A Symbolic Approach to Predicate Abstraction[*]

Shuvendu K. Lahiri[1], Randal E. Bryant[1], and Byron Cook[2]

[1] Carnegie Mellon University, Pittsburgh, PA
shuvendu@ece.cmu.edu,Randy.Bryant@cs.cmu.edu
[2] Microsoft Corporation, Redmond, WA
bycook@microsoft.com

Abstract. Predicate abstraction is a useful form of abstraction for the verification of transition systems with large or infinite state spaces. One of the main bottlenecks of this approach is the extremely large number of decision procedures calls that are required to construct the abstract state space. In this paper we propose the use of a symbolic decision procedure and its application for predicate abstraction. The advantage of the approach is that it reduces the number of calls to the decision procedure exponentially and also provides for reducing the re-computations inherent in the current approaches. We provide two implementations of the symbolic decision procedure: one based on BDDs which leverages the current advances in early quantification algorithms, and the other based on SAT-solvers. We also demonstrate our approach with quantified predicates for verifying parameterized systems. We illustrate the effectiveness of this approach on benchmarks from the verification of microprocessors, communication protocols, parameterized systems, and Microsoft Windows device drivers.

1 Introduction

Abstraction is crucial in the verification of systems that have large data values, memories, and parameterized processes. These systems include microprocessors with large data values and memories, parameterized cache coherence protocols, and software programs with arbitrarily large values. *Predicate abstraction*, first proposed by Graf and Saidi [15] as a special case of the general framework of *abstract interpretation* [10], has been used in the verification of protocols [15, 22], parameterized systems [11, 12] and software programs [1, 13]. In predicate abstraction, a finite set of predicates is defined over the concrete set of states. These predicates are used to construct a finite state abstraction of the concrete system. The automation in generating the finite abstract model makes this scheme attractive in combining deductive and algorithmic approaches for infinite state verification.

One of the main problems in predicate abstraction is that it typically makes a large number of theorem prover calls when computing the abstract transition relation or the abstract state space. Most of the current methods, including Saidi and Shankar [22], Das, Dill and Park [12], Ball et al. [1], Flanagan and

[*] This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029.001.

Qadeer [13] require a number of validity checks that can be exponential in the number of predicates.

A number of tools [1, 15] address this problem by only approximating the abstract state space — resulting in a weaker abstract transition relation. Das and Dill [11] have proposed *refining* the abstract transition relation based on counterexamples, starting with an initial transition relation. This technique can work well — particularly for systems with sparse abstract state graphs. However, this technique may still require a potentially exponential number of calls to a decision procedure. Namjoshi and Kurshan [20] have proposed an alternative technique by syntactically transforming a concrete system to an abstract system. Instead of using a theorem prover, they propose the use of syntactic strategies to eliminate first-order quantifiers. However, the paper does not report empirical results to demonstrate the effectiveness of the approach.

In this work, we present a technique to perform predicate abstraction that reduces the number of calls to decision procedures exponentially. We formulate a symbolic representation of the predicate abstraction step, reduce it to a quantified Boolean formula and then use Boolean techniques (based on Binary Decision Diagrams (BDDs) [4] and Boolean Satisfiability solvers (SAT)) to generate a symbolic representation of the abstract transition relation or abstract state space. Our work is motivated by the recent work in UCLID [6, 24], which transforms quantifier-free first-order formulas into Boolean formulas and uses Boolean techniques to solve the resulting problems.

The advantage of our approach is three-fold: First, we can leverage efficient Boolean quantification algorithms [14] based on BDDs and recent advances in SAT-based techniques for quantification [9, 18]. Second, the single call to the symbolic decision procedure eliminates the overhead of multiple (potentially exponential) calls to decision procedures (e.g., initializing data structures, libraries, system calls in certain cases). Third, in previous work, the decision procedures cannot exploit the *pruning* across different calls and result in a lot of re-computation. The *learning* and *pruning* mechanisms in modern SAT solvers allow us to prevent the re-computations.

Our work considers both quantifier-free and quantified predicates. The quantified predicates are used in the verification of parameterized systems and systems with unbounded resources. Experimental results indicate that our method outperforms previous methods by orders of magnitude on a large set of examples drawn from the verification of microprocessors, communication protocols, parameterized systems and Microsoft Windows device drivers.

The paper is organized as follows. In Section 2, we provide some background on predicate abstraction. Section 3 describes how predicate abstraction can be implemented using our symbolic decision procedure. We also describe several ways of implementing the decision procedure. Section 4 describes the handling of universally quantified predicates for verifying parameterized systems. Section 5 presents the results of our experiments.

