Path Predicate Abstraction by Complete Interval Property Checking

Joakim Urdahl, Dominik Stoffel, Jörg Bormann*, Markus Wedler, Wolfgang Kunz
Dept. of Electrical and Computer Eng., U. of Kaiserslautern, Germany *Abstract RT Solutions, Munich, Germany

***Abstract*—This paper describes a method to create an abstract model from a set of properties fulfilling a certain completeness criterion. The proposed abstraction can be understood as a *path predicate abstraction*. As in predicate abstraction, certain concrete states (called *important* states) are abstracted by predicates on the state variables. Additionally, paths between important states are abstracted by path predicates that trigger single transitions in the abstract model. As results, the non-important states are abstracted away and the abstract model becomes time-abstract as it is no longer cycle-accurate. Transitions in the abstract model represent finite sequences of transitions in the concrete model. In order to make this abstraction sound for proving liveness and safety properties it is necessary to put certain restrictions on the choice of state predicates. We show that *Complete Interval Property Checking (C-IPC)* can be used to create such an abstraction. Our experimental results include an industrial case study and demonstrate that our method can prove global system properties which are beyond the scope of conventional model checking.**

I. Introduction

Even after years of progress in the area of formal property checking, simulation is still the predominant technique in industrial practice for verifying the hardware of Systems-on-Chip (SoCs). There are at least two reasons for that: first, formal techniques provide a rigorous correctness proof only for a selected piece of design behaviour. Ensuring such local correctness is considered valuable, especially in corner cases of a design. However, in many industrial flows only parts of the overall design behavior are covered by properties and confidence in the correctness of the overall design largely still depends on simulation examining global input stimuli to the design and their output responses. Sophisticated methodologies have been developed to achieve full design coverage by local properties according to rigorous completeness metrics, as we will consider them in this paper. This can indeed contribute to replace simulation for SoC module verification by formal techniques. However, even in such a scenario simulation will still be needed for chip-level verification. This is the second reason for the prevailing role of simulation in SoC hardware verification. Formulating and proving global system properties spanning across different SoC modules on the Register-Transfer-Level (RTL) of an SoC is clearly beyond the capacity of current tools. As an alternative more viable than proving global properties at the RTL we may resort to system-level verification based on abstract design descriptions as they are supported by languages like SystemC [1]. However, unless the design refinements from these high levels to lower implementation levels become fully automated and can possibly be supported by new generation equivalence checking tools it is apparent that chip-level simulation at the RTL will still be needed and will contribute substantially to the overall verification costs.

The techniques proposed in this paper intend to make a step toward proving global properties for an RTL design implementation. We will prove properties that span over multiple SoC modules and which are significantly more complex than what can be proved with available property checkers. Our approach is not based on boosting the performance of the proof engines. Instead we propose a methodology to create design abstractions based on sets of properties fulfilling a certain completeness criterion.

The proposed approach leverages the significant advances that have been made over the last years in developing *systematic* procedures of writing properties for comprehensively capturing the behavior of a design. In particular, property checking formulations such as in Symbolic Trajectory Evaluation (STE) are very suitable for a systematic approach [2], [3], [4]. STE employs a specific style of formulating properties making it natural to compose properties into a more comprehensive design description which is successfully used in industrial practice. This paper is based on a related industrial property checking formulation called *Interval Property Checking (IPC)* [5], [6]. IPC stands for proving so called *operation properties* or *interval properties* on a bounded circuit model based on satisfiability (SAT) solving. From a computational perspective it can therefore also be seen as a variant of Bounded Model Checking (BMC) [7] while STE is more related to symbolic simulation and has a different way of representing and processing state sets based on Binary Decision Diagrams (BDDs). Moreover, in this paper we state a rigorous completeness criterion for sets of IPC properties [8], [9] which is a prerequisite for the proposed abstraction. Nevertheless, we believe that the proposed methodology and formalisms could also be adapted to property checking formulations other than IPC, in particular to STE and related approaches.

Abstraction in model checking is almost as old as model checking itself. The most popular abstraction techniques can be classified in being based on localization reduction [10] and predicate abstraction [11]. There is tremendous progress to integrate these abstractions into algorithms that automatically search for an appropriate abstraction such as [12], [13], [14]. Such techniques contribute substantially to increasing the scope of model checking to designs with several hundred state variables. This is often adequate for proving properties in SoC

module verification as described above. However, if designs with thousands of state variables have to be handled and chip-level properties must be proved on the RTL additional concepts for even stronger abstractions are required.

In this paper we propose to create an abstraction based on complete sets of IPC properties. This means as a starting point of our approach we assume that individual SoC modules have first been verified using IPC and a complete set of properties is available. In [15] so called *cando-objects* were proposed as abstract but still cycle-accurate design descriptions obtained from IPC properties. In contrast, the abstraction proposed here is time-abstract and is referred to as a *path predicate abstraction*. It leads to sound models for verifying both safety and liveness properties if certain restrictions on the state predicates are fulfilled. In Section II we first introduce basic notations. Section III introduces the proposed abstraction and shows that it can be used to prove safety and liveness properties which are also valid on the concrete model. Then, in Section IV we explain how this abstraction is created through the IPC methodology. In Section V we present experimental results also including an industrial case study.

II. Notations

A Kripke model is a finite state transition structure (S,I,R,A,L) with a set of states S, a set of initial states $I \subseteq S$, a transition relation $R \subseteq S \times S$, a set of atomic formulas A, and a valuation function $L : A \mapsto S$.

We consider *state predicates*, $\eta(s)$, $S(s)$, $X(s)$, $Z(s)$, $Y(s)$, that are evaluated for any concrete state s. In Kripke models derived from Moore FSMs the state variables and input variables may serve as atomic formulas. We may distinguish between the two kinds of state variables in our notation. The *original state variables* are denoted by z_i, the *input variables* of the original Moore FSM by x_j. If $S(s)$ is expressed only in terms of input state variables then the predicate describes an input *trigger condition* denoted by $X(s)$. If $S(s)$ is expressed only in terms of original state variables then we write $Z(s)$. $Y(s)$ denotes output values of the Moore machine in a state s. $T(s,s')$ is the characteristic function of the transition relation R.

An l-sequence π_l is a sequence of $l+1$ states $(s_0,s_1,\ldots,s_l)$. An l-sequence predicate $\sigma(\pi_l) = \sigma((s_0,s_1,\ldots,s_l))$ is a Boolean function characterizing a set of l-sequences; l is called the *length* of the predicate.

