A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions [*]

Sanjit A. Seshia sanjit.seshia@cs.cmu.edu
Shuvendu K. Lahiri shuvendu@ece.cmu.edu
Randal E. Bryant randy.bryant@cs.cmu.edu

School of Computer Science & Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213

ABSTRACT

SAT-based decision procedures for quantifier-free fragments of first-order logic have proved to be useful in formal verification. These decision procedures are either based on encoding atomic subformulas with Boolean variables, or by encoding integer variables as bit-vectors. Based on evaluating these two encoding methods on a diverse set of hardware and software benchmarks, we conclude that neither method is robust to variations in formula characteristics. We therefore propose a new hybrid technique that combines the two methods. We give experimental results showing that the hybrid method can significantly outperform either approach as well as other decision procedures.

Categories and Subject Descriptors

I.1 [**Symbolic and Algebraic Manipulation**]: Expressions and Their Representation, Algorithms; I.2.3 [**Deduction and Theorem Proving**]; F.4.1 [**Mathematical Logic**]: Mechanical theorem proving