Deciding CLU Logic Formulas via Boolean and Pseudo-Boolean Encodings

Randal E. Bryant[1,2], Shuvendu K. Lahiri[2], and Sanjit A. Seshia[1]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
{Randy.Bryant, Sanjit.Seshia}@cs.cmu.edu
[2] Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA
shuvendu@ece.cmu.edu

Abstract. UCLID is a tool for specifying and verifying systems expressed in a quantifier-free first-order logic, called CLU logic, that includes uninterpreted functions, equality, ordering, constrained lambda expressions, and counter arithmetic. In previous work, we presented different variants of a decision procedure for this logic that are all based on on an efficient translation of a CLU formula to a Boolean formula, which is checked using a Boolean satisfiability solver. In this paper, we present another variant based on the PBS tool that integrates a pseudo-Boolean constraint solver with a SAT solver. We then present an empirical evaluation of all decision procedure variants on a set of benchmark formulas generated from various UCLID models.

1 Introduction

Several systems, such as communication protocols with unbounded channels, and networks of an arbitrary number of identical processes, have state variables of unbounded integer or unbounded integer-valued array type. Others with very large integer-valued state variables and memories, such as superscalar out-of-order processors, are also often conveniently modeled as infinite-state systems.

UCLID [6] is a verification tool for verifying safety properties of such systems. It is based on modeling systems in a logic of counter arithmetic with lambda expressions and uninterpreted functions, abbreviated as CLU. CLU is a fragment of quantifier-free first-order logic that includes uninterpreted functions, equality, ordering, constrained lambda expressions, and successor ("+1") and predecessor ("-1") arithmetic operations. A safety property for a UCLID model is a formula in CLU logic. Thus, a central part of UCLID's operation is deciding the validity of CLU formulas. In previous work [6], we presented an efficient decision procedure for CLU. This procedure is based on an efficient validity-preserving translation of a CLU formula to a Boolean formula, which is checked using a Boolean satisfiability (SAT) solver. Thus, we are able to leverage the recent advances in SAT solving.

Separation logic is the fragment of CLU logic obtained by dropping constrained lambda expressions and uninterpreted functions of nonzero arity. Recently, there has been work [11] on efficiently deciding separation formulas via a different translation to SAT, which is an extension of an earlier technique for encoding equations with Boolean variables [7]. This translation method has also been incorporated into UCLID as a different variant of its decision procedure for CLU.

In this paper, we present yet another variant of our decision procedure for CLU. This variant is based on an encoding of a separation formula as a CNF formula and a pseudo-Boolean (0-1 integer) constraint satisfaction problem. We use the PBS tool [1] that integrates a pseudo-Boolean constraint solver with a SAT solver. The second contribution of this paper is an empirical evaluation of all decision procedure variants on a set of benchmark formulas generated from various UCLID models.

The rest of the paper is organized as follows. We give some background in Section 2, describing the CLU logic, separation logic, and the PBS tool. We then describe the three variants of the decision procedure that we compare in this paper. We first describe the common part that all three share in Section 3, and then describe the variants in Section 4. We analyze these variants empirically in Section 5, and conclude in Section 6.

2 Background

2.1 The CLU Logic

Expressions in CLU describe means of computing four different types of values. *Boolean* expressions, also termed *formulas*, yield **true** or **false**. *Integer* expressions, also referred to as *terms*, yield integer values. *Predicate* expressions denote functions from integers to Boolean values. *Function* expressions, on the other hand, denote functions from integers to integers. Figure 1 summarizes the expression syntax.

The simplest Boolean expressions are **true** and **false**. Boolean expressions can also be formed by comparing two integer expressions for equality or for ordering, or by applying a predicate expression to a list of integer expressions, or by combining Boolean expressions using Boolean connectives. Integer expressions can be integer variables[3], or can be formed by applying a function expression (including interpreted functions **succ** ("+1") and **pred** ("-1")) to a list of integer expressions, or by applying the *ITE* (for "if-then-else") operator. Function (predicate) expressions can be either function (predicate) symbols, representing uninterpreted functions (predicates), or lambda expressions, defining the value of the function as an integer (Boolean) expression containing references to a set

[3] Integer variables are used only as the formal arguments of lambda expressions

$$
\begin{aligned}
\textit{bool-expr} ::= {} & \mathbf{true} \mid \mathbf{false} \mid \neg \textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \\
& \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\
& \mid \textit{predicate-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{int-expr} ::= {} & \textit{int-var} \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
& \mid \mathbf{succ}(\textit{int-expr}) \mid \mathbf{pred}(\textit{int-expr}) \\
& \mid \textit{function-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{predicate-expr} ::= {} & \textit{predicate-symbol} \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \textit{bool-expr} \\
\textit{function-expr} ::= {} & \textit{function-symbol} \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \textit{int-expr}
\end{aligned}
$$

Fig. 1. CLU Syntax.

of argument variables. We will omit parentheses for function and predicate symbols with zero arguments, writing a instead of $a()$. We refer to function symbols of zero arity as *symbolic constants*, and to predicate symbols of zero arity as *symbolic Boolean constants*.

An integer variable x is said to be *bound* in expression E when it occurs inside a lambda expression for which x is one of the argument variables. We say that an expression is *well-formed* when it contains no unbound variables. The value of a well-formed expression in CLU is defined relative to an interpretation I of the function and predicate symbols. We shall omit the details in this paper. A well-formed formula F is *true under interpretation* I if $[F]_I$ is **true**. It is *valid* when it is true under all possible interpretations.

2.2 Separation Logic

Separation logic is a restriction of CLU obtained by dropping lambda expressions and uninterpreted function and predicate symbols of nonzero arity; i.e., containing only symbolic constants and symbolic Boolean constants. The syntax is summarized in Figure 2. The semantics are the same as for CLU expressions.

$$
\begin{aligned}
\textit{bool-expr} ::= {} & \mathbf{true} \mid \mathbf{false} \mid \neg \textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \\
& \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \mid \textit{symb-bool-const} \\
\textit{int-expr} ::= {} & \textit{symb-const} \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
& \mid \mathbf{succ}(\textit{int-expr}) \mid \mathbf{pred}(\textit{int-expr})
\end{aligned}
$$

Fig. 2. Separation Logic Syntax.

