UNIVERSITY OF CALIFORNIA
SANTA CRUZ

Chai- A TOOL FOR SYNCHRONOUS INTERFACES

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Vaibhav Bhandari

December 2003

The thesis of Vaibhav Bhandari
is approved:

Professor Luca de Alfaro, Chair

Professor Jim Whitehead

Professor Scott Brandt

I think it's someone different now.

Frank Talamantes
Vice Provost and Dean of Graduate Studies

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Interface models [CdAHM02] help to decompose a design into components that can be implemented independently. Interface models can represent both the input or environment requirements of a module and its output behavior or guarantees. Thus, interface modules enable the decomposition of a global design problem into the design of smaller components, with the guarantee that the components, once implemented, will work together correctly. In this thesis we present CHAI, a tool for applying interface models to hardware design. Specifically we elaborate on the implementation issues for CHAI, and on the construction of a smooth path from standard hardware design languages such as Verilog to CHAI.

is a module the same as a component? In software architecture research, the answer to this question varies.

Interface models capture the behavior of a system component, and the interaction between the component and its environment. Since interfaces allow a designer to model assumptions about the environment they can effectively handle formalization of component based designs. They aid component based design by allowing:

- **Top-down design decomposition.** A design is decomposed into components that can be designed and implemented separately. The component model is used to specify the task of each component; and the interface specification ensures that the components, once implemented, can work together correctly.

- **Component Reuse**. In component-based approach, designs are created by combining pre-existing and application-specific components promoting component reuse. The models of the components and their interfaces help in selecting and combining the components, and in checking that the component interfaces are compatible with one another.

- **Compositional verification.** In order to verify a complete design, each component is studied with the help of assumptions about its environment. The results for the single components are then combined into an analysis for the complete system. Interface models capture both the assumptions about the environment, and the component behavior. If interface models are compatible one can be sure that the component implementations are compatible.

To enable usage of interfaces models we have implemented a tool - **Chai**. CHAI is intended to be a vehicle for experimentation with interfaces and related compositional verification algorithms and methodologies. CHAI is an extension of MOCHA [?], and it follows a software architecture similar to VIS [RGA+96]; it is written entirely in C and its shell user interface is provided by Tcl. In CHAI **Designers and application developers** can customize their application or design with their own

italics, not bold

user interface by writing Tcl scripts while a **Algorithm developers and researchers** italics
can develop new algorithms by writing C code.

The input language of Chai for interfaces is **interface modules**. Interface modules are built on Reactive Modules. citation? Reactive Modules enable only the description of the output behavior of a component, Interface Modules add to this the capability of describing input assumptions.

In order to build a smooth path from standard design languages to Chai, we have implemented a translator, mv2rm, from BLIF-MV (Berkeley Logic Interchange citation Format - Multivariate) to the input language of Chai. As many design languages, such as Verilog, VHDL, and Esterel, can be translated into BLIF-MV, the translator mv2rm opens the way to the use of interface models in the design and analysis of real hardware.

1.1 Organization of this thesis

Chapter 2 describes and defines interface modules, their composition and compatibility. In Chapter 3 we explore Chai and introduce its roots. It explains modelling with interfaces and sheds some light on implementation of interface modules. Chapter 4 presents conversion of hardware description languages to interface modules. Specifically it elaborates mv2rm, a tool implemented to convert from BLIF-MV to Reactive Modules. Chapter 5 concludes this work.

Chapter 2

Interfaces

2.1 Interface Modules

No component is designed in isolation: a component must interact with an *environment* (either the user, or other design components), and produce some useful output or behavior. *Interface modules* provide a way of modeling both aspects of component development: the input (or design) assumptions, and the output guarantees (or output behavior).

To give ~~For~~ the eager reader ~~to get~~ a taste of interface modules, Figure 2.1 illustrates a simple synchronous interface of a 8-bit ± 1 adder controlled by a binary counter. The interface formalism is described as an interface module with state variables partitioned as inputs and outputs. The acceptable state variable changes are described by transition relations of input atoms, and possible output state variable changes are described as transition relations of output atoms. We will consider parts of this example ~~as~~

```
interface Counter
  input  vars: cl:     bool;
  output vars: q0, q1: bool;

  input atom
    controls cl
    init
      [] true -> cl :=nondet
    update
      [] true -> cl':=nondet
  endatom
  output atom
    controls q0,q1
    reads cl, q0, q1
    init
      [] true -> q0:=1; q1:=1;
    update
      [] cl               -> \
                 q1':=1; q0':=1
      [] ~cl &  q1 &  q0 -> \
                 q1':=1; q0':=0
      [] ~cl &  q1 & ~q0 -> \
                 q1':=0; q0':=1
      [] ~cl & ~q1 &  q0 -> \
                 q1':=0; q0':=0
      [] ~cl & ~q1 & ~q0 -> \
                 q1':=1; q0':=1
  endatom
end interface
```

```
interface Adder
  input  vars: q0, q1: bool; \
                    di: [0..7];
  output vars: do: [0..7];

  input atom
    controls q0, q1
    init
      [] true -> q0:=1
      [] true -> q1:=1
    update
      [] true -> q0':=1
      [] true -> q1':=1
  endatom
  output atom
    controls d0
    reads q0, q1
    init
      [] true -> do:=nondet
    update
      []  q0 &  q1 -> do':=di'
      [] ~q0 &  q1 -> do':=di'+1
      []  q0 & ~q1 -> do':=di'-1
  endatom
end interface
```

Figure 2.1: A counter and a ±1 adder, modelled as a ~~s~~ynchronous interface.

~~illustrations through~~ in coming sections to explain various aspects of interfaces.

2.2 Basics

This section is modelled after [dA01]. In order to formally define INTERFACE MODULES we present a few relevant terms.

Variables. Consider an infinite global set W of *typed variables,* from which the variables of the modules will be drawn. Each variable $x \in W$ has an associated *domain,* or set of possible values, which we denote $D(x)$. We treat $D(x)$ as a finite set, restricting our attention to finite-state systems.

States. Given a finite set $V \subseteq W$ of variables, a *state* s over V is a function that associates with each variable $x \in V$ a value $s(x) \in D(x)$; we denote by $S[V]$ the set of all possible states over the variables V. Note that, formally, the type of $s \in S[V]$ is $\prod_{x \in V}(x \mapsto D(x))$. Given a state $s \in S[V]$ and a subset $U \subseteq V$ of variables, we denote by $s[U] \in S[U]$ the restriction of s to the variables in U: precisely, $s[U]$ is defined by $s[U](x) = s(x)$ for all $x \in U$. For any two sets V, U of variables, and states $s \in S[V]$ and $t \in S[U]$, we write $s \simeq t$ if $s(x) = t(x)$ for all shared variables $x \in V \cap U$.

State predicates. We assume a logical language Lang in which assertions about the values of the variables in W can be written. For example, if all variables are boolean, then Lang can be taken to be predicate logic with the addition of the quantifiers $\forall$ and $\exists$ over the booleans. We say that a formula $\phi \in$ Lang is *over* a set V of variables if it only involves variables of V; such a formula is also called a *predicate* over V. We denote by `Preds[V]` the set of all formulas over the set of variable V. Given a formula ϕ over V and a state $s \in S[V]$, we write $s \models \phi$ to denote the fact that ϕ is true under the interpretation that assigns to every variable $x \in V$ the value $s(x)$. In particular, a formula ϕ over V defines the set of states $[\![\phi]\!]_V = \{s \in S[V] \mid s \models \phi\}$.

Illustration 1 *Consider the set of boolean variables $V = \{x, y, z\}$. The set $S[V]$ consists of $2^3 = 8$ elements. If we take* Lang *to be propositional logic, then the formula $x \wedge \neg y$ is satisfied by the two states $(x = \mathrm{T}, y = \mathrm{F}, z = \mathrm{F}), (x = \mathrm{T}, y = \mathrm{F}, z = \mathrm{T}) \in S[V]$. If we take* Lang *to be quantified boolean formulas, then the formula $\exists w \,.\, (w \equiv x \wedge w \equiv \neg y \wedge w \equiv z)$ is satisfied by the two states $(x = \mathrm{T}, y = \mathrm{F}, z = \mathrm{T}), (x = \mathrm{F}, y = \mathrm{T}, z = \mathrm{F}) \in S[V]$.* ∎

Why use $\circ$: $\ominus$ instead of the more common x_t, x_{t+1}, x_{t-1}?