Note that we allow an l-sequence predicate to be applied also to a longer sequence, i.e., to an m-sequence $(s_0,s_1,\ldots,s_l,\ldots,s_m)$ with $m > l$. The predicate is then evaluated on the l-prefix $(s_0,s_1,\ldots,s_l)$ of the sequence. In case we would like to evaluate the predicate on an l-subsequence other than the prefix we need to shift the predicate in time using the *next* operator defined as follows:

$$next(\sigma_l,n)((s_0,s_1,\ldots,s_{n-1},\ s_n,s_{n+1},\ldots,s_{n+l})) := \sigma_l((s_n,s_{n+1},\ldots,s_{n+l})).$$

The $next(\sigma_l,n)$ operator shifts the starting point of the evaluation of a predicate σ_l to the n-th state in a sequence; $next(\sigma_l,n)$ is a sequence predicate of length $(n+l)$.

The usual Boolean operators $\vee$, $\wedge$, $\neg$, $\Rightarrow$ are also applicable to l-sequence predicates. If l_{max} is the largest length of all sequence predicates in a Boolean expression built with these operators, then the value of the expression is defined only for m-sequences with length $m \geq l_{max}$.

We also define a concatenation operation $\odot$ for l-sequence predicates:

$$\sigma_l \odot \sigma_k = \sigma_l \wedge next(\sigma_k,l)$$

This predicate evaluates to true for all sequences that begin with a sequence of length l characterized by σ_l and continue with a sequence of length k characterized by σ_k, where the last state in the l-sequence is the first state in the k-sequence.

A special l-sequence predicate called $any_l(\pi_l)$ is defined to evaluate to true for every sequence π_l of length l.

Together with the transition relation of the Kripke model, an l-sequence predicate becomes an *l-path predicate*:

$$P_l(\pi_l) = P_l((s_0,s_1,\ldots,s_l)) = \sigma((s_0,s_1,\ldots,s_l)) \wedge \bigwedge_{i=1}^{l} T(s_{i-1},s_i)$$

We define the general path predicate *ispath*:

$$ispath((s_0,s_1,\ldots,s_l)) = \bigwedge_{i=1}^{l} T(s_{i-1},s_i)$$

It represents an unrolling of the transition relation into l time frames and evaluates to true if the l-sequence given as its argument is a valid path in the Kripke model.

III. Path Predicate Abstraction

A. Abstract and Concrete Kripke Model

In *path predicate abstraction* we consider a concrete Kripke model (S,I,R,A,L) and an abstract Kripke model denoted by $(\hat{S},\hat{I},\hat{R},\hat{A},\hat{L})$. The two are related to each other based on a mapping of *important states* of the concrete model to abstract states and a mapping of *finite paths between important states* to abstract transitions.

Important states are identified and characterized using state predicates $\eta_i(s)$. The vector of important state predicate values, $(\eta_1(s),\eta_2(s),\ldots)$, defines an abstract state value for every concrete state s. This is the abstraction function, $\alpha(s) := (\eta_1(s),\eta_2(s),\ldots)$ mapping a concrete state to an abstract state. The set $\hat{A}$ of atomic formulas of the abstract Kripke model comprises one state variable $\hat{a}_i$ for every important state predicate $\eta_i(s)$.

Definition 1: An *important-state predicate* $\eta_i(s)$ is a predicate evaluating to *true* for a set of concrete important states s and to *false* for all other states. The disjunction of all $\eta_i(s)$ is a state predicate $\Psi(s) = \eta_1(s) \vee \eta_2(s) \vee \ldots$ characterizing the set of all important states. Finally, we require that the η_i satisfy the *important-state requirements* stated in Def. 3, below.

Definition 2: An *operational l-path* between two important states, $s_B \in S$ and $s_E \in S$, is an l-path $(s_B,s_1,\ldots,s_{l-1},s_E)$ with $l > 0$ such that $\Psi(s_B) = true$ and $\Psi(s_E) = true$ and

all intermediate states $s_1,\ldots,s_{l-1}$ are unimportant states, i.e., $\Psi(s_1)=\ldots=\Psi(s_{l-1})=\mathit{false}$.

The important state predicates cannot be chosen arbitrarily. Instead, the choice must satisfy two constraints in order to be useful for the proposed abstraction.

Definition 3: The important-state predicates are defined to fulfill the following *important-state requirements*:

1) For all pairs of (concrete) important states $s_B, s_E \in S$ between which there exists an operational l-path, there is an $l_{\max}$ such that every operational l-path between s_B and s_E is of length $l_{\max}$ or shorter: $l \leq l_{\max}$.
2) For every pair of important-state predicates, $\eta_B(s)$, $\eta_E(s)$, such that there exists a finite operational path $(\tilde{s},\ldots,\tilde{s}')$ with $\eta_B(\tilde{s})=\mathit{true}$ and $\eta_E(\tilde{s}')=\mathit{true}$ it holds that there also exists an operational path $(s,\ldots,s')$ for *every* state s satisfying $\eta_B(s)=\mathit{true}$ and *some* state s' satisfying $\eta_E(s')=\mathit{true}$.

The first constraint requires that *all cyclic paths in the concrete model intersect an important state*, i.e., there are only finite operational paths between important states. The second constraint is more difficult to understand: it ensures that abstract paths assembled from abstract transitions can always be mapped to some concrete path, i.e., there are no false abstract paths. This requirement is "automatically" fulfilled by the operation-oriented property checking technique introduced later.

Definition 4: We consider an abstraction function α such that the important-state predicates η_i fulfill the requirements of Def. 3. Then, the transition relation $\hat{R}\subseteq\hat{S}\times\hat{S}$ of the abstract model is given by:

$$\hat{R}=\{(\hat{s},\hat{s}')|\exists l:\exists(s_0,s_1,\ldots,s_l):\mathit{ispath}((s_0,s_1,\ldots,s_l))$$
$$\wedge\alpha(s_0)=\hat{s}\wedge\alpha(s_l)=\hat{s}'\wedge\neg\Psi(s_1)\wedge\ldots\wedge\neg\Psi(s_{l-1})$$

In this definition, $(s_0,s_1,\ldots,s_l)$ denotes an operational path of length l (i.e., l transitions and $l+1$ states) in the concrete Kripke model. The transition relation contains all pairs $(\hat{s},\hat{s}')$ where $\hat{s}$ and $\hat{s}'$ are head and tail of a path between important states such that all intermediate states are non-important.

