Scaling Probabilistic Timing Verification of Hardware Using Abstractions in Design Source Code

Jayanand Asok Kumar, Lingyi Liu and Shobha Vasudevan
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{jasokku2, liu187, shobhav}@illinois.edu

***Abstract*—Sources of randomness such as physical process variations and input pattern variations make hardware timing a statistical measure. It is desirable to verify statistical timing properties at the higher levels of design such as the Register Transfer Level (RTL). The RTL design can be modeled as a Discrete Time Markov Chain (DTMC) and probabilistic model checking then applied to verify that the DTMC satisfies a desired timing specification. However, we find that such an approach does not scale beyond 10^{10} states. In this paper, we introduce an abstraction methodology to scale this approach to large designs. Instead of considering the entire space of data values that can be assigned to the design input variables, we perform a *value-based interval* abstraction by considering only those intervals of input values that are relevant to a given timing property. We employ *symbolic execution* on the RTL source code to automatically derive such intervals for the design inputs, with respect to a given timing property. We use these intervals to construct smaller abstract DTMCs and thereby make the corresponding probabilistic model checking problems more tractable. We show that our abstraction is sound since we do not remove any probabilistic behavior that is relevant to the property of interest. We demonstrate the effectiveness of our technique using multiple designs used in communication systems such as FFT, filters and several modules of a real world H.264 decoder. We use our technique to successfully verify timing of an H.264 module, for which the concrete model contains more that 10^{80} states, by constructing an abstract model with approximately only 10^{10} states.**

I. Introduction

Adaptive techniques like voltage and frequency scaling, process variations that affect physical device parameters [1], aging effects [2] and physical faults [3] contribute significantly to the stochastic nature of contemporary hardware. As a consequence, the timing associated with hardware computations is also statistical in nature. Recent high performance designs [4] allow long computations to make timing errors that can eventually be corrected. Therefore, contemporary semiconductor environments are increasingly interested in the question: *"What is the probability that the correct hardware output is available with a delay less than a timing specification T?"* . Such information, if available at the higher levels of design such as Register Transfer Level (RTL), would facilitate informed choices early in the hardware design cycle and avoid oversights that may prove costly in the later stages.

Traditionally, RTL verification checks functional correctness and adherence to timing specification is considered at the lower, circuit level in the design cycle. Due to the growing sources of variations in lower level hardware, it is desirable to incorporate statistical timing into the definitions of correctness at the higher, RTL. Viewing RTL designs as probabilistic entities, with non-deterministic notions of correctness opens the door to using formal verification for new sources of uncertainty like process variation and aging.

Probabilistic model checking based techniques [5] [6] [7] can be used for verifying timing properties of hardware designs in the presence of statistical variations. However, from our experience [8], we find that such an approach is limited by the capacity of the probabilistic model checking engine to less than 10^{10} states.

In this work, we present a *value-based interval abstraction* technique to mitigate the state space explosion during probabilistic model checking of RTL designs. We perform our abstraction with respect to the timing property $P[Delay < T]$. We treat RTL source code descriptions as "programs" [9]. As in the case of non-probabilistic RTL verification [10] [11], we perform our property-specific abstraction by using static analysis at the RTL design source code level (Figure 1). Abstractions performed in non-probabilistic verification produce smaller Kripke structures. As an analogue, our abstraction produces smaller DTMCs that make probabilistic model checking feasible for large RTL designs. In the abstract DTMC that we obtain, all the states of the original DTMC that are not relevant to the specified timing property are lumped together.

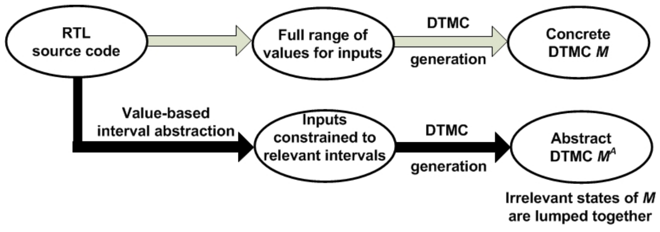

RTL source code
Full range of values for inputs
DTMC generation
Concrete DTMC M
Value-based interval abstraction
Inputs constrained to relevant intervals
DTMC generation
Abstract DTMC M^A
Irrelevant states of M are lumped together

Fig. 1. Our value-based interval abstractions are applied at design source code level, leading to smaller DTMCs.

We demonstrate the application of our technique on multiple practically useful designs that could not otherwise be verified due to space limitations. Timing cannot be compromised in microprocessor control logic. It is only in the datapath that probabilistic timing is acceptable. Datapath verification is notoriously hard [12] to get guarantees for. Our abstractions are intended to verify probabilistic timing on the datapath of RTL designs. We consider filters and FFT blocks that are widely used in communication/DSP systems, as well as an H.264 decoder. We show consistent and significant reductions in state space which make probabilistic model checking of these RTL designs feasible. For example, we are able to model check a module in the H.264 decoder for which the concrete model contains over 10^{80} states. The abstract DTMC contains approximately 10^{10} states.

In previous work [8] [13], we introduced a methodology for formally verifying the statistical timing properties of an RTL design. We statically analyzed Verilog [14] RTL source code to determine statistical correlation among the RTL signals (*i.e*, variables in RTL). We combined this statistical information with gate level delay models and represented RTL designs as finite DTMCs. We then used the probabilistic model checking tool, PRISM [15], to verify that an RTL DTMC M satisfies a statistical timing property ϕ, denoted by $M \models \phi$.

We use this framework to describe our value-based interval abstractions. We are interested in properties ϕ of the form $P[exp(V) < T]$, where exp is a real valued function that is defined over the set of RTL variables V and T is a user-specified value. $exp(V) < T$ is a *predicate* that evaluates to TRUE or FALSE in a DTMC state depending on the numeric values assigned to V in that state (for example, $Delay < T$). When we verify $M \models \phi$, we are actually computing the probability of being in a DTMC state where the predicate is TRUE. Therefore, among all possible concrete states of M, only those states where the predicate $exp(V) < T$ evaluates to TRUE are relevant. Each state of the DTMC M corresponds to a unique assignment of values to the input variables in the RTL design [8]. We restrict inputs to intervals of values (*value-based intervals*) such that only the relevant states of M are generated during DTMC construction. All the irrelevant states are lumped together to a single representative state. Lumping is a well-known abstraction approach for DTMCs [16] [17]. We show that, using this elegant abstraction, we are able to handle complex RTL designs. The complexity of our technique is not as much in the abstractions as in the process of obtaining them automatically from hardware descriptions.