Transition predicates. In order to be able to define *relations,* in addition to sets of states, we introduce the following notation. For each state variable x, we introduce a new variable $\circ x$ (read: "next x"), with $D(x) = D(\circ x)$, that denotes the value of the state variable x in the successor state. Given a set $V \subseteq W$ of variables, we let $\circ V = \{\circ x \mid x \in V\}$ be the corresponding set of next variables. We denote the converse of $\circ$ by $\ominus$ (read: "previous"): precisely, we let $\ominus \circ x = x$ for all variables x. Given a predicate ϕ, we denote by $\circ \phi$ the result of replacing every variable x in ϕ with $\circ x$, and by $\ominus \phi$ the result of replacing every $\circ x$ in ϕ with x; thus, $\ominus \circ \phi = \phi$.

In a transition predicate the standard variables refer to the current state, and the next variables refer to the successor state. Given a predicate ρ over $V \cup \circ U$, and states $s \in S[V]$ and $t \in S[U]$, we write $(s, t) \models \rho$ to denote the fact that ρ is true when every $x \in V$ has value $s(x)$, and every $\circ y \in \circ U$ has value $t(y)$. A transition predicate $\rho \in \mathtt{Preds}[\mathtt{V}, \circ \mathtt{U}]$ defines a relation

$$[\![\rho]\!]_{V, \circ U} = \{(s, t) \in S[V] \times S[U] \mid (s, t) \models \rho\}.$$

Illustration 2 *Consider the set of boolean variables $V = \{x, y\}$. The transition* *predicate* $(\bigcirc x \equiv y) \wedge \neg \bigcirc y$ *defines the transition that copies the value of* y *into* x, *and sets* y *to* F. ∎

2.3 Synchronous Interfaces, Formally

I thought synchronous, as applied to ~~electric~~ digital circuits, means that it is clocked. Your defin just implies no feedback. You should note that there is a difference here in use of terms.

Interfaces as implemented in CHAI are synchronous in nature, i.e, they do not depend on next values of inputs. We formally define them as follows.

Definition 1 (Interface Modules) *An* Interface Module $M = \langle V^i_M, V^o_M, V^r_M, \theta^i_M, \theta^o_M, \tau^i_M, \tau^o_M \rangle$ *consists of the following elements:*

- *A set* V^i_M *of* input variables, *and a set* V^o_M *of* output variables. *The two sets must be disjoint:* $V^i_M \cap V^o_M = \emptyset$. *We indicate by* $V_M = V^i_M \cup V^o_M$ *the set of all state variables of* M.
- *A set* V^r_M *of* reserved variables, *such that* $V^i_M \cup V^o_M \subseteq V^r_M$. *The set* V^r_M *contains variables that are reserved for use by the module, and constitute the module* name space.
- *A predicate* $\theta^i_M \in \mathtt{Preds}[\mathtt{V^i_M}]$ *defining the legal initial values for the input variables.*
- *A predicate* $\theta^o_M \in \mathtt{Preds}[\mathtt{V^o_M}]$ *defining the initial values of the output variables.*
- *An* input transition predicate $\tau^i_M \in \mathtt{Preds}[\mathtt{V_M} \cup \bigcirc \mathtt{V^i_M}]$, *such that for all* $s \in S[V_M]$, *there is some* $t \in S[V^i_M]$ *such that* $(s,t) \models \tau^i_M$. *The predicate* τ^i_M *specifies what are the legal value changes for the input variables.*

- *An* output transition predicate $\tau^o_M \in \texttt{Preds}[\mathtt{V_M} \cup \mathtt{OV^o_M}]$, *such that for all* $s \in S[V_M]$, *there is some* $t \in S[V^o_M]$ *such that* $(s,t) \models \tau^o_M$. *The predicate* τ^o_M *specifies how the module can update the values of the output variables.* ∎

Thus, associated with a module is a set of initial states, a transition relation, and a language. The set of initial states consists of the states that correspond to both possible initial values for the output variables, and legal initial values for the input variables. The transition relation consists of the state transitions that are both possible for the output variables, and legal for the input variables. The language of a module consists of all the possible infinite sequences of states that satisfy the initial conditions and the transition relations.

Definition 2 (Set of [initial] states, transition relation, trace, and language) *Consider a module* $M = \langle V^i_M, V^o_M, V^\tau_M, \theta^i_M, \theta^o_M, \tau^i_M, \tau^o_M \rangle$.

- *The set of* states *of* M *is* $S_M = S[V_M]$.
- *The set of* initial states *of* M *is* $I_M = \{s \in S_M \mid s \models \theta^i_M \wedge \theta^o_M\}$.
- *The* transition relation *of* M *is* $R_M = \{(s,t) \in S_M \times S_M \mid (s,t) \models \tau^i_M \wedge \tau^o_M\}$.
- *A* path *of* M *from* $s \in S_M$ *is an infinite sequence* $s = s_0, s_1, s_2, \ldots$ *of* S_M *such that* $(s_k, s_{k+1}) \in R_M$ *for all* $k > 0$.
- *A* trace *of* M *is a path* $s_0, s_1, s_2, \ldots$ *such that* $s_0 \in I_M$.
- *The* language *of* M *is the set* $L_{(M)}$ *consisting of all traces of* M. ∎

The requirement on τ^i_M and τ^o_M ensures that every state in S_M has a successor that satisfies both the input and output transition relations, ensuring that from every state, there is a transition that is both possible for the module, and legal for the environment. Note that if $\theta^i_M \wedge \theta^o_M$ is unsatisfiable, then $L_{(M)} = \emptyset$.

Synchronous interface modules as defined above are an example of *Moore* modules, in which the next value of the output and internal variables can depend on the current state, but not on the next value of the input variables. For instance, if the state variables of a module M are x (input) and y (output), then in a transition from $s \in S_M$ to $t \in S_M$ the next value $t(y)$ of y can depend on the old values $s(x)$ and $s(y)$, but not on the new value $t(x)$ of the input variable.

2.4 Composition

Two interface modules are compatible if there is an environment in which they can work together. Compatible interfaces on composing lead to a new satisfiable input assumption that ensures that no local error state is reachable.

Consider the example in Figure 2.1. The adder `Adder` has two control inputs q_0 and q_1, data inputs $i_7 \cdots i_0$, and data outputs $o_7 \cdots o_0$.

Fig 2.1 calls them $di_7 \ldots di_0$ & $do_7 \ldots do_0$

- When $q_0 = q_1 = 1$, the adder leaves the input unchanged: the next value of $o_7 \cdots o_0$ is equal to $i_7 \cdots i_0$.
- When $q_0 = 0$ and $q_1 = 1$, the next outputs are given by $[o'_7 \cdots o'_0] = [i_7 \cdots i_0] + 1 \texttt{mod} 2^8$, where primed variables denote the values at the next clock cycle, and

I thought this was the point of the O & Θ notation.. why don't you write:

$[\bigcirc o_7 \ldots \bigcirc o_0] = [i_7 \ldots i_0] + 1 \bmod 2^8$?

$[o'_7 \cdots o'_0]$ is the integer encoded in binary by $o'_7 \cdots o'_0$.

- When $q_1 = 0$ and $q_0 = 1$, we have $[o'_7 \cdots o'_0] = [i_7 \cdots i_0] - 1 \bmod 2^8$.
- The adder is designed with the assumption that q_1 and q_0 are not both 0: hence, the input transition relation of `Adder` states that $q'_0 q'_1 \neq 00$.

The point of this is just illustrative, right? Otherwise, it doesn't make a lot of sense to add +1, 0, -1 constantly.

In order to cycle between adding $0, +1, -1$, the control inputs q_0 and q_1 are connected to the outputs q_1 and q_0 of a two-bit count-to-zero counter `Counter`. The counter has only one input, `cl`: when `cl` $= 0$, then $q'_1 q'_0 = 11$; otherwise, $[q'_1 q'_0] = [q_1 q_0] - 1 \bmod 4$.

When the counter is connected to the adder, the joint system can take a transition to a state where $q_1 q_0 = 00$, violating the adder's input assumptions. In spite of this, the counter and the adder are compatible, since there is a way to use them together: to avoid the incompatible transition, it suffices to assert `cl` $= 0$ early enough in the count-to-zero cycle of the counter. To reflect this, when we compose `Counter` and `Adder`, we synthesize for their composition `Counter`$\|$`Adder` a new input assumption, that ensures that the input assumptions of both `Counter` and `Adder` are satisfied.

