Executable Protocol Specification in ESL*

E. Clarke[1] S. German[2] Y. Lu[1] H. Veith[1,3] D. Wang[1]

[1] Carnegie Mellon University [2] IBM T. J. Watson Research Center [3] TU Vienna

Abstract. Hardware specifications in English are frequently ambiguous and often self-contradictory. We propose a new logic ESL which facilitates formal specification of hardware protocols. Our logic is closely related to LTL but can express all regular safety properties. We have developed a protocol synthesis methodology which generates Mealy machines from ESL specifications. The Mealy machines can be automatically translated into executable code either in Verilog or SMV. Our methodology exploits the observation that protocols are naturally composed of many semantically distinct components. This structure is reflected in the syntax of ESL specifications. We use a modified LTL tableau construction to build a Mealy machine for each component. The Mealy machines are connected together in a Verilog or SMV framework. In many cases this makes it possible to circumvent the state explosion problem during code generation and to identify conflicts between components during simulation or model checking. We have implemented a tool based on the logic and used it to specify and verify a significant part of the PCI bus protocol.

1 Introduction

Motivation. The verification of bus protocols, i.e., of communication protocols between hardware devices, as in the case of the well-known PCI bus, is a central problem in hardware verification. Although bus protocol design and verification have become increasingly important due to the integration of diverse components in IP Core-based designs, even standard bus protocols are usually described in English which makes specifications often ambiguous, contradictory, and certainly non-executable.

Example 1. It is often the case that English documentation attaches different meanings to a single name in different situations, as in the following definition for "data phase completion" from PCI 2.2 [19], page 10: *A data phase is completed on any clock both IRDY# and TRDY# are asserted.* On Page 27, however, there is a different definition: *A data phase completes when IRDY# and [TRDY# or STOP#] are asserted.* The obvious problem with these definitions is whether data phase completion should include the case when both IRDY# and STOP# are asserted.

By carefully reading the English documentation, one can also find many contradictory statements, as in the following example from PCI 2.2 [19], page 50: *Once the master has detected the missing DEVSEL#, FRAME# is deasserted and IRDY# is deasserted.* On Page 51, however,

* This research is sponsored by the Gigascale Research Center (GSRC), the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Max Kade Foundation. One of the authors is also supported by Austrian Science Fund Project N Z29-INF. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of GSRC, NSF, or the United States Government.

it is said that *Once a master has asserted IRDY#, it can not change IRDY# or FRAME# until the current data phase completes.* The reason why these two are contradictory is that the first sentence allows FRAME# to be deasserted without the current data phase being complete, while the second one disallows this.

Traditional hardware description languages are usually not well-suited for protocol specification because they are based on existing concrete designs (or abstractions thereof) instead of specifications, and their execution model therefore focuses on single-cycle transitions. With protocols, the specification is naturally represented by constraints on signals which may connect relatively distant time points.

Another problem of transition-system based approaches is that naive composition of participants in the protocol may cover up important protocol inconsistencies due to synchronization faults or write conflicts among non-cooperative participants. It is important that the specification language be executable, i.e., that a machine model can be computed from the specification. This is a trivial property for hardware description languages, but not for protocol specifications.

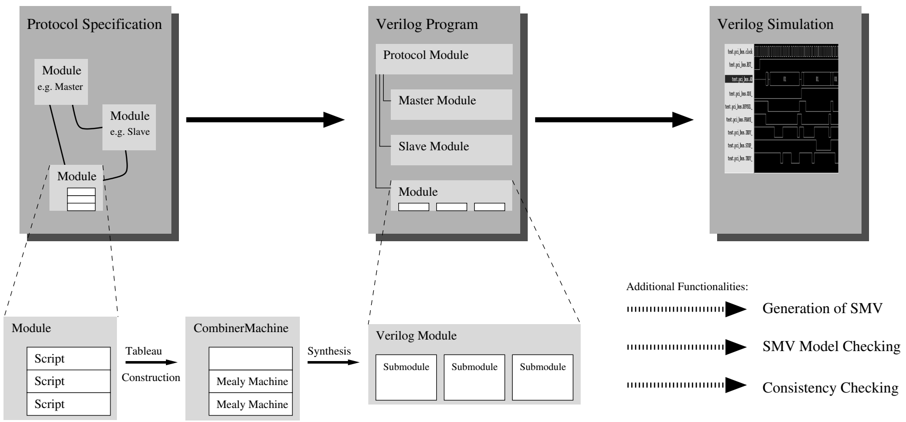

Fig. 1. System Overview

The ESL logic. We propose the new logical language **ESL** which facilitates specification, verification, and simulation of protocols. ESL is based on a variant of linear temporal logic (LTL) where atomic propositions are constraints on signals in the protocol, e.g. $\texttt{REQ} \in \{0, 2\}$. The core of ESL are *ESL scripts*, i.e., finite collections of executable temporal formulas. More precisely, a script is a finite collection of axioms $\Phi_{\mathbf{P}} \Rightarrow \Phi_{\mathbf{X}}$ where $\Phi_{\mathbf{P}}$ is a temporal formula involving only the temporal operator $\mathbf{P}$ (immediate past-time), and $\Phi_{\mathbf{X}}$ is a temporal formula involving only the temporal operator $\mathbf{X}$, e.g. $\mathbf{P}\ a \Rightarrow b \vee \mathbf{X}\ \mathbf{X}\ a$. Atoms in $\Phi_{\mathbf{P}}$ are interpreted as tests of previous signals, while $\Phi_{\mathbf{X}}$ asserts constraints in the future. Scripts may contain local variables which are not visible in the traces (i.e., models) of the script.

An *ESL module* $\mathcal{E}$ is a finite collection $\mathcal{S}_1 || \dots || \mathcal{S}_n$ of *ESL scripts*. The intended semantics is that ESL scripts describe Mealy Machines and that the ESL module specifies the synchronous composition of the Mealy machines. Different scripts of a module potentially conflict if they employ common output signals. Note that in general two scripts $\mathcal{S}_1||\mathcal{S}_2$ describe two different Mealy machines, while the conjunction $\mathcal{S}_1 \wedge \mathcal{S}_2$ of two scripts is itself a script which describes a single Mealy machine.

A finite collection $\mathcal{P}$ of ESL modules with pairwise disjoint output signals is called an *ESL protocol*. The semantics of an ESL protocol is the synchronous composition of its modules. ESL modules cannot directly conflict with each other, but they may have receptiveness failures (see Section 5).

In real protocols, ESL scripts describe distinct functionalities and features of hardware devices, while ESL modules correspond to the devices themselves. Note that different scripts can constrain the same signal, which is important when translating natural language specifications.

Experiments and Results. We show that semantically ESL expresses exactly the regular safety properties. This property is crucial for synthesizing executable models of the protocol, and distinguishes it from other executable logics, e.g. [12]. A fragment of LUSTRE has been shown to capture the same expressive power [14] in the context of synchronous data-flow languages for real-time systems. We present efficient tableau and synthesis algorithms to obtain executable models of hardware protocols, cf. Figure 1. The tricky aspect of synthesizing hardware protocols is to group the signals occuring in the specification into output signals and input signals. To this aim, we introduce a marked tableau construction which preserves the distinction between past observations and future assertions. Since each script is converted into a Mealy machine separately, there is no state explosion during the translation phase.

The execution of ESL in Verilog and SMV makes it possible to use the power of well-known and highly developed verification paradigms; in particular, it is possible to verify important features about the ESL protocol, such as synchronization contradictions between different scripts and receptiveness [2] of the ESL modules.

Another important debugging method is property checking, i.e., existing Verilog monitors and model checking tools can be easily used to debug ESL protocols. To improve coverage of the Verilog simulation, a dynamically biased random simulation test bench [20] can also be written directly in ESL scripts.