2.3 The PBS solver

A pseudo-Boolean constraint is an integer constraint of the form $\sum_i a_i x_i \bowtie b$, where the a_is and b are arbitrary integer constants, $x_i \in \{0,1\}$ are pseudo-Boolean variables, and $\bowtie \in \{=, \geq, \leq\}$ is a relational operator. The PBS solver [1] takes as input a set of pseudo-Boolean constraints and a purely Boolean formula (expressed in conjunctive normal form), and outputs "satisfiable" if the Boolean formula *and* all the pseudo-Boolean constraints are satisfiable, and "unsatisfiable' otherwise. The key point is that variables can be shared between the Boolean formula and the pseudo-Boolean constraints. Thus, assignments to shared variables are propagated between the pure-Boolean and pseudo-Boolean solvers to detect conflicts and make forced assignments. For details, we refer the reader to the PBS paper [1].

3 Decision Procedure for CLU

Assume we start with a well-formed formula F_{clu} in CLU expressing some desired system property. The decision procedure must determine whether it is *valid*, i.e., true under all possible interpretations of the function and predicate symbols.

For purposes of this paper, we consider the decision procedure as operating in two phases. In the first phase, we perform a validity-preserving transformation of a F_{clu} to a formula F_{const} in separation logic. In this phase, we also compute *range* information that is used in the next phase. In the second phase, we use different encoding methods to decide the separation logic formula. Thus, the different variants of our decision procedure differ only in the second phase. In this section, we give a brief description of the first phase of transformation. A detailed description may be found in a previous paper [6]. The second phase is described in Section 4.

3.1 Transforming CLU to Separation Logic

Expand Lambda Applications. Since CLU syntax does not permit recursion or iteration, each lambda application can be expanded by *beta-substitution*, i.e., by replacing each argument variable with the corresponding argument term. Let us call the resulting formula F_{exp}.

Identify P-Function Symbols. As described in [5], we can exploit restricted uses of equations to greatly reduce the number of interpretations that must be encoded when we reduce the formula to propositional logic. To do this, we automatically analyze F_{exp} to determine those function symbols that satisfy the restrictions of "p-functions". The general idea is to determine the polarity of each equation, i.e., whether it appears under an even (positive) or odd (negative) number of negations. Terms can then be classified as either p-terms, i.e., used only under positive equalities, or g-terms, i.e., general terms that appear under other equalities or under inequalities. Function symbols for which all applications are p-terms can then be classified as p-function symbols, and the others are general "g-function" symbols. Applications of p-function symbols can be encoded in propositional logic with fewer symbolic variables than can those of g-function symbols.

Remove Function and Predicate Applications. We replace all applications of uninterpreted functions (or predicates) of nonzero arity by terms containing only symbolic constants (or symbolic Boolean constants). Each function application is replaced by a nested series of ITE operations. As an example, if function symbol f has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$, then we would generate 3 new symbolic constants vf_1, vf_2, and vf_3. We would then replace all instances of $f(a_1)$ by vf_1, all instances of $f(a_2)$ by $ITE(a_2 = a_1,\ vf_1,\ vf_2)$, and all instances of $f(a_3)$ by $ITE(a_3 = a_1,\ vf_1,\ ITE(a_3 = a_2,\ vf_2,\ vf_3))$.

Predicate applications can be removed by a similar process. In eliminating applications of some predicate p, we introduce symbolic Boolean constants $vp_1, vp_2, \ldots$.

This leaves us with a formula F_{const} containing only symbolic constants, ITEs, successors, predecessors, equations, inequalities, and logical connectives — viz., a formula in separation logic.

3.2 Computing Range Information

Two of the three decision procedure variants utilize the "small-model" property of CLU. This allows us to restrict ourselves to interpretations whose domain size equals the number of distinct terms in F_{clu}. We can then perform a *finite instantiation* of each symbolic constant.

We now describe the steps used to compute the size of this finite instantiation. We measure size of the finite instantiation of a symbolic constant in terms of the number of bits required to encode that constant.

Partition into Subdomains. We first split the set of symbolic constants V into two sets V_p and V_g. V_p consists of those symbolic constants occurring in F_{exp} that were classified as p-function applications, as well as those constants vf_i that were introduced when eliminating an application of some p-function symbol f. The remaining symbolic constants are in V_g.

We then partition the set of symbolic constants into classes $V_1, \ldots, V_n$. Each constant in V_p is assigned to its own class. Constants in V_g are then grouped into classes, with two constants being in the same class if their values may be compared by equations or inequalities. The idea is that if two constants are never compared, we need only consider interpretations in which they get distinct values.

Compute Ranges. For each symbolic constant v in F_{const} we must determine the maximum amount it can be incremented or decremented by successor and predecessor operations. We do this by labeling each distinct term T in F_{const}

by its lower bound $l(T)$ and its upper bound $u(T)$. These bounds indicate the range over which the term may be decremented or incremented.

The labeling can be implemented as a fixed-point computation, starting with $l(T) = u(T) = 0$ for each term T. Labels are then updated according to the following rules:

Term T	Lower Bound	Upper Bound
$ITE(F, T_1, T_2)$	$l(T_1) \leftarrow \min(l(T_1), l(T))$ $l(T_2) \leftarrow \min(l(T_2), l(T))$	$u(T_1) \leftarrow \max(u(T_1), u(T))$ $u(T_2) \leftarrow \max(u(T_2), u(T))$
$\mathbf{succ}(T_1)$	$l(T_1) \leftarrow \min(l(T_1), l(T) + 1)$	$u(T_1) \leftarrow \max(u(T_1), u(T) + 1)$
$\mathbf{pred}(T_1)$	$l(T_1) \leftarrow \min(l(T_1), l(T) - 1)$	$u(T_1) \leftarrow \max(u(T_1), u(T) - 1)$

Eventually, this process will reach a point where the bounds do not change. We then use the values of $l(v)$ and $u(v)$ to determine the range of offsets for symbolic constant v.