2 Background

Fig. 1 displays the syntax of the Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU), a fragment of first-order logic extended with equality, inequality, and counters. An *expression* in CLU can evaluate to truth values (*bool-expr*), integers (*int-expr*), functions (*function-expr*) or predicates (*predicate-expr*). The logic can be used to describe systems in the tool UCLID [6].

$$\begin{array}{rl}
\textit{bool-expr} ::= & \textbf{true} \mid \textbf{false} \mid \textit{bool-symbol} \\
& \mid \neg \textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \\
& \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\
& \mid \textit{predicate-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{int-expr} ::= & \textit{int-var} \mid \textit{int-symbol} \\
& \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
& \mid \textit{int-expr} + \textit{int-constant} \\
& \mid \textit{function-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{predicate-expr} ::= & \textit{predicate-symbol} \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \textit{bool-expr} \\
\textit{function-expr} ::= & \textit{function-symbol} \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \textit{int-expr}
\end{array}$$

Fig. 1. CLU Expression Syntax. Expressions can denote computations of Boolean values, integers, or functions yielding Boolean values or integers.

A system description is a four-tuple[1] $(\mathcal{S}, \mathcal{K}, \delta, q^0)$ where:

- $\mathcal{S}$ is a set of symbols denoting the state elements.
- $\mathcal{K}$ is a set of symbols denoting the parameters of the system.
- δ represents the transition function for each state element in $\mathcal{S}$, as CLU expressions over $\mathcal{S} \cup \mathcal{K}$.
- q^0 represents the set of initial state expressions for each state element in $\mathcal{S}$, as CLU expressions over $\mathcal{K}$.

State variables can be integers, Booleans, functions over integers or predicates over integers. The functions and predicates represent unbounded mutable arrays of integers and Booleans, respectively. The symbols in $\mathcal{K}$ are the parameters for the system, and can be used to denote the size of a buffer in the system, the functionality of an ALU in a processor, or the number of processes in a protocol. They can also specify a set of initial states of the system. The logic has been used to model and verify out-of-order processors with unbounded resources and parameterized cache-coherence protocols [6, 17].

For a set of symbols $\mathcal{U}$, an interpretation $\sigma_{\mathcal{U}}$ assigns type-consistent values to each of the symbols in the set $\mathcal{U}$. For any expression Ψ over $\mathcal{U}$, $\langle \Psi \rangle_{\sigma_{\mathcal{U}}}$ denotes the evaluation of the expression under $\sigma_{\mathcal{U}}$.

[1] We can encode inputs to the system at each step by a *generator of arbitrary values* as described in previous work by Velev and Bryant [25].

If q denotes a set of expressions (one for each state element), then $\langle q \rangle_{\sigma_{\mathcal{K}}}$ applies $\sigma_{\mathcal{K}}$ point-wise to each expression of q. If Ψ represents an expression, then $\Psi\,[Y/\mathcal{U}]$ *substitutes* the expressions in Y point-wise for the symbols in $\mathcal{U}$.

A state s of the concrete system is an interpretation to the symbols in $\mathcal{S}$. Given an interpretation $\sigma_{\mathcal{K}}$ to the symbols in $\mathcal{K}$, the initial state of the concrete system is given as $\langle q^0 \rangle_{\sigma_{\mathcal{K}}}$, and an *execution sequence* of the concrete system is given by $\langle q^0 \rangle_{\sigma_{\mathcal{K}}}, \ldots, \langle q^i \rangle_{\sigma_{\mathcal{K}}}, \langle q^{i+1} \rangle_{\sigma_{\mathcal{K}}}, \ldots$, where $q^{i+1} = \delta\left[q^i/\mathcal{S}\right]$. A concrete state s is *reachable* if it appears in an execution sequence for any interpretation $\sigma_{\mathcal{K}}$.

For the rest of the paper, we will represent a set of states as either a set or by a Boolean expression over $\mathcal{S} \cup \mathcal{K}$.

2.1 Predicate Abstraction

The predicate abstraction algorithm generates a finite state abstraction from a large or infinite state system. The finite model can then be used in the place of the original system when performing model checking or deriving invariants. Let $\Phi \doteq \{\phi_1, \ldots, \phi_k\}$ be the set of predicates which are used for inducing the abstract state space. Each of these predicates is a Boolean expression over the set of symbols in $\mathcal{S} \cup \mathcal{K}$. Let $\mathcal{B} \doteq \{b_1, \ldots, b_k\}$ denote the Boolean variables in the abstract system such that value of b_i denotes the evaluation of the predicate ϕ_i over the concrete state. An abstract state s_a is an interpretation to the variables in $\mathcal{B}$. The *abstraction* and the *concretization* functions α and γ are defined over sets of concrete and abstract states respectively.

If Ψ_c describes a set of concrete states (as an expression over $\mathcal{S} \cup \mathcal{K}$), then:

$$\alpha(\Psi_c) \doteq \{s_a \mid \exists s_c \in \Psi_c \; \exists \sigma_{\mathcal{K}} \; s.t. \bigwedge_{i \in 1,\ldots,k} \langle b_i \rangle_{s_a} \Leftrightarrow \langle \phi_i \rangle_{s_c \cup \sigma_{\mathcal{K}}}\}$$

If Ψ_a represents a set of abstract states, then

$$\gamma(\Psi_a) \doteq \{s_c \mid \alpha(\{s_c\}) \in \Psi_a\}$$

In fact, the concretization function γ is implemented as the substitution, $\gamma(\Psi_a) \doteq \Psi_a\,[\Phi/\mathcal{B}]$. For each predicate ϕ_i, let ϕ'_i represent the result of substituting the next-state expression for each symbol in $\mathcal{S}$. That is, $\phi'_i \doteq \phi_i\,[\delta(\mathcal{S})/\mathcal{S}]$. Let $\Phi' \doteq \{\phi'_1, \ldots, \phi'_k\}$.