In a case study, we have used ESL to specify the PCI bus protocol Rev 2.2 [19]. Several errors were identified, including some errors from the English specification. Verilog monitors and temporal formulas from [23] have been checked against the generated Verilog and SMV models.

Related Work. Algorithms for synthesizing concurrent and distributed programs from temporal logic specifications have been developed for CTL by Clarke and Emerson [7, 8] and for LTL by Manna and Wolper [16]. Both methods synthesize global state transition systems based on an interleaved asynchronous model of computation and then use projection to obtain controllers for individual processes. Neither of these methods has been very successful because of the computational complexity of the decision procedures involved. Protocol synthesis, especially synthesis of bus protocols, is easier, since many implementation details are already given. Our methodology automatically exploits this information by incorporating it into the synthesis process.

Various formalisms have been used in hardware specification and synthesis [11, 15, 18, 21]. Recently, Shen and Arvind used term rewriting systems to specify and verify ISA and out-of-order implementations of processors [22].

Gabbay has developed an executable logic [12] for AI applications in which programs are written using rules of the form "If A holds in the past then do B". Since liveness properties can be expressed in his logic, it is not suitable for Verilog simulation.

Originally, our work was motivated by the approach of Shimizu, Dill and Hu [23]. They use monitors to specify bus protocols formally, based on rules of the form "predicate on past states $\Rightarrow$ predicate on current state". A coding style is defined that promotes readability, discourages errors, and guarantees receptiveness. We use a similar definition of receptiveness (see Section 5). Although our paper is also concerned with formal specification of bus protocols, there are several important distinctions between the two papers. First, our formalization is a *declarative formalism* based on temporal logic. Second, their monitors are restricted to interface signals. Consequently, constructing appropriate monitors can be tricky. For example, identification of "master abort" in the PCI bus protocol involves observing the bus for several cycles. Third, both Verilog and SMV models can be obtained from our specifications. Our Verilog model can directly generate valid simulation patterns and can, therefore, be simulated with part of the real design. In the monitor approach, the Verilog versions of the monitors are only applicable to a complete design, because monitors can not be used to generate valid patterns.

Consistency problems arise at different levels of specifications. Bryant et al [4] have developed a notion of consistency by identifying combinational dependencies. They show how to derive the consistency model from a modular specification where individual modules are specified as Kripke structures and give an algorithm to check the system for consistency. Very recently, the system FoCs has been developed at IBM Haifa [1]. FoCs automatically generates simulation checkers from RCTL specifications.

A more detailed exposition of the related work and proofs will be given in the full version of this paper.

Structure of the Paper In Section 2, we describe the logical framework of ESL. Section 3 contains the protocol description language which is used as input to our tool. In Section 4, the translation algorithms are presented. Section 5 shows how to debug and verify protocols. Finally, Section 6 contains our practical experiments with the PCI bus protocol. In Section 7, we briefly summarize our work, and outline future work.

2 The ESL logic

In this section we describe the linear time logic which underlies the implementation of the ESL system. A more user-friendly input language for ESL will be described in the following section.

2.1 Temporal Logic on Signal Traces

Let $V = \{v_1, \ldots, v_n\}$ be a set of variables (signals) where each v_i has finite domain D_{v_i}. A variable v_i is Boolean, if its domain D_{v_i} is $\{0, 1\}$. An *atom* is an expression

$v_i = d$, where $d \in D_{v_i}$. The finite set of atoms is denoted by $\mathbf{Atoms}$. If v_i is Boolean, then v_i abbreviates $v_i = 1$. *Literals* are atoms and negated atoms. For a set A of atoms, $\mathrm{var}(A) = \{v : \text{for some value } d \in D_i, v = d \in A\}$ is the set of variables appearing in A. A *type* is a consistent conjunction $\bigwedge_{1 \le i \le n} \alpha_i$ of literals. As is common in logic, we shall often write types as sets $\{\alpha_1, \ldots, \alpha_n\}$. A *complete type* is a type which determines the values of all variables. Note that each complete type can be assumed to contain only (unnegated) atoms.

Example 2. Suppose that $V = \{\mathtt{REQ}, \mathtt{COM}\}$, where $D_{\mathtt{REQ}} = \{1, 2, 3\}$, and $D_{\mathtt{COM}} = \{0, 1\}$. Then $\mathbf{Atoms} = \{\mathtt{REQ} = 1, \mathtt{REQ} = 2, \mathtt{REQ} = 3, \mathtt{COM} = 0, \mathtt{COM} = 1\}$. Examples of two types σ_1, σ_2 are:

1. $\sigma_1 = \{\mathtt{REQ} = 1, \mathtt{COM} \neq 0\}$, or as a conjunction, σ_1 becomes $\mathtt{REQ} = 1 \wedge \mathtt{COM} \neq 0$.
2. $\sigma_2 = \{\mathtt{REQ} \neq 2, \mathtt{COM} = 1\}$, or as a conjunction $\mathtt{REQ} \neq 2 \wedge \mathtt{COM} = 1$.

σ_1 is a complete type, because it determines the values of both REQ and COM. σ_1 is equivalent to the type $\{\mathtt{REQ} = 1, \mathtt{COM} = 1\}$. σ_2 does not determine the value of REQ, because both $\mathtt{REQ} = 1$ and $\mathtt{REQ} = 3$ are consistent with σ_2. Therefore, σ_2 is not complete.

ESL is based on a discrete time model, where time points are given by natural numbers from $N = \{1, 2, \ldots\}$. A *signal trace* is an infinite sequence $S = s_1 s_2 \cdots$, where each s_i determines the values of all variables at time i. S can be viewed as an infinite string, where the alphabet consists of *complete types*. Following the previous example,

$$S = \{\mathtt{REQ} = 1, \mathtt{COM} = 0\}\{\mathtt{REQ} = 1, \mathtt{COM} = 1\}\{\mathtt{REQ} = 3, \mathtt{COM} = 1\}\ldots$$

is a signal trace. Thus, the alphabet of signal traces is given by

$$\Sigma = \{\sigma \in 2^{\mathbf{Atoms}} : \sigma \text{ is a complete type without negation}\}.$$

The set of signal traces is given by Σ^ω.

Traces which do not determine the values of all signals are important for us as well (see Section 4.1). For this purpose, we use the alphabet Θ which consists of all types, i.e., *partial assignments* to the signals. Note that Θ is a superset of the signal trace alphabet Σ. Formally, we define

$$\Theta = \{\vartheta : \vartheta \text{ is a type}\}.$$

The set of traces is given by Θ^ω.

Remark. To keep the presentation simple, we tacitly identify two elements of Θ, if they are logically equivalent, e.g., $\mathtt{COM} = 1$ and $\mathtt{COM} \neq 0$. Thus, a rigid formal definition of Θ would require that the alphabet Θ is given by the finite number of equivalence classes of types. For example, we may use the lexicographically minimal types as representatives of their equivalence classes.

Since $\Sigma \subseteq \Theta$, every signal trace is a trace, and all definitions about traces in principle also apply to signal traces. The main difference between Σ and Θ is that each element of Σ determines the values of all signals, while Θ may contain partial information. On the other hand, with each element ϑ of Θ we can associate all elements of Σ which are consistent with ϑ. To this end, we define the function $comp : \Theta \to 2^{\Sigma}$ by

$$comp(\vartheta) = \{\sigma \in \Sigma : \sigma \Rightarrow \vartheta\}.$$

$comp$ is called the *completion function*, because it maps a type ϑ to the set of all complete types consistent with ϑ. In other words, $comp$ maps a partial assignment to the set of possible complete assignments. Let $T = t_1 t_2 \cdots \in \Theta^{\omega}$ be a trace. Then the set of signal traces described by T is given by

$$comp(T) = \{S = s_1 s_2 \cdots \in \Sigma^{\omega} : \text{for all } i,\, s_i \in comp(t_i)\}.$$

For a set L of traces $comp(L) = \{comp(T) : T \in L\}$. Given two sets L_1, L_2 of traces, we define $L_1 \approx L_2$ iff $comp(L_1) = comp(L_2)$, i.e., L_1 and L_2 describe the same set of signal traces.