For each class V_i we compute its range as:

$$range(V_i) = \sum_{v \in V_i} (u(v) - l(v) + 1).$$

This determines the size of the finite instantiation we need to consider for each symbolic constant in V_i, as described in Section 4.

4 Methods for Encoding Separation Constraints

We now describe three different methods of deciding the separation logic formula F_{const}. The first two, described in Sections 4.1 and 4.2, work by translating F_{const} to a Boolean formula F_{bool}. We then use a SAT solver to check if $\neg F_{bool}$ is satisfiable. If $\neg F_{bool}$ is unsatisfiable, then we have determined that the original formula F_{clu} is valid, otherwise, it is invalid. The third method, described in Section 4.3, is based on translating $\neg F_{const}$ to a set S of pseudo-Boolean constraints along with a Boolean formula F_{bool}. These are then input to the PBS solver to check the satisfiability of $\neg F_{const}$.

4.1 Small Domain Instantiation with Binary Arithmetic (SD-BA Method)

Suppose there are K different classes $V_1, \ldots, V_K$ generated in the first phase as described above. Let M be the maximum value of $range(V_i)$ for any class V_i. Let $k = \lceil \log_2 K \rceil$ and $m = \lceil \log_2 M \rceil$. Then we encode each symbolic constant as a vector of $k + m$ Boolean formulas $\boldsymbol{v}$. For variable v in class V_i, the high order k elements of $\boldsymbol{v}$ correspond to the binary encoding of i. If class V_i contains just a single constant v, then the low order m elements of $\boldsymbol{v}$ are simply the binary representation of $-l(v)$. Since $l(v)$ must be less than or equal to zero, the effect of this is to bias the value used to encode variable v such that this value will never be decremented below zero by any of the **pred** operations. Otherwise, for each variable v we must introduce m' Boolean variables $\boldsymbol{x_v} \doteq x_v^{m'-1}, \ldots, x_v^0$, where $m' = \lceil \log_2 |V_i| \rceil$. The low order m elements of $\boldsymbol{v}$ are then the Boolean formulas expressing the bit-level representation of $\boldsymbol{x_v} - l(v)$.

We then recursively translate F_{const} into a symbolic Boolean formula, where each term is represented as a vector of $k+m$ formulas and each subformula as a single Boolean formula. Each symbolic constant v is represented by the vector $\boldsymbol{v}$, while each symbolic Boolean constant is represented by a Boolean variable. ITE operators are translated to perform a bit-wise multiplexing of the arguments. Successor and predecessor operations are translated as bit-level incrementers and decrementers. Equations and inequalities are translated as comparators. Boolean connectives are translated as Boolean operators. Let F_{bool} denote the resulting Boolean formula.

This translation process takes advantage of the positive equality structure of the formula in a manner similar to that described in [5]. Each symbolic constant in V_p is assigned a fixed bit pattern, greatly reducing the number of Boolean variables required.

4.2 Per-Constraint Boolean Variables (EIJ Method)

Consider the separation logic formula F_{const}. We translate it to a Boolean formula in three steps:

1. Eliminate *ITE* expressions in F_{const}: Consider the directed acyclic graph (DAG) representation of F_{const}. The leaves of this DAG are symbolic constants. We eliminate *ITE* expressions from this DAG by pushing **succ** and **pred** function applications to the leaves of the DAG, and then by translating the formula so that equality and inequality comparisons are only between ground terms (symbolic constants and iterated applications of **succ** and **pred** to them) at the leaves, and not between *ITE* expressions, using an extensions of the method described in [5].

Note that, by using positive equality, equalities of the form $v_i = v_j + c$, where either of symbolic constants v_i or v_j are in V_p, can be directly reduced to **false**.

2. Replace separation constraints by Boolean variables: At this point, the leaves of the formula DAG are constraints of the form $v_i \bowtie v_j + c$ where $\bowtie \in \{=, <\}$, v_i and v_j are symbolic constants, and c is an integer constant. Conceptually, we replace each constraint $v_i \bowtie v_j + c$ in the formula with a corresponding Boolean variable $e_{i,j}^{\bowtie,c}$. Let the resulting formula be F_{bvar}.

3. Add "transitivity" constraints: The final step is to constrain the $e_{i,j}^{\bowtie,c}$ Boolean variables so as to disallow assignments to these variables that do not have any corresponding evaluation to the symbolic constants v_i and v_j. We do this by imposing a set of constraints on those Boolean variables. These constraints can be viewed as "transitivity" constraints, that capture the semantics of relational operators with offsets. For example, we would impose the constraint $e_{i,j}^{=,0} \wedge e_{j,k}^{=,0} \implies e_{i,k}^{=,0}$. Let F_{trans} denote the conjunction of all the imposed constraints.

The final Boolean formula F_{bool} is $(F_{trans} \implies F_{bvar})$.

We omit a detailed description of this method from the paper. The interested reader is referred to [7, 11].

4.3 Pseudo-Boolean Encoding (PBS Method)

As described in Section 3.2, CLU has a small model property, due to which we can decide the validity of a CLU formula by considering only finite instantiations of symbolic constants. We now show that this property also enables the use of a pseudo-Boolean solver, by a suitable encoding of equations and inequalities.

The first step is to negate the formula F_{const}, and to eliminate the *ITE* expressions in $\neg F_{const}$ just as described in Section 4.2.