Let us now define a *cube over* $\mathcal{B}$, $c_{\mathcal{B}}$, to be a clause $l_{i_1} \wedge \ldots \wedge l_{i_m}$, where each l_{i_j} is a *literal* (either b_{i_j} or $\neg b_{i_j}$, where $b_{i_j} \in \mathcal{B}$). A *cube over* $\mathcal{B}$ is *complete* if all the symbols in $\mathcal{B}$ are present as literals in the cube. A complete cube over $\mathcal{B}$ corresponds to an abstract state.

Now, we define the computation of the abstract state space using the current methods [12, 22]. To obtain the set of abstract initial states, $\Psi^0_{\mathcal{B}}$, the cubes over $\mathcal{B}$ are enumerated and a complete cube $c_{\mathcal{B}}$ is included in $\Psi^0_{\mathcal{B}}$ iff the expression $\gamma(c_{\mathcal{B}})\left[q^0/\mathcal{S}\right]$ is satisfiable. The set of reachable abstract states are computed by performing a fixpoint computation, starting from $\Psi^0_{\mathcal{B}}$ and computing $\Psi^1_{\mathcal{B}}, \ldots, \Psi^r_{\mathcal{B}}$, the states reachable within $1, \ldots, r$ steps from $\Psi^0_{\mathcal{B}}$ until $\Psi^{r+1}_{\mathcal{B}} \implies \Psi^r_{\mathcal{B}}$. The main operation in this process is the computation of the set of abstract successor states $\Psi'_{\mathcal{B}}$, for a set of states $\Psi_{\mathcal{B}}$.

Current methods compute $\Psi'_{\mathcal{B}}$ iteratively by enumerating *cubes over* $\mathcal{B}$ and including a complete cube $c_{\mathcal{B}}$ in $\Psi'_{\mathcal{B}}$, iff the expression $\gamma(\Psi_{\mathcal{B}}) \wedge c_{\mathcal{B}}\,[\Phi'/\mathcal{B}]$ is satisfiable. The satisfiability is checked using decision procedures for (quantifier-free) first-order logic. Since an exponential number of cubes over $\mathcal{B}$ have to be enumerated in the worst case, a very large number of decision procedure calls are involved in the computation of the abstract state space.

In the next section, we demonstrate how to avoid the possibly exponentially number of calls to decision procedure by providing a symbolic method to obtain the set of initial states $\Psi^0_{\mathcal{B}}$ and the set of successor states $\Psi'_{\mathcal{B}}$, given $\Psi_{\mathcal{B}}$.

3 Symbolic Predicate Abstraction

Decision procedures for CLU [6, 24] translate a CLU formula F_{clu} to a propositional formula $\widetilde{F_{clu}}$, such that F_{clu} is satisfiable iff $\widetilde{F_{clu}}$ is satisfiable. Different methods have been proposed for translating a CLU formula to the propositional formula. These methods differ in ways to enforce the constraints for various theories (uninterpreted functions, inequality, equality etc.) to construct the final propositional formula. We will give intuitive description of the different methods as required in the different parts of the paper. Details of the procedures are outside the scope of this paper, and interested readers are referred to previous work on this subject [5, 6, 24].

Now, for every Boolean subexpression E of F_{clu}, let $\widetilde{E}$ denote the *corresponding* Boolean subexpression in the final propositional formula (if it exists). The final formula is denoted as $\widetilde{F_{clu}}$[2]. Let Σ be the set of symbols in the CLU formula F_{clu} (can include Boolean, integer and function symbols) and $\widetilde{\Sigma}$ be the set of (Boolean) symbols in the final formula $\widetilde{F_{clu}}$, where the set of Boolean symbols in $\widetilde{\Sigma}$ include all the Boolean symbols in Σ. The different Boolean encoding methods preserve the valuation of the Boolean symbols in the original formula:

Proposition 1. *There exists an interpretation σ over Σ, such that $\langle F_{clu}\rangle_{\sigma}$ is* **true**, *iff there exists an interpretation σ' over $\widetilde{\Sigma}$, such that $\langle \widetilde{F_{clu}}\rangle_{\sigma'}$ is* **true** *and for every Boolean symbol b in F_{clu}, $\langle b\rangle_{\sigma} \Leftrightarrow \langle b\rangle_{\sigma'}$.*

