{Q}$-model, has been developed and proven practical in circuit design verifications. In fact, both tern expressions and Set expressions are the integrated components of the Voss system. Establishing such a relationship between $\mathcal{Q}$-model and $\mathcal{F}$-model allows us conduct verification in Voss and then to interpret the result in $\mathcal{F}$-model.

5.1 Interpreting Set Expressions to the $\mathcal{Q}$-Model

In $\mathcal{Q}$, the functions AND, NOT, $\sqcup$, and UNION are as defined in the following "truth-tables":

	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	$\bot$	$\top$
0	0	0	1	$\top$
1	$\bot$	1	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

AND

$\bot$	0	1	$\top$
$\bot$	1	0	$\top$

NOT

	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\sqcup$

	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\bot$	0
1	1	$\bot$	1	1
$\top$	$\bot$	0	1	$\top$

UNION

Note that the extensions of AND, and NOT to $\mathcal{Q}$ is carefully done so that both functions are monotonic. This is to accommodate the requirement that every next-state function in a $\mathcal{Q}$-model has to be monotonic [1].

Given a tern expression t, it can be evaluated to a value in $\mathcal{Q}$ the same way as we evaluate t to $\mathcal{F}$. To distinguish the evaluation of a tern expression to $\mathcal{Q}$ and that to $\mathcal{F}$, we add subscripts to the evaluation function $\mathcal{E}$: *i.e.* $\mathcal{E}_{\mathcal{Q}}$ ($\mathcal{E}_{\mathcal{F}}$) evaluates a tern expression to $\mathcal{Q}$ ($\mathcal{F}$). Similarly, we add subscripts to the semantic function $[\![S]\!]$ *i.e.* $[\![S]\!]_{\mathcal{Q}}$ and $[\![S]\!]_{\mathcal{F}}$ respectively, in order to distinguish the different interpretations in $\mathcal{Q}$ and $\mathcal{F}$.

We now interpret Set expressions in $\mathcal{Q}$: given a Set expression S,

1. $[\![\mathsf{Empty}]\!]_{\mathcal{Q}}(s) = \vec{\bot}$.
2. $[\![\mathsf{Element}\ n,(g,v)]\!]_{\mathcal{Q}}(s)$ is $\psi : \mathcal{U} \rightarrow \mathcal{Q}$, such that for every $a \in \mathcal{U}$,

$$\psi(a) = \begin{cases} \mathcal{E}_{\mathcal{Q}}(v,s) & \text{if } \mathcal{E}_{\mathcal{Q}}(g,s) = 1 \text{ and } n = a \text{ and } \mathcal{E}_{\mathcal{Q}}(v,s) \neq X \\ \bot & < \text{Otherwise} \end{cases}$$

3. $[\![\mathsf{Union}\ S_1\ S_2]\!]_{\mathcal{Q}}(s) = \text{UNION}\ [\![S_1]\!]_{\mathcal{Q}}(s)\ [\![S_2]\!]_{\mathcal{Q}}(s)$.
4. $[\![\mathsf{Join}\ S_1\ S_2]\!]_{\mathcal{Q}}(s) = [\![S_1]\!]_{\mathcal{Q}}(s) \sqcup [\![S_2]\!]_{\mathcal{Q}}(s)$.

5.2 The Relationship Between $[\![S]\!]_{\mathcal{F}}$ and $[\![S]\!]_{\mathcal{Q}}$

The difference between $[\![S]\!]_{\mathcal{Q}}$ and $[\![S]\!]_{\mathcal{F}}$ is the treatments when $\mathcal{E}(v,s) = X$, $\mathcal{E}(g,s) = 1$, and $n = a$ (in $[\![\mathsf{Element}\ n,(g,v)]\!]_{\mathcal{Q}}(s)$ and $[\![\mathsf{Element}\ n,(g,v)]\!]_{\mathcal{F}}(s)$). In this situation, $[\![S]\!]_{\mathcal{Q}}(s)(a) = \bot$ whereas $[\![S]\!]_{\mathcal{Q}}(s)(a) = X$. This is because that, in $\mathcal{Q}$, there is no distinction between driven unknown value and undriven unknown value, they both are represented by $\bot$. To further illustrate this subtle difference, let us compare the Set expression Empty with

Element 'n' (One, X)