Besides mapping a set of concrete states into a single abstract state as in in standard predicate abstraction, the proposed path predicate abstraction also maps a set of concrete paths into a single abstract transition. Therefore, we refer to the proposed abstraction as a "path predicate abstraction". Note, however, that such path predicate abstraction only leads to sound models since we require certain conditions on the state predicates to be fulfilled as stated in Def. 3.

B. Model Checking on the Abstract Model

In this section we show that the proposed abstraction can be used to prove CTL safety and liveness properties of the concrete model. Similar results could be obtained for other temporal logics such as LTL.

Theorem 1: Consider a formula $\hat{f}$ for the abstract model from Table I. The formula has the form $\hat{f}=<\text{CTL operator}>\hat{p}(\hat{a}_1,\hat{a}_2,\ldots)$, with $\hat{p}$ being a Boolean formula of only $\hat{a}_i\in\hat{A}$, i.e., atomic formulas of the abstract model. The corresponding formula f from Table I for the concrete model has the form $f=<\text{CTL operator}>(\Psi\wedge p)$ or the form $f=<\text{CTL operator}>(\Psi\Rightarrow p)$, where p is the Boolean formula obtained by replacing the $\hat{a}_i$ in $\hat{p}$ by their corresponding important-state predicates: $p=\hat{p}(\hat{a}_1:=\eta_1(s),\hat{a}_2:=\eta_2(s),\ldots)$.

abstract formula $\hat{f}$	concrete formula f
$\mathsf{EF}\,\hat{p}$	$\mathsf{EF}\,(\Psi\wedge p)$
$\mathsf{EG}\,\hat{p}$	$\mathsf{EG}\,(\Psi\Rightarrow p)$
$\mathsf{AF}\,\hat{p}$	$\mathsf{AF}\,(\Psi\wedge p)$
$\mathsf{AG}\,\hat{p}$	$\mathsf{AG}\,(\Psi\Rightarrow p)$

TABLE I
ABSTRACT FORMULAS VS CONCRETE FORMULAS

If and only if the formula $\hat{f}$ holds for a state $\hat{s}\in\hat{S}$ of the abstract model then the corresponding formula f from Table I holds for the corresponding concrete states, i.e., for all states $s\in S$ of the concrete model such that $\hat{s}=\alpha(s)$.

Proof: We prove the theorem for the first row of Table I. First it is proved that if $\mathsf{EF}\,\hat{p}$ holds in an abstract state then $\mathsf{EF}\,(\Psi\wedge p)$ holds in all corresponding concrete states.

If $\mathsf{EF}\,\hat{p}$ holds in a state $\hat{s}_0$ in the abstract model then there exists a finite path $(\hat{s}_0,\hat{s}_1,\ldots,\hat{s}_n)$ of n abstract transitions such that $\hat{p}$ holds in $\hat{s}_n$. For every abstract transition $(\hat{s}_i,\hat{s}_{i+1})$, according to Def. 4, there exists an operational finite l-path $(s_{i,0},s_{i,1},\ldots,s_{i,l})$ from an important concrete state $s_{i,0}$ such that $\alpha(s_{i,0})=\hat{s}_i$ to an important concrete state $s_{i,l}$ such that $\alpha(s_{i,l})=\hat{s}_{i+1}$. Then, according to requirement 2 of Def. 3, for *every* important state s_i such that $\alpha(s_i)=\hat{s}_i$ there exists an operational l-path $(s_{i,0},s_{i,1},\ldots,s_{i,l})$ from *every* important state $s_{i,0}$ such that $\alpha(s_{i,0})=\hat{s}_i$ to some important state $s_{i,l}$ such that $\alpha(s_{i,l})=\hat{s}_{i+1}$. (Note that l may be different for every path.) The same argument holds for the important states mapped to $\hat{s}_{i+1}$. Hence, there must exist a finite path $(s_{0,0},s_{0,1},\ldots,s_{1,0},s_{1,1},\ldots,s_{n,l})$ from every important state $s_{0,0}$ such that $\alpha(s_{0,0})=\hat{s}_0$ to some important state $s_{n,l}$ such that $\alpha(s_{n,l})=\hat{s}_n$. Since $\hat{p}$ holds in $\hat{s}_n$ and, therefore, $\Psi\wedge p$ holds in all important states $s_{n,l}$ mapped to $\hat{s}_n$, this means that $\mathsf{EF}\,(\Psi\wedge p)$ holds in all important states $s_{0,0}$ mapped to $\hat{s}_0$.

We now prove that if $\mathsf{EF}\,(\Psi\wedge p)$ holds in an important concrete state then $\mathsf{EF}\,\hat{p}$ holds in the corresponding abstract state. If $\mathsf{EF}\,(\Psi\wedge p)$ holds in an important concrete state s_0 then there exists a finite path $(s_0,s_1,\ldots,s_n)$ from s_0 to an important concrete state s_n such that $(\Psi\wedge p)$ holds in s_n. The path can be split up into segments $(s_i,s_{i+1},\ldots,s_{j-1},s_j)$ such that s_i and s_j are important states and the intermediate states $s_{i+1}\ldots,s_{j-1}$ are non-important states. According to Def. 4, for every such segment of the concrete path there exists an abstract transition $(\hat{s}_i,\hat{s}_j)\in\hat{R}$ such that $\alpha(s_i)=\hat{s}_i$ and $\alpha(s_j)=\hat{s}_j$. Hence, there exists an abstract path from $\hat{s}_0$ to $\hat{s}_n$. Because $(\Psi\wedge p)$ holds in the concrete state s_n the abstract property $\hat{p}$ holds in $\hat{s}_n$. Therefore, there exists an abstract path from $\hat{s}_0$ to a state where $\hat{p}$ holds, i.e., $\mathsf{EF}\,\hat{p}$ holds in $\hat{s}_0$.