Now consider a CLU formula F over the symbols $\Sigma \cup \mathcal{A}$, where $\mathcal{A}$ contains Boolean symbols only and $\Sigma \cap \mathcal{A} = \{\}$. The Boolean formula $\widetilde{F}$ is a formula over $\widetilde{\Sigma} \cup \mathcal{A}$, where $\widetilde{\Sigma}$ are the Boolean symbols in the final propositional formula other than symbols in $\mathcal{A}$. Using proposition 1, we can show the following:

Proposition 2. *For any interpretation $\sigma_{\mathcal{A}}$ over the symbols in $\mathcal{A}$, $\langle \exists \Sigma : F\rangle_{\sigma_{\mathcal{A}}} \Leftrightarrow \langle \exists \widetilde{\Sigma} : \widetilde{F}\rangle_{\sigma_{\mathcal{A}}}$.*

Now let us get back to the problem of obtaining the initial set of abstract states $\Psi^0_{\mathcal{B}}$ and obtaining the expression for the successors, $\Psi'_{\mathcal{B}}$ for the set of abstract

[2] There can be a Boolean subexpression E_1 in F_{clu} which may not have a corresponding Boolean subexpression $\widetilde{E_1}$ in $\widetilde{F_{clu}}$. This can arise because of optimizations like $(\mathbf{true} \vee E_1 \longrightarrow \mathbf{true})$.

states in $\Psi_{\mathcal{B}}$. The set of initial states for the abstract system is given by the expression

$$\Psi^0_{\mathcal{B}} \doteq \exists \mathcal{K} : \bigwedge_{i \in \{1,\ldots,k\}} b_i \Leftrightarrow \phi_i \left[q^0/\mathcal{S}\right] \tag{1}$$

Similarly, given $\Psi_{\mathcal{B}}$, one can obtain the set of successors using the following equation:

$$\Psi'_{\mathcal{B}} \doteq \exists \mathcal{S} : \exists \mathcal{K} : \gamma(\Psi_{\mathcal{B}}) \wedge \bigwedge_{i \in \{1,\ldots,k\}} b_i \Leftrightarrow \phi'_i \tag{2}$$

The correctness of the encodings follows from the following proposition:

Proposition 3. *For any complete cube $c_{\mathcal{B}}$ over $\mathcal{B}$, $c_{\mathcal{B}} \Rightarrow \Psi^0_{\mathcal{B}}$ iff the expression $\gamma(c_{\mathcal{B}}) \left[q^0/\mathcal{S}\right]$ is satisfiable. Similarly, for any complete cube $c_{\mathcal{B}}$ over $\mathcal{B}$, $c_{\mathcal{B}} \Rightarrow \Psi'_{\mathcal{B}}$ iff the expression $\gamma(\Psi_{\mathcal{B}}) \wedge c_{\mathcal{B}} \left[\Phi'/\mathcal{B}\right]$ is satisfiable.*

Notice that in Equations 1 and 2, the only free variables are the set of Boolean symbols $\mathcal{B}$. Hence one can use Proposition 2 to reduce these second-order formulas (there can be function symbols in $\mathcal{S} \cup \mathcal{K}$) to an existentially quantified Boolean formula. For example, the result of propositional encoding of the formula in Equation 2 (with $\Sigma \doteq \mathcal{S} \cup \mathcal{K}$) yields the following structure[3]:

$$\exists \widetilde{\Sigma} : C \wedge \widetilde{\gamma(\Psi_{\mathcal{B}})} \wedge \bigwedge_{i \in \{1,\ldots,k\}} b_i \Leftrightarrow \widetilde{\phi'_i} \tag{3}$$

where C denotes a set of propositional constraints over the symbols in $\widetilde{\Sigma}$ to enforce the semantics of different first-order theories (uninterpreted functions, equality, inequality etc.).

This formulation is also applicable to predicate abstraction techniques which derive the *Weakest Boolean Precondition* (*WBP*) [1] over the set of predicates Φ, given the *weakest precondition* $WP(\delta, \Psi_{\mathcal{S}})$ of a CLU expression $\Psi_{\mathcal{S}}$ with respect to the transition function δ. In this case, a cube $c_{\mathcal{B}}$ is included in *WBP* if $c_{\mathcal{B}} \left[\Phi/\mathcal{B}\right] \Rightarrow WP(\delta, \Psi_{\mathcal{S}})$ is *valid*. The set of cubes which are not included in *WBP* is given as:

$$\overline{WBP} \doteq \exists \mathcal{S} : \exists \mathcal{K} : \neg\, WP(\delta, \Psi_{\mathcal{S}}) \wedge \bigwedge_{i \in \{1,\ldots,k\}} b_i \Leftrightarrow \phi_i \tag{4}$$

and $WBP \doteq \neg\, \overline{WBP}$.