Value-based intervals for RTL inputs can be used to construct an abstract DTMC M^A by lumping the irrelevant states together even at the model construction stage in the probabilistic model checker. In order to derive these intervals, we first consider the predicate $exp(V) < T$ as a symbolic constraint on the values of variables in V. We rewrite such symbolic constraints as constraints that are expressed over the input variables. We achieve this by performing *symbolic execution* [18] on the RTL source code. We use an integer constraint solver to obtain lower and upper bound values of the intervals for these inputs by maximizing (or minimizing) the value of the input for which the predicate $exp(V) < T$ is satisfied. We use these intervals while describing the model in the probabilistic model checker which then constructs the abstract DTMC M^A and checks if $M^A \models \phi$. We show that the abstract DTMC M^A is an exact reduction of the concrete DTMC M, *i.e.* $M^A \models \phi$ iff $M \models \phi$.

The value of our work is twofold. Firstly, we scale probabilistic model checking, by adapting symbolic execution techniques to hardware, and integrating them with other techniques known in software. To the best of our knowledge, we are the first to use these set of techniques in the context of probabilistic model checking for RTL designs. Secondly, we demonstrate that, using our technique, it is feasible to reason reliably about very low level physical variations.

II. Preliminaries: Probabilistic model checking of RTL designs

We now describe the framework that we use for formally representing RTL designs in order to employ probabilistic model checking. We reuse some of the model definitions from our previous work [13].

We shall use the following example RTL source code in order to illustrate the steps of our abstraction technique (Section IV).

```
always @(posedge clk)
if (sel)
  O1 <= I1 + I2;
else
  O1 <= 4*I2 + I3;
end
```

where $I1$, $I2$, $I3$ are the inputs, sel is a Boolean control variable and $O1$ is the output. All input and output variables are of 10 bits and can therefore be assigned 1024 different numeric values.

The *always @(posedge clk)* blocks can be thought of as processes that are executed in parallel at every rising edge of the hardware clock signal which is considered as a time step. At any time step t, the <= operator evaluates the *right-hand side* (RHS) value and assigns it to the *left-hand side* (LHS) variable in the next step $t + 1$. Typical data intensive RTL designs that are used in communication/DSP systems mostly perform arithmetic operations.

A. Variables in RTL designs

In RTL source code, variables are used to represent the data that is processed in hardware. A variable v can be assigned integer values in the range [l u] where l and u denote the lower and upper bound, respectively. For an N-bit variable, we assume the default range of values to be [0 2^N-1]. We refer to the probability distribution of v as the PMF of v. We define a set of variables V to be independent if all the variables in V are mutually independent. The joint PMF of an independent set V is simply a product of the individual PMFs of the variables $v \in V$.

We assume knowledge of the distribution of primary input variables and that they are independently distributed. However, our approach is not limited to designs with independent primary inputs. We also assume stationary[*] probability distributions for our inputs, and therefore for all variables in the system. Such an assumption is reasonable since statistical timing/aging analysis of hardware datapaths is often performed by considering time-invariant statistical distributions for input data [2] [20]. The PMFs of stationary variables are independent of time. Therefore, the values assigned to stationary variables in two different time steps are statistically independent of each other.

Let V be the set of all variables in a system and $I \subset V$ be the set of input variables. In order to find the PMFs of a variable $v \in V$ from the PMFs of I, we need to find the function f such that $v = f(I)$. We call f a *system function*. For a variable v, $f(I)$ is the symbolic expression that includes inputs, or the "formula" that corresponds to its evaluation. $f(I)$ may comprise Boolean or arithmetic operators that are allowed in the source code. The *support* of v, denoted by $Sup(v)$, is the set of all input variables in the expression corresponding to $f(I)$. $Sup(v)$ is a subset of I, *i.e.* $Sup(v) \subseteq I$.

The values assigned to the control variables activates/selects one among several possible paths in the RTL design. Each path may result in a different assignment to a variable. Therefore, for each path i, a system function f_i needs to be defined for each variable v that is of interest. However, $Sup(v)$ is computed by considering all possible paths. In the RTL example, there are two possible paths (sel=0 and sel=1) and $Sup(O1) = \{I1, I2, I3\}$.

B. Modeling RTL designs as DTMCs

In an extension of [9], we model both input and process variations. Since our abstraction technique is not dependent on the type of variation, we consider only input variations in this work. Therefore, the probabilistic behavior of a variable of interest, v, can be completely represented by the inputs $Sup(v)$, along with their joint PMF. We now describe the process of representing an RTL design by using a finite-state probabilistic system, namely a finite DTMC.

A DTMC can be completely specified by using a triple (S, $Trans$, μ_0) where S is the set of state variables, $Trans$ is the probabilistic state transition relation and μ_0 is the initial state. Each state μ of the DTMC corresponds to a unique assignment of values to the variables in S.

We construct the DTMC model M for a variable v, with the support of v being the state variables ($S = Sup(v)$). We define the initial state by setting the value of all state variables to 0. Each hardware clock cycle corresponds to a time step in which new values are assigned to the variables. Therefore, each such time step corresponds to a DTMC transition from one state μ to another state μ' that corresponds to the new assignment of values to $Sup(v)$.

The probability of a transition to a new state μ' is equal to the probability with which the corresponding new values of the state variables are assigned to $Sup(v)$. Since all the variables in the $Sup(v)$ are assumed to be independent, we obtain the state transition probabilities by taking the product of the individual probabilities of all variables (Section II.A). If we do not assume independence for the inputs, we would use the specified joint PMF of $Sup(v)$. All such possible state transitions labeled with the corresponding probabilities constitute $Trans$ for the DTMC M.

If a set of variables Π are of interest, we construct the corresponding DTMC M such that

$$\begin{aligned} S &= Sup(\Pi) \\ &= \bigcup_{v \in \Pi} Sup(v) \end{aligned} \qquad (1)$$

In the RTL example, $O1$ is the variable of interest. Therefore, we construct the corresponding DTMC with $Sup(O1) = \{I1, I2, I3\}$ as state variables.

C. Model checking a statistical timing property in RTL

We now describe the notion of delay in RTL presented in [8] and how we formally represent a statistical timing property.