The proof for the second row of Table I for EG formulas is very similar to the above proof for EF formulas. The only

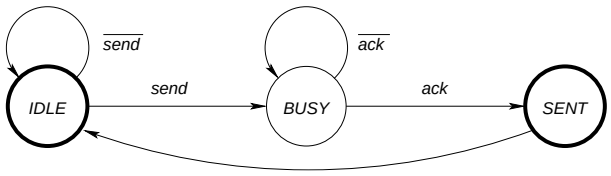

Fig. 1. Concrete FSM

difference is that in the translation from abstract properties to concrete properties, the properties are evaluated only on important states, i.e., $\hat{p}$ maps to $(\Psi \Rightarrow p)$ and vice versa. We omit this proof for reasons of space. The proof for the third row of Table I follows directly from $\mathsf{AF}\, p = \neg\mathsf{EG}\, \neg p$. Likewise, the proof for the fourth row follows from $\mathsf{AG}\, p = \neg\mathsf{EF}\, \neg p$. ■

The proposed path predicate abstraction is related to the notion of stuttering bisimulation [16]. It also decomposes infinite runs into segments of finite length that are matched segment by segment. However, we only require the important starting and ending states of the segments to be matched by the abstraction function and do not care about the intermediate state predicates. Furthermore, instead of using a theorem proving approach we use IPC to establish this weaker correlation of the models.

IV. Complete Interval Property Checking

In this section we revisit Interval Property Checking (IPC) [5], [6]. We also restate Completeness Checking for sets of IPC properties as proposed in [8], [9]. An important purpose of this paper is to show that both together can be used to create a path predicate abstraction as described in Section III.

Interval property checking is based on standard Mealy- or Moore-type finite state machine models. CTL model checking is based on Kripke models. A Moore model can be translated into a Kripke model in a straightforward way by introducing "state variables" (i.e., atomic formulas) for every input. We encode the state space of the Kripke model by the state vector $\underline{s} = (\underline{s_z}, \underline{s_x})$. The sub-state vector s_x represents the set of input values. Every state s contains in its sub-state vector s_x what combination of input values made the system transition into the state s. (This is equivalent to latching the input variables.)

In the following sections, we use the FSM of Figure 1 as a simple running example. The Moore machine stays in *IDLE* until it receives a *send* command. When the command comes it moves to state *BUSY*, sending out a request (output, not shown). In state *BUSY* it waits for an acknowledge *ack*. When the acknowledge comes it moves to state *SENT* where it signals completion to its client (output, not shown). Then it moves back to state *IDLE*.

In our IPC-based abstraction, we adopt an operational view on the design to come up with a complete set of IPC properties. An IPC operation property covers the behavior of a design moving from one *important* state to another important state within a finite time interval [6]. Operations in industrial practice typically describe one or several computational steps in a SoC module such as instructions of a processor, or processing steps of communication structures such as in transactions of a protocol. An operation (interval) property typically spans up to a few hundred clock cycles and can have up to a few million gates in its cone of influence. By unfolding the design into its operations IPC provides a functional view on the design that is orthogonal to the conventional structural view at the RT level. Industrial practice has proved that this is very effective in finding bugs.

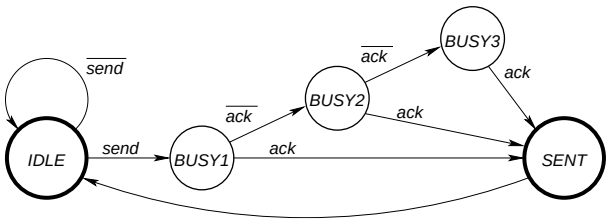

Fig. 2. Example of concrete FSM with a timing constraint on *ack*

P_1:	assume:	at t: *IDLE* ; at t: $\overline{\textit{send}}$;
	prove:	at $t+1$: *IDLE* ;
P_2:	assume:	at t: *IDLE* ; at t: *send* ; at $t+1$: *ack* ;
	prove:	at $t+2$: *SENT* ;
P_3:	assume:	at t: *IDLE* ; at t: *send* ; at $t+1$: $\overline{\textit{ack}}$; at $t+2$: *ack* ;
	prove:	at $t+3$: *SENT* ;
P_4:	assume:	at t: *IDLE* ; at t: *send* ; at $t+1$: $\overline{\textit{ack}}$; at $t+2$: $\overline{\textit{ack}}$; at $t+3$: *ack* ;
	prove:	at $t+4$: *SENT* ;
P_5:	assume:	at t: *SENT* ;
	prove:	at $t+1$: *IDLE* ;

TABLE II
Operational IPC properties

A. Operations and Important States

Consider the example of Figure 1. The verification engineer chooses *IDLE* and *SENT* to be the important states — this is indicated by bold circles. We can identify three basic operations in this design: one staying in *IDLE*, one moving from *IDLE* to *SENT* and one moving back from *SENT* to *IDLE*.

Obviously, there is an infinite path in the Moore model between states *IDLE* and *SENT* that cannot be represented in an IPC property. A technical solution is to add an input constraint to the model. In our example, we assume that *ack* is asserted at most three clock cycles after entering state *BUSY*. Note that the verification as well as the abstraction of Section IV-E are based on the validity of such a constraint. In most practical cases, however, constraints can be justified by RT-level verification of other modules of the system. Figure 2 shows the Moore model resulting from the input constraint.

Table II shows a set of five IPC properties describing all possible operations between the important states *IDLE* and *SENT* in the Moore FSM of Fig. 2. Note that IPC properties are always formulated over finite time intervals, hence the requirement 1 of Def. 3 is always fulfilled if path predicate abstraction is based on IPC.

Figure 3 shows the concrete Kripke model of our example. Since there are 5 states in the constrained Moore FSM we need (at least) 3 state variables for encoding them. The state encoding is chosen as follows: *IDLE* = 000, *BUSY1* = 100, *BUSY2* = 101, *BUSY3* = 110, *SENT* = 111. We need 2 more

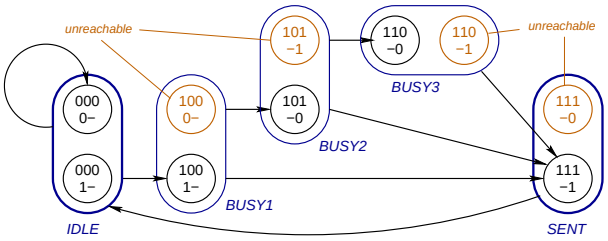

Fig. 3. Concrete Kripke model

state variables to encode the input variables *send* and *ack* . The state transition graph of the Kripke model in Figure 3 shows the 5-bit state codes inside each node.

In our examples, if we group states to abstract states this is indicated by drawing extra circles around these states. The enclosed states may themselves be abstract states. When we draw a transition edge such that it ends (or begins) at a surrounding circle we implicitly mean it to end (or begin, respectively) at every state represented by that circle.