Similarly, one can obtain a symbolic expression for the abstract transition relation as:

$$\delta(\mathcal{B}, \mathcal{B}') \doteq \exists \mathcal{S} : \exists \mathcal{K} : \bigwedge_{i \in \{1,\ldots,k\}} b_i \Leftrightarrow \phi_i \wedge \bigwedge_{i \in \{1,\ldots,k\}} b_i{}' \Leftrightarrow \phi'_i \tag{5}$$

Example 1. Consider a concrete system with two integer state variables, x and y. Thus $\mathcal{S} = \{x, y\}$. The initial state q^0 is given as $q^0_x = c_0$ and $q^0_y = c_0$. The transition function δ is given as $\delta_x \doteq f(y)$ and $\delta_y \doteq f(x)$. The only parameters

[3] The translation needs to ensure that $\widetilde{(\neg\phi'_i)}$ is syntactically same as $\neg\widetilde{(\phi'_i)}$. This requirement is violated for certain optimizations that push $\neg$ to the leaves of the formula [23]. Hence we have to disable such transformations in the Boolean translation.

which appear in this system are $\mathcal{K} = \{c_0, f\}$. Consider the predicate $\phi_1 \doteq x = y$ over the system.

The initial abstract state is given as $\Psi_B^0 \doteq \exists c_0 : b_1 \Leftrightarrow (c_0 = c_0) \doteq b_1$.

For the first iteration, $\Psi_B \doteq \Psi_B^0$. Now $\gamma(b_1) \doteq x = y$ and $\phi_1' \doteq (f(y) = f(x))$. The set of successor states Ψ_B' is denoted as

$$\exists x : \exists y : \exists f : x = y \wedge b_1 \Leftrightarrow (f(y) = f(x))$$

We eliminate the function symbols in the formula by the method of Bryant et al. [5]. $f(y)$ is replaced by a fresh symbolic constant vf_1. $f(x)$ is replaced by the expression $ITE(x = y, vf_1, vf_2)$ to preserve functional consistency. After eliminating the ITE, the equation becomes

$$\exists x : \exists y : \exists vf_1 : \exists vf_2 : x = y \wedge b_1 \Leftrightarrow (x = y \wedge vf_1 = vf_1 \vee \neg(x = y) \wedge vf_1 = vf_2)$$

For this example, we use the encoding of inequalities using separation predicates (SEP) method [24]. Each inequality is encoded with a fresh Boolean variable and transitivity constraints are imposed. We use the Boolean variables $e_{x,y}$ to encode $x = y$ and Boolean variable e_f to encode $vf_1 = vf_2$. In this case, no transitivity constraints are required. The quantified Boolean formula for Ψ_B' becomes

$$\Psi_B' \doteq \exists e_{x,y}, e_f : e_{x,y} \wedge b_1 \Leftrightarrow (e_{x,y} \vee \neg e_{x,y} \wedge e_f)$$

The solution to this equation is simply b_1. The set of successor states after the first iteration, $\Psi_B^1 \doteq \Psi_B'$. The reachability analysis terminates since $\Psi_B^0 \Leftrightarrow \Psi_B^1$ and the set of reachable abstract state is given as b_1.

We have thus reduced the problem of computing the set of initial abstract states Ψ_B^0 and the set of successors Ψ_B' to the problem of computing the set of solutions to an existentially quantified Boolean formula. In the next few sections, we shall exploit the structure of the formula to efficiently compute the set of solutions. For the rest of the discussion, we assume that we want to solve the existentially quantified Boolean formula $\exists \widetilde{\Sigma} : \widetilde{F}$.

3.1 Using BDD-based approaches

We can obtain the set of solutions to the quantified Boolean formula $\exists \widetilde{\Sigma} : \widetilde{F}$ by constructing the BDD for $\widetilde{F}$ and then existentially quantifying out the variables in $\widetilde{\Sigma}$ using BDD quantification operators. The resultant BDD is a symbolic representation of the formula $\exists \widetilde{\Sigma} : \widetilde{F}$.

The naive method of constructing the BDD for $\widetilde{F}$ and then quantifying out the symbols in $\widetilde{\Sigma}$ is very inefficient, since the size of the intermediate BDD representing $\widetilde{F}$ could be very large. We show that we can exploit the syntactic formula structure to leverage most the efficient quantification techniques (e.g. early quantification [14]) from image computation in symbolic model checking.

Equation 3 resembles the equation for post-image computation in symbolic model checking where $C \wedge \widetilde{\gamma(\Psi_B)}$ represents the set of current states and the transition relation for the state variable b_i is given as $\widetilde{\phi_i'}$. We can treat all the symbols in $\widetilde{\Sigma}$ as present state variables. In fact, the next state for the variables in $\widetilde{\Sigma}$ are left unconstrained and can also be interpreted as inputs. We use NuSMV's [8] `INIT` expression to specify $C \wedge \gamma(\widetilde{\Psi_{\mathcal{B}}})$ as the set of initial states and for each state variable b_i, specify the next-state transition as $\widetilde{\phi}'_i$. We then perform *one* step of post image computation to obtain the set of successor states.