To determine the new input assumption, we **solve a game** between Input, which chooses the next values of `cl` and $i_7 \cdots i_0$, and Output, which chooses the next values of q_0, q_1, and $o_7 \cdots o_0$. The goal of Input is to avoid a transition to $q_1 q_0 = 00$. At the states where $q_1 q_0 = 01$, Input can win if `cl` $= 0$, since we will have $q'_1 q'_0 = 11$; but Input cannot win if `cl` $= 1$. By choosing `cl'` $= 0$, Input can also win from the states where $q_1q_0 = 10$. Finally, Input can always win from $q_1q_0 = 11$, for all $\texttt{cl}'$. Thus, we associate with $\texttt{Counter}\|\texttt{Adder}$ a new input assumption encoded by the transition relation requiring that whenever $q_1q_0 = 10$, then $\texttt{cl}' = 0$. The input requirement $q_1q_0 \neq 00$ of the adder gives rise, in the composite system, to the requirement that the reset-to-1 occurs early in the count-to-zero cycle of the counter. ∎

2.4.1 Composition and Compatibility, Formally

Two Synchronous interfaces M and N are *composable* if $V_M^o \cap V_N^o = \emptyset$. If M and N are composable, we merge them into a single interface P as follows. We let $V_P^o = V_M^o \cup V_N^o$ and $V_P^i = (V_M^i \cup V_N^i) \setminus V_P^o$. The output behavior of P is simply the joint output behavior of M and N, since each interface is free to choose how to update its output variables: hence, $\theta_P^o = \theta_M^o \wedge \theta_N^o$ and $\tau_P^o = \tau_M^o \wedge \tau_N^o$. On the other hand, we cannot simply adopt the symmetrical definition for the input assumptions. A syntactic reason is that $\theta_M^i \wedge \theta_N^i$ and $\tau_M^i \wedge \tau_N^i$ may contain variables in $(V_P^o)'$. But a deeper reason is that we may need to strengthen the input assumptions of P further, in order to ensure that the input assumptions of M and N hold. If we can find such a further strengthening θ^i and τ^i, then M and N are said to be *compatible,* and $P = M\|N$ with θ_P^i and τ_P^i being the weakest such strengthenings; otherwise, we say that M and N are incompatible, and $M\|N$ is undefined. Hence, informally, M and N are compatible if they can be used together under some assumptions.

Definition 3 (Compatibility and composition of synchronous interfaces) *For any two synchronous interfaces M and N, we say that M and N are* composable *if*

$V_M^o \cap V_N^o = \emptyset$. If M and N are composable, let $V_P^o = V_M^o \cup V_N^o$, $V_P^i = (V_M^i \cup V_N^i) \setminus V_P^o$, $V_P = V_P^o \cup V_P^i$, $\theta_P^o = \theta_M^o \wedge \theta_N^o$, and $\tau_P^o = \tau_M^o \wedge \tau_N^o$.

The interfaces M and N are compatible *(written $M \wr\wr N$) if they are composable, and if there are predicates θ^i on V_P^i and τ^i on $V_P \cup (V_P^i)'$ such that (i) θ^i is satisfiable; (ii) $\forall V_P . \exists (V_P^i)' . \tau^i$ holds; (iii) for all $s_0, s_1, s_2, \ldots \in \mathtt{Traces}(V_P^i, V_P^o, \theta^i, \theta_P^o, \tau^i, \tau_P^o)$ we have $s_0 \models \theta_M^i \wedge \theta_N^i$ and, for all $k \geq 0$, $(s_k, s_{k+1}) \models \tau_M^i \wedge \tau_N^i$.*

The composition *$P = M \| N$ is defined if and only if $M \wr\wr N$, in which case P is obtained by taking for the input predicate θ_P^i and for the input transition relation τ_P^i the weakest predicates such that the above condition holds.* ∎

2.5 Composition Algorithm

To compute $M \| N$, we consider a game between Input and Output [CdAHM02]. At each round of the game, Output chooses new values for the output variables V_P^o according to τ_P^o; simultaneously and independently, Input chooses (unconstrained) new values for the input variables V_P^i. The goal of Input is to ensure that the resulting behavior satisfies $\theta_M^i \wedge \theta_P^i$ at the initial state, and $\tau_M^i \wedge \tau_N^i$ at all state transitions. If Input can win the game, then M and N are compatible, and the most general strategy for Input will give rise to θ_P^i and τ_P^i; otherwise, M and N are incompatible. The algorithm for computing θ_P^i and τ_P^i proceeds by computing iterative approximations to τ_P^i, and to the set C of states from which Input can win the game. We let $C_0 = \mathrm{T}$

and, for $k \geq 0$:

$$\widetilde{\tau}_{k+1} = \forall (V_P^o)'.\Big(\tau_P^o \rightarrow (\tau_M^i \wedge \tau_N^i \wedge C_k')\Big) \qquad C_{k+1} = C_k \wedge \exists (V_P^i)'.\widetilde{\tau}_{k+1}. \tag{2.1}$$

Note that $\widetilde{\tau}_{k+1}$ is a predicate on $V_P^o \cup V_P^i \cup (V_P^i)'$. Hence, $\widetilde{\tau}_{k+1}$ ensures that, regardless of how V_P^o are chosen, from C_{k+1} we have that (i) for one step, τ_M^i and τ_N^i are satisfied; and (ii) the step leads to C_k. Thus, indicating by $C_* = \lim_{k\to\infty} C_k$ and $\widetilde{\tau}_* = \lim_{k\to\infty} \widetilde{\tau}_k$ the fixpoints of (2.1) we have that C_* represents the set of states from which Input can win the game, and $\widetilde{\tau}_*$ represents the most liberal Input strategy for winning the game. This suggests us to take $\tau_P^i = \widetilde{\tau}_*$. However, this is not always the weakest choice, as required by Definition 3: a weaker choice is $\tau_P^i = \neg C_* \vee \widetilde{\tau}_*$, or equivalently $\tau_P^i = C_* \rightarrow \widetilde{\tau}_*$. Contrary to $\tau_P^i = \widetilde{\tau}_*$, this weaker choice ensures that the interface P is non-blocking. We remark that the choices $\tau_P^i = \widetilde{\tau}_*$ and $\tau_P^i = C_* \rightarrow \widetilde{\tau}_*$ differ only at non-reachable states. Since the state-space of P is finite, by monotonicity of (2.1) we can compute the fixpoint C_* and $\widetilde{\tau}_*$ in a finite number of iterations. Finally, we define the input initial condition of P by $\theta_P^i = \forall V^o.(\theta_P^o \rightarrow (\theta_M^i \wedge \theta_N^i \wedge C_*))$. The following algorithm summarizes these results.

Algorithm 1 *Given two composable Synchronous interfaces M and N, let $C_0 = \mathrm{T}$, and for $k > 0$, let the predicates C_k and $\widetilde{\tau}_k$ be as defined by (2.1). Let $\widetilde{\tau}_* = \lim_{k\to\infty} \widetilde{\tau}_k$ and $C_* = \lim_{k\to\infty} C_k$; the limits can be computed with a finite number of iterations, and let $\theta_*^i = \forall V^o.\Big(\theta_P^o \rightarrow (\theta_M^i \wedge \theta_N^i \wedge C_*)\Big)$. Then the interfaces M and N are* compatible *iff* θ^i_* *is satisfiable; in this case their composition* $P = M \| N$ *is given by*

$$\begin{aligned} V^o_P &= V^o_M \cup V^o_N & \tau^o_P &= \tau^o_M \wedge \tau^o_N & \theta^o_P &= \theta^o_M \wedge \theta^o_N \\ V^i_P &= (V^i_M \cup V^i_N) \setminus V^o & \tau^i_P &= C_* \to \widetilde{\tau}_* & \theta^i_P &= \theta^i_*. \end{aligned} \quad \blacksquare$$

Implementation Technique. To obtain an efficient implementation, both the input and the output transition relations should be represented using a conjunctively decomposed representation, where a relation τ is represented by a list of BDDs $\tau_1, \tau_2, \ldots, \tau_n$ such that $\tau = \wedge_{i=1}^n \tau_i$. When computing $P = M\|N$, the list for τ^o_P can be readily obtained by concatenating the lists for τ^o_M and τ^o_N. Moreover, assume that τ^o_P is represented as $\bigwedge_{i=1}^n \tau^o_i$, and that $\tau^i_M \wedge \tau^i_N$ is represented as $\bigwedge_{j=1}^m \tau^o_j$. Given C_k, from (2.1) the conjunctive decomposition is $\bigwedge_{j=1}^{m+1} \widetilde{\tau}_{k+1,j}$ for $\widetilde{\tau}_{k+1}$ by taking $\widetilde{\tau}_{k+1,m+1} = \neg\exists(V^o_P)'.(\tau^o_P \wedge \neg C'_k)$ and, for $1 \le j \le m$, by taking $\widetilde{\tau}_{k+1,j} = \neg\exists(V^o_P)'.(\tau^o_P \wedge \neg\tau^i_j)$. Also $C_{k+1} = \exists(V^i_P)'. \bigwedge_{j=1}^{m+1} \widetilde{\tau}_{k+1,j}$. All these operations can be performed using image computation techniques. On reaching k such that $C_k \equiv C_{k+1}$, the BDDs $\widetilde{\tau}_{k,1}, \ldots, \widetilde{\tau}_{k,m+1}$ form a conjunctive decomposition for $\widetilde{\tau}_*$. Since the two transition relations $\widetilde{\tau}_*$ and $C_* \to \widetilde{\tau}_*$ differ only for the behavior at non-reachable states, we can take directly $\tau^i_P = \widetilde{\tau}_*$, obtaining again a conjunctive decomposition.