Example 3. Let REQ and COM be as in Example 2. Then the trace $T = (\{\mathtt{COM} = 1, \mathtt{REQ} = 1\}\{\mathtt{COM} = 0, \mathtt{REQ} \neq 2\})^{\omega}$ does not determine REQ in all positions. It is easy to see that $comp(T)$ is given by the ω-regular expression $(\{\mathtt{COM} = 1, \mathtt{REQ} = 1\}\{\mathtt{COM} = 0, \mathtt{REQ} = 1\} | \{\mathtt{COM} = 1, \mathtt{REQ} = 1\}\{\mathtt{COM} = 0, \mathtt{REQ} = 3\}))^{\omega}$.

A *trace property* L is a set of signal traces. L_n denotes the set of finite prefixes of length n of words in L. A *safety property* is a trace property L, such that for each $t \notin L$, there exists a finite prefix r of t, such that $r \notin L_{|r|}$. In other words, traces not in L are recognized by finite prefixes.

For a set S of atoms and a set $X \subseteq V$ of variables, let S_X denote the restriction of S to atoms containing variables from X only. Similarly, Σ_X denotes the alphabet of complete types for the set of variables X.

V is partitioned into two sets V_G and V_L of *global* and *local* variables. The distinction between global and local variables will be justified in Section 2.2. Thus, S_{V_G} denotes the restriction of S to *global atoms*, i.e., atoms using global variables. Similarly, Σ_{V_G} denotes the alphabet of complete types for the set of variables V_G. Let $\mathbf{global}$ be the projection function which maps Σ to Σ_{V_G} by

$$\mathbf{global}(\sigma) = \sigma \cap \mathbf{Atoms}_{V_G}.$$

Intuitively, the function $\mathbf{global}$ removes all local atoms from the alphabet. With each signal trace $S = s_1 s_2 \cdots$ we associate the global signal trace $\mathbf{global}(S) = \mathbf{global}(s_1)\mathbf{global}(s_2) \cdots$ over $2^{\mathbf{Atoms}_{V_G}}$.

Example 4. Let $S = (\{R = 1, a = 2\}\{R = 0, a = 3\})^{\omega}$ be a signal trace where a is a local variable. Then $\mathbf{global}(S) = (\{R = 1\}\{R = 0\})^{\omega}$.

Given two sets L_1, L_2 of traces, we define $L_1 \approx_{\mathrm{gl}} L_2$ iff $\mathbf{global}(comp(L_1)) = \mathbf{global}(comp(L_2))$, i.e., L_1 and L_2 describe the same set of global signal traces.

Lemma 1. *If L is a safety property, then $\mathbf{global}(L)$ is a safety property.*

We consider a linear time logic with temporal operators $\mathbf{X}$, $\mathbf{P}$, and $\mathbf{G}$. Let $T = t_1 t_2 \ldots$ be a trace. We inductively define the semantics for atomic formulas f and temporal operators $\mathbf{X}$, $\mathbf{P}$, $\mathbf{G}$ as follows:

$T, i \models f$ iff $t_i \Rightarrow f$ (Here, we use the fact that t_i is a type, and thus a formula.)
$T, i \models \mathbf{X}\ \varphi$ iff $T, i+1 \models \varphi$
$T, i \models \mathbf{P}\ \varphi$ iff $i \geq 2$, and $T, i-1 \models \varphi$
$T, i \models \mathbf{G}\ \varphi$ iff $\forall j \geq i\ T, j \models \varphi$

The semantics of the Boolean operators is defined as usual. Existential quantification over traces is defined as follows: Let v be a variable, and let $T = t_1 t_2 \cdots$ be a trace over $\Sigma_{V-\{v\}}$, i.e., a trace which does not assign values to v. Then we define

$T, i \models \exists v.\varphi$ iff there exists an infinite sequence $a_1 a_2 \ldots$ of values for v such that the trace $S = (t_1 \cup \{v = a_1\})(t_2 \cup \{v = a_2\}) \ldots$ satisfies φ at time i, i.e., $S, i \models \varphi$. Given a formula φ, $\mathbf{Traces}(\varphi)$ denotes the set of *signal traces* T such that $T, 1 \models \varphi$.

Traces can be combined in a very natural manner:

Definition 1. *Let $T = t_1 t_2 \cdots$ and $S = s_1 s_2 \cdots$ be traces. Then the combined trace $T \bowtie S$ is given by the infinite sequence $u_1 u_2 \cdots$ where*

$$u_i = \begin{cases} t_i \wedge s_i & \textit{if } t_i \textit{ and } s_i \textit{ are consistent} \\ \{\text{false}\} & otherwise \end{cases}$$

We say that T and S are compatible *if there is no i such that $u_i = \{\text{false}\}$. Otherwise we say that T and S contradict. Let L_1 and L_2 be sets of traces. Then $L_1 \bowtie L_2 = \{T_1 \bowtie T_2 : T_1 \in L_1, T_2 \in L_2\}$.*

The above definitions easily generalize to more than two traces. Note that the operation $\bowtie$ introduces the new symbol false in the alphabet of traces.

The following important example demonstrates why the operator $\bowtie$ is different from conjunction, and why we need it to analyze traces.

Example 5. Consider the two formulas $\mathbf{G}\ (a \Rightarrow \mathbf{X}\ b)$ and $\mathbf{G}\ (a \Rightarrow \mathbf{X}\ \neg b)$. Their sets of traces are given by

$$\mathbf{Traces}(\mathbf{G}\ (a \Rightarrow \mathbf{X}\ b)) \approx (\{a\}\{b\}|\{\neg a\})^\omega$$

and

$$\mathbf{Traces}(\mathbf{G}\ (a \Rightarrow \mathbf{X}\ \neg b)) \approx (\{a\}\{\neg b\}|\{\neg a\})^\omega.$$

When viewed as specifications of different devices, the two formulas are intuitively contradictory when the input signal a becomes true. In fact, in the set of combined traces

$$\mathbf{Traces}(\mathbf{G}\ (a \Rightarrow \mathbf{X}\ b)) \bowtie \mathbf{Traces}(\mathbf{G}\ (a \Rightarrow \mathbf{X}\ \neg b)) \approx (\{a\}\{\text{false}\}|\{\neg a\})^\omega$$

the contradictions become visible immediately after a becomes true. On the other hand, the naive conjunction $\mathbf{G}\ (a \Rightarrow \mathbf{X}\ b) \wedge \mathbf{G}\ (a \Rightarrow \mathbf{X}\ \neg b)$ of the formulas is equivalent to $\mathbf{G}\ \neg a$, their set of traces is given by $\mathbf{Traces}(\mathbf{G}\ \neg a) = (\{\neg a, b\}|\{\neg a, \neg b\})^\omega$, and thus the potential contradiction vanishes.

2.2 ESL scripts

The following definition describes the fragment of linear time logic used in ESL.