B. Interval Property Checking

An *operation property* or *interval property* P is a pair (A_l, C_l) where both A_l (called *assumption*) and C_l (called *commitment*) are l-sequence predicates. The property checker proves that if the assumption holds on the design (given by the l-path predicate *ispath*()) the commitment does too, for all starting states s_0:

$$A((s_0,s_1,\ldots,s_l)) \wedge ispath((s_0,s_1,\ldots,s_l)) \Rightarrow C((s_0,s_1,\ldots,s_l))$$

Both sequence predicates A_l and C_l are defined over sequences of length l. The parameter l is called the *length* of the property. Since the property is implicitly checked for all possible starting states s_0 (not just the initial state of the system) it is a safety property. The implication can be rewritten in the following equivalent form:

$$ispath(\pi_l) \Rightarrow (A_l(\pi_l) \Rightarrow C_l(\pi_l))$$

where $\pi_l = (s_0,s_1,\ldots,s_l)$ is an l-sequence and $ispath(\pi_l) = \bigwedge_{i=1}^{l} T(s_{i-1},s_i)$ is the unrolling of the transition relation into l time frames. The property check can be formulated as a SAT problem that searches for a a path π_l in the Kripke model where the implication does not hold. The path π_l is then a *counterexample* of the property. It is a false counterexample if the state s_0 in the path is unreachable from the initial state.

In order to rule out unreachable counterexamples in practice, it is common to add invariants to the proof problem [6]. The strengthened proof problem looks like this:

$$(\Phi(s_0) \wedge ispath(\pi_l)) \Rightarrow (A_l(\pi_l) \Rightarrow C_l(\pi_l))$$

where $\Phi(s)$ is a state predicate characterizing an over-approximation of the reachable state set and s_0 is the head (i.e., the starting state) of the l-sequence π_l. If we re-write the implication in the following equivalent form:

$$ispath(\pi_l) \Rightarrow ((\Phi(s_0) \wedge A_l(\pi_l)) \Rightarrow C_l(\pi_l))$$

we can see that the predicate $\Phi(s_0)$ may simply be included in the assumption part of the property in order to add it to the proof.

The properties we consider in this paper have a special form. The assumption A_l of a property P is an l-sequence predicate of the form

$$A_l((s_0,s_1,\ldots,s_l)) = Z(s_0) \wedge X_l((s_0,s_1,\ldots,s_l)). \quad (1)$$

Here, $Z(s_0)$ is a state predicate characterizing an important state from which the operation starts, and $X_l(\pi_l)$ characterizes a trigger sequence for the operation. The predicate $Z(s_0)$ is expressed only in state variables of the Moore machine, i.e., it is independent of input variables.

The commitment C_l is an l-sequence predicate of the form

$$\begin{aligned} C_l((s_0,s_1,\ldots,s_l)) = Y_l((s_0,s_1,\ldots,s_l)) \wedge Z(s_l) \\ \wedge \neg\Psi(s_1) \wedge \ldots \wedge \neg\Psi(s_{l-1}) \end{aligned} \quad (2)$$

The state predicate $Z(s_l)$ characterizes the important state in which the operation ends. Again, $Z(s_l)$ refers only to state variables of the Moore machine and not to input variables. The output sequences produced in the operation are characterized by $Y_l(\pi_l)$. The state predicate $\neg\Psi(s_i)$ checks that every intermediate state s_i visited in the operation is an un-important state. This is not needed in conventional IPC but is inserted here to fulfill Def. 4.

Writing the properties in this way ensures that we only consider operational paths as defined in Def. 2. In practice, we can obtain the desired forms of Eq. 1 and 2 by following some coding conventions for writing properties, e.g., by defining appropriate macros as supported by commercial tools.

To continue our running example, the assumptions and commitments of the five properties are given by the l-sequence predicates listed below. In the commitments, the important-state predicate $\Psi(s)$ is given by $\Psi(s) = IDLE\ (s) \vee SENT\ (s)$.

$$\begin{aligned}
A_1((s_0,s_1)) &= IDLE\ (s_0) \wedge \neg send\ (s_1) \\
C_1((s_0,s_1)) &= IDLE\ (s_1) \\
A_2((s_0,s_1,s_2)) &= IDLE\ (s_0) \wedge send\ (s_1) \wedge ack\ (s_2) \\
C_2((s_0,s_1,s_2)) &= SENT\ (s_2) \wedge \neg\Psi(s_1) \\
A_3((s_0,s_1,s_2,s_3)) &= IDLE\ (s_0) \wedge \\
&\quad send\ (s_1) \wedge \neg ack\ (s_2) \wedge ack\ (s_3) \\
C_3((s_0,s_1,s_2,s_3)) &= SENT\ (s_3) \wedge \neg\Psi(s_1) \wedge \neg\Psi(s_2) \\
A_4((s_0,s_1,s_2,s_3,s_4)) &= IDLE\ (s_0) \wedge send\ (s_1) \wedge \\
&\quad \wedge \neg ack\ (s_2) \wedge \neg ack\ (s_3) \wedge ack\ (s_4) \\
C_4((s_0,s_1,s_2,s_3,s_4)) &= SENT\ (s_4) \wedge \neg\Psi(s_1) \wedge \neg\Psi(s_2) \wedge \neg\Psi(s_3) \\
A_5((s_0,s_1)) &= SENT\ (s_0) \\
C_5((s_0,s_1)) &= IDLE\ (s_1)
\end{aligned}$$

C. Property Language

In industrial practice, IPC properties can be formulated, for example, in SVA, or in ITL (*InTerval Language*), a proprietary language developed by OneSpin Solutions [5] that is well adapted to interval property checking. This language can be mapped to a subset of LTL as described in the following.

Definition 5: An *interval LTL formula* is an LTL formula that is built using only the Boolean operators $\wedge$, $\vee$, $\neg$ and the "next-state" operator X.

Let us define a generalized next-state operator X^t that denotes finite nestings of the next-state operator, i.e., if p is an interval LTL formula, then $\mathsf{X}^t(p) = \mathsf{X}\left(\mathsf{X}^{t-1}\right)$ for $t > 0$ and $\mathsf{X}^0(p) = p$.

Definition 6: An interval LTL formula is in time-normal form if the generalized next-state operator X^t is applied only to atomic formulas.

Since in LTL, $\mathsf{X}(a \vee b) = \mathsf{X}a \vee \mathsf{X}b$ and $\mathsf{X}(a \vee b) = \mathsf{X}a \vee \mathsf{X}b$ and $\neg \mathsf{X}a = \mathsf{X}\neg a$, any interval LTL formula can be translated to time-normal form. It is easy to see how an interval LTL formula can be used to specify an l-sequence predicate: The generalized next-state operator refers to the state variables of the system at the different "time" points in the sequence.