3.2 Using SAT-based Approaches

The main difference between the image computation step in traditional model checking and the quantified equation in Equation 3 is that the number of bound variables (in $\widetilde{\Sigma}$) often exceeds the number of variables in $\mathcal{B}$. These variables arise because of the boolean encoding of integers in the CLU formula. In many cases, the number of variables in $\mathcal{B}$ is only 5% of the variables in $\widetilde{\Sigma}$ (for instance 400 Boolean symbols in $\widetilde{\Sigma}$ for just 3 predicates). In these cases, the cost of constructing the BDD and quantifying the variables in $\widetilde{\Sigma}$ is expensive. Instead we can use SAT-based methods to compute the set of solutions to Equation 3.

SAT-based quantification engines [18, 9] enumerate solutions over $\widetilde{\Sigma} \cup \mathcal{B}$ for the expression $\widetilde{F}$. Since we are only interested in the interpretations of symbols in $\mathcal{B}$, the part of the assignment which corresponds to symbols in $\widetilde{\Sigma}$ is projected out. To prevent the SAT checker from computing the same assignment to the symbols in $\mathcal{B}$ again, a *blocking clause* over the variables in $\mathcal{B}$ is added to the set of conflict clauses to block this assignment. The most challenging part in the entire procedure is to find a minimal blocking clause, which assigns values to a minimal subset of literals over $\mathcal{B}$. We have integrated one such tool, SATMC developed by Daniel Kroening [9], as the SAT-based quantification engine. It uses heuristics to efficiently add blocking clauses for the variables in $\mathcal{B}$ using the data structures and algorithms of ZChaff [19]. We omit the details of the procedure from this paper.

Although SAT-based quantification uses an enumeration technique, there are several advantages over current approaches which use a decision procedure repeatedly to obtain the set of cubes. First, we can take advantage of the *learning* from the SAT solvers, since the same data structure is maintained throughout the computation. Secondly, the ability to perform non-chronological backtracking provides more flexibility to obtain better cubes than the approach in Das, Dill and Park [12], where the order of variables involved in splitting the cubes is fixed. Lastly, we can remove the overhead of invoking the decision procedure repeatedly to obtain the set of solutions.

4 Universally Quantified Predicates

To verify systems with function or predicate state elements, we need the ability to specify quantified predicates. The function and predicate state elements allow us to model unbounded arrays of integers, truth values or enumerated types. These unbounded arrays can be used to model memories, queues or network of identical processes. For example, if pc is a state element which maps an integer to an enumerated set of states, then $pc(i)$ denotes the state of the i^{th} process in the system. Properties for such parameterized systems are expressed as quantified formulas. To state the property of mutual exclusion, one has to state $\forall i, j : i \neq j \Rightarrow \neg(pc(i) = \texttt{CS} \wedge pc(j) = \texttt{CS})$, where CS is the state of a process in the critical section.

However, introducing universally quantified predicates (predicates of the form $\forall \bar{i} : \phi(\bar{i})$, where $\phi(\bar{i})$ is a quantifier-free CLU expression and $\bar{i}$ is a vector of integer variables) adds two dimensions to the problem:

1. For quantifier-free predicates, concretization of the reachable abstract state space yields the strongest expression involving the predicates (Boolean combination of the predicates) that approximates the concrete reachable states. This expression serves as an invariant for the system. For parameterized systems, we have found that the inductive invariant can often be expressed as $\forall \bar{i} : P(\bar{i})$, where P is a quantifier-free CLU expression [17, 21]. However, given quantified predicates $\forall \bar{i} : \phi_1(\bar{i}), \ldots, \forall \bar{i} : \phi_k(\bar{i})$, a Boolean combination of the predicates does not always yield the strongest expression of the form $\forall \bar{i} : P(\bar{i})$, where $P(\bar{i})$ is a Boolean combination of $\phi_1(\bar{i}), \ldots, \phi_k(\bar{i})$. For example, one cannot obtain the expression $\forall i : \phi_1(i) \vee \phi_2(i)$ using a combination of the predicates $\forall i : \phi_1(i)$ and $\forall i : \phi_2(i)$.
2. Introducing universally quantified predicates requires us to check satisfiability of first-order formulas with both universal and existential quantifiers, which is an undecidable task. Hence we need sound quantifier instantiation strategies to eliminate the universal quantifiers.

To address the first problem, the user provides the set of predicates $\Phi \doteq \{\phi_1(\bar{i}), \ldots, \phi_k(\bar{i})\}$ with an implicit quantifier over $\bar{i}$ (similar to the work by Flanagan and Qadeer [13]). As before, we associate a Boolean variable b_i with $\phi_i(\bar{i})$. If $\Psi_{\mathcal{B}}$ be an expression over the symbols in $\mathcal{B}$, then Equation 2 can be written as:

$$\Psi'_{\mathcal{B}} \doteq \exists \mathcal{S} : \exists \mathcal{K} : \left(\forall \bar{i} : \Psi_{\mathcal{B}}\left[\Phi(\bar{i})/\mathcal{B}\right]\right) \wedge \exists \bar{j} : \bigwedge_{i \in \{1,\ldots,k\}} b_i \Leftrightarrow \phi'_i(\bar{j}) \tag{6}$$

Now we need to address the second problem. First, the existential quantifiers over $\bar{j}$ are pulled outside. The universal quantifiers over $\bar{i}$ are instantiated using sound instantiation techniques present in UCLID [17]. The resulting formula has the same existentially quantified structure as that of Equation 2 and thus can be solved using BDD or SAT-based quantification as before.