I don't know what this acronym means

Chapter 3

Chai

citation

This chapter starts with introduction to Mocha, on which Chai is built. It is then is followed by a tutorial on modelling and verification using ChaiTowards the end of the chapter the details of Interface Modules, their implementation and usage are presented.

3.1 The Starting Point- Mocha

Chai is based on Mocha, it extends the functionality of Mocha and adds an input language to suit interfaces.

Mocha is an interactive environment for modular verification of heterogeneous systems. Mocha relies on the modelling framework of reactive modules unlike traditional use of state transition graphs. Its input language is machine readable variant of reactive modules.
cite
In Chai we extend Reactive Modules to interface modules

of digital components.

a

what does heterogeneous mean in this context?

as explained in next chapter.

MOCHA supports the following functionalities [RAT00]:

- **System specification** in the language of Reactive Modules. Reactive Modules allow the formal specification of heterogeneous systems with synchronous, asynchronous, and real-time components. Reactive Modules support modular and hierarchical structuring and reasoning principles.

- **System execution** by randomized, user-guided, or mixed-mode trace generation. In mixed-mode trace generation, the user plays a game against MOCHA and guides the execution of some modules, while MOCHA controls the execution of other modules.

- **Requirement specification** in Alternating Temporal Logic cite. The logic ATL allows the formal specification of requirements that refer to collaborative as well as adversarial relationships between modules. The popular logic CTL cite is a sublanguage of ATL.

- **Requirement verification** by ATL model checking. The symbolic model checker in both implementations is based on BDD engines developed by the UC Berkeley VIS project. For invariant checking, MOCHA supports both symbolic and enumerative search. cite

- **Implementation verification** by checking trace containment between implementation and specification modules. MOCHA supports containment checking if the specification module has no hidden state, and simulation checking otherwise. For decomposing proofs, MOCHA supports an assume-guarantee principle.

- **Reachability analysis** of real-time systems.

In MOCHA the basic structuring units, or the molecules of a system, are *reactive modules* [dA01]. The modules have a well-defined interface given by a set of *external (or input)* variables and a set of *interface (or output)* variables. A module may also have a set of *private* variables. All variables are typed, and MOCHA supports a standard set of finite and infinite types, such as booleans and integers. A module is built from *atoms*, each grouping together a set of *controlled* (interface or private) variables with exclusive updating rights. *Updating* is defined by two nondeterministic guarded commands: an *initialization* command and an *update* command. In these commands unprimed variables, such as x, refer to the old value of the corresponding variable, and primed variables, such as x', refer to the new value of the corresponding variable. An atom is said to *await* another atom if its initialization or update commands refer to primed variables that are controlled by the other atom. The variables change their values over time in a sequence of *rounds*. The first round consists of the execution of the initialization command of each atom, and the subsequent rounds consist of the execution of the update command of each atom, in an order consistent with the await dependencies. A round of an atom is therefore a *subround* of the module. If no guard of the update command is enabled, then the atom idles, i.e., the values of the variables do not change.

in what sense are they nondet.? I assume. initialization comes before update...

Modules can be *composed* if they have disjoint sets of interface variables, and their union of atom sets does not contain a circular await dependency. Given a specification SystemSpec of a system and model of user behavior as UserSpec, specification module Spec is defined as:

```
module Spec is UserSpec || SystemSpec
```

For encapsulation REACTIVEMODULES allows the *hiding* of interface variables, and for instantiation it allows the *renaming* of interface and external variables. Hiding and parallel composition permit hierarchical descriptions of complex systems.

3.2 Tutorial Introduction to Chai

Chai is available for free public download from [DL03a]. It requires the GLU BDD package and Tcl7.2. The installation instructions and relevant notes are available at [DL03a].

3.2.1 Modelling with Chai

Consider modelling ~~of a~~ the 2-bit down counter shown in Figure 3.1 as an interface module. The counter counts from 3 to 0 on its normal run. Whenever the reset is high the counter resets itself to start down counting from 3. Thus we have the input, ~~as~~ reset, and the outputs, ~~as the~~ counting bits $b0$ and $b1$.

To model this counter as an interface we should first understand the input assumption and output guarantee of the system. In case of the downcounter, the input assumption is fairly trivial, the input *reset* can be either 1 or 0. So we choose it to be non-deterministic. The output guarantee is that counter counts from $3 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow 3$

I dont understand your use of this term in this context.

when ~~its~~ it is not reset and on being reset starts counting from 3. We represent this as guarded commands. If the state of counter (reset,b1,b0) is (0,1,0) then in the next state it should move to zero if the reset stays low i.e (0,0,0) low so we have a guarded command as shown below.

I don't understand this term in this context.

```
[] ~reset &  b1 & ~b0 -> b1' := false; b0' := true
```

Figure 3.2 shows the interface module for the down counter described above 3.1.

I don't see this line in Fig 3.2.

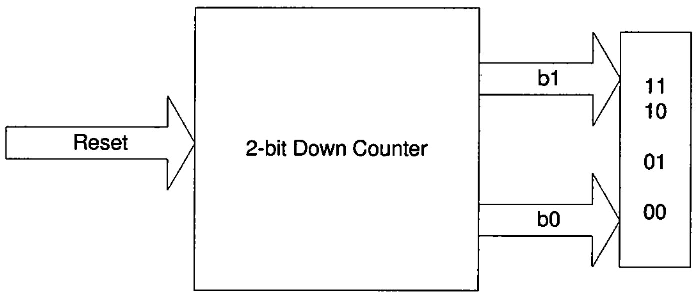

Figure 3.1: A 2-bit Down Counter

3.2.2 Running Chai

All the interface module definitions have to be entered into a single file named typically with the suffix .intf; in our case, (say) this file is downcounter.intf ~~Figure 3.2~~. ~~for the inter~~

CHAI is invoked by typing `chai` at the shell prompt.

```
kala 6> chai
Welcome to CHAI 1.0
Please report any problems to dvl@cse.ucsc.edu
chai 1.0 >
```

The interface is read and parsed with the `read_intf` command. As shown below CHAI displays the names of the modules that were successfully parsed. In the case of a parse

If x0 & x1 represent the counter state, aren't they outputs?

these names are completely different from the i/o vars shown in Fig 3.1

Isn't reset an input?

```
interface downcounter
  input vars: x0:    bool; x1: bool;
  output vars: toone: bool;
```

reset?

```
output atom controls x0, x1 reads x0, x1, toone
  init
      [] true -> x0' := true; x1' := true;
  update
      []  toone              -> x1' := true;  x0' := true
      [] ~toone &  x1 &  x0 -> x1' := true;  x0' := false
      [] ~toone &  x1 & ~x0 -> x1' := false; x0' := true
      [] ~toone & ~x1 &  x0 -> x1' := false; x0' := false
      [] ~toone & ~x1 & ~x0 -> x1' := true;  x0' := true
  endatom
input atom controls toone
  init
      [] true -> toone' := nondet
  update
      [] true -> toone' := nondet
  endatom
endinterface
```

This doesn't seem like the right version of this figure.

Figure 3.2: A 2-bit Down Counter Modelled As An Interface Module

error, an appropriate message is displayed.

```
kala 68> chai
Welcome to CHAI 1.0
Please report any problems to dvl@cse.ucsc.edu
chai 1.0 > read_intf downcounter.intf
...
DEBUG Interface Created: downcounter
chai 1.0 >
```

CHAI provides many methods and tools for verifying the correctness of a design: execution (i.e., simulation), invariant checking, refinement checking, ATL model checking, interface composition. Interface composition being the distinguishing feature of CHAI we detail it here, rest of the features are described in [RAT00].

the

The command for interface composition is compose_intf.

```
chai 1.0 > compose_intf
Usage: compose_intf <outIntf> <Intf1> <Intf2>
```

It checks if the interfaces Intf1 and Intf2 can be composed as a new interface outIntf.