Definition 2. ESL scripts

(i) A *retrospective formula* is a formula which contains no temporal operators except **P** .
(ii) A *prospective formula* is a formula which contains no temporal operators except $\mathbf{X}$.
(iii) A *script axiom* Φ is a formula of the form $\Phi_{\mathbf{P}} \Rightarrow \Phi_{\mathbf{X}}$ where $\Phi_{\mathbf{P}}$ is a retrospective formula, and $\Phi_{\mathbf{X}}$ is a prospective formula.
(iv) An *ESL script* $\mathcal{S}$ is a conjunction $\bigwedge_{1\leq j\leq k} \Phi_j$ of script axioms Φ_j.
(v) With each script $\mathcal{S}$, we associate the formula $\mathbf{G}\ (\mathcal{S}) = \mathbf{G}\ \bigwedge_{1\leq j\leq k} \Phi_j$.

Intuitively, atoms in $\Phi_{\mathbf{P}}$ are interpreted as tests of previous signals, while $\Phi_{\mathbf{X}}$ asserts constraints in the future. For simplicity, we assume that no local variable appears in two different ESL scripts.

Example 6. Consider the ESL script $\mathbf{P}\ a \wedge a \Rightarrow (\mathbf{X}\ a \vee \mathbf{X}\ \mathbf{X}\ a)$. The script says that if a held true in two subsequent cycles, then a must hold true in one of the two following cycles.

The following lemma says that the temporal operator $\mathbf{P}$ is redundant.

Lemma 2. *Let $\mathcal{S}$ be an ESL script. Then $\mathbf{G}(\mathcal{S})$ is equivalent to a formula of the form $\mathbf{G}\ (\varphi_G) \wedge \varphi_{\mathrm{Init}}$, where φ_G and φ_{Init} are temporal logic formulas which contain no temporal operators except $\mathbf{X}$.*

The following theorem states that the projection operator $\mathbf{global}$ achieves the same effect as existential quantification.

Proposition 1. *Let $\mathcal{S}$ be an ESL script, and $l_1, \ldots, l_n$ its local variables. Then* $\mathbf{global}(\mathbf{Traces}(\mathbf{G}\ \mathcal{S})) = \mathbf{Traces}(\exists l_1 \cdots l_n \mathbf{G}(\mathcal{S}))$.

Thus, projection amounts to a kind of implicit existential quantification. The effect of this quantification is characterized by the following theorem.

Theorem 1. *On global signals, ESL scripts capture the regular safety properties. Formally, for each regular safety property L over Σ_{V_G}, there exists an ESL script $\mathcal{S}$ such that* $\mathbf{global}(\mathbf{Traces}(\mathbf{G}(\mathcal{S}))) = L$*, and vice versa.*

We conclude from Theorem 1 that projection extends the expressive power of the logic to capture all regular safety properties on *global variables*. We will show in the next section that in praxis the complexity of our logic does not increase significantly. Thus, the addition of local variables appears to be a good choice which balances expressive power and complexity.

Corollary 1. *For global variables, all past time temporal operators, as well as the weak until operator* $\mathbf{W}$ *are expressible by ESL scripts.*

2.3 Regular tableaus

The tableau of an LTL formula φ is a labeled generalized Büchi automaton T that accepts exactly the sequences over $(2^{\mathbf{Atoms}_\varphi})^\omega$ that satisfy φ [13]. (Here, $\mathbf{Atoms}_\varphi$ denotes the set of atomic propositions appearing in φ.) In this section, we define *regular* tableaus by adapting LTL tableaus to ESL.

Definition 3. Regular Tableau
A *regular tableau* $\mathcal{T}$ is a tuple $\langle S^\mathcal{T}, S_0^\mathcal{T}, A^\mathcal{T}, L^\mathcal{T}, R^\mathcal{T}\rangle$ where

- $S^\mathcal{T}$ is a finite set of states, $S_0^\mathcal{T} \subseteq S^\mathcal{T}$ is a set of initial states, and $A^\mathcal{T}$ is a finite set of atoms.
- $L^\mathcal{T} : S^\mathcal{T} \to \Theta$ is a labeling function which labels states by types, where Θ is the set of types over $A^\mathcal{T}$.
- $R^\mathcal{T} \subseteq S^\mathcal{T} \times S^\mathcal{T}$ is the transition relation of the tableau.

Since by Theorem 1, ESL defines only safety properties, tableaus for ESL do not need all the expressive power of ω-automata. Moreover, the states of regular tableaus are labeled only by sets of atoms (and not by temporal formulas). Intuitively, temporal formulas can be omitted from tableaus because we have local variables which carry the information that is usually carried by the temporal formulas.

Definition 4. Regular Tableau Acceptance
A trace $T = t_1 t_2 \cdots \in \Theta^\omega$ is accepted by $\mathcal{T}$ if there exists a sequence $s_1 s_2 \cdots \in (S^\mathcal{T})^\omega$ such that

(i) $s_1 \in S_0^\mathcal{T}$
(ii) $s_1 s_2 \cdots$ is an infinite path in the graph given by the transition relation $R^\mathcal{T}$.
(iii) For all i, $t_i \Rightarrow L^\mathcal{T}(s_i)$.

The language $\mathcal{L}(\mathcal{T})$ is the set of traces T accepted by tableau $\mathcal{T}$. Let $\mathcal{S}$ be an ESL script, and $\mathcal{T}$ a tableau. $\mathcal{T}$ is a *correct tableau* for $\mathcal{S}$, if $\mathcal{L}(\mathcal{T}) \approx \mathbf{Traces}(\mathbf{G}(\mathcal{S}))$, i.e., if the traces generated by the tableau define exactly the signal traces which satisfy $\mathcal{S}$.

3 ESL Protocols

ESL facilitates modular specification of protocols, in particular hardware protocols. ESL protocols consist of modules which in turn consist of scripts. Under the intended semantics of ESL, modules correspond to distinct devices (e.g. a master and a slave device), while scripts describe independent functionalities of the devices.

Formally, an ESL *module* $\mathcal{E}$ is a finite collection $\mathcal{S}_1, \ldots, \mathcal{S}_n$ of scripts. Each script $\mathcal{S}_i$ is given by a finite conjunction $\bigwedge \varphi_j$ of specifications. The intended semantics is that ESL scripts describe Mealy Machines and that the ESL module specifies the synchronous composition of the Mealy machines. Therefore, we shall write $\mathcal{S}_1 || \mathcal{S}_2 \ldots || \mathcal{S}_n$ to denote $\mathcal{E}$. Different scripts for the same module potentially conflict if they employ common output signals, i.e., common global variables which model output signals. Scripts may contain local variables which are not visible to other scripts.

Our aim is to build a machine $M_{\mathcal{E}}$ such that the language accepted by $M_{\mathcal{E}}$ coincides with the combined traces of the scripts on global variables, i.e.,

$$\mathcal{L}(M_{\mathcal{E}}) \approx_{\mathrm{gl}} \mathbf{Traces}(\mathcal{S}_1) \bowtie \cdots \bowtie \mathbf{Traces}(\mathcal{S}_n).$$

As shown in Example 5, trace combination will enable us to identify contradictions between scripts.

A finite collection $\mathcal{P}$ of ESL modules with pairwise disjoint output signals is called an *ESL protocol*. The semantics of protocols is given by the semantics of the *scripts* of its constituent modules. ESL modules of a protocol have fewer sources of conflict among each other than ESL scripts because they do not have common output signals.

3.1 Input Language for ESL Protocols

An ESL protocol must contain exactly one protocol module which starts with the keyword **protocol**. Each module can have three types of variables: **input**, **output** and **local** variables. Each **script** can include more than one specification, which start with the keyword **spec**.

To facilitate easy description of properties and protocols, we augment the basic ESL language with syntactic sugar, such as default values, additional temporal operators, and parametrized macros. In particular, we use the weak until operator **W** and the operator $keep(a)$ which asserts that the current value of signal a is the same as in the previous step. Note that by Lemma 2, we can add all the past temporal operators to the language because they only define regular safety properties. Parameters are used to adapt specifications to hardware configurations of varying size.