The ITL language can be used to specify interval LTL formulas and, hence, l-sequence predicates, using convenient syntax extensions. Consider the example of the property set shown in Table II. The "assume" and "prove" keywords are used to identify the assumption and commitment formulas, respectively. Each formula is a list of sub-formulas that are implicitly conjoined. A subformula is a Boolean expression over design variables, preceded by the definition of a time point using the "at" keyword. The time point "at t:" corresponds to the operator X^t as defined above.

For usability, ITL has many more syntactic extensions. For example, several sub-properties can be considered together disjunctively in a single property. In our example, properties P_2, P_3 and P_4 would result from a single "property" statement in ITL, succinctly describing the operation moving from *IDLE* to *SENT*. Also, expressions can be encapsulated for re-use and code structuring in so-called *macros*. For example, in our property set we have two state predicates, *IDLE* and *SENT* that have been formulated as ITL macros over the state variables of the design. They define the important states that will be the states of the abstract model.

D. Complete Interval Property Checking

In this section, we describe *Complete Interval Property Checking (C-IPC)* [8], [9]. It is based on a completeness criterion developed independently also by Claessen [17]. We will see that operation properties match well with this notion of completeness and that the completeness check becomes computationally tractable in combination with IPC.

The completeness criterion in [9], [8], [17] answers the question whether a set of properties fully describes the input/output behavior of a design implementation. The property suite is called *complete* if for every input sequence the property suite defines a unique output sequence that is to be produced by the implementation, according to some *determination requirements*. The basic idea presented in this section is to prove this inductively by considering chains of operation properties.

The determination requirements specify the times and circumstances when specific output signals need to be *determined* through the design. As an example: data on a bus only needs to be determined when the "data valid" signal is asserted. A determination requirement for the data signal could be written as "if (datavalid = true) then determined(data)". In general, a determination requirement is a pair (o, σ_s) for a signal o (here: data) and a guard σ_s given as an l-sequence predicate (here: datavalid) characterizing the temporal conditions when the signal o is to be determined. A signal is called *determined* by an operation at a certain time point if its value at this time point can be uniquely calculated from the start state Z of the operation, from its trigger condition X, or from other determined signals. These other determined signals can, for example, belong to the operands of a data path. If the operation performs an addition then the result signals are determined if the input operands are determined. It is checked for the reset state of the system that it fulfills all determination requirements. This is the induction base of an inductive proof.

In C-IPC the set of operation properties written by the verification engineer completely covers the state transition graph of the design's finite state machine. Any input/output sequence produced by the design, starting from reset, can be split up into a corresponding sequence of operations, each defined by one operation property. For each individual operation we can verify the functionality and we can check whether the determination requirements are fulfilled in that operation, provided the previous operation did also fulfill its own determination requirements. This is the induction step of an inductive proof.

Definition 7: A property set is complete if two arbitrary finite state machines satisfying all properties in the set are sequentially equivalent in the signals specified in the determination requirements at the time points characterized by the guards of the determination requirements. $\diamond$

Completeness of a set of $n+1$ properties $V = \{P_0, P_1, \ldots, P_n\}$, with P_0 being the reset property, is checked in the following way. Besides the determination requirements mentioned above, the user specifies a *property graph* $G = (V, E)$ where the nodes $V = \{P_i\}$ are the properties. Each property P_i is a pair (A_i, C_i) where both the assumption A_i and the commitment C_i are l-sequence predicates; l is called the length of the property P_i. Every property P has its own length l_P. The edges of the property graph describe the concatenation (sequencing) of operations. There is an edge $(P_j, P_k) \in E$ if the operation specified by P_k can take place immediately after the operation specified by P_j. (This is the case if operation P_j starts in the important state that is reached by operation P_k.)

Note that, in principle, the property graph could be determined automatically from the set of properties. However, for better debugging an incomplete and possibly incorrect property suite the user is required to specify the property graph which only involves a small extra effort.

The completeness engine performs three checks on the property graph G: a *case split test*, a *successor test* and a *determination test*, all described below. It is important to note that the completeness checks are carried out without consideration of the design.

1) Case Split Test: The case split test checks that all paths between important states in the design are described by at least one property in the property suite, i.e., that all input scenarios in an important state are covered. The set of important states is given by the commitments $\{C_i\}$ of the properties $\{P_i\}$. For every important state (given by a commitment C_P) reached in an operation P it is checked whether the disjunction of the assumptions $\{A_{Q_j}\}$ of all successor properties Q_j completely

covers the commitment C_P, i.e., for every path starting in a substate of the important state C_P there exists an operation property Q_j whose assumption A_{Q_j} describes the path. Let $\{A_{Q_1}, A_{Q_2}, \ldots\}$ be the set of assumptions, then the case split test checks if

$$C_P \odot any_{l_Q} \Rightarrow any_{l_P} \odot (A_{Q_1} \vee A_{Q_2} \vee \ldots)$$

In this expression, l_P is the length of property P and l_Q is the length of the longest successor property Q_j. The any_l sequence predicate defined in Section II is used to make both sides of the implication a sequence predicate of length $l_P + l_Q$.

If the case split test succeeds this means that for every possible input trace of the system there exists a chain of properties that is executed. However, this chain may not be uniquely determined. Therefore, the following successor test is performed.

2) Successor Test: The successor test checks whether the execution of an operation Q is completely determined by every predecessor operation P. For every predecessor/successor pair $(P,Q) \in E$ it is checked whether the assumption A_Q of property Q depends solely on inputs or on signals *determined* by the predecessor P.

The successor test creates a SAT instance that is satisfied if there exist two state sequences, π_1 and π_2, such that π_1 represents an execution of operation P followed by operation Q and the other represents an execution of operation P followed by another operation not being Q, with the additional constraint that the inputs and determined variables are the same in both sequences. The execution of P followed by Q is expressed through $(A_P \wedge C_P) \odot A_Q$, the execution of P followed by not-Q is expressed through $(A_P \wedge C_P) \odot \neg A_Q$. If the SAT check succeeds then, according to A_Q, triggering of the operation Q is decided non-deterministically. This is the case if the assumption A_Q was written such that it depends on some state variables other than inputs and variables determined by P.

What is most important for our work here is that the successor test (as a side product) makes sure that for all pairs $(P,Q) \in E$:

$$any_{l_P} \odot A_Q \Rightarrow C_P \odot any_{l_Q}.$$

The expression states that the successor operation Q always starts in an (important) state s_l that is reached by a predecessor operation P.