We then replace each equation or inequality C in $\neg F_{const}$ with a corresponding Boolean variable t_C. The formula F_{bool} obtained after substituting equations and inequalities (hereafter collectively referred to as "constraints") in $\neg F_{const}$ with their corresponding Boolean variables is thus purely Boolean. We encode the equivalences $t_C \Leftrightarrow C$ by pseudo-Boolean constraints. Thus, the variables t_C occur as both pure Boolean and pseudo-Boolean variables. The symbolic constants are encoded as pseudo-Boolean bit vectors of length equal to the size of their finite instantiation.

The central problem is thus to encode equivalences $t_C \Leftrightarrow C$ as pseudo-Boolean constraints. There are two types of constraints derivable in the syntax of CLU: equations and "less-than" inequalities. We consider encoding both these types of constraints in turn.

Consider the inequality $v_i < v_j + c$. Since v_i and v_j are integer-valued, this inequality is equivalent to $v_i \leq v_j + c - 1$. Let V_{ij} denote the class containing the symbolic constants v_i and v_j. The size of the finite instantiation for v_i and v_j is Λ bits, where $\Lambda = \lceil \lg(range(V_{ij})) \rceil$. Let $v_{i,0}, v_{i,1}, \ldots, v_{i,\Lambda-1}$ denote the pseudo-Boolean variables in the pseudo-Boolean bit-vector encoding v_i (and likewise for v_j). We will hereafter use v_i and v_j as short-hand for a linear combination of their constituent pseudo-Boolean bits, with the precise definition of v_i (and similarly for v_j) given here:

$$v_i = \sum_{\lambda=0}^{\Lambda-1} v_{i,\lambda} 2^{\lambda} \qquad (1)$$

Let t denote the (pseudo-)Boolean variable that encodes this inequality. We represent the equivalence $t \Leftrightarrow (v_i \leq v_j + c - 1)$ by the following two pseudo-

Boolean constraints:

$$v_i - v_j + t(2^{\Lambda} + 1 - c) \leq 2^{\Lambda} \tag{2}$$
$$-v_i + v_j + t(-2^{\Lambda} - c) \leq -c \tag{3}$$

The inequalities 2 and 3 represent the implications $t \implies (v_i \leq v_j + c - 1)$ and $\neg t \implies (v_i > v_j + c - 1)$ respectively. To see this, consider the case $t = 1$. In this case, the two inequalities reduce to

$$v_i - v_j \leq c - 1$$
$$-v_i + v_j \leq 2^{\Lambda}$$

The first inequality is the one we want satisfied if $t = 1$. The second inequality is trivially satisfied, because v_i and v_j are both non-negative and bounded above by 2^{Λ}. The case $t = 0$ can be analyzed in a symmetric fashion.

The encoding of equations is only slightly more complicated. Firstly, we cannot encode an equation directly with a single (pseudo-)Boolean variable because the negated equation cannot be represented as a pseudo-Boolean constraint. Instead, we represent an equation $v_i = v_j + c$ as two inequalities $v_i \geq v_j + c$ and $v_i \leq v_j + c$, and encode each of these inequalities using two (pseudo-)Boolean variables t_1 and t_2 respectively. The equivalences $t_1 \Leftrightarrow (v_i \geq v_j + c)$ and $t_2 \Leftrightarrow (v_i \leq v_j + c)$ are represented by four pseudo-Boolean constraints, listed here in order corresponding to the cases $t_1 = 1$, $t_1 = 0$, $t_2 = 1$, and $t_2 = 0$:

$$-v_i + v_j + t_1(c + 2^{\Lambda}) \leq 2^{\Lambda} \tag{4}$$
$$v_i - v_j + t_1(c - 1 - 2^{\Lambda}) \leq c - 1 \tag{5}$$
$$v_i - v_j + t_2(-c + 2^{\Lambda}) \leq 2^{\Lambda} \tag{6}$$
$$-v_i + v_j + t_2(-c - 1 - 2^{\Lambda}) \leq -c - 1 \tag{7}$$

In this manner, we generate a set S of pseudo-Boolean constraints that represent the relation between the constraints C in $\neg F_{const}$ and the variables t_C that encode them, along with a purely Boolean formula F_{bool}. The set S and formula F_{bool} are then fed as input to the PBS solver to decide satisfiability of $\neg F_{const}$, which in turn decides the validity of F_{const}.

5 Analysis

We now analyze the variants of our decision procedure empirically using a large set of benchmarks. The benchmarks we used, along with the descriptions of where they arise and the characteristics of the formulas, are summarized in Figure 3.

5.1 Results

The total time taken to decide a CLU formula consists of (i) time spent in the decision procedure to convert the CLU formula to an equivalent propositional

Benchmark	Description	Formula Characteristics	
		Has inequalities?	Other
DLX processor	There are 3 models: (i) a 5-stage DLX processor with finite-state data memory model (dlx1) [4], (ii) a DLX processor with a memory model that preserves data-forwarding semantics (dlx2), and (iii) a variant of dlx2 that additionally uses a virtual to real address translation box (dlx3). Formulas were generated in correspondence checking [4].	No	
German's cache coherence protocol	A directory-based cache-coherence protocol with unbounded number of clients. Multiple formulas were generated by checking the cache-coherence property after symbolically simulating the system for increasing number of steps.	No	
Memory unit of Elf™ processor.	Multiple formulas were generated by checking the memory unit of the Elf pipeline against an ISA model by symbolically simulating for an increasing number of steps.	No	
Code validation suite	These formulas were obtained from software verification problems in the code-validation project [9].	No	
Two queues	Multiple formulas were obtained by comparing two different queue implementations by symbolically simulating for an increasing number of steps.	No	Very few Boolean variables in large propositional DAG (e.g., 40 steps, 40 vars, 41937 DAG nodes).
Out-of-order processor	This is a model of an simple out-of-order processor with arithmetic instructions and unbounded resources [8]. Two different sets of formulas (`ooo.tag` and `ooo.rf`) were generated after symbolically simulating the model for an increasing number of steps.	Yes	
Invariant checking	The formulas in this section are generated during deductive verification of the cache-coherence protocol and the out-of-order processors (mentioned above).	Yes (very many)	Very many uninterpreted functions, very few p-functions.