When we run this command on interface downcounter (Figure 3.1) and a dual output gate (Figure 3.3). The composition gives a compatible interface as following CHAI

```
interface gate
  input vars: y0: bool; y1: bool;
  output vars:  x0: bool; x1: bool;
output atom controls y0, y1 reads x0, x1, y0, y1
  init
      [] true -> y0' := false; y1' := nondet
      [] true -> y1' := false; y0' := nondet
  update
      [] ~x0 & ~x1 -> y0' := true;  y1' := true
      [] ~x0 &  x1 -> y0' := true;  y1' := false
      []  x0 & ~x1 -> y0' := false; y1' := true
      []  x0 &  x1 -> y0' := y0;    y1' := y1
  endatom
input atom controls x0, x1 reads x0, x1
  init
      [] true -> x0' := nondet; x1' := nondet
  update
      []  ~x0 & ~x1 -> x0' := false;  x1' := nondet
      []  ~x0 & ~x1 -> x1' := false;  x0' := nondet
      []   x0 |  x1 -> x0' := nondet; x1' := nondet
  endatom
endinterface
```

Figure 3.3: A Simple Gate

session shows.

```
chai 1.0 > read_intf downcounter.intf
...
chai 1.0 > read_intf gate.intf
...
chai 1.0 > compose_intf GC gate downcounter
Interfaces gate and downcounter are compatible.
```

3.3 Interface Modules

INTERFACE MODULES consist of input and output variables controlled by input and output atoms. In this section we look at them in detail.

3.3.1 Variables

The state of an interface module is described by a set of *state variables,.* They are in turn partitioned into sets of *input* and *output* variables. The **input variables**

ital

represent inputs to the interface, their value can be read, but not changed, by the interface module. The input variables are denoted by **inputs vars:** clause. The

a

output variables represent outputs of the interface, and their value can be changed

ital

(and read) by the interface module. The output variables are denoted by **output vars:** clause.

Consider the description of the downcounter as an interface (Figure 3.2). The variable are declared in the beginning as:

```
input vars:  reset: bool;
output vars: b0: bool; b1: bool;
```

This is different from ~~the~~ Fig. 3.2!!

Add a sentence explaining this in English. "reset is defined as an input variable, of type boolean, while b0, b1 are ..."

3.3.2 Intialization and Transitions

The variables in an interface module have to be initialized at system reset and assigned new values at each clock tick. The input and output variables achieve this through **input atoms** and **output atoms** respectively.

ital ital

Input atoms describe the initialization and update of input variables. The

Why are they called "atoms" and not "actions"?

transition relations in input atom model the design assumptions about the inputs provided by the environment and a set of *initial inputs* specify the desired initial condition of the environment. For example the input variable reset of downcounter is described with an input atom as follows:

```
input atom controls reset
  init
       [] true -> reset' := nondet
  update
       [] true -> reset' := nondet
  endatom
endinterface
```

also, is different from Fig 3.2

— Explain in English. What does "[]" mean? What does "nondet" mean?

Output atoms describe the initialization and update of output variables. The transition relations in output atom model the possible changes of output variables describing the behavior of the module. The initial outputs specify the initial conditions of the module. For example the output variables $b0, b1$ of downcounter are described with an output atom as follows:

```
output atom controls b0, b1 reads b0, b1, reset
  init
       [] true -> b0' := true; b1' := true;
  update
       []  reset              -> b1' := true;  b0' := true
       [] ~reset &  b1 &  b0 -> b1' := true;  b0' := false
       [] ~reset &  b1 & ~b0 -> b1' := false; b0' := true
       [] ~reset & ~b1 &  b0 -> b1' := false; b0' := false
       [] ~reset & ~b1 & ~b0 -> b1' := true;  b0' := true
  endatom
```

Describe in English. What does "controls" mean? What does "reads" mean? Why is the "nondet" keyword not used here?

3.3.3 Syntax

only? At least?

Any .intf file contains one interface definition. The interface definition has the syntax described in Figure 3.4. The CHAI environment knows the interface by

```
interface <interface-name>
    input vars: <input-list>
    output vars: <output-list>

    [input | output] <atom>
    ...
    [input | output] <atom>
endinterface
```

Figure 3.4: Syntax of Interface Module

what does it mean to control a variable?

interface-name. Each state variable in an interface ~~has to~~ is controlled by one and only one atom. The *input atoms* describe the input assumptions while the *output atoms* state the output guarantees.

3.3.4 Semantics

The semantics of an interface module is essentially a simple game between the module and its environment.

The behavior of an interface module consists of an infinite sequence of states starting from an initial state (trace). Starting with initial state each successive state is generated by the module and by its environment. The modules chooses the new values of the output variables according to the output transition relation, while the environment must choose the new values of the input variables according to the input transition relation.

If the module is able to fulfill all its output guarantees even for a single set of environment variables then the module is said to be compatible with its environment.

3.3.5 Implementation

The formalism of interface modules as described here is implemented in CHAI. The parser of MOCHA was changed to accommodate the input language of interface modules. The parser splits the interfaces in to input assumption and output guarantee REACTIVE MODULES which in turn are converted in to finite state machines (FSM) represented as binary decision diagrams. The FSMs are then composed into a single interface. The resulting interface is referred in the CHAI environment by the name it had in its interface definition file.

Composition and compatibility checking for interfaces as presented in section 2.5 is implemented by extending the CUDD BDD package and the VIS BDD manipulation package [RGA+96] in the CHAI environment. Using the techniques explained in section 2.5, the size (number of BDD variables) of the interfaces that CHAI is able to check for compatibility, and compose, is roughly equivalent to the size of the models that MOCHA[R. 98] can verify with respect to safety properties.

3.4 Chai Operations

This section presents various commands available to work in the CHAI design and verification environment. There are different data structures on which one can work at different levels of granularity and different amount of control. Here we present a summary of commands which each relevant data structure or abstraction can handle. The detailed list of CHAI commands and functions can be found online at [DL03b].

3.4.1 Interfaces

command

Interface modules can be read in the CHAI environment by the command read_intf. Two interfaces can be composed by the compose_intf. One can use check_intf_ref to check if one interface is a refinement of the other. The levels of various variables in the module can be known by poking it with print_levels.

what does this mean? I don't recall seeing a definition of this in Section 2.

3.4.2 Reactive Modules

Reactive modules can be read in the CHAI environment by using the commands read_module. Two reactive modules can be composed by using the compose command. A module can be renamed by using the ren directive and a new instance of it can be created using the let command. One can see the atoms comprising a module by using the show_atoms command.

3.4.3 BDDs and FSMs

Binary decision diagrams are at the bottom of data structure hierarchy. They form the core of the implementation. MDD (Multivariate Decision Diagrams) and FSMs (Finite State Machines) follow next.

m d d

A module can be converted in to a FSM using the commands fsm. An interface can be made from FSM representations by using the commands make_intf.

A dump of BDDs from a module file can be obtained by using the commands dump_bdd in a module. Various operations like not,and,or can be performed directly on BDDs. The truth value of a BDD can be checked using true command.

~~what does this~~

An example here would be nice.

3.4.4 Invariants

Invariants can be read in the CHAI environment by the `read_inv` command. Invariants in alternating temporal logic are read using `atl_read`. To check wether a module satisfies an invariant one can use the command `inv_check` on an instance of module and the invariant.

3.4.5 Summary of Important Commands

A handy list of ~~most relevant~~ important chai commands is presented below:

- **read_intf**. This commands reads and validates an interface module description from the specified file.

- **sl_make_intf**. This command given two modules, one describing the input evolution, the other describing the output evolution, creates a single new interface module that combines the two.

- **sl_make_intf_out**. This command creates an interface having only an output portion (with no input assumptions).

- **sl_compose_intf**. This command given two interfaces, composes them and checks if they are compatible. If they are not compatible, says so. If they are compatible, says so, and returns the composition.

- **sl_check_intf_ref**. This commands given two interfaces intf1 and intf2, checks whether intf2 is a refinement of intf1.

- **sl_print_intf**. This command prints an interface on the ~~CHAI~~ console.
- **sl_copy**. This command makes a copy of an interface and references it by the name specified by the commandline parameter.
- **sl_compose**. This command composes two FSMs even if they share controlled variables and references the composition by the name specified by the commandline parameter.
- **sl_reach_histonly**. This command computes the set of reachable states, projected onto the history variables only.

Chapter 4

Interfaces for Hardware Design

One of the goals of Chai is to formally verify hardware designs at the Register Transfer Logic (RTL) level. ~~It~~ Chai has a Binary Decision Diagram (BDD) based formal verification engine and has algorithms to compose and verify designs. A developer can use these algorithms to formally verify existing hardware designs. Tools for converting existing component based hardware designs to the CHAI environment are required to be able to provide such a comprehensive test bed for component-based design verification algorithms and methodologies. This chapter describes the conversion of hardware designs coded in Verilog, Esterel or VHDL to Interface Modules, as summarized in Figure 4.1.