Due to space restrictions, we describe the input language by example. The following example is a fragment of the specification of the PCI bus in ESL. It consists of one master and one arbiter.

module pci_master
  input $\mathtt{clock}, \mathtt{GNT}$: bool;
  output $\mathtt{REQ} = 0$: bool;
  local $\mathtt{status} : \{\mathtt{COMP}, \mathtt{MABORT}, \mathtt{MTO}\}$;
  local $\mathtt{retry\_req}, \mathtt{last\_cycle}, \mathtt{addr\_phase}$: bool;
  script **spec** $\mathtt{retry\_req} \Rightarrow \mathbf{X}\ (\mathtt{retry\_req}\ \mathbf{W}\ \mathtt{addr\_phase})$;
  spec $\mathtt{last\_cycle} \Rightarrow \neg\mathtt{retry\_req}$;
  script **spec** $\mathtt{GNT} \Rightarrow \mathbf{X}(\mathtt{keep(REQ)})\ \ \mathbf{W}\ \ \mathtt{status} = \mathtt{COMP}$;
endmodule

module pci_arbiter
  parameter N;
  input $\mathtt{RST}, \mathtt{REQ}$[1 to N]: bool;
  output $\mathtt{GNT}$[1 to N]: bool;
  local $\mathtt{tok}$[1 to N]: bool;
  script spec
    $\mathtt{RST}\ \wedge\ \mathtt{tok}[i] \Rightarrow \mathbf{X}\ (\mathtt{tok}[(i+1)\ \mathrm{Mod\ N}] = 1\ \wedge\ \mathrm{tok[i]} = 0)\ \mathbf{for}\ \mathrm{i} = 1\ \mathbf{to}\ \mathrm{N}$;
endmodule

protocol pci_bus
    constant N = 2;
    modules
        $master$[1 to N]: pci_master,
        $arbiter$(N): pci_arbiter;
    connection
        $master[i].\mathtt{GNT} = arbiter.\mathtt{GNT}[i]$ **for** $i = 1$ **to** N;
        $arbiter.\mathtt{REQ}[i] = master[i].\mathtt{REQ}[i]$ **for** $i = 1$ **to** N;
endprotocol

This example includes two modules and one protocol module. Each module includes one or two scripts. In the module *pci_master*, `REQ` has Boolean type and is initialized to 0. The local variable *status* denotes whether the PCI master aborts the transaction or the time out happens. In the module *pci_arbiter*, we define a parameter N which will be instantiated when the module gets instantiated. A formula $\varphi(i)$ **for** $i = 1$ **to** N is equivalent to $\varphi(1) \wedge \varphi(2) \wedge \ldots \wedge \varphi(N)$. For example, the script of the module pci_arbiter is instantiated as a conjunction of two formulas:

$$(RST \wedge token[2] \Rightarrow \mathbf{X}\ (token[1] := 1 \wedge token[2] := 0)) \wedge$$
$$(RST \wedge token[1] \Rightarrow \mathbf{X}\ (token[2] := 1 \wedge token[1] := 0))$$

The protocol module explicitly connects the modules *pci_master* and *pci_aribter* by matching the corresponding inputs and outputs.

4 Synthesis of Executable Models

Given an ESL protocol, our procedure comprises three main steps to transform executable models into either Verilog or SMV programs.

1. *Preprocessing.* In this step, the ESL protocol is parsed and type checked to make sure each variable is declared and each axiom is well typed. For example, for a Boolean variable x, an atom $x = 3$ is not allowed. Furthermore, the parameters are instantiated and the macros are expanded.
2. *Module Synthesis.* This step converts each ESL module separately into a Verilog or SMV module. An overview of the algorithm **ModuleSynthesis** given in Figure 2. In the algorithm, we first generate a regular tableau for each script and translate the tableau into a nondeterministic automaton. Then we determinize the automaton and translate it into a Mealy machine which can be easily implemented in Verilog or SMV. The combiner connects each Mealy machine. Details of each step will be described in the following three subsections.
3. *Module Connection.* In this final step, the inputs and outputs from different ESL modules are connected. In the synchronous bus protocol design, combinational dependency loops between the signals are not allowed. This step identifies all combinational dependencies within individual modules or between modules. A short outline of this step is given in Section 4.4.

Algorithm ModuleSynthesis$(\mathcal{S}_1 \| \cdots \| \mathcal{S}_n)$
foreach $\mathcal{S}_i$
$\mathcal{T}_{\mathcal{S}_i} = \mathbf{MTableau}(\mathcal{S}_i)$
$\mathcal{N}_{\mathcal{S}_i} = \mathbf{Automaton}(\mathcal{T}_{\mathcal{S}_i})$
$\mathcal{N}^{\mathcal{D}}_{\mathcal{S}_i} = \mathbf{Powerset}(\mathcal{N}_{\mathcal{S}_i})$
$M_{\mathcal{S}_i} = \mathbf{GenMealy}(\mathcal{N}^{\mathcal{D}}_{\mathcal{S}_i})$
return Combine$(M_{\mathcal{S}_1}, \ldots, M_{\mathcal{S}_n})$

Fig. 2. Algorithm to synthesize scripts

Algorithm MTableau$(\mathcal{S})$
mark $\mathcal{S}$
$\mathcal{T}_{\mathcal{S}} := \mathrm{Tableau}(\mathcal{S})$
for all states $s \in S^{\mathcal{T}_{\mathcal{S}}}$
if $\mathrm{clean}(L^{\mathcal{T}_{\mathcal{S}}})$ **is inconsistent**
then remove s **from** $\mathcal{T}_{\mathcal{S}}$
return $\mathcal{T}_{\mathcal{S}}$

Fig. 3. Tableau Construction.

4.1 Tableau construction

Given an ESL script $\mathcal{S}$, our aim is to generate a Mealy machine whose operational behavior is specified by the script. Consider the two simple scripts $a = 1 \Rightarrow b = 1$ and $b = 0 \Rightarrow a = 0$, where a and b are Boolean variables. Logically, these scripts are equivalent (since $a \Rightarrow b$ is equivalent to $\neg b \Rightarrow \neg a$.) However, as specifications for Mealy machines they should intuitively describe different machines: the formula $a = 1 \Rightarrow b = 1$ describes a Mealy machine which asserts $b = 1$ if it observes $a = 1$, while $b = 0 \Rightarrow a = 0$ describes a Mealy machine which asserts $a = 0$ if it observes $b = 0$. We conclude from this example that for the operational behavior of the Mealy machines it is important to know for each occurrence of a variable whether it belongs to the retrospective or the prospective part of a script. Variables from the retrospective part eventually will become inputs of Mealy machines, and variables from the prospective part will become outputs of Mealy machines.

In our methodology, we first build a tableau for $\mathcal{S}$, and then translate it further to a Mealy machine. As argued above, it is important for our tableau construction not to lose the information which variables will be used as outputs of the Mealy machine later on. Therefore, we distinguish such variables by marking them with a symbol $\bullet$.

Definition 5. Given a set of variables $V = \{v_1, \ldots, v_n\}$, $V^\bullet$ is a set $\{v_1^\bullet, \ldots, v_n^\bullet\}$ of new variables called *marked* variables.

Given an ESL axiom $\varphi = \Phi_{\mathbf{P}} \Rightarrow \Phi_{\mathbf{X}}$ and the corresponding alphabet Θ, the *marked axiom* φ_m is given by $\Phi_{\mathbf{P}} \Rightarrow \Phi^\bullet_{\mathbf{X}}$ where $\Phi^\bullet_{\mathbf{X}}$ is obtained from $\Phi_{\mathbf{X}}$ by replacing each occurrence of a variable v by the marked variable $v^\bullet$. ESL scripts are marked by marking all their axioms.