Since $[\![\mathsf{Empty}]\!]_{\mathcal{Q}} = \vec{\perp}$ and $[\![\mathsf{Element}\ 'n'(\mathsf{One}, \mathsf{X})]\!]_{\mathcal{Q}} = \vec{\perp}$, the two expressions are equivalent in $\mathcal{Q}$. On the other hand, $[\![\mathsf{Empty}]\!]_{\mathcal{Q}} = \vec{\perp}$, $[\![\mathsf{Element}\ 'n'(\mathsf{One}, \mathsf{X})]\!]_{\mathcal{Q}}$ is a function f such that $f(n) = X$, therefore, $[\![\mathsf{Empty}]\!]_{\mathcal{F}} \neq [\![\mathsf{Element}\ 'n'(\mathsf{One}, \mathsf{X})]\!]_{\mathcal{Q}}$. Although very subtle, this difference is exactly we have been after: Empty represents a finite state machine whose value on the node 'n' is undriven unknown whereas

Element 'n' (One, X)

represents a finite state machine whose value on the node 'n' is driven unknown.

The following theorem establishes an important relationship between $[\![S]\!]_{\mathcal{Q}}$ and $[\![S]\!]_{\mathcal{F}}$:

Theorem 4. *Let S be any Set expression such that $[\![S]\!]_{\mathcal{Q}}$ is monotonic. Let $Y = [\![S]\!]_{\mathcal{F}}$ and $Y' = [\![S]\!]_{\mathcal{Q}}$. For any trajectory formulae A, and C,*

1. *if $\delta(C) \sqsubseteq \tau_{Y'}(A)$, then $\delta(C) \sqsubseteq \tau_Y(A)$.*
2. *if $\delta(C) \sqsubseteq \tau_Y(A)$, and $\tau_Y(A)$ does not contain any $\top$, then $\delta(C) \sqsubseteq \tau_{Y'}(A)$.*

This theorem allows us to conduct verification in the Voss system, which models circuits in $\mathcal{Q}$, and interpret the verification results in $\mathcal{F}$: if $\delta(C) \sqsubseteq \tau_{Y'}(A)$, then $\tau_C \sqsubseteq \tau_Y(A)$. If $\delta(C) \not\sqsubseteq \tau_{Y'}(A)$, then either $\delta(C) \not\sqsubseteq \tau_Y(A)$, or $\tau_Y(A)$ contains $\top$ elements. Neither case is desirable thus $A \Rightarrow C$ is not an assertion of the given circuit. Disallowing $\top$ elements in $\tau_Y(A)$ is a way to obtain pessimistic verification results. The Voss system is capable of checking and reporting whenever a trajectory ($\tau_Y(A)$) contains a $\top$ value.

6 Application

6.1 The Voss System

The Voss system, a formal hardware verification system developed at the University of British Columbia, consists of three major components: a highly efficient implementation of OBDDs; an event driven symbolic simulator with very comprehensive delay and race analysis capabilities [2]. From a user's point of view, the Voss system verifies assertions in the form of

FSM fsm (ante, cons)

where fsm denotes a finite-state machine and the pair (ante, cons) represents a trajectory assertion to be verified. The finite-state machine fsm is specified by a next-state function.

The following table summaries several verification experiments conducted by the Voss system:

Circuits	# Transistors	Time (sec)	Memory
64 × 32-bit moving data stack	16,470	35	1.2M
64 × 32-bit stationary data stack	15,383	200	3.2M
UART	9840	400	9.0M
32-bit Tamarack (synchronous)	7,214	320	10.1M
32-bit Tamarack (asynchronous)	7,214	530	10.1M
32-bit RISC core (32 regs)	16,100	7,300	35.4M

6.2 Verification by Composition

Originally, our work on verification by composition was motivated by verification of a system of more than one components. It is often the case that the designer of one component does not have finite state machines of other components when his design needs to be verified. In this situation, traditional verification methods may not be applicable due to a lack of a complete description of implementation *e.g.* a finite state machine. Although the designer could make assumptions about the environment (behaviors of other components) in order to valid his own design, such assumptions may disguise design errors even verifications produce positive answers.

To briefly illustrate our approach to the above mentioned problem, let us examine various verification exercises performed on the Tamarack microprocessor [3, 4]. Originally, a memory is an integrated component of the microprocessor's behavioral specification [3]. However, various Tamarack implementations which we are aware of do not contain such a memory module. Instead, memory is treated as part of the environment in which the microprocessor operates. To correlate the verification results to the original specification, assumptions of the behavior of the memory module have to be made.