4.1 HDL to BLIF-MV

Verilog to BLIF-MV conversion is possible by the tool vl2mv [Che94]. VHDL to BLIF-MV conversion can be achieved by the tool prevail [BBD+96] while Esterel

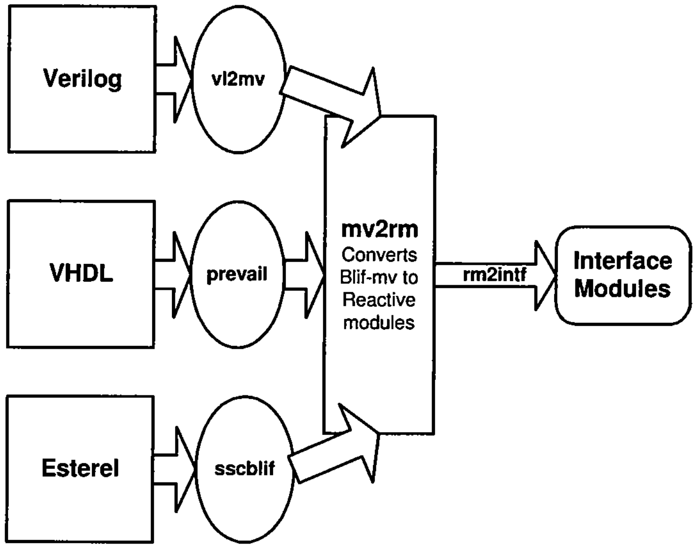

Figure 4.1: Converting Hardware Description Languages to Interfaces

has a back-end SSCBlif [ssc] to output code in BLIF-MV format. Thus,BLIF-MV is a rich intermediate format.

4.2 BLIF-MV

BLIF-MV is an acronym for Berkeley Logic Interchange Format - Multivariate. It is successor of BLIF, and it primarily adds non-determinism. BLIF-MV format is designed to represent non-deterministic sequential systems in hierarchical fashion. A system can be composed of interacting sequential systems, each of which can be again described as a collection of communicating sequential systems. In BLIF-MV, there is an implicit assumption that the whole system is clocked by a single global clock although the clock is never declared in BLIF-MV.

We implemented a tool to convert BLIF-MV representation of a of a design to reactive modules. Next two sections describes the tool MV2RM and the translation process.

4.2.1 MV2RM

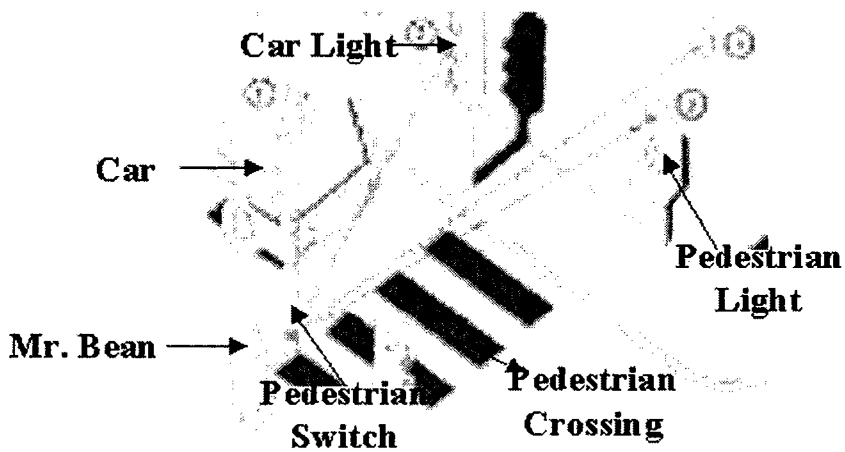

Figure 4.2: Pedestrian Crossing this figure is barely readable

Figure 4.2 illustrates a simple daily life scenario of a pedestrian crossing. Here a simple traffic light controller manages ~~between~~ both car ~~lights~~ traffic lights and pedestrian lights. car lights ~~are~~ ~~what~~ are the lights on a car Figure 4.3 presents the example encoded in BLIF-MVWhen the *CarSignal* is asserted and Mr. Bean asserts *Button* to cross the street, ~~then~~ the *ControlLogic* de-asserts the *CarSignal* at next clock-tick. When *CarSignal* is de-asserted the *PedestrianSignal* gets asserted. Finally, now Mr. Bean is able to cross the road!

The left column in Figure 4.3 is the BLIF-MV model [BFM00] and the right column is translation to REACTIVE MODULES done by MV2RM.

Figure 4.4 shows the architecture of MV2RM. The file `lights.mv` is fed to the lexer of MV2RM. The translation is two-pass. MV2RM is implemented in functional language OCAML [Inr].

In pass one the **sub-circuit analyzer** parses the input models BLIF-MV to analyze the parameters of the subcircuits used to make an appropriate hide variable list when the model is converted in to a REACTIVE MODULES.

In pass two the **parser** goes through all the BLIF-MV constructs to make an abstract syntax tree. The validation of input is done in this phase. At the end of the pass the code generator runs over the abstract syntax tree to emit REACTIVE MODULES code.

4.3 Translating BLIF-MV to Reactive Modules

In this section we describe the translation logic used by MV2RM. Each important construct of BLIF-MV is described with a relevant example and the corresponding strategy of conversion is detailed. The Full BNF grammar of BLIF-MV is in the appendix and the documentation of the OCAML implementation of MV2RM is given at [Bha03].

4.3.1 Multi-valued Variables

A multi-valued variable is a variable that can take a finite number of values. There are two classes of multi-valued variables. The class of *enumerative variables* consists of variables whose domain is the n integers $\{0, \ldots, n-1\}$.

```
.model Lights
.inputs Button
.outputs CarSignal
        PedestrianSignal
.subckt ControlLogic CL
   PresentSignal=CarSignal
   Button=Button NextSignal=Tmp
.latch Tmp CarSignal
.reset CarSignal
0
1
.table CarSignal -> \
            PedestrianSignal
0 1
1 0
.end

.model ControlLogic
.inputs PresentSignal Button
.outputs NextSignal
.table PresentSignal \
        Button -> NextSignal
.default 1
1 1 0
.end
```

```
module a_Lights
 external
    Button: (0..1) ;
    Tmp: (0..1)
 interface
    CarSignal: (0..1) ;
    PedestrianSignal: (0..1)

 atom
   controls  PedestrianSignal
   awaits  CarSignal
     init update
     []  CarSignal' = 1  ->
            PedestrianSignal':= 0
     []  CarSignal' = 0  ->
            PedestrianSignal':=1
 endatom
 atom
   controls  CarSignal
   reads  Tmp
     init
     update
     []  true ->  CarSignal':=Tmp
 endatom
 endmodule

module a_ControlLogic
 external
    PresentSignal: (0..1) ;
    Button: (0..1)
 interface
    NextSignal: (0..1)

 atom
   controls  NextSignal
   awaits  PresentSignal , Button
     init update
     []  PresentSignal' = 1  &
           Button' = 1  -> NextSignal':= 0
     [] default -> NextSignal':=1
 endatom
 endmodule

 -- Generated by mv2rm DONOT edit
 -- Report bugs to <vaibhav@cse.ucsc.edu>
 ControlLogic:= a_ControlLogic
 b_Lights:= a_Lights
 ||  ControlLogic [PresentSignal, Button \
  , NextSignal := CarSignal , Button , Tmp]

 Lights := hide  Tmp  in b_Lights endhide
```

Figure 4.3: Pedestrian Light Controller in BLIF-MV and REACTIVE MODULES

· Move to previous page
· Use smaller font

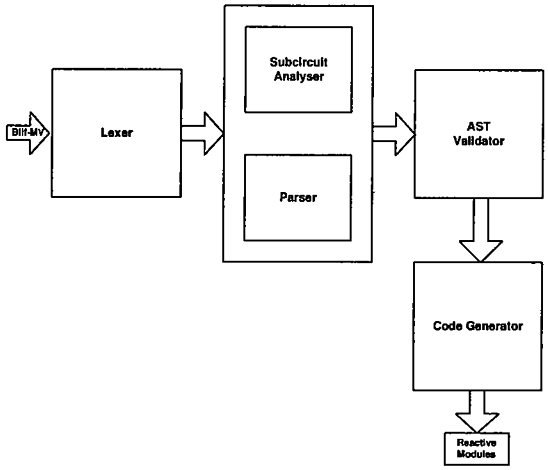

Figure 4.4: Architecture of MV2RM

BLIF-MV	REACTIVE MODULES
model	module
inputs	interface
outputs	external
undefined var	private
table	atom (awaited)
reset	atom
latch	atom (read)
subckt	composition

Figure 4.5: Mapping of BLIF-MV constructs to REACTIVE MODULES

```
.mv <variable-name-list> <number-of-values>
```

The second class are *symbolic variables*, which can take a set of arbitrary values. Symbolic variables are declared as follows.

```
.mv <variable-name-list> <number-of-values> <value-list>
```

REACTIVE MODULES have multi-valued variables. Table 4.1 shows how enumerative and symbolic variables are translated. REACTIVE MODULES also have typed variables which can be used to represent symbolic variables.