Having established that there exists a unique chain of operations for every input trace it remains to be shown that these operations determine the output signals as stated in the determination requirements. This is the task of the determination test.

3) Determination Test: The determination test checks whether each property Q fulfills its determination requirements provided the predecessor operation P, in turn, fulfilled its determination requirements.

The test creates a SAT instance that is satisfied if a determination requirement is violated. The satisfying set represents two state sequences, π_1 and π_2, that both represent an execution of operation P followed by operation Q, with the additional constraint that the inputs and the variables determined by P are the same in both sequences, such that π_1 and π_2 have different values for some signal that should be determined by Q.

The three completeness tests all contribute to an inductive proof. The induction is rooted at the reset, represented by the reset property P_0 that does not have a predecessor. The induction base is established through a separate *reset test* that checks whether reset can always be applied deterministically and whether reset fulfills all determination requirements.

E. Abstraction by C-IPC

It is now shown that C-IPC with a set of properties written in the form of Eq. 1 and Eq. 2 of Section IV-B leads to an abstract Kripke model that is a path-predicate abstraction of the design under verification according to Section III.

As described above, the methodology produces a set of properties, V, and a property graph $G = (V,E)$ for which the completeness tests have been successfully carried out. A basic element of the created abstraction are the *important states* given by state predicates that are used in the properties to characterize the starting states s_0 of an operation in the assumption and the ending states s_l of the operation in the commitment. The important-state predicates defining the abstraction function $\alpha(s)$ are given by the set of all important-state predicates $\{Z_i(s)\}$ appearing in the properties: $\alpha(s) := (Z_1(s), Z_2(s), \ldots)$. The abstraction function maps every concrete state of the design to an abstract state.

It must be shown that the transition relation $\hat{R}$ of the abstract Kripke model is given by the set of properties in the following way: there is a transition from one abstract state $\hat{s}$ to another one $\hat{s}'$ if and only if there exists a proven property P describing an operation that starts in the important state $\hat{s}$ and that ends in $\hat{s}'$ according to Def. 4. Moreover, the requirements for the state predicates of Def. 3 must be fulfilled.

The IPC proof engine, when proving the property P, verifies for a given pair of important states forming an abstract transition $(\hat{s}, \hat{s}')$ that there exists a corresponding operational path as given in Def. 4. It is obvious that the first requirement of Def. 3 is always fulfilled in IPC. Since every operation is proved for all concrete important states described by a state predicate $Z_i(s)$ and a trigger condition $X_i(s)$ the Kripke model will also fulfill the second requirement of Def. 3. For *all* concrete states fulfilling $Z_i(s)$ there is a path in the Kripke model to some state fulfilling the ending state condition of the operation and the trigger condition that leads into this state.

It remains to be shown that there is a property for every abstract transition fulfilling Def. 4, and for every property there is an abstract transition. This follows from the case split test and the successor test. The case split test makes sure that for every path leaving an important state in the concrete model there is a property, i.e., an abstract transition, describing that path. The successor test makes sure that properties describe only paths actually starting in an important state reached by some other property, i.e., for every abstract transition there also exists a succeeding abstract transition.

Thus, the abstraction produced by means of C-IPC fulfills all requirements as stated in Section III and is sound to prove safety and liveness properties for the concrete system.

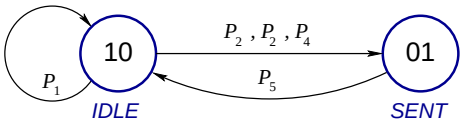

Fig. 4. Abstract Kripke model

Fig. 4 shows the abstract Kripke model of our example. The model has two important states *IDLE* and *SENT* . There is an edge between two important states if there is an IPC property describing a path between the two.

V. Experiments

Two sets of experiments were made to evaluate the usefulness of C-IPC-based abstraction. The first set is a case study on an experimental serial bus system [18]. The second set of experiments was made on a system built using Infineon's Flexible Peripheral Interconnect (FPI) bus.

In both experiments we have a set of modules communicating over a bus. *Clients* connect to the bus through *bus agents*. Each bus agent has one interface to the bus and another interface to the client. (The client could, e.g., be a CPU core or a peripheral.)

A. Serial Bus System

The communication system used in the first set of experiments is a custom-made serial bus. The protocol uses certain elements from different "real-world" serial communication protocols; for example, it uses CSMA (Carrier Sense Multiple Access) with bitwise arbitration as in CAN, and synchronization is done as in RS232 using start and stop bits.

Using C-IPC with OneSpin 360MV the bus agent was verified and a complete set of properties was obtained. The corresponding abstract state machine was manually translated into VHDL. This step will have to be automated in our ongoing work, but is here guided by a coding convention that makes the abstract states and abstract transitions obvious. Note that only the bus agents were abstracted. The clients and the interface between a client and its bus agent remained the same so that properties could be checked on the concrete and the abstract system. The clients were designed to implement a token passing mechanism among them.

Number of agents	Concrete system CPU time	Concrete system Memory	Abstract system CPU Time	Abstract system Memory
3	0.32s	78MB	0.04s	37MB
5	1.75s	158MB	0.12s	43MB
8	1min 46s	735MB	0.38s	74MB
12	54min 59s	1372MB	1.03s	109MB
15	—	—	1.89s	155MB
30	—	—	9.09s	514MB

TABLE III
IPC property checked on concrete and abstract system

Table III shows the results for checking an IPC property on different abstract system configurations using OneSpin 360MV. The design was made such that the number of bus participants can be configured by a parameter. The property checks that after reset, token passing is triggered ensuring that there is only one master in the system. Table III shows in each row the number of bus participants and the CPU time and memory consumption for checking the property on the concrete system and on the abstract system. The experiments were run on an Intel Core 2 Duo at 3GHz with 4GB main memory.

For the serial bus system, the particular strength of path predicate abstraction becomes apparent. Each individual agent in the system has 129 state variables in the concrete and 89 state variables in the abstract model. While this reduction of about 30% is not drastic the main reduction in proof complexity comes from temporal abstraction: The individual operations in the concrete model, having lengths of up to 35 cycles, are mapped to abstract single-cycle transitions. A system transaction taking more than a hundred clock cycles of serial transmission is therefore mapped to only a few transitions in the abstract model, reducing temporal length of properties by factors as low as 1/35.