1) Modeling delay in RTL: We consider delay in terms of RTL assignment statements. The delay of an RTL assignment depends on the operator and the values of the operands in the RHS. We consider real-valued analytical functions exp, which we call *macromodels* [8], that estimate the delay of an operator based on the value of the operands.

For each RTL operator, we derive the macromodel exp by performing extensive simulations of a gate-level implementation of the operator. We repeat this for several possible implementations of each RTL operator and construct a library of macromodels. We perform this whole macromodeling process offline for a given technology library.

In the RTL example, the delay of $I1 + I2$ can be computed by using the macromodel $exp(I1, I2)$ corresponding to the specified adder implementation. With a Ripple Carry Adder implementation, exp is a polynomial function of the number of carry bits in the addition of $I1$ and $I2$. Further details of the macromodeling process can be found in [8].

Each state in the RTL DTMC is associated with an RTL delay which can be computed based on the values of the RTL inputs (*i.e.*, state variables) in that state. We "tag" each state with the associated RTL delay which we compute by using the appropriate macromodel. Each DTMC transition represents a change in value of the RTL inputs and does not contain any information regarding the RTL delay.

2) Specifying an RTL timing property: We wish to compute the probability that the RTL delay meets a timing requirement T. The delay of an RTL block can be expressed as a combination of the macromodels of all the operators in the block [8]. Let this RTL delay be denoted by $exp(\Pi)$, which is an expression defined over a set of variables, $\Pi \subseteq V$. We define probabilistic invariants Γ [21] based on the timing requirement of the design, given by

$$\Gamma \triangleq P[exp(\Pi) < T] \ ^{\dagger} \qquad (2)$$

[*]A function of stationary variables is also stationary [19].

[†]In place of $<$, we allow for the use of other relational operators as well.

where T is a real-valued constant and $exp(\Pi) < T$ is the *predicate* that is of interest to us. P[Predicate = TRUE] denotes the probability that the predicate is satisfied (*i.e.* TRUE) by an assignment of concrete numeric values to Π. Γ is the probability of being in a state (*i.e.*, an input pattern) where the tagged delay is less than T.

We formally define probabilistic timing properties ϕ of the form,

$$\phi \triangleq \Gamma \leq p \tag{3}$$

where $p \in$[0,1] is a specification of the design. We allow logical comparison operators other than $\leq$ to be used for comparing the probabilistic invariant with p.

We are interested in computing Γ for values of Π at some time step t. For all the variables in Π, we consider the values assigned to them in the same time step. Since we assume the probabilities to be stationary, the value of t does not affect the correctness of our approach. In this paper, we omit the index t in order to simplify our notation. In this regard, our properties can be thought of stationary/steady-state properties that are not dependent on time.

We employ probabilistic model checking and verify that a DTMC M satisfies a property ϕ, denoted by $M \models \phi$. The model checking procedure for properties described in Equation 3 involves the computation of the invariant Γ and comparing it with p. If p is not specified, verifying $M \models \phi$ is equivalent to the computation of Γ. In this paper, we use PRISM [15] as the probabilistic model checking engine.

Probabilistic model checking explores all possible behaviors of the DTMC (*i.e.* all values of RTL inputs) and therefore, computes the exact probability with which the timing requirement is met.

D. Describing DTMC models in PRISM

In PRISM, we describe a DTMC M by defining the assignments to each state variable $s_v \in Sup(\Pi)$, independently. Let s_v correspond to an N-bit input variable in RTL. s_v can be assigned a value $j \in 0, 1, ..2^N - 1$ with probability p_j. We model this in PRISM by the statement,

$p_j : (s'_v = j);$

Therefore, there are 2^N statements corresponding to the description of s_v. If there are K such N-bit variables, $2^N * K$ assignment statements are required. PRISM supports assignment statements for multiple state variables. However, this approach would require 2^{NK} statements, which is inefficient.

III. Our abstraction using value-based intervals

In this section, we define and establish criteria for performing value-based interval abstractions on probabilistic systems of our interest, namely RTL designs. We perform our abstraction by statically analyzing the RTL source code. In our approach, we directly generate the abstract DTMC M^A without generating the concrete DTMC M first.

Let Λ be the predicate that is specified in the property ϕ. Let Π be the set of RTL variables over which Λ is expressed. We construct the DTMC M using $Sup(\Pi)$ as state variables.

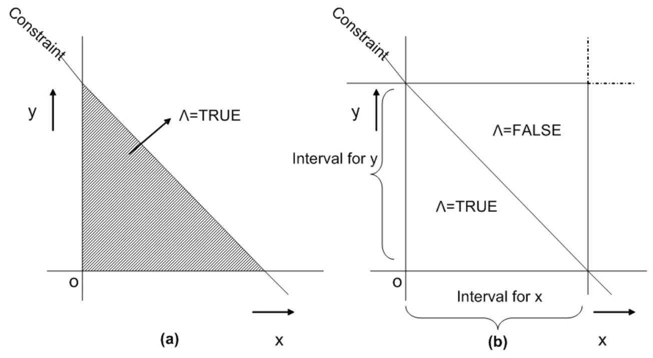


Fig. 2. a) Exact constraint for values of inputs x,y b) Conservative value-based intervals for inputs x,y

We wish to verify whether M satisfies ϕ, denoted by $M \models \phi$. In other words, we wish to compute the probability of being in a DTMC state where Λ is TRUE. This can be achieved by considering a smaller DTMC M^A that contains all the states of M where Λ = TRUE. M^A is the abstract DTMC model corresponding to the concrete DTMC M. Since each state of M corresponds to a unique assignment of concrete numeric values to the input variables $Sup(\Pi)$, the construction of M^A corresponds to retaining only those values of inputs for which Λ = TRUE. All other values of the inputs are inconsequential and can be lumped together by using a single representative value. This forms the basis of our data abstraction technique.

Λ = TRUE imposes a constraint on the values that can be assigned to the variables in Π. In order to construct an abstract DTMC M^A, we wish to use this constraint to determine the concrete values of the input variables $Sup(\Pi)$ that need to be considered. We achieve this by using RTL symbolic execution [18] to rewrite the constraint on variables Π as a constraint on inputs $Sup(\Pi)$. Symbolic execution statically explores all possible paths through the RTL design and determines a constraint C_i on the values of $Sup(\Pi)$, for each path i.