Fig. 3. Benchmarks. All formulas had occurrences of **succ** and equalities. Elf is a trademark of Motorola Inc.

formula for small-domain and EIJ encoding, or to a pseudo-Boolean formula for the PBS encoding and (ii) time taken by a SAT solver (Chaff) or PBS to solve the resulting formula. For large benchmarks, the second component of the time dominates the overall time. Note that for SD-BA and PBS, the conversion procedure is polynomial time (after expanding lambda expressions). However, EIJ encoding can add exponential number of constraints (in the size of the lambda-free formula) to convert it to a propositional formula.

We also used a *hybrid* variant to combine the SD-BA and EIJ method without producing an exponentially large formula. This variant, termed as "Hybrid", uses the EIJ method for encoding equality constraints for variable classes (mentioned in Section 3.2) whose members are compared for equality (or disequality) alone, and do not have **succ** or **pred** applied to any of the members. The SD-BA method is used for encoding constraints for all other variable classes. Since the

EIJ method for equalities only result in a polynomial number of constraints being added, we still have a polynomial transformation to the propositional formula.

In addition to these variants, we ran experiments with the Cooperating Validity Checker (CVC) developed at Stanford [2]. Note that CVC can handle a more general logic than CLU, including, in addition, linear real arithmetic and inductive data types.

All the experiments were performed on a 1.4 GHz P4 machine with 256 MB memory running Linux. We used the zChaff version of Chaff, while we used a beta-version of the PBS solver that uses a custom-built version of Chaff. We ran PBS with the "-D 1" option.

We present the salient features of the results for each benchmark below.

DLX processor. Figure 4 contains the results for this benchmark. We observe

File	#-vars	#-pvars	PBS Time		SD-BA Time		EIJ Time		Hybrid Time		CVC
			Conversion	PBS	Conversion	Chaff	Conversion	Chaff	Conversion	Chaff	Time
dlx1	105	73	4.63	2.9	3.67	7.98	7.98	0.74	7.410	1.07	18.83
dlx2	98	57	5.480	4.74	3.780	22.87	11.620	2.27	11.410	1.59	24.45
dlx3	106	65	5.190	3.69	3.810	10.2	11.510	1.8	10.86	1.58	14.48

Fig. 4. Results on DLX processor. "vars" is the number of integer symbolic constants in the CLU formula after eliminating function applications, and "p-vars" is number of those symbolic constants that correspond to p-function applications. The 3 columns correspond to the time taken by the 3 methods. "Conversion" is time to translate the CLU formula to a Boolean formula. "PBS" is the time taken by the PBS solver and "Chaff" is the time taken by Chaff SAT solver.

that the time to translate a CLU formula to a Boolean formula is the smallest for the small-domain encoding and largest for the EIJ encoding. This is because both PBS and EIJ encoding require the elimination of ITE constructs which can cause a quadratic blowup in the formula size. The EIJ encoding incurs the additional overhead of having to compute transitivity constraints. The trend is identical in the rest of the benchmarks, and furthermore, the relative differences are within a factor of 2 for the larger benchmarks. The time taken by PBS or SAT dominates translation time for larger benchmarks. Therefore, in the rest of the examples, we shall ignore translation time and concentrate on the PBS or SAT time.

German's cache coherence protocol. Figure 5 shows the results. CVC ran out of memory for all the formulas in this category and hence does not appear in the plot.

Memory unit of Elf™ processor. Figure 6 contains the time taken by the three methods on these benchmarks. PBS ran out of memory (> 400MB) beyond 8 steps of symbolic simulation. EIJ method is unable to complete beyond 8 steps of simulation because of the very larger number of transitivity constraints.

Code validation suite. Figure 7 shows the time taken for each of the 10 benchmarks. Most of these benchmarks finish off within a second with EIJ encoding

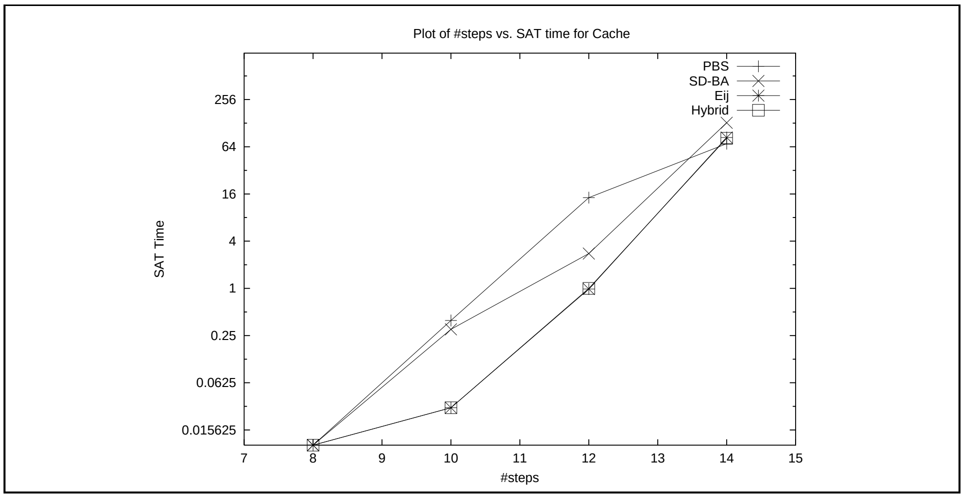

Fig. 5. Results for Cache Protocol.

and PBS. Small domain encoding however is slow by more than an order of magnitude on most examples. The missing bars indicate time less than 0.01 seconds.

Two queues. Figure 8(b) illustrates the result for the *Two queues* benchmark. Notice that small-domain encoding outperforms the other methods. For this example, the PBS and EIJ encoding methods run out of memory during the ITE elimination stage. We also used BDDs [3] instead of a SAT solver and the results demonstrate that BDDs are extremely efficient in solving this benchmark. This is probably due to the small number of Boolean variables.