Number of agents	Concrete System CPU Time	Concrete System Memory	Abstract System CPU Time	Abstract System Memory
2	10s	117MB	4s	119MB
3	26s	115MB	9s	346MB
4	1min 16s	461MB	15s	428MB
5	—	—	58s	577MB

TABLE IV
Safety property checked using induction

Table IV shows the results for checking a safety property using the induction prover built into OneSpin 360 MV. The safety property ensures that at any time there is only one master. For more than 4 agents the property cannot be proven on the concrete system, while on the abstract system it is proven in very short CPU time.

B. Industrial FPI Bus System

A more comprehensive evaluation of the proposed method using CTL model checking on the abstract model was done in an industrial case study. The Flexible Peripheral Interconnect bus (FPI bus) owned by Infineon Technologies is used for our experiments. It is an on-chip bus system similar to the industry standard AMBA. The throughput of the FPI bus is optimized by pipelining of transactions and extensive use of combinational logic. This makes it particularly interesting to examine how our approach can be used to abstract from such high-performance implementations and how a "clean" model at the transaction level can be obtained.

The FPI bus is a modular system consisting of master/slave interfaces, a BCU, an address decoder and a bus multiplexer. C-IPC was applied to obtain complete property sets for the modules. From the complete property sets we derived the abstract modules.

In our experiment we implemented our abstraction in the Cadence SMV input language. By extensive use of macros in our IPC-based verification tool (OneSpin 360 MV) the signals of the SoC modules were encapsulated and named so that a one-to-one mapping with the signals of the abstract module was obtained. The implementation of the abstraction also here was a manual step. Correctness can be ensured easily due to the one-to-one mapping between the macros created in OneSpin 360MV and the design description used for Cadence SMV. In this way, abstract modules for the master agent and the BCU were derived. For the slave agent, the address decoder and the bus multiplexer the abstract modules were not derived from C-IPC but created ad-hoc and integrated with the master agent and the BCU to form an abstract system.

	Master agent	BCU
RT code inspection, lines of code	4,000	1,500
Number of properties	17	6
Total runtime of properties	1h 19min	15s
Total runtime of completeness checks	41s	10s

TABLE V
FPI BUS MODULE VERIFICATION

Table V shows some information on the complexity of deriving the abstract modules by C-IPC. Specifically, it presents the approximate number of lines of RTL code which had to be inspected in order to create our abstract models. In general, the manual effort spent in C-IPC is about 2,000 lines of code per person month for an average verification engineer. This figure proved quite accurate also in the case study conducted here.

Based on these industrial SoC modules we assembled a system of three master agents, two slaves, the arbiter as well as bus multiplexers and address decoders. If several complete property suites are composed to completely describe a design assembled from several modules additional checks need to be applied in the completeness methodology to ensure the correctness of the integration conditions [8].

As a result of the proposed methodology the abstract model was obtained for the assembled FPI bus. While the concrete system contained 2,624 state variables only 75 state variables were included in the abstract system. We now used Cadence SMV to prove several liveness and safety properties on the abstract system. All properties are proven on the abstract model within a few minutes using less than 500 MB.

As a liveness property, we have proved that any request from a master will finish successfully within a fixed time under the constraint that a master peripheral only sends requests complying with the protocol, that the starvation prevention is switched on, and that a slave does not stay busy forever. As an example of a safety property, we prove that the bus is correctly driven at any time. Specifically, we proved that the various enable signals (data, address, ready) are one-hot-encoded. According to Theorem 1 this property holds only in the important states. By adding local properties proving that the enable signals do not change value in-between important states we obtain an unrestricted proof of the safety property that now holds for both the important and the unimportant states of the concrete model.

VI. Conclusion

In this paper we presented a methodology to leverage the results of a complete property checking methodology, C-IPC, to create abstractions for system-level verification. Our approach can be understood also as a light-weight theorem proving approach. In theorem proving, building a stack of models to prove system properties is very common. Our results show that such a paradigm is also feasible for property checking by an appropriate methodology. Future work will explore how the proposed abstraction can be integrated into a SystemC-based design and verification flow.

References

[1] D. Kroening and N. Sharygina, "Formal verification of system c by automatic hardware/software partitioning," *Formal Methods and Models for Co-Design*, 2005.

[2] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," *IEEE Transactions on VLSI Systems*, vol. 11, no. 3, pp. 345–353, 2003.

[3] A. J. Hu, J. Casas, and J. Yangpa, "Reasoning about GSTE assertion graphs," in *Proc. CHARME*. Springer, 2003, pp. 170–184, Lecture Notes in Computer Science Vol. 2860.

[4] R. Sebastiani, E. Singerman, S. Tonetta, and M. Y. Vardi, "GSTE is partitioned model checking." in *Proc. International Conference on Computer-Aided Verification (CAV)*, 2004.

[5] Onespin Solutions GmbH, Germany. OneSpin 360MV.

[6] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *IEEE Transactions on Computer-Aided Design*, vol. 27, no. 11, pp. 2068–2082, November 2008.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.

[8] J. Bormann, "Vollständige Verifikation," Dissertation, Technische Universität Kaiserslautern, 2009.

[9] J. Bormann and H. Busch, "Method for determining the quality of a set of properties," European Patent Application, Publication Number EP1764715, 09 2005.

[10] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes – The Automata-Theoretic Approach*. Princeton University Press, 1994.

[11] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *Proc. International Conference Computer Aided Verification (CAV)*, ser. LNCS, vol. 1254. London, UK: Springer-Verlag, 1997, pp. 72–83.

[12] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[13] E. Clarke, M. Talupur, H. Veith, and D. Wang, "SAT-based predicate abstraction for hardware verification," in *Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 2919. Springer, 5 2003, pp. 78–92.

[14] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word-level predicate abstraction and refinement techniques for verifying RTL Verilog," *IEEE Trans. on CAD*, vol. 27, no. 2, pp. 366–379, 2008.

[15] M. Schickel, V. Nimbler, M. Braun, and H. Eveking, "On consistency and completeness of property sets: Exploiting the property-based design process," in *Proc. Forum on Design Languages*, 2006.

[16] P. Manolios and S. K. Srinivasan, "A refinement-based compositional reasoning framework for pipelined machine verification," *IEEE Transactions on VLSI Systems*, vol. 16, pp. 353–364, 2008.

[17] K. Claessen, "A coverage analysis for safety property lists," in *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE Computer Society, 2007, pp. 139–145.

[18] H. Lu, "A case study on the verification of abstract system models derived through interval property checking," Master's thesis, University of Kaiserslautern, 2009.