There are mainly two reasons justifying separation of processor and memory in various verification efforts:

- The complexity of memory would make verification by automated verifiers, such as Voss, computationally intractable.
- Tamarack designs usually do not contain a memory module. Therefore, a complete implementation of Tamarack is usually not available to verification exercises.

However, the correctness of an implementation can only be established in an environment which realizes the memory module faithfully. Conceptually, our approach, *i.e.* verification by composition, proceeds as follows:

1. Obtaining a specification for the processing unit of Tamarack and verify an implementation of the unit against the specification. The work reported in [4] accomplished this in hybrid hardware verification system [5, 6].
2. Obtaining a specification of the memory module and then generating an abstract circuit model automatically:
   Automatically generate a Set expression S_m from the trajectory assertions of memory. In [7], we reported a constructive method which generates a finite-state machine from a given set of trajectory assertions such that $[\![S]\!]_{\mathcal{F}}$ is the next-state function of some circuit model of the given set of assertions. We also proved that such a finite state machine is the smallest machine in terms of the number of internal states [8].
3. Translate the next-state function of Tamarack to an equivalent Set expression S_t, according to the interpretation of Set expressions presented in Section 3.2.2[2].
4. Perform the Union operation on S_m and S_t to obtain a new Set expression $S = Union\ S_1\ S_2$..
5. Finally, verify the original Tamarack specification against S by the Voss system. An important difference between this verification and other verification exercise, such as [4], is that, if succeeds, guarantees the correctness of the processor/ memory composition.

7 Conclusion

In this paper, we presented a 5-element circuit domain which is compositional. The purpose of devising this model is to facilitate verification by composition. The major results can be summarized as follows:

1. We presented a language for finite state machines (SetExpressions). This language is used to describe circuits behaviors. Expressions written in the language can be interpreted to $\mathcal{F}$, as well as to $\mathcal{Q}$. An operator in the language is designed for finite-state machine composition. The semantics of this operator is consistent to our intuitive understanding of "connecting two circuit node together". The major theorem concerning this operator is that it preserves the properties of the finite-state machines which are being composed (Theorem 2).
2. The finite-state machine description can also be interpreted to the model which is used by the Voss system, a circuit verification system. A theorem

[2] This is an ongoing work.

(Theorem 4) shows that under certain condition, two interpretations of the same finite-state machine description achieve same verification results. This theorem allows us to perform circuit verification by using the well-developed Voss system, and then interpret the verification result in the model presented in this paper.

References

1. SEGER, C.-J., AND BRYANT, R. Formal verification of digital circuits by symbolic evaluation of partially-ordered trajectories. Tech. Rep. Technical Report 93-8, The Computer Science Department, The University of British Columbia, The Computer Science Department, The University of B.C. Vancouver B.C. V6T 1Z4, 1993. To appear in *Journal of Formal Methods in System Design.*
2. SEGER, C.-J. Voss – a formal hardware verification system, user's guide. Tech. Rep. Technical Report 93-45, The Computer Science Department, The University of British Columbia, The Computer Science Department, The University of B.C. Vancouver B.C. V6T 1Z4, 1993.
3. JOYCE, J. *Multi-level Verification of Microprocessor-Based Systems.* PhD thesis, Computer Laboratory, Cambridge University, 1989.
4. ZHU, Z., JOYCE, J., AND SEGER, C.-J. Verification of the tamarack-3 microprocessor in a hybrid verification environment. In *Proceedings of 1993 international meeting on Higer Order Logic and its Applications, Lecture Notes in Computer Science, Vol 780* (1993), Springer-Verlag.
5. SEGER, C.-J., AND JOYCE, J. A mathematically precise two-level formal verification methodology",. Tech. Rep. Report-92-34, Computer Science Department, The University of British Columbia, 1992.
6. JOYCE, J., AND SEGER, C.-J. Linking bdd-based symbolic evaluation to interactive theorem-proving. In *Proceedings of 30th Design Automation Conference* (1993).
7. ZHU, Z. Construction of circuit models from trajectory specifications. in progress, Janurary 1994.
8. ZHU, Z., AND SEGER, C.-J. Model construction from trajectory assertion and the completeness of a hardware inference system. In *The proceedings of the Sixth International Conference on Computer Aided Verification (CAV94), Lecture Notes in Computer Science, Vol 818* (1994), Springer-Verlag.