Let X be a set of variables or atoms etc. Then $\mathbf{umk}(X)$ denotes the subset of X where no element contains a marked variable, and $\mathbf{mkd}(X)$ denotes $X - \mathbf{umk}(X)$. The function $\mathbf{clean}(X)$ removes all markers from the elements of X, e.g. $\mathbf{clean}(V^\bullet) = V$.

In Figure 3, we outline the tableau algorithm **MTableau**$(\mathcal{S})$ for ESL scripts. In the first step, the algorithm marks the script as described above. In the second step we use the standard LTL tableau algorithm described in [13] to generate a tableau for the marked script. Note that from the point of view of the tableau construction algorithm, a variable v and its marked version $v^\bullet$ are different. In the third step, however, we exclude those states from the tableau where the assertions are inconsistent with the observations, i.e., those states whose labelling would become inconsistent if the marks were removed. Thus, the resulting marked tableau is a correct tableau for $\mathcal{S}$ if the markers are ignored.

Lemma 3. *Let $\mathcal{S}$ be a script. Let L be the set of traces accepted by the tableau $\mathbf{MTableau}(\mathcal{S})$. Then $\mathbf{clean}(L) \approx \mathbf{Traces}(\mathcal{S})$, i.e., after unmarking, the traces of the tableau are the same as the traces of the script.*

Our actual implementation of the algorithm **MTableau** removes inconsistent states on-the-fly *during* constructing tableaus. Thus, fewer intermediate states are created, and less memory is used. In principle, other tableau algorithms [9] could be used, too.

Note that local and global variables are not distinguished by the tableau algorithm. The tableau traces in general contain local variables.

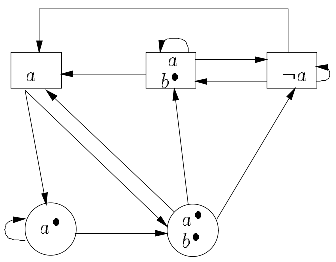

Fig. 4. An example tableau. Boxes denote initial states.

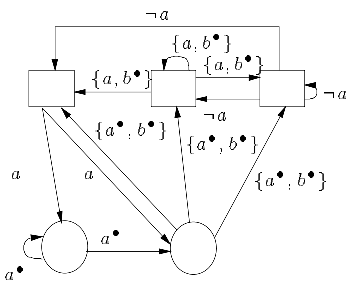

Fig. 5. The corresponding automaton

The following example highlights the construction of tableaus for marked scripts. We will use this example as a running example throughout the paper.

Example 7. Consider the following axiom: $(a \Rightarrow b \vee \mathbf{X}\ a)$ The meaning of the axiom is: whenever a is asserted, then either b is asserted at the same time point or a is asserted in the next time point. The corresponding marked axiom is: $(a \Rightarrow b^\bullet \vee \mathbf{X}\ a^\bullet)$ The tableau for this axiom is shown in Figure 4.

Before describing how to translate tableaus into automata, we first give a formal definition of automata.

Definition 6. A *nondeterministic ESL automaton* N is a 4-tuple $\langle S, S_0, \Theta, \delta\rangle$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, Θ is the set of types, and $\delta \subseteq S \times \Theta \times S$ is the transition relation. A trace $T = t_1 t_2 \cdots \in \Theta^\omega$ is accepted by N if there exists a sequence $s_1 s_2 \cdots \in S^\omega$ such that $s_1 \in S_0$, $s_1 s_2 \cdots$ is an infinite sequence of states, and for all i, there exists a type $\vartheta \in \Theta$ such that $t_i \Rightarrow \vartheta$, and $(s_i, \vartheta, s_{i+1}) \in \delta$. The language $\mathcal{L}(N)$ is the set of traces accepted by N.

Given a tableau $\mathcal{T} = \langle S^\mathcal{T}, S_0^\mathcal{T}, A^\mathcal{T}, L^\mathcal{T}, R^\mathcal{T}\rangle$, the algorithm $\mathbf{Automaton}(\mathcal{T})$ computes the automaton $\mathcal{N}_\mathcal{T} = \langle S, S_0, \Theta, \delta\rangle$ where $S = S^\mathcal{T}$, $S_0 = S_0^\mathcal{T}$, and $\delta = \{(s, \vartheta, s') \mid \vartheta \in \Theta, (s, s') \in R^\mathcal{T}, L^\mathcal{T}(s) = \vartheta\}$. Then the following lemma holds trivially.

Lemma 4. *The tableau $\mathcal{T}$ and the ESL automaton* $\mathbf{Automaton}(\mathcal{T})$ *accept the same languages. Formally,* $\mathcal{L}(\mathcal{T}) = \mathcal{L}(\mathbf{Automaton}(\mathcal{T}))$.

4.2 Mealy machine synthesis

Since ESL automata are nondeterministic, we cannot translate them directly into Verilog. In this section, we describe how to synthesize automata into deterministic Mealy machines which can then be easily translated into Verilog programs. We proceed in two steps. First, we use a powerset algorithm to determinize the automata. Then, we use the variable markers to determine the inputs and outputs for each state of the Mealy machines.

Since ESL automata do not have Büchi constraints, we use the traditional method for automata determinization by powersets which is described in many textbooks [25].

Although the algorithm **Powerset** is potentially exponential, the resulting automata in our experiments are often much smaller than the original automata. In Figure 6, results for 62 scripts in our PCI protocol specification are shown, where for each script, the ratio of the size of the deterministic automaton compared with that of the original non-deterministic automaton is shown. It can be seen that, in our experiments, the deterministic automata are always smaller than the nondeterministic automata. On average, the automata can be compressed to about 25% of their original size. The intuitive explanation for this behavior is that the powerset algorithm clusters related states into one state. For the example in Figure 5, the automaton after determinization is shown in Figure 7. Since the deterministic automaton accepts the same language as the original

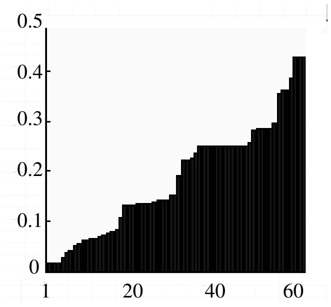

Fig. 6. Compression obtained by determinization

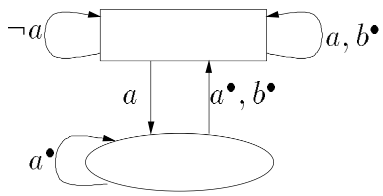

Fig. 7. Automaton after determinization

automaton does, the behavior of the original scripts is maintained after determinization.

Finally, we describe how to generate Mealy machines from deterministic automata. The Mealy machine model presented in the following definition takes into account the type concept which we use for traces.

Definition 7. Mealy machine
A *Mealy machine* M is a tuple $\langle S, S_0, I, O, \lambda, R \rangle$ where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, I is a finite set of input variables, O is a finite set of output

variables ($I \cap O = \emptyset$), $\lambda : S \times \Sigma_I \to \Sigma_O$ is the output function, and $R \subseteq S \times \Sigma_I \times S$ is the transition relation. M is deterministic if for every input σ every state has a unique successor i.e., for all $s, s' \in S$ and $\sigma \in \Sigma_I$, $R(s, \sigma, s') \wedge R(s, \sigma, s'') \to s' = s''$. M accepts a trace $T = t_1 t_2 \cdots$ over alphabet $\Sigma_I \cup \Sigma_O$ if there exists an infinite sequence $S = s_1 s_2 \cdots$ of states such that *(i)* $s_1 \in S_0$; *(ii)* for all indices i, $R(s_i, \mathbf{umk}(t_i), s_{i+1})$; *(iii)* for all indices i, $\lambda(s_i, \mathbf{umk}(t_i)) = \mathbf{mkd}(t_i)$. The set of traces accepted by M is denoted by $\mathcal{L}(M)$.