Each constraint C_i specifies an exact bound on the values of $Sup(\Pi)$ for which Λ =TRUE on path i. However, in general, C_i is specified jointly over multiple input variables in $Sup(\Pi)$. C_i cannot be included in the PRISM model description since we define assignments to input variables independently (Section II.D). Therefore, we use a constraint solver (ILP) with C_i to derive *value-based intervals* for each input variable in $Sup(\Pi)$. Since we wish to compute the probability of Λ = TRUE for all paths through the design, we construct an abstract interval ψ_{abs} for v that includes all the values from the intervals computed using each C_i.

Finally, we use the abstract intervals for each $v \in Sup(\Pi)$ in order to construct the abstract DTMC M^A. We then verify $M \models \phi$ by checking $M^A \models \phi$

For each $v \in Sup(\Pi)$, we consider all values of v such that there is a possible assignment of values to the other input variables $\in Sup(\Pi) \setminus \{v\}$ that would satisfy Λ =TRUE. Therefore, the value-based intervals that we construct are conservative (Figure 2). M^A may contain states from M in

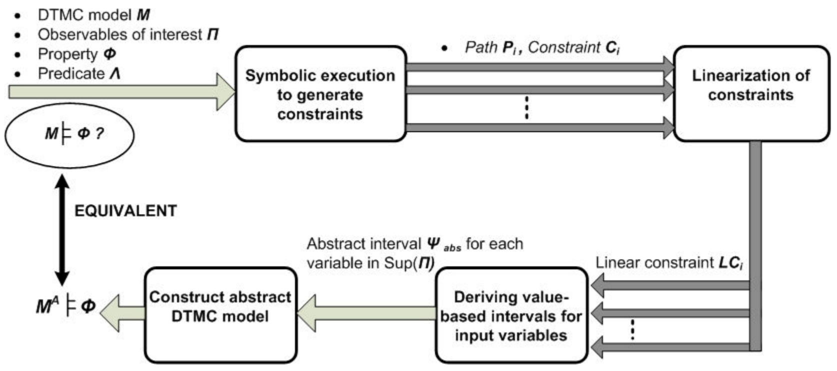

DTMC model M
Observables of interest Π
Property Φ
Predicate Λ
$M \models \Phi$?
EQUIVALENT
$M^A \models \Phi$
Symbolic execution to generate constraints
Path P_i, Constraint C_i
Linearization of constraints
Linear constraint LC_i
Deriving value-based intervals for input variables
Abstract interval Ψ_{abs} for each variable in Sup(Π)
Construct abstract DTMC model

Fig. 3. Block diagram showing the stages of our predicate-based data abstraction technique. The labels on the arrows show the outputs of each stage.

which Λ=FALSE. However, we do not discard any state from M in which Λ= TRUE. We show that our abstraction is sound with respect to the probabilistic property of interest.

Most existing abstractions for probabilistic model checking, excepting a few recent ones such as [22] [23], operate on the concrete DTMC. Instead, we perform our abstraction entirely at the RTL source code level prior to DTMC construction. Since we specify the abstracted intervals in the PRISM model description, we directly generate the abstract DTMC and circumvent the capacity issues of PRISM that are associated with generating the larger concrete DTMC.

IV. Algorithm for value-based interval abstraction

We wish to construct an abstract DTMC M^A in order to determine $P[(exp(\Pi) < T)]$, where $exp(\Pi) < T$ is the predicate of interest, Λ. Figure 3 shows the different steps in our abstraction technique. We now describe each of these steps in detail.

We shall illustrate our technique using the RTL example in Section II. Let $P[O1 < 100]$ be the invariant that we wish to compute.

A. Symbolic execution to generate constraints

We use symbolic execution to explore each possible path i in the RTL design and generate a corresponding constraint C_i on the input variables. For each path i, let v=$f_i(Sup(v))$ where f_i is the system function for variable v. Therefore, a predicate $exp(v) < T$ can be written as $exp(f_i(Sup(v))) < T$ which is a constraint C_i on the values of the input variables $Sup(v)$.

Symbolic execution refers to the execution of a single path with symbolic inputs. Symbolic execution of a path generates symbolic expressions that are a logical conjunction of the guards (conditional expression of branches) and assignments to the variables used in guards along that path. Symbolic execution is well known in software [24]. In recent work [18], symbolic execution has been introduced for RTL source code. The RTL symbolic execution engine works on the CFG and expression tree structure of each RTL "program" statement. For each single statement or conditional expression in the design, the expression tree structure exactly records the corresponding assignment or expressions and is linked to corresponding CFG node.

The RTL symbolic execution engine [18] considers exactly one path i of the CFG at any given time. At each CFG node in path i, the corresponding expression tree is traversed and output as symbolic expression. When a variable $v \in \Pi$ is encountered, the corresponding system function $f_i(Sup(v))$ is output by the engine. Every occurrence of $v \in \Pi$ in $exp(V) < T$ is substituted with the corresponding system function $f_i(Sup(v))$. We thus obtain the constraint C_i on $Sup(v)$ corresponding to the path i. We repeat this for all possible paths i in the RTL design and obtain the corresponding constraints on the input variables. Further details of the RTL symbolic execution engine, along with an optimization strategy for path exploration, can be found in [18].

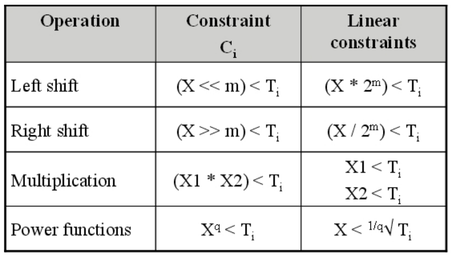
In the RTL example (Section II), we obtain the linear constraints $I1$+$I2$ <100 and $4 * I2$+$I3$ <100 corresponding to the paths sel=1 and sel=0, respectively.

B. Linearizing the constraints

A linear constraint will have *terms* on the left hand side that are separated by +/- signs. Each term can be a variable multiplied by a constant numeric value. Since datapaths of RTL designs comprise mainly of arithmetic operators, each constraint C_i is typically a linear constraint that is defined over the input variables. However, if the constraints are not linear, we transform them into a set of linear constraints.

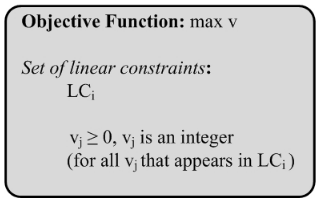
In Figure 4, we outline a set of rules for transforming non-linear operations into linear constraints. All the rules that we have defined can be extended easily for relational operators other than <. The terms $(X >> m)$ and $(X << m)$ represent shifting the variable X by m bits towards the right and left, respectively. These operations are equivalent to division and multiplication by 2^m, respectively.

If there is a term corresponding to multiplication of non-constants, we split the term into a set of linear constraints. Let $X1 * X2$ be the non-linear term in the constraint C_i. We treat $X1 * X2$ as an input variable and compute its upper bound T_i (Section IV.D). We then rewrite this term as two linear constraints $LC1$ and $LC2$, as in Figure 4.



Operation	Constraint C_i	Linear constraints
Left shift	$(X << m) < T_i$	$(X * 2^m) < T_i$
Right shift	$(X >> m) < T_i$	$(X / 2^m) < T_i$
Multiplication	$(X1 * X2) < T_i$	$X1 < T_i$ $X2 < T_i$
Power functions	$X^q < T_i$	$X < {}^{1/q}\sqrt{T_i}$

(a)



Objective Function: max v

Set of linear constraints:
LC_i

$v_j \geq 0$, v_j is an integer
(for all v_j that appears in LC_i)

(b)

Fig. 4. a) Rules for linearizing constraints, b) ILP instance for computing upper bound of v.

Concatenation of variables is supported in RTL designs. Let $X1$, $X2$ be $n1$-bit and $n2$-bit variables, respectively. The variables can be treated as strings of bits and concatenated to get a string of $n1+n2$ bits, represented by the term $\{X1, X2\}$. This algebraic operation corresponding to this term can be rewritten as the linear expression $X1 * 2^{n2} + X2$.

We apply the above rules recursively to each non-linear constraint and derive a set of linear constraints LC_1 to LC_{NumLC}, where $NumLC$ is the total number of linear constraints. The rules that we have defined in Figure 4 are not complete, since RTL designs support several other operators. However, our rules are sufficient for the large class of datapath RTL designs that are used in DSP systems.

In general, the predicate can be expressed as a polynomial function over variables Π. In such cases, we can define rules to convert non-linear terms such as $(X^q < T_i)$ into corresponding linear terms $(X < \sqrt[1/q]{T_i})$ (for $q > 1$ and non-negative X). However, in this paper, we only consider predicates that are linear functions over Π.

C. Deriving value-based intervals for input variables

We consider a linear constraint LC_i. For each input variable v that appears in the expression for LC_i, we wish to compute the interval $\psi^{(i)} = [l^{(i)}\ u^{(i)}]$ of values that can be assigned to it. We achieve this by formulating an instance of the ILP problem.

Figure 4 shows the ILP instance for computing the upper bound of v. Each ILP instance comprises one linear constraint LC_i, and a set of constraints that force all variables v_j (including v) that appear in LC_i to be non-negative integers. The objective of the the ILP problem is to maximize the integer value of v such that all the constraints are satisfied.

If the ILP instance has an optimal solution, we set $u^{(i)}$ to be equal to that solution. If the ILP instance is "unbounded", it implies that all non-negative integer values for v will satisfy the given set of constraints. In this case, we set $u^{(i)}$ to the default upper bound (*i.e.* 2^N -1, as in Section II.A) and mark v as a *free variable*. Similarly, we compute $l^{(i)}$ by changing the objective function to $min\ v$.

We perform this interval computation for all linear constraints. If a variable is marked to be free, we do not compute its intervals for any of the subsequent linear constraints.

Finally, we compute the most conservative interval for each input variable v, by computing the union of the intervals $\psi^{(i)}$ that are obtained using the linear constraints LC_i. We call this the *abstraction interval* ψ_{abs} for the variable v.

$$\psi_{abs} = \bigcup_{i=1}^{NumLC} \psi^{(i)} \tag{4}$$

In the RTL example (Section II), we compute the intervals [0 99] for both $I1$ and $I2$ based on the sel=1 path. Based on the constraint in the sel=0 path, we compute the intervals [0 24] and [0 99] for the variables $I2$ and $I3$, respectively. After computing the union of the two intervals for $I2$, we observe that ψ_{abs} for all $I1$, $I2$ and $I3$ is equal to [0 99].

If there is a "-" sign in the left hand side of the constraint, all the variables that appear in the constraint will be unbounded. For example, it is possible for each value of the variable $X1$ (and $X2$) to satisfy the constraint $X1 - X2 < 100$. However, it is possible to compute a lower bound for $X1$ if the $>$ operator is used in the constraint instead of $<$.

In this work, we have considered *unsigned* arithmetic where RTL variables are interpreted to have non-negative integer values. However, the rules in Figure 4 can be easily extended to the case of *signed* arithmetic, where negative integer values are also allowed.

D. Describing the abstract DTMC model

We use the abstraction intervals in order to describe an abstract DTMC model M^A in PRISM. For each variable $\{v : v \in Sup(\Pi)$ and v is not free$\}$, we update the assignment statements (Section II.D) in the corresponding module in PRISM. We select a non-negative integer $z \notin \psi_{abs}$. For each statement that assigns a numeric value $j \notin \psi_{abs}$ to v, we replace j with z. Therefore, z is a single value that we use to represent all numeric values of v that lie outside the abstraction interval. With the updated description, the DTMC constructed by PRISM is the abstract model M^A.

In the RTL example (Section II), we use the numeric value 100 to represent all values of $I2$ outside the interval [0 99]. Therefore, all DTMC states corresponding to values of $I2$ outside [0 99] are lumped to a single state in which $I2$ = 100.

V. Mapping from concrete to abstract DTMCs

We now describe how our abstraction maps the states and transitions of the concrete DTMC to those of the abstract DTMC. We will denote abstractions with respect to Λ by $\alpha(\cdot, \Lambda)$.

We first define the abstraction for states. For each state μ in the concrete DTMC M, $\alpha(\mu, \Lambda) = \mu^A$, where μ^A is a state in the abstract DTMC M^A. There are two possible outcomes under the abstraction:

1) μ^A=μ
   This necessarily happens if Λ = TRUE in state μ of DTMC M. This can also happen if Λ = FALSE in μ but there exists at least one state variable whose valuation in μ lies within its corresponding abstraction interval ψ_{abs}.
2) μ^A=$\mu^\ulcorner$
   This implies that Λ = FALSE in state μ of DTMC M and the values assigned to all state variables lie outside their corresponding abstraction intervals. All such states in M are mapped to a single representative state $\mu^\ulcorner$ in M^A

We now define the effect of α on the state transition probabilities of M. Let $p(\mu_1^A \to \mu_2^A)$ denote the probability of transition from state μ_1^A to state μ_2^A in M^A. We consider the following cases:

Case 1: $\mu_1^A \neq \mu^\ulcorner$ and $\mu_2^A \neq \mu^\ulcorner$

$$p(\mu_1^A \to \mu_2^A) = p(\mu_1 \to \mu_2) \tag{5}$$

where $\alpha(\mu_1, \Lambda)$ = μ_1^A, $\alpha(\mu_2, \Lambda)$ = μ_2^A

Case 2: $\mu_1^A \neq \mu^\ulcorner$ and μ_2^A = $\mu^\ulcorner$

$$p(\mu_1^A \to \mu_2^A) = \sum_{\mu_i \in M : \alpha(\mu_i, \Lambda) = \mu^\ulcorner} p(\mu_1 \to \mu_i) \tag{6}$$

where $\alpha(\mu_1, \Lambda)$ = μ_1^A.

Case 3: μ_1^A=$\mu^\ulcorner$ and $\mu_2^A \neq \mu^\ulcorner$

$$p(\mu_1^A \to \mu_2^A) = p(\mu_i \to \mu_2) \tag{7}$$

where $\alpha(\mu_2, \Lambda)$ = μ_2^A and μ_i is any state in M. Since the probability distributions for all input variables are stationary (Section II.A), the probability of reaching any state $\mu_2 \in M$ is independent of the previous state. Therefore, the exact identity of state μ_i in Equation 7 is irrelevant and our abstraction does not remove any relevant probabilistic behavior. The same applies for Equation 5 as well.

Proof of correctness:
We now present a brief proof intuition for the soundness of our technique. We wish to prove that $M \models \phi$ is equivalent to $M^A \models \phi$. We achieve this by using the Strong Lumping Theorem [16] [17] to show that M^A is a probabilistic bisimulation [25] of M with respect to ϕ.

α can be thought of as an equivalence relation between the states in M and M^A. α relates a state μ in M to the state $\mu^A = \alpha(\mu, \Lambda)$ in M^A. By construction, α also preserves the valuation of Λ. Therefore, μ^A is locally equivalent to μ with regard to Λ. In fact, all states μ^A in M^A can be viewed as equivalence classes of M under the relation α. With the exception of $\mu^\ulcorner$, all equivalence classes μ^A contain only one state.

The abstract model M^A can be thought of as a quotient DTMC that comprises of equivalence classes defined by α. Equations 5, 6 and 7 can then be used to invoke the Strong Lumping Theorem and prove that M^A is a probabilistic bisimulation of M.

VI. Experimental Results

We implement the RTL symbolic execution algorithm using C++. We perform all our experiments on an Intel i5 2.67GHz quad-core machine with 16GB of memory. We use *lpsolve* [26], an open source ILP solver, in order to solve the set of integer linear constraints and derive the value-based intervals for the inputs.

We demonstrate the effectiveness of our methodology by applying it on two sets of data-intensive RTL designs. The first set of designs comprise `fir`, `elliptic` and `fft8` all of which are high-level synthesis benchmarks [27] that are commonly used in communication/DSP systems. Filter coefficients are fixed and stored in a ROM table. We consider constant values for these coefficients. `Inter_pred_LPE`, `Inter_pred_pipeline` and `Inter_pred_sliding_window` are different modules from a real-world H.264 decoder[‡] and constitute our second set of designs. In this work, we analyze each of the H.264 modules independently.

In Table I, *Number of paths* represents the total number of paths that needs to be explored during symbolic execution. This number is with regard to the variables which appear in the predicate (described in Table II) of the specified property. Since our designs do not contain multiplication of variables with each other, there should be exactly one linear constraint per path (Section IV.B). However, in some paths, all variables are assigned a constant value and therefore, the predicate is vacuously TRUE or FALSE. We discard these paths and consider only the linear constraints (*Number of constraints*) corresponding to the remaining paths while formulating the ILP instances.

In Table I, *Number of inputs* represents the total number of input variables (and their bitwidths) on which the variables in the predicate depend, *i.e.* $Sup(\Pi)$. Therefore, each of these input variables appear in at least one of the linear constraints. However, in each linear constraint, at most a small subset of these input variables are present. Therefore, each ILP instance is small and the corresponding runtime of *lpsolve* is also negligibly small. The total abstraction time, which includes the time for both the generation of linear constraints and the ILP solver, is less than 10 seconds in all our experiments.

For each of the designs that we consider, we specify a property that is defined over some internal data variables. Table II provides a description of all the predicates that we define in order to specify the properties of interest. `fir`

[‡]www.opencores.org

TABLE I
SIZES OF THE ILP INSTANCES THAT WE USE TO DERIVE THE INTERVALS FOR INPUT VARIABLES.

Design name	Predicate name	Number of inputs	Number of paths	Number of constraints	Abstraction time
`fir`	p8	6 (8-bit)	1	1	<10s
`elliptic`	p9	12 (8-bit)	1	1	<10s
`fft8`	p10	8 (16-bit)	4	4	<10s
`Inter_pred_LPE`	p1	5 (8-bit)	180	131	<10s
`Inter_pred_LPE`	p2	5 (8-bit)	180	131	<10s
`Inter_pred_LPE`	p3	5 (8-bit)	180	131	<10s
`Inter_pred_pipeline`	p4	32 (8-bit)	1936	1932	<10s
`Inter_pred_pipeline`	p5	3 (8-bit)	8	5	<10s
`Inter_pred_sliding_window`	p6	19 (8-bit)	29	21	<10s
`Inter_pred_sliding_window`	p7	16 (8-bit)	29	21	<10s

TABLE II
DESCRIPTION OF THE PREDICATES THAT WE USE TO SPECIFY PROPERTIES OF OUR INTEREST. TO VERIFY THESE PROPERTIES, WE COMPUTE P[PREDICATE = TRUE].

Predicate name	Predicate description
p1	bilinear0_A + bilinear0_B < 8
p2	bilinear0_A + bilinear0_B < 6
p3	bilinear0_A + bilinear0_B < 4
p4	8*Inter_blk_mvx + Inter_blk_mvy < 2
p5	Inter_pred_out0 < 200
p6	Inter_pix_copy0 < 2
p7	Inter_H_window_0_0 < 3
p8	y<30
p9	outp < 30
p10	s3r < 127

and `elliptic` are filter designs in which it is common to check whether the output is less than a user-defined threshold. Therefore, we define the predicates p8 and p9 over the output variables y and $outp$, respectively. Although not exact models, these predicates can be viewed as being representative of certain timing properties of the design. For example, y can be an input to an adder block (Section II.C.1) for which the timing constraint requires that $y < 30$, as in p8. For our experiments, we consider predicates that are linear functions over a set of RTL variables and we use the "<" relational operator. For each of the H.264 modules, we consider multiple predicates.

Table III demonstrates the reduction in state-space provided by our abstraction method. PRISM runs out of memory while trying to construct any of the concrete DTMC models and therefore, these designs can not be model checked. We estimate the number of states in the concrete model based on the total number of combinations of values that can be assigned to the corresponding input variables. There is no reason to believe that the RTL inputs, which are data variables, are restricted and we use their full range of values to estimate the concrete state-space. In all the designs, with the exception of `fft8`, we are able to obtain significant reductions in state space by using our abstraction technique and PRISM successfully constructs the corresponding abstract DTMCs. We approximately represent the number of states in the abstract DTMC model as powers of 2, in order to facilitate comparison with the concrete number of states. Model checking of the smaller abstract DTMCs by PRISM requires only a few seconds.

p1, p2 and p3 are all the same predicate that differ only in the constraint values that are specified in the RHS. We observe that as the constraint values get smaller, the number of relevant data values (and hence states) also decrease. Our technique is extremely effective when the predicate is TRUE for only a small fraction of the possible data values. Although our technique would still be sound for larger constraint values, the reduction that we achieve may be far more modest.

Since model checking could not be completed for the concrete DTMCs in Table III, we do not present a comparison of the model checking results for these designs. Instead, as a proof of concept, we construct smaller versions (smaller bitwidth for inputs) of `fir` and `elliptic` and verify that the results computed using the concrete DTMC and the abstract DTMC are exactly the same (Table IV). We choose the smaller bitwidths such that PRISM model checks the concrete DTMC. For example, we consider 3-bit data for `fir`(small) and the runtime is <10s. We do not consider a smaller `fft8` since our abstraction does not provide any reductions for it (Table III).

In `fft8`, we are not able to demonstrate any reduction in state-space using our abstraction. This is due to "-" operator in the RTL design. As described in Section IV.B, a "-" sign on the left hand side of a "less than" constraint will result in unconstrained values for all the input variables. `JPEG encoder` is another design for which we cannot obtain reductions. The module of the encoder design that we consider is control-intensive and therefore, the number of paths that need to be explored by the symbolic execution algorithm is huge (Section IV.A). For this design, we stopped the symbolic execution engine after 1hr of exploring paths and generating the corresponding constraints. We could not use this incomplete set of constraints since all possible paths in the design need to be considered in order to guarantee correctness of our abstraction.

In all the designs mentioned above, we find that the control paths are independent of the values of data. This is fairly common for a large class of data-intensive designs that are

TABLE III
REDUCTIONS IN NUMBER OF STATES THAT WE ACHIEVE BY USING OUR ABSTRACTION

		Concrete DTMC		Abstract DTMC	
Design name	Predicate name	Number of states	Model checking time	Number of states	Model checking time
`fir`	p8	2^{56}	*Out of memory*	2^{28}	<2s
`elliptic`	p9	2^{96}	*Out of memory*	$\approx 2^{29.73}$	<2s
`fft8`	p10	2^{16}	*Out of memory*	2^{16}	*Out of memory*
`Inter_pred_LPE`	p1	2^{40}	*Out of memory*	$\approx 2^{15.85}$	<2s
`Inter_pred_LPE`	p2	2^{40}	*Out of memory*	$\approx 2^{14.04}$	<2s
`Inter_pred_LPE`	p3	2^{40}	*Out of memory*	$\approx 2^{11.61}$	<2s
`Inter_pred_pipeline`	p4	2^{256}	*Out of memory*	2^{32}	<2s
`Inter_pred_pipeline`	p5	2^{24}	*Out of memory*	$\approx 2^{22.95}$	<2s
`Inter_pred_sliding_window`	p6	2^{152}	*Out of memory*	$\approx 2^{30.11}$	<2s
`Inter_pred_sliding_window`	p7	2^{128}	*Out of memory*	2^{32}	<2s

TABLE IV
DEMONSTRATING CORRECTNESS OF OUR ABSTRACTIONS USING SMALLER, CONTRIVED VERSIONS OF BENCHMARKS DESIGNS SINCE THE CONCRETE DTMCS CANNOT BE CONSTRUCTED FOR THE ACTUAL SIZES.

		Concrete DTMC		Abstract DTMC	
Design name	Predicate	Number of states	P[Predicate = TRUE] (PRISM result)	Number of states	P[Predicate = TRUE] (PRISM result)
`fir` (small)	$(y < 12)$	2^{24}	4.6539x10^{-4}	$\approx 2^{15.57}$	4.6539x10^{-4}
`elliptic` (small)	$(outp < 30)$	2^{30}	9.6485x10^{-7}	$\approx 2^{14.39}$	9.6485x10^{-7}

commonly used in DSP systems. For example, typical control variables that we observe are *counters* that are not data-dependent. Since control variables control the selection of paths and since we wish to consider all possible paths, we cannot constrain the values of such variables. In non-DSP designs, the control variables may depend on input data variables and therefore, all such input variables must also be unconstrained. In these cases, the overall reduction achieved by our abstraction technique may not be very large.

In RTL designs, it is possible that arithmetic operations can result in an overflow (or underflow) due to insufficient number of bits that are assigned to store the results. Ideally, such incorrect computations should not be allowed. Our technique cannot detect such overflow errors. Typically, overflows are prevented in DSP designs by assigning sufficient number of bits to the different variables in the design. Our abstraction techniques can be applied on such designs.

VII. RELATED WORK AND CONCLUSION

In the realm of software verification, there exist several techniques [28] [29] for predicate abstraction. Properties regarding program correctness/safety can be expressed using a set of predicates, that are either specified or automatically inferred. These predicates can be used to abstract a program and convert it into a Boolean program on which the properties can be easily verified. More generally, *abstract interpretation* [30] is the theory of reasoning with the approximate semantics of a large program rather than the set of all possible concrete behaviors. However, unlike predicate abstraction, all such abstractions are not necessarily property-specific. In all these abstractions, the concrete numeric values of data can either be completely abstracted out of the program or can be restricted to finite intervals [31].

Data abstraction techniques have been applied even in the context of hardware verification [10]. These techniques employ predicate abstraction in order to focus on the verification of Boolean control logic for which the exact numeric values of datapath variables are inconsequential. In [11], RTL designs are verified by restricting data values to intervals that are imposed by the execution of the RTL program. Therefore, these intervals are not property-specific.

Abstraction techniques have been employed in the context of probabilistic systems as well [32] [33] [22] [23]. In [23], the abstraction is performed on the source code itself. However, this technique is intended for probabilistic software and cannot be extended to RTL designs. In [22], the authors present a predicate-based abstraction for Markov Decision Processes (MDPs). They employ an SMT solver in order to implement this abstraction at the level of the PRISM language itself. However, this implementation is very inefficient for the bulky PRISM descriptions that are used for RTL designs (Section II.D).

In conclusion, we have presented a property-specific value-based interval abstraction technique that is applied at the source code level. We intend our abstraction for scaling probabilistic model checking of hardware designs. Widespread adoption of formal verification is feasible only if it remains relevant and practicable in critical, emerging areas of need like variation-aware timing verification. Our work represents a strategic step in this direction.

References

[1] K. A. Bowman, M. Orshansky, and S. S. Sapatnekar, "Tutorial ii: Variability and its impact on design," in *Proc. of ISQED'06*, 2006, p. 5.

[2] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar, "NBTI-aware synthesis of digital circuits," in *Proc. of DAC'07*, 2007, pp. 370–375.

[3] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in *Proc. of DSN'04*, 2004, p. 61.

[4] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. of MICRO'03*, Dec. 2003.

[5] M. Z. Kwiatkowska, G. Norman, and R. Segala, "Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM," in *Proc. of CAV'01*, 2001, pp. 194–206.

[6] M. Kwiatkowska, G. Norman, and D. Parker, "Symmetry Reduction for Probabilistic Model Checking," in *Proc. of CAV'06*, 2006, pp. 234–248.

[7] J.-P. Katoen, "Advances in Probabilistic Model Checking," in *Proc. of VMCAI'10*, 2010, p. 25.

[8] J. A. Kumar and S. Vasudevan, "Variation-Conscious Formal Timing Verification in RTL," in *Proc. of VLSI Design'11*, 2011, extended version available at http://users.crhc.illinois.edu/jasokku2/docs/TCAD_final.pdf.

[9] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing of hardware description languages," in *Proc. of CHARME'99*, 1999, pp. 298–312.

[10] E. Clarke, O. Grumberg, and et al., "High level verification of control intensive systems using predicate abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1512–1542, 2003.

[11] V. P. Nazanin, N. Mansouri, and R. Vemuri, "Automatic Data Path Abstraction for Verification of Large Scale Designs," in *Proc. of ICCD'98*, 1998, pp. 192–194.

[12] P. Johannsen, "BOOSTER: Speeding Up RTL Property Checking of Digital Designs by Word-Level Abstraction," in *Proc. of CAV'01*, 2001, pp. 373–377.

[13] J. A. Kumar and S. Vasudevan, "Automatic compositional reasoning for probabilistic model checking of hardware designs," in *Proc. of QEST'10*, 2010, pp. 143–152.

[14] "Verilog Reference Manual," http://eesun.free.fr/DOC/VERILOG/verilog_manual1.html.

[15] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 2.0: A tool for probabilistic model checking," in *Proc. of QEST'04*, 2004, pp. 322–323.

[16] J. Kemeny and J. Snell, *Finite Markov chains*, repr ed., ser. University series in undergraduate mathematics. New York: VanNostrand, 1969.

[17] S. Derisavi, H. Hermanns, and W. H. Sanders, "Optimal state-space lumping in Markov chains," *Information Processing Letters*, vol. 87, no. 6, pp. 309 – 315, 2003.

[18] L. Liu and S. Vasudevan, "Efficient Validation Input Generation in RTL by Hybridized Source Code Analysis," in *Proc. of DATE'11*.

[19] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, 2005.

[20] L. Wan and D. Chen, "Dynatune: circuit-level optimization for timing speculation considering dynamic path behavior," in *Proc. of ICCAD'09*, 2009.

[21] T. S. Hoang, Z. Jin, K. Robinson, A. McIver, and C. Morgan, "Probabilistic invariants for probabilistic machines," in *Proc. of ZB'03*. Springer, 2003, pp. 240–259.

[22] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "Game-Based Probabilistic Predicate Abstraction in PRISM," *ENTCS*, vol. 220, no. 3, pp. 5–21, 2008.

[23] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "Abstraction Refinement for Probabilistic Software," in *Proc. of VMCAI'09*, 2009, pp. 182–197.

[24] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[25] K. G. Larsen and A. Skou, "Bisimulation through probabilistic testing," *Information and Computation*, vol. 94, no. 1, pp. 1–28, 1991.

[26] M. Berkelaar, K. Eikland, and P. Notebaert, "lp_solve 5.5, open source (mixed-integer) linear programming system," Software, May 1 2004, available at http://lpsolve.sourceforge.net/5.5/. Last accessed Dec, 18 2009. [Online]. Available: http://lpsolve.sourceforge.net/5.5/

[27] P. R. Panda and N. D. Dutt, "1995 high level synthesis design repository," in *ISSS*, 1995, pp. 170–174, available at http://ftp.ics.uci.edu/pub/hlsynth/.

[28] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside microsoft," in *Proc. of IFM'04*, 2004, pp. 1–20.

[29] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," *SIGPLAN Not.*, vol. 39, no. 1, pp. 232–244, 2004.

[30] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points," in *Proc. of POPL'77*. ACM, 1977, pp. 238–252.

[31] D. Monniaux, "A minimalistic look at widening operators," *Higher Order Symbol. Comput.*, vol. 22, no. 2, pp. 145–154, 2009.

[32] L. D. Alfaro and P. Roy, "Magnifying-Lens Abstraction for Markov Decision Processes," in *Proc. of CAV'07*, 2007, pp. 325–338.

[33] P. R. D'argenio, H. E. Jensen, and K. G. Larsen, "Reachability analysis of probabilistic systems by successive refinements," in *Proc. of PAPM/PROBMIV'01*, 2001, pp. 39–56.