Out-of-order processor. Figures 9 and 10 illustrate the results for the `ooo.tag` and `ooo.rf` sets of formulas. The EIJ and Hybrid methods perform best.

Invariant checking. Table 11 shows the results for the out-of-order processor and cache coherence protocols. PBS method ran out of memory ($>$ 400 MB) on all these examples. The EIJ method did not finish due to a very large number of transitivity constraints. The large number of transitivity constraints arise mainly because of the absence of p-function symbols in all these benchmarks and the presence of large number of inequalities and **succ** in the formulas. Since the validity of these formula rely on the *integer* interpretation of the terms, CVC, which interprets terms over rationals, could not be used to check the validity.

5.2 Discussion

As noted earlier, the time to translate a CLU formula to a Boolean formula follows the order SD-BA $<$ PBS $<$ EIJ, due to the overhead of ITE elimination for PBS and EIJ, and of transitivity constraint generation for EIJ.

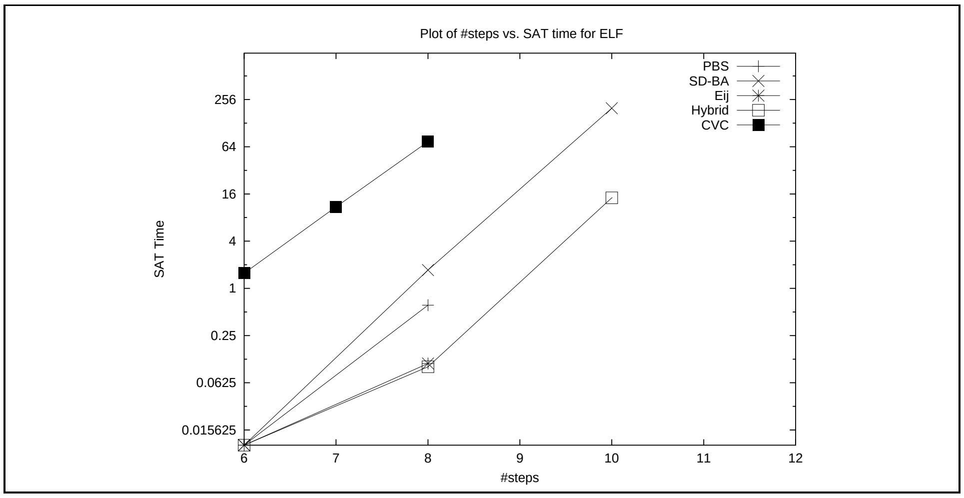

Fig. 6. Results for Elf Processor.

Encoding separation constraints directly by Boolean variables, as done in the PBS and EIJ methods, appears to assist the SAT solver better than small-domain encoding. This is probably because, in many formulas, separation constraints control the values sub-expressions of the formula evaluate to, and so assigning to the corresponding Boolean variables is more likely to lead to a conflict or a satisfying assignment. On the other hand, for SD-BA, the SAT solver must correlate the assignment to the various bits of variables of a separation constraint so as to come up with an assignment to the whole constraint itself, so the cost of finding the right correlation is more. Our hypothesis is supported by results for the DLX, cache protocol, and code validation benchmarks.

While encoding constraints more naturally, the PBS and EIJ methods have to ensure that the assignment to the separation constraint Boolean variables respects transitivity constraints. The SD-BA method requires no such transitivity constraints. When the number of transitivity constraints is small, as for the DLX, cache protocol, and out-of-order benchmarks (see Figure 5.2), EIJ (and PBS to a lesser extent) does well. But if there are very many transitivity constraints to be enforced, both PBS and EIJ methods perform poorly compared to SD-BA, as manifested in the invariant checking benchmarks.

The EIJ method explicitly imposes transitivity constraints by adding clauses to the original formula. In the worst case, it might add exponentially many clauses (in the number of original constraints). We see an example of the blowup in transitivity constraints on the invariant checking benchmarks (Figure 11).

The PBS method imposes transitivity constraints implicitly, by detecting how assignments made to Boolean variables encoding separation constraints can lead to conflicts between their corresponding pseudo-Boolean constraints. To see this, suppose the SAT portion makes an assignment to two separation constraint

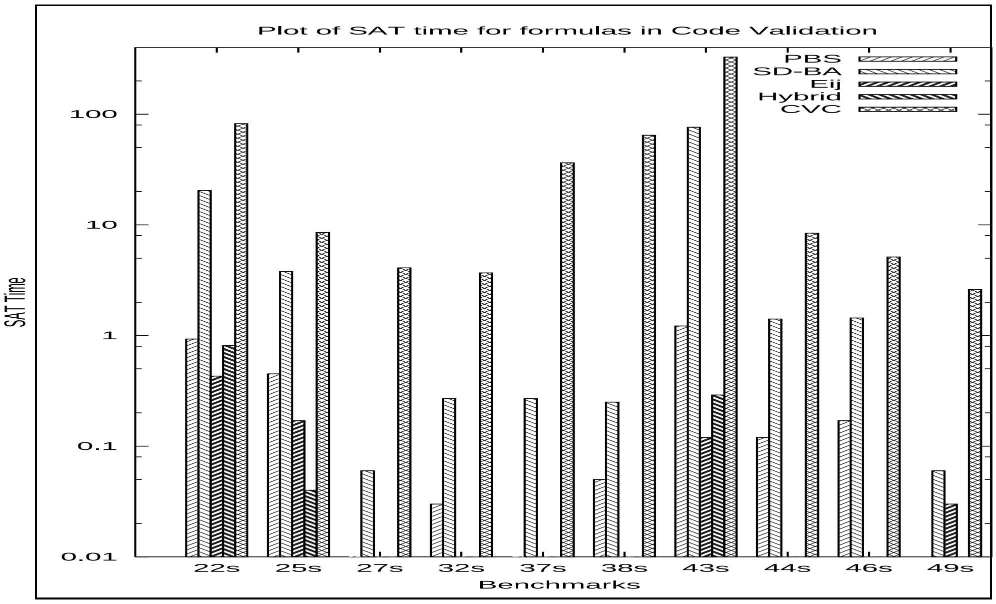

Fig. 7. Results for Code Validation Benchmarks.