Given a deterministic automaton $N = \langle S, S_0, \Omega, \delta, F\rangle$, we generate a Mealy machine $M = \langle S, S_0, I, O, \lambda, R\rangle$ such that N and M have the same sets of states and initial states. Let `MAX` be the smallest number such that no state in N has more than `MAX` immediate successors.

The input variables I of the Mealy Machine M are the unmarked variables of the automaton N, and the new variable nd, $D_{nd} = \{1, \ldots, \mathtt{MAX}\}$. Intuitively, the new variable nd will be used to determine the successor state among the `MAX` possible successor states. The output variables O of M are the marked variables of N, i.e., $O = \mathbf{mkd}(\Omega)$. The output function λ and the transition relation R are defined by the algorithm **GenMealy** shown in Figure 8.

Algorithm GenMealy(N)
  foreach $s \in S$
    $P = \{\mathbf{umk}(a) \mid \exists s' \in S, \delta(s, a, s')\}$
    for each $a \in P$
      $Q = \{b \mid \mathbf{umk}(b) = a, \exists s' \in S, \delta(s, b, s')\}$
      i = 1
      for each $b \in Q$
        $\lambda = \lambda \cup \{(s, a \wedge nd = i, \mathbf{mkd}(b))\}$
        if $\delta(s, b, s') = \mathtt{true}$ **then**
          $R = R \cup \{(s, a \wedge nd = i, s')\}$
        i = i + 1
      choose some $b \in Q$
      for j=i **to** MAX
        $\lambda = \lambda \cup \{(s, a \wedge nd = j, \mathbf{mkd}(b))\}$
        $R = R \cup \{(s, a \wedge nd = j, s')\}$

Fig. 8. Pseudo-code for **GenMealy**

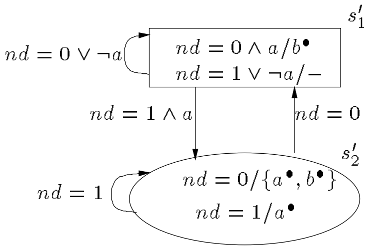

Fig. 9. A deterministic Mealy machine

Example 8. The Mealy machine obtained from the automaton in Figure 5 is shown in Figure 9. nd is the new unconstrained internal signal. The labelling in state s_2 denotes the input/output function, for example $nd = 1/a^\bullet$ represents input $nd = 1$ and output is $a^\bullet = 1$.

Theorem 2. *Let $\mathcal{S}$ be an ESL script. Then the language of the synthesized Mealy machine coincides with $\mathbf{Traces}(\mathcal{S})$ on global variables. Formally,*

$$\mathbf{clean}(\mathbf{Traces}(\mathbf{ModuleSynthesis}(\mathcal{S}))) \approx_{\mathrm{gl}} \mathbf{Traces}(\mathcal{S}).$$

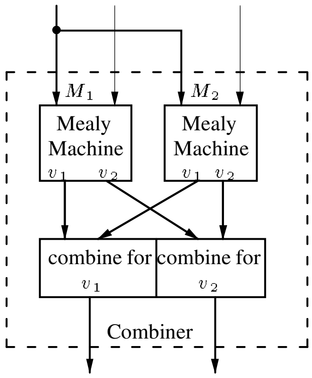

Fig. 10. Combiners for a Module

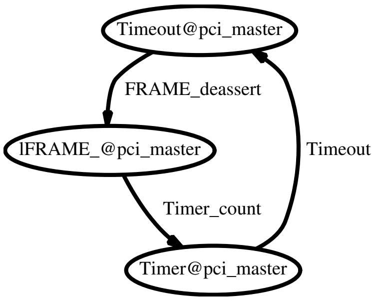

Fig. 11. Identified Combinational dependency loops

The obtained Mealy machine can be translated into a Verilog program easily. Details are omitted due to the space restrictions.

4.3 Combiner generation

Recall that each ESL script $\mathcal{S}$ is translated into a Mealy machine, and that different scripts can assign values to the same signal. In Section 2 we defined the operation $\bowtie$ to combine two traces. As exemplified in Example 5, the operation $\bowtie$ on traces is not the same as conjunction of scripts.

On the level of Mealy machines, we introduce *combiners* which perform the operation $\bowtie$ on traces. A combiner machine is defined as follows.

Definition 8. Given a sequence $M_1, \ldots, M_m$ of Mealy machines with possibly nondisjoint alphabets, the *combiner* $C(M_1, \ldots, M_m)$ is a Mealy machine whose input alphabet is the union of the input alphabets of the M_i and whose output alphabet is obtained from the union of the output alphabets of the M_i by removing the marks. On input of a symbol σ, $C(M_1, \ldots, M_m)$ simulates each of the Mealy machines, collects the outputs $o_1, \ldots, o_m$ of the Mealy machines, and nondeterministically outputs one element of $\mathbf{clean}(comp(o_1 \bowtie o_2 \bowtie \ \ldots \ \bowtie o_m))$.

Thus, if the output of the Mealy machines is consistent, the combiner will output it; if it is inconsistent, then the combiner will output $\{false\}$; if the output does not determine the values of all signals, the combiner will nondeterministically choose consistent values for the unspecified or under-constrained signals.

The algorithm **Combine** generates Verilog code for the combiner. In the Verilog translation, each Mealy machine is translated into a Verilog module. The combiner itself is a module which uses the Mealy machine modules as subprograms and combines their outputs.

In the traditional approach (i.e., synthesizing the conjunction of *all* scripts), the number of tableau states for the conjunction of the scripts can become exponentially larger. In fact, our techniques are tailored to find inconsistencies between scripts; in conjoined formulas, inconsistent behaviors would be eliminated.

Finally, we can formally state the correctness of our algorithms, cf. Section 3.

Theorem 3. *Let $\mathcal{E} = \mathcal{S}_1 || \cdots || \mathcal{S}_n$ be an ESL module. Then*

$$\mathcal{L}(\mathbf{ModuleSynthesis}(\mathcal{E})) \approx_{\text{gl}} \mathbf{Traces}(\mathcal{S}_1) \bowtie \cdots \bowtie \mathbf{Traces}(\mathcal{S}_n).$$

4.4 Module Connection

In Verilog, connecting the inputs and outputs of modules can be done hierarchically. For synchronous bus designs, combinational loops are not allowed. Therefore, in the variable dependency graph, we use standard algorithms for strongly-connected components [6] to identify combinational loops. Our tool will report combinational loops to the users. Figure 11 shows an example combinational loop identified during PCI bus protocol debugging. In the Figure, $Timeout$, $lFRAME\_$, and $Timer$ are signals in module pci_master, and $FRAME\_deassert$, $Timer\_count$ and $Timeout$ are names of scripts in this module. Each edge represents a dependency between the two signals introduced by the script, e.g., signal $lFRAME\_$ depends on $Timeout$ in script $FRAME\_deassert$.

5 Debugging Protocols in ESL

For a protocol specified in ESL, it is essential to have debugging capabilities which verify that the specified protocol is indeed what the designers want to specify. In this section, we describe special properties of the generated Verilog and SMV code which facilitate debugging. Due to space restrictions, we describe only some of the debugging capabilities which we implemented. The code generator for Verilog and SMV can be extended easily to handle other capabilities.

Synchronization Contradiction. In ESL, two scripts can potentially assert contradictory values to signals, cf. Example 5. To detect such cases, a flag OC is defined in the Verilog code for the combiner. As soon as the combiner computes $\{\text{false}\}$, the flag is automatically set to 1. According to Theorem 3, the flag OC is set to 1 if and only if the traces of the scripts contradict. Therefore, to verify the absence of synchronization contradictions (i.e., consistency) it suffices to check for unreachability of $OC = 1$.