.mv signal 3	$\Rightarrow$	signal: (0..2)
.mv signal 3 STOP READY2GO GO	$\Rightarrow$	signal: {STOP, READY2GO, GO}

Table 4.1: Multivalued Variables Translated

4.3.2 Tables

A table is an abstract representation of a physical gate. A table is driven by inputs and generates outputs following its functionality. Although a real gate generates an output deterministically depending on what inputs are supplied, tables in BLIF-MV can represent non-deterministic behaviors as well. The functionality of the table is described as a symbolic relation, i.e. the table enumerates symbolically all the valid combination of values among the inputs and the outputs. A table without input represents a constant generator. If the table allows more than one value for its output, then the table is a *nondeterministic* constant generator, which we call *pseudo input.* Tables are declared in the following way.

```
.table <in-1> <in-2> ... <in-n> -> <out-1> <out-2>... <out-m>
<relation>
...
<relation>
```

ital

The table is translated as an **atom** in REACTIVE MODULES. The input variables are read fresh for each clock tick, i.e., they are **awaited** and the output variables are controlled.

ital

A *relation* of BLIF-MV is a white-space separated non-null list of $n + m$ strings, giving a valid combination of values among inputs and outputs. The i-th string in a relation specifies a set of values for the i-th variable in the input/output

```
.table PresentSignal \
       Button -> NextSignal
.default 1
1 1 0
.end
```

```
atom
  controls  NextSignal
  awaits  PresentSignal , Button
    init update
    []  PresentSignal' = 1  &
      Button' = 1  -> NextSignal':= 0
    [] default -> NextSignal':=1
endatom
```

Table 4.2: Tables translated

declaration of `.table`. A relation of BLIF-MV is translated as a **guarded command** of REACTIVE MODULES. In each update round the values of controlled variables (outputs in tables) are based on the guarded commands (relations).

The `.default` construct of BLIF-MV is used to define a default output for the input patterns not specified in the given relation. A default construct BLIF-MV is translated as a default guarded command of REACTIVE MODULES.

Table 4.2 shows a BLIF-MV .table snippet from Figure 4.3 translated into REACTIVE MODULES. Note that the atom has **awaited** variables.

ital no bold

4.3.3 Latches and Reset Tables

A latch models a storage element, which retains the value of the input at the last clock tick. A latch has **only one** input and output. Every latch has to be initialized by a **reset** statement. A latch is allowed to have more than one initial value, in which case the latch takes an initial value non-deterministically from the specified values. Thus a latch can be seen as a multi-valued flip-flop with possibly multiple initial states. The reset statement specifies the values **latched variables** can take when the system is reset.

<pre>.latch Tmp CarSignal
.reset CarSignal
0
1</pre>
	<pre>atom
	  controls CarSignal
	  reads Tmp
	    init
	    [] true -> CarSignal':= 1
	    [] true -> CarSignal':= 0
	    update
	    [] true -> CarSignal':=Tmp
	endatom</pre>

Table 4.3: Latch and Reset Statements Translated

A latch is declared as follows:

```
.latch <latch-input> <latch-output>
```

The reset statement for a latch is as follows:

```
.reset <option-reset-input> latch_output
<reset-in-0> <value-0>
...
```

MV2RM parser goes through the input file combines the **latch** and **reset** statements for a particular variable. The *latch-output* variable is controlled by the atom and initialized according to the relations in the reset statement. It is updated to the value of *latch-input.* Note that the value of *latch-input* is **read** by the atom and not awaited, implying that it does not wait for the variable to be update in current round bu t rather picks its value from the last round. Table 4.3 illustrates the conversion.

4.3.4 Models

`.model` is the prime construct of BLIF-MV used to define a basic component of a hierarchical system.

Any BLIF-MV file contains one or more model definitions. In case of multiple models in a single file the first model is considered to the `root` model or the model having `.root` construct in second line of its definition . A model looks like Figure 4.6.

```
.model <model-name>
.inputs <input-list>
.outputs <output-list>
<command>
...
<command>
.end
```

Figure 4.6: Blif-MV model syntax

- *model-name* is a string by which the model is referred in the system. The equiv- construct to a alent ~~of~~ model in REACTIVE MODULES is a **module**. Each model is abstracted as a module. Table **??** illustrates the conversion. no bold ??

- *input-list* is a white-space separated list of strings (terminated by the end of the line) giving the formal input terminals for the model being declared. If this is the root model, then signals can be identified as the primary inputs of this system. The input variables are mapped as **external** variables while translating to REACTIVE MODULES. ital

- *output-list* is a white-space separated list of strings (terminated by the end of the line) giving the formal output terminals for the model being declared. If this is the root model, then signals can be identified as the primary output of this system. The input variables are mapped as **interface** variables while translating to REACTIVE MODULES. ital

.model ControlLogic
.inputs PresentSignal Button
.outputs NextSignal
.names PresentSignal \
   Button -> NextSignal
.def 1
1 1 0
.end
	module a_ControlLogic
	 external
	   PresentSignal: (0..1) ;
	   Button: (0..1)
	 interface
	   NextSignal: (0..1)
	
	 atom
	  controls NextSignal
	  awaits PresentSignal , Button
	   init update
	   [] PresentSignal' = 1 & \
	   Button' = 1 -> NextSignal':= 0
	   [] default -> NextSignal':=1
	 endatom
	 endmodule
	 ControlLogic:= a_ControlLogic

Table 4.4: `.model` translated

- *command* is one of `.mv`, `.table`, `.latch`, `.reset` and `.subckt`, which defines the detailed functionality of the model. Translation of `.subckt` is described in next section while rest are detailed in previous sections. Undeclared variables are mapped as private variables while translating to REACTIVE MODULES.

4.3.5 Subcircuits

In a model, another model can be instantiated as a subcircuit using the `.subckt` construct. It is the contruct which enables hierarchical composition in BLIF-MV.

```
.subckt <model-name> <instance-name> <formal-actual-list>
```

This construct instantiates a reference model *model-name* as an instance *instance-name* in the current model. *formal-actual-list* specifies the association between each formal variable in *model-name* and its corresponding actual variable in the current model. Formal variables are declared in the reference model, while actual variables are variables declared in the current model. A *formal-actual-list* is a list of assignments separated by a white space. The declaration of *formal-actual-list* is of form:

```
formal-1 = actual-1 formal-2 = actual-2 ... formal-n = actual-n
```

The `.subckt` construct is replaced by composition construct of REACTIVE MODULES. A model M with subckts A1..An is represented as $M = a\_M \parallel A1 \parallel A2 \ldots \parallel An$, here a_M is the base model M without the subcircuits.

Special processing is ~~to be done~~ performed for the actual parameters passed to the subcircuits in the base model. To do subcircuit analysis and special processing we maintain a modelTab, which is a hash table of models, hashed by the modelname and containing the 3 lists, inlist, outlist, subcktlist of the model. If the actual passed parameter from the base model is an output and the associated formal parameter in subcircuit is an output in the subcircuit, then it is made an input in the base model (a_module). This is because on composition it will be an output in $module = a\_module \parallel subckt$. If the actual passed parameter from the base model is a private variable and associated formal variable is an output in the subciruit, then it is made an input in the base model (a_module) and added to a hidelist. This is because on composition it will be output in $module = a\_module \parallel subckt$ but it should be hidden from the world by $module = hide\ hidelist \in\ a\_module \parallel subckt$.

Consider the translation shown in table 4.5, and note that the private variable `Tmp` in model `Lights` is added to hidelist and is made an output (external) variable in

```
.subckt ControlLogic CL \
    PresentSignal=CarSignal \
    Button=Button NextSignal=Tmp
```

```
ControlLogic:= a_ControlLogic
b_Lights:= a_Lights
 ||  ControlLogic \
     [ PresentSignal , Button , NextSignal \
     := CarSignal , Button , Tmp  ]
Lights := hide  Tmp  in b_Lights endhide
```

Table 4.5: Partial .subckt Translation

the base model a_Lights.