Boolean variables t_1 and t_2 that violates a transitivity constraint. The only way the violation can be detected is by searching through the space of assignments to the pseudo-Boolean variables appearing in the constraints encoded by t_1 and t_2. On the contrary, for the EIJ method, since transitivity constraints occur as CNF clauses, if the assignment to two Boolean variables violates a transitivity constraint, it is relatively easier to detect the violation and quickly backtrack from it. This appears to be the reason why EIJ performs better than PBS on benchmarks where the number of transitivity constraints is not very large.

The Hybrid method combines EIJ with SD-BA to avoid exponential growth in transitivity constraints. Hence, it performs the best on all benchmarks except the invariant checking ones, where the ITE elimination stage produces a huge blowup in the formula size. Also, CVC does not perform well on any of the benchmarks, possibly because the overhead of lazily enforcing transitivity constraints is significantly large.

Finally, we present a classification of the methods analyzed in this section. The classification is performed along two axes – the method of Boolean encoding, and whether transitivity constraints are enforced lazily or eagerly – and is summarized in Figure 13. The EIJ method encodes transitivity constraints explicitly and encodes each separation constraint using a single Boolean variable. The PBS method lazily (implicitly) handles transitivity constraints, and uses a mixture of small domain encoding and per-constraint Boolean variables, where the small domain variables get utilized in detecting the violation of a transitivity constraint. The SD-BA method uses a pure small-domain encoding which enforces transitiv-

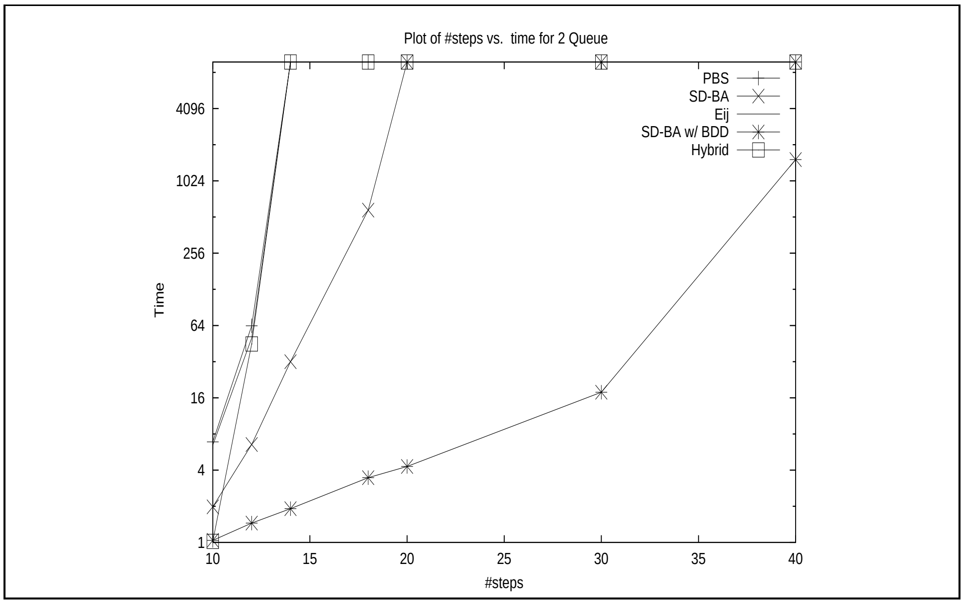

Fig. 8. Results for 2 Queue Example.

ity constraints by construction. The Hybrid method eagerly enforces transivity constraints, and uses a combination of small domain and per-constraint encoding. Finally, CVC uses only per-constraint encoding and adopts a lazy approach to enforcing transitivity constraints.

6 Conclusions

We have presented three different ways to translate a separation logic formula to a Boolean formula, along with empirical results on benchmark formulas generated from UCLID models. The results indicate that the relative performance is dependent on the characteristics of the benchmark. EIJ and PBS methods work well when the overheads of ITE elimination and transitivity constraints are small; otherwise, SD-BA works well. The Hybrid method tries to combine the best of the EIJ and SD-BA methods. All these methods work better than the general purpose checker CVC.

There is scope for improvement in the EIJ method as the number of transitivity constraints can be minimized by a careful analysis of the Boolean connectives in the formula [10]. The PBS solver is still in an early prototype stage, and further refinement could improve its performance.

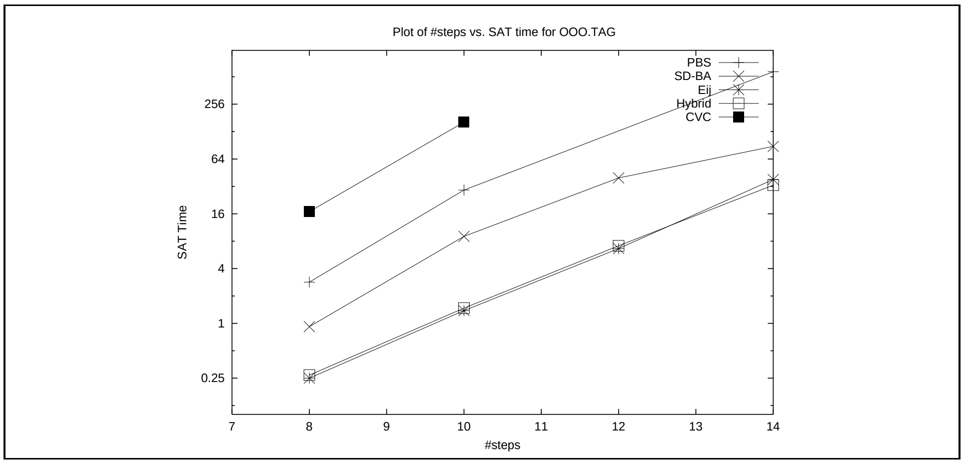

Fig. 9. Results for OOO tag Benchmarks.