Receptiveness. A machine is receptive [2] in an environment, if starting from any state, for all possible inputs from the environment, the machine can produce valid output and transit to a valid next state. Receptiveness is among the most important properties of protocols. In ESL, receptiveness questions arise on the level of *modules*. Receptiveness is interesting for a module in connection with the whole protocol, but also for a single module with unconstrained inputs. For each module M, a special flag BI_M is defined in the Verilog code for M. If a Mealy machine belonging to M gets stuck, i.e., has no valid transition for the given input, then the flag BI_M is set to 1.

The absence of synchronization contradictions and the unreachability of BI_M for each M guarantees the receptiveness of the ESL protocol. Deadend checks for receptiveness have been proposed previously for Moore machines in [23].

Property Checking. The specification of protocols often can be partitioned into specifications of a fundamental and operational character, and another set of additional specifications which were logically redundant for a correct protocol, but are aimed at finding out if the protocol is correctly specified.

In ESL, we build Verilog and SMV models based on the operational specifications, and then use a Verilog simulator and the SMV model checker to verify the additional specifications.

6 Experimental Results

We have built a prototype system in Standard ML, which includes about 13,000 lines of code. To test the capability of our tool, we have specified a subset of PCI bus protocol [19]. The specification consists of 118 formulas and 1028 lines including English comments for each formula. The specification includes five module types: master sequencer, master backend, target sequencer, target backend, and arbiter. One master sequencer and one master backend form a PCI master; conversely, one target sequencer and one target backend form a PCI target. We have specified a biased random test bench [20] in the master backend, because in PCI, the test bench involves transaction request generation, which is an integral part of the behavior of the master backend. Using the parameter concept in ESL, the number of modules on the bus can be easily modified. Independently, [23] developed a specification methodology for the PCI protocol, in which explicitly declared signals are used for monitoring protocol violation.

The specified subset of the PCI protocol includes burst transaction, master-initiated fast-back-to-back transaction, master and target termination conditions, initial and subsequent data latencies, target decode latency, master reissue of retried request. We are currently working on configuration cycles, bus parking and parity checking. We plan to specify the complete PCI bus protocol in future work.

Our algorithm is very efficient, and it only takes about 15 seconds to generate the SMV or Verilog model from 62 ESL scripts on a 550MHz Pentium **III** machine with 256 MB memory. The generated Verilog is about 7000 lines of code while the generated SMV is about 6100 lines of code. Using the Cadence Verilog-XL simulator, we are able to simulate the generated Verilog model. Many easy errors have been identified by checking synchronization contradiction in Verilog simulation, including some errors in the protocol. For example, the following two statements from [19] can be contradictory when a PCI target can perform fast decode.

1. In page 26, A PCI target is required to assert its TRDY# signal in a read transaction unconditionally when the data is valid.
2. In page 47, The first data phase on a read transaction requires a turnaround-cycle (enforced by the target via TRDY#).

The sixty nine Verilog monitors from [24] are used to characterize correctness of the PCI bus protocol. In order to verify that our Verilog model is correct, we connect the monitors with our model to check whether our model violates the monitors.

Model checking can formally verify the correctness of a given formula. However, it is limited by the size of the model. We use it on a limited configuration of the protocol, namely one master, one target, one dummy arbiter. After abstracting the data path, e.g., the width of the bus, we have successfully model checked 70 temporal properties which are derived from the temporal properties given by [24]. The verification takes 450MB memory and 2 minutes on a 360MHz Sun 6500 Enterprise server.

7 Conclusions

We have proposed a new logic ESL for formal specification of hardware protocols. Our synthesis methodology generates executable code in Verilog or SMV from ESL specifications. A significant part of the PCI bus protocol has been specified, simulated, and verified using our tool. Our experimental results clearly demonstrate that our synthesis methodology is feasible for systems of realistic complexity.

In the future, we plan to complete the PCI bus specification and to experiment with other industrial bus protocols such as the IBM CoreConnect bus and the Intel P6 bus. To achieve this goal, we are investigating various extensions of ESL. In particular, we believe it is possible to incorporate our logic into Verilog without significantly changing the syntax of the language. Such an extension should also make our tool easier for engineers to use.

Finally, we believe that game theoretic notions of consistency [10, 3] are necessary under certain circumstances. We intend to investigate various notions of consistency for our logic and devise algorithms for verifying them.

Acknowledgment

We are grateful to David Dill and Kanna Shimizu for discussions and comments on our paper, and for providing the PCI properties[24].

References

1. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar and Y. Wolfsthal,"FoCs - Automatic Generation of Simulation Checkers from Formal Specifications", *CAV 00: Computer-Aided Verification*, Lecture Notes in Computer Science 1855, 538-542. Springer-Verlag, 2000.
2. R. Alur and T.A. Henzinger. Reactive Modules, *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, 207-218. IEEE Computer Society Press, 1996.
3. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symposium on Foundations of Computer Science*, 100-109, 1997.
4. R. E. Bryant, P. Chauhan, E. M. Clarke, A. Goel. A Theory of Consistency for Modular Synchronous Systems. Formal Methods in Computer-Aided Design, 2000
5. P. Chauhan, E. Clarke, Y. Lu, and D. Wang. Verifying IP-Core based System-On-Chip Designs. In *Proceedings of the IEEE ASIC Conference*, 27-31, 1999.
6. T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithm. *MIT Press*, 1990.
7. E. Clarke and E. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Logic of Programs: Workshop*, Yorktown Heights, NY, May 1981 Lecture Notes in Computer Science, Vol. 131, Springer-Verlag. 1981.

8. E. Emerson and E. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. In *Science of Computer Programming*, Vol 2, 241-266. 1982.
9. E. Clarke, O. Grumberg, and D. Peled. Model Checking. *MIT Publishers*, 1999.
10. David Dill. Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits. *MIT Press*, 1989.
11. M. Fujita and S. Kono. Synthesis of Controllers from Interval Temporal Logic Specification. *International Workshop on Logic Synthesis*, May, 1993.
12. D. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, B. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, Vol. 398, 409-448. Springer Verlag, LNCS 398, 1989.
13. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.
14. N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language Lustre. In *Sixth International Symp. on Lucid and Intensional Programming*, ISLIP'93, Quebec City, Canada, April 1993. Universit'e Laval.
15. L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, March 1994.
16. Z. Manna, P. Wolper: Synthesis of Communicating Processes from Temporal Logic Specifications, *ACM TOPLAS*, Vol.6, N.1, Jan. 1984, 68-93.
17. K.L. McMillan. Symbolic Model Checking. *Kluwer Academic Publishers*, 1993.
18. B. Moszkowski. Executing temporal logic programs. *Cambridge University Press*, 1986.
19. PCI Special Interest Group. PCI Local Bus Specification Rev 2.2. Dec. 1998.
20. J. Yuan, K. Shultz, C. Pixley, and H. Miller. Modeling Design Constraints and Biasing in Simulation Using BDDs. In *International Conference on Computer-Aided Design*. 584-589, November 7-11, 1999
21. A. Seawright, and F. Brewer. Synthesis from Production-Based Specifications. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 194-199, 1992.
22. X. Shen, and Arvind. Design and Verification of Speculative Processors. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems*, June 1998, Marstrand, Sweden.
23. K. Shimizu, D. Dill, and A. Hu. Monitor Based Formal Specification of PCI. Formal Methods in Computer-Aided Design, 2000.
24. K. Shimizu. http://radish.stanford.edu/pci.
25. M. Sipser. Introduction to the Theory of Computation. *PWS Publishing Company*. 1997.