4.4 Reactive Modules to Interfaces modules

We have implemented a tiny tool `rms2intf` which integrates the input assumption and output guarantee reactive modules to form an interface module.

```
kala 39> rms2intf -h

Usage: rms2intf inputAssumption.rm outputGuarantee.rm
```

Figure 4.7 shows the input assumption of a 2-bit down counter in left hand column and the output guarantee in the right hand column. All the atoms in input assumptions weaved as input atoms in the interface representation and the output guarantee atoms become output atoms. Figure 4.8 shows the interface output from `rms2intf`. Note that instead of interface and external we term the variables as output and input.

The hardware design extracted in above fashion can be input in the CHAI environment in the form of reactive modules or interface modules as illustrated by Figure 4.9

```
module counterI
  external  x0: bool; x1: bool;
  interface toone: bool;

atom controls toone
  init
       [] true -> toone' := nondet
  update
       [] true -> toone' := nondet
  endatom
endmodule
```

```
module counterO
  interface x0: bool; x1: bool;
  external  toone: bool;

atom controls x0, x1 reads x0, x1, toone
  init
       [] true -> x0' := true; x1' := true;
  update
       []  toone              -> \
                x1' := true;  x0' := true
       [] ~toone &  x1 &  x0 -> \
               x1' := true;  x0' := false
       [] ~toone &  x1 & ~x0 -> \
               x1' := false; x0' := true
       [] ~toone & ~x1 &  x0 -> \
               x1' := false; x0' := false
       [] ~toone & ~x1 & ~x0 -> \
               x1' := true;  x0' := true
  endatom
endmodule
```

Figure 4.7: Reactive Modules Representing a Down Counter

This is the same as Fig 3.2, but Fig 3.2 doesn't agree with the text.

```
interface counterIO
input vars: x0:    bool; x1: bool;
output vars:  toone: bool;

output atom controls x0, x1 reads x0, x1, toone
init
    [] true -> x0' := true; x1' := true;
update
    []  toone              -> x1' := true;  x0' := true
    [] ~toone &  x1 &  x0 -> x1' := true;  x0' := false
    [] ~toone &  x1 & ~x0 -> x1' := false; x0' := true
    [] ~toone & ~x1 &  x0 -> x1' := false; x0' := false
    [] ~toone & ~x1 & ~x0 -> x1' := true;  x0' := true
endatom
input atom controls toone
init
    [] true -> toone' := nondet
update
    [] true -> toone' := nondet
endatom
endinterface
```

Figure 4.8: Interface Representation of Down Counter by `rms2intf`

```
kala 100> chai
Welcome to CHAI 1.0
Please report any problems to dvl@cse.ucsc.edu
chai 1.0 > read_module counterO.rm
Module counterO is composed and checked in.
parse successful.
chai 1.0 > read_intf counterIO.intf
...
Done..
DEBUG PrsReadIntfCmd :  counterIO.intf.I
Module counterOI is composed and checked in. parse successful.
Module counterOO is composed and checked in. parse successful.
...
DEBUG Interface Created: counterIO
chai 1.0 > exit
Thank you for using CHAI 1.0
```

Figure 4.9: HDLs in CHAI

Chapter 5

Conclusion and Future Work

CHAI provides an environment to experiment with interface modules and game semantics. A developer can make use of algorithms present in chai to develop new verification algorithms or use the existing algorithms to verify the designs. For instance, CHAI provides a host of BDD based algorithms to find the predecessor and post regions on a set of states and compute a set of reachable states for error detection. This facility is currently being used for development of error detection algorithms in DVLAB [dA03].

The streamlined path for using hardware description languages within CHAI makes a plethora of designs coded in high level languages like Verilog available for verification research with CHAI. At DVLAB [dA03] several ~~a host~~ publicly available hardware designs coded in Verilog are used as test suites for development of **early error-detection** no bold **algorithms**. Thus, CHAI accelerates and promotes application and validation of verification research on proven industry designs.

We are working towards a comprehensive manual for CHAI and releasing a stable version. Future work would essentially comprise of supporting CHAI, maintaining MV2RM, streamlining the input language and integrating proven compositional verification research and methodologies back in to the CHAI code-base.

Appendix A

BLIF-MV BNF

```
main:
    models TokEOF
;
models:
    model models
  |
;
model:
      head decl body \
      TokEnd TokEOL
;
head:
    TokModel TokVar TokEOL
;
decl:
      vdecl
  |
;
vdecl:
      vdecl input
  | input
  | vdecl output
  | output
;
```

```
iovals:
      TokVar iovals
  | TokVar
;
body:
   table body
  | subckt body
  | mv  body
  | reset  body
  | latch body
  |
;
table:
      TokTable tabvars \
      TokEOL relations

;
mv:
      TokMV mvvars TokVal \
      values TokEOL
;
mvvars:
      TokVar COMMA mvvars
  | TokVar
;
```

```
input:
      TokInputs iovals TokEOL
;
output:
    TokInputs iovals TokEOL
;
subckt:
      TokSubckt TokVar TokVar
        param_passing TokEOL
;
param_passing:
    formal_actual param_passing
    |
;
formal_actual:
      TokVar ASSIGN TokVar
; latch:
      TokLatch TokVar TokVar
                    TokEOL
;
tabvars:
      inList ARROW outList
  |   TokTabVar inList

  | TokTabVar
;
inList:
    TokTabVar inList
  | TokTabVar
  |
;
outList:
    TokTabVar outList
  | TokTabVar
;
```

```
values:
      TokVar values
   | TokVal values
   |
;
reset:
  TokReset tabvars \
  TokEOL relations
;
relations:
      vdefault relations
   | relation relations
   |
;
vdefault:
  TokTabDef valList TokEOL
;
relation:
      valList TokEOL
;
valList:
      valb valList
   | valb
;
valb:
      TokTabVal
   |   TokTabVar
   | ASSIGN  TokTabVar
   | HYPHEN
   | LBRACE TokTabVal HYPHEN
         TokTabVal RBRACE
   | LPAREN vals RPAREN
   | NOT valb
;
vals:
      TokTabVal COMMA vals
   | TokTabVal
;
```

Appendix B

Grammar Of Interface Modules

```
interface <interface-name>
    input vars: <input-list>
    output vars: <output-list>

    [input | output] <atom>
    ...
    [input | output] <atom>
endinterface
```

Bibliography

[BBD+96] Dominique Borrione, H. Bouamama, David Deharbe, C. Le Faou, and Ayman Wahba. HDL-based integration of formal methods and CAD tools in the PREVAIL environment. In *Formal Methods in Computer-Aided Design*, pages 450–467, 1996. publisher? (this is a book, right?)

[BFM00] David Basin, Stefan Friedrich, and Sebastian Mödersheim. B2M: A Semantic Based Tool for BLIF Hardware Descriptions. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000*, volume 1954, pages 91–107, Austin, Tx, USA, 2000. Springer-Verlag.
LNCS

[Bha03] Vaibhav Bhandari. MV2RM Documentation. http://www.cse.ucsc.edu/dvlab/mv2rm/, June 2003.

[CdAHM02] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces, volume 2404, pages 414–427, 2002.
conference name, location?
LNCS

[Che94] S. Cheng. Compiling verilog into automata, 1994.

[dA01] Luca de Alfaro. Introduction to the theory of discrete systems. Lecture Notes, 2001.

is this a book, or a conference paper, or a web site or a tech report? I have no idea.

Unpublished

[dA03] ~~Prof.~~ Luca de Alfaro. Design and verification lab. http://www.cse.ucsc.edu/dvlab/, 2003.

what is the difference between these? If they have different URLs, shouldn't they have different titles?

[DL03a] Design and Verification Lab. CHAI. http://www.cse.ucsc.edu/dvlab/chai, 2003.

[DL03b] Design and Verification Lab. CHAI. http://www.cse.ucsc.edu/dvlab/chai/docs/html, 2003.

[Inr] Inria. Ocaml. http://caml.inria.fr/.

AHM+98
[~~R. 98~~] R. Alur ~~and~~, T.A. Henzinger ~~and~~, F.Y.C. Mang ~~and~~, S. Qadeer, ~~and~~ S.K. Rajamani and S. Tasiran. MOCHA : Modularity in model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, pages 521–525. Springer-Verlag, 1998.

HKM+00
[~~RAT00~~] T. A. Henzinger, S. C. Krishnan, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, R. Alur, L. de Alfaro and S. Tasira. *MOCHA user manual.* University of California, Berkeley, 2000.

BHS-V+
[~~RGA+~~96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, 1996. Springer Verlag.

[ssc] Sscblif. http://www.infeig.unige.ch/lab/doc/html/esterel/sscblif1.html.