Acknowledgments

We thank Ofer Strichman for providing the decision procedure for separation logic based on the per-constraint Boolean variable encoding and also for the code validation benchmarks. We also thank Fadi Aloul for providing us with the PBS solver. The cache coherence protocol was provided by Steven German.

References

1. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo-boolean solver. In *Proc. Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, pages 346–353, 2002.
2. C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference on Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 236–249. Springer-Verlag, July 2002.
3. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
4. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV '99)*, LNCS 1633, pages 470–482. Springer-Verlag, July 1999.
5. R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.
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 E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference*

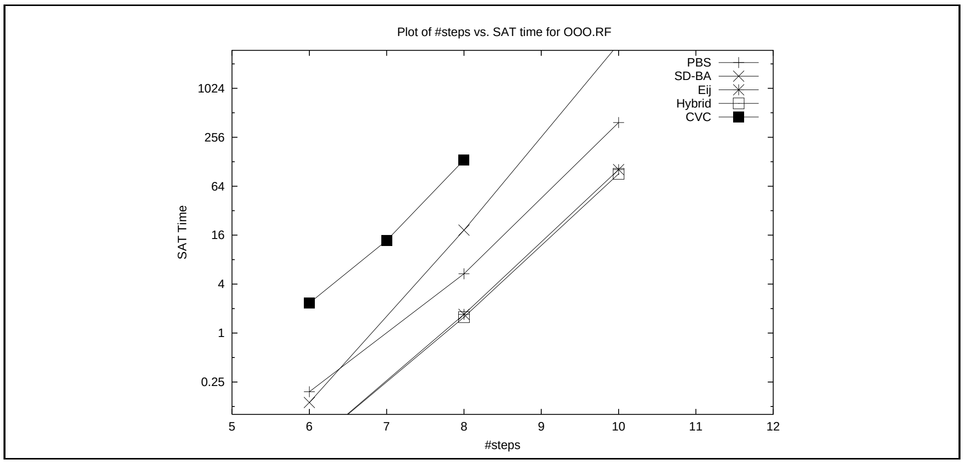

Fig. 10. Results for OOO Rf Benchmarks.

Model	File	# vars	# pvars	PBS Time	SD-BA Time	Hybrid Time	EIJ Time	EIJ # sep	EIJ # tr-constr	CVC
Cache	c1	53	1	*	163.04	855.84	*	1197	170904	n/a
coherence	c2	116	1	*	3.34	15.64	*	2235	> 1000000	n/a
protocol	c3	78	1	*	3	7.26	*	929	221872	n/a
	c4	144	1	*	1417.78	1727.62	*	3854	> 1000000	n/a
	c5	135	1	*	86.28	720.08	*	3074	> 1000000	n/a
Out-	o1	229	19	*	28.45	107.36	*	1310	97310	n/a
-of-	o2	294	25	*	300.78	*	*	2080	175896	n/a
-order	o3	258	20	*	398.74	1228,4	*	1566	111577	n/a
unit	o4	315	36	*	277.78	*	*	2355	215061	n/a
	o5	159	12	*	231.62	*	*	703	33850	n/a

Fig. 11. Results from invariant checking benchmarks. "vars" is the number of integer symbolic constants in the CLU formula after eliminating function applications, and "p-vars" is number of those symbolic constants that correspond to p-function applications. The 3 columns correspond to the time taken by the 3 methods. For EIJ method, "sep" represents the number of separation predicates in the formula and "tr-constr" refers to the number of transitivity constraints for this encoding. A "*" means that the run did not finish.

on Computer-Aided Verification (CAV'02), LNCS 2404, pages 78–92. Springer-Verlag, July 2002.

7. R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. In A. Emerson and P. Sistla, editors, *Computer-Aided Verification (CAV 2000)*, LNCS 1855. Springer-Verlag, July 2000.
8. Shuvendu Lahiri, Sanjit A. Seshia, and Randal E. Bryant. Modeling and verification of out-of-order microprocessors using uclid. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, (to appear), November 2002.
9. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domain instantiations. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 455–469. Springer-Verlag, July 1999.
10. O. Strichman. Optimizations in decision procedures for propositional linear inequalities. Technical Report CMU-CS-02-133, Carnegie Mellon University, 2002.

Benchmark	# of steps	# of sep	# of tr-constr
Elf	6	2	3
	7	25	70
	8	80	813
	9	253	14352
	10	579	$>$ 1000000
OOO.Rf	6	10	52
	8	31	459
	10	64	1606
OOO.Tag	8	31	459
	10	64	1606
	12	109	3877
	14	166	7656
Cache	6	0	0
	8	0	0
	10	0	0
	12	0	0
dlx1	-	40	72
dlx2	-	40	72
dlx3	-	40	72

Fig. 12. Transitivity constraints for EIJ. "steps" is the number of steps of symbolic simulation, "sep" represents the number of separation predicates in the formula and "tr-constr" refers to the number of transitivity constraints for this encoding.

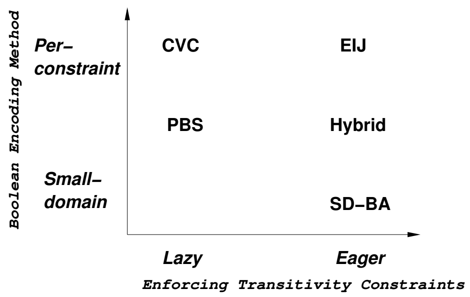

Fig. 13. A classification of decision procedures.

11. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference on Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 209–222. Springer-Verlag, July 2002.