Finally, if $\Psi^*_{\mathcal{B}}$ is the set of reachable states over $\mathcal{B}$, then the invariant of the concrete system is $\forall \bar{i} : \Psi^*_{\mathcal{B}}\left[\Phi(\bar{i})/\mathcal{B}\right]$.

5 Results

We have built a prototype of the methods discussed into the tool UCLID [6]. To compare the effectiveness of the approach, we compare against an implementation of a recursive case-splitting based approach suggested by Das, Dill and

Park [12]. The approach is based on checking satisfiability of individual cubes as mentioned in Section 2. The Stanford Validity Checker[4] (SVC) [2] is used as the decision procedure for checking first order formulas. Various optimizations (such as considering cubes in the increasing order of lengths) are performed to reduce the number of calls to the decision procedure from the exponential worst case.

We have analyzed the different ways to encode the first order formula as a Boolean formula. In the rest of the discussion, we use the function elimination strategy by Bryant et al. [5]. Integers are encoded using two methods:

- FI : The domain of each integer is restricted to a finite but large enough set of values that preserve satisfiability [6].
- SEP : Each separation predicate $x \bowtie y + c$ is encoded as a Boolean variable and transitivity constraints are imposed [24].

NuSMV is used as the BDD-based model checker. We use dynamic variable ordering with the *sifting* heuristics and IWLS95 heuristics for quantifier scheduling. SATMC is the SAT-based model checker and is used to solve the quantification step. All the experiments were performed on a 2.2 GHz Pentium 4 machine with 900MB of memory.

Hardware Benchmarks We have used the predicate abstraction engine in the context of verification of pipelined DLX processor, an unbounded out-of-order processor (OOO), communication protocols and mutual exclusion protocols. Below we describe some of them.

Predicate abstraction was used to approximate the reachable set of states for the DLX pipelined processor. This is used to restrict the set of initial states for the Burch and Dill commutative diagram approach [7]. Current processor verification methods that employ Burch and Dill's technique [5, 16] requires the user to manually provide invariants to restrict the most general state of the system. The approximate state space was also used to verify the absence of data-hazards and correctness of stalling logic for the processor.

Quantified predicates are used to derive candidate invariants for the deductive verification of an out-of-order processor with unbounded resources [17]. Various infinite-state protocols have also been chosen to demonstrate the effectiveness of the approach. We have chosen the two process Bakery algorithm (Bakery-2) and a parameterized semaphore protocol [21]. We also chose a communication protocol called the Bounded Retransmission Protocol (BRP) which was described by Graf and Saidi [15] for demonstrating the use of predicate abstraction.

Fig 2 illustrates the performance of the different approaches on a set of benchmarks. For DLXb, the number of calls to SVC reported are even before the first iteration for the explicit version completes.

[4] We have experimented with other decision procedures (e.g. CVC [3],UCLID [6]) but have not found any significant improvement for this application.

Example	Preds	Iter	Explicit Method		Symbolic Time (sec)			
					SATMC-based		NuSMV-based	
			SVC Time	# of Calls	FI	SEP	FI	SEP
Semaphore	8	5	37.0	513	16.3	9.7	5.1	2.6
DLXa	5	3	24.5	199	5.3	4.0	2.5	2.3
DLXb	23	5	> 1600	> 16800	> 1000	> 1000	> 1000	535.0
Bakery-2	10	4	59.0	383	9.1	7.1	2.9	2.7
BRP	22	9	561.2	5040	190.2	221.0	25.4	39.9
OOO	8	2	*	*	33.0	-	> 1000	> 1000

Fig. 2. Hardware example results. "Iter" is the number of iteration for abstract reachability analysis, "SVC Time" is the time spent inside SVC decision procedure, "# of Calls" denote the number of calls to SVC, "FI" and "SEP" denote the Boolean Encoding using finite-instantiation and separation predicates respectively. A "*" indicates that SVC produces spurious answer due to rational interpretation. A "-" indicates an unexpected core dump with separation predicate encoding.

Example	# of Predicates	Explicit		SATMC			
		Calls	Time	SEP		FI	
				# Prop-vars	Time (sec)	# Prop-vars	Time (sec)
sl.0	12	955	129.5	64	3.8	60	2.6
sl.20	10	631	81.5	77	4.9	74	3.0
sl.56	11	821	110.2	65	5.4	54	2.6
sl.65	10	469	57.2	50	3.0	46	1.7
dr.10	19	>7576	>1000	162	9.2	115	9.9
dr.13	20	>7351	>1000	234	44.7	161	35.3
dr.14	20	>7189	>1000	232	103.5	157	25.6
dr.15	23	>7237	>1000	336	68.2	198	700.4
dr.16	13	2023	172.2	96	10.8	129	30.6
dr.17	15	3041	507	82	5.4	105	6.1
dr.18	18	>7099	>1000	153	130.6	180	49.7
dr.3	13	2023	355	100	9.0	125	7.0
dr.6	13	3355	596	96	8.1	129	7.8

Fig. 3. Results over SLAM formulas. "Calls" denotes the number of calls to the decision procedure SVC, "Prop-vars" denotes the number of propositional variables in the final propositional encoding. "SEP" denotes the method of encoding using separation predicates, and "FI" denotes the encoding using finite-instantiation. For the explicit version, the process was stopped after 1000s spent in the decision procedure.

Software Benchmarks For software benchmarks, we generated several problem instances from the SLAM [1] toolkit for Microsoft Windows device-driver verification. For each of these examples, the expression for the weakest precondition (*WP*) and the set of predicates are supplied. The tool computes the *Weakest Boolean Precondition* (*WPB*) (as described in Section 3). We have run more than 100 such examples. In Fig 3, we report the results on some of the benchmarks with greater than 10 predicates. The symbolic method also outperforms the explicit method on smaller set of predicates. On all these examples, the SATMC based solver outperformed the NuSMV based method. SATMC solver took at most a few seconds to solve most examples, whereas NuSMV took several minutes to solve the bigger examples. This is primarily due to the large number of Boolean variables in the final encoding.

5.1 Discussion

We have found that the BDD-based approach is more sensitive to the number of variables in the final encoding rather than the number of predicates. This is because the size of the intermediate BDD depends on the sum of the number of quantified and unquantified variables. This makes it useful for applications where the number of predicates are large, but the Boolean encoding has a smaller number of Boolean variables. This is evident in the example with Semaphore, DLXa, DLXb, Bakery-2 etc. where the NuSMV method outperforms the SATMC-based method. The SATMC based approach is more robust in the presence of large number of bound variables. This is evident in the case of the software verification benchmarks, where the number of Boolean variables is in excess of 100.

The large number of calls to the decision procedure for the DLXb example can be explained as follows. For this model, the set of reachable states is extremely dense and results in a dense abstract state space too. Therefore, the number of cubes to enumerate is very large. This is one reason why even the SATMC based approach takes a long time to solve the example.

For parameterized systems like the out-of-order processor or cache-coherence protocols, we have found the SATMC based method to outperform the NuSMV-based methods on most examples. This is because, even though the number of predicates is small (typically 10–15), the instantiation of quantifiers produces a lot of terms in the first-order formula, which translates to a large number of Boolean variables (almost 500) in the final formula.

Acknowledgements The first author is grateful to Pankaj Chauhan for his help in understanding NuSMV and his suggestions. We are also grateful to Daniel Kroening for letting us use SATMC, and spending time to create an interface to our tool. We are also grateful to Sanjit Seshia and Amit Goel for their comments and help with the tools.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI '01)*, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.
2. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, November 1996.
3. C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Computer-Aided Verification (CAV '02)*, LNCS 2404, 2002.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), August 1986.
5. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer-Aided Verification (CAV '99)*, LNCS 1633, July 1999.

6. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, July 2002.
7. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, LNCS 818, June 1994.
8. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *Computer-Aided Verification (CAV'99)*, LNCS 1633, July 1999.
9. E. Clarke, D. Kroening, and P. Chauhan. Fixpoint computation for circuits using Symbolic Simulation and SAT. In *Preparation*, 2003.
10. D. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Fourth Annual Symposium on Principles of Programming Languages (POPL '77)*, 1977.
11. S. Das and D. Dill. Successive approximation of abstract transition relations. In *IEEE Symposium of Logic in Computer Science(LICS '01)*, June 2001.
12. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verification (CAV '99)*, LNCS 1633, July 1999.
13. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '02)*, 2002.
14. D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Computer-Aided Verification (CAV '94)*, LNCS 818, June 1994.
15. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254, June 1997.
16. S. K. Lahiri, C. Pixley, and K. Albin. Experience with term level modeling and verification of the MCORE microprocessor core. In *Proc. IEEE High Level Design Validation and Test (HLDVT 2001)*, November 2001.
17. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, November 2002.
18. K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Computer Aided Verification, (CAV '02)*, LNCS 2404, 2002.
19. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, 2001.
20. K. Namjoshi and R. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer-Aided Verification (CAV 2000)*, LNCS 1855, July 2000.
21. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, 2001.
22. H. Saidi and N. Shankar. Abstract and Model Check while you Prove. In *Computer-Aided Verification (CAV '99)*, LNCS 1633, July 1999.
23. O. Strichmann. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, November 2002.
24. O. Strichmann, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with sat. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, July 2002.
25. M. N. Velev and R. E. Bryant. Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions and Branch Predication. In *37th Design Automation Conference (DAC '00)*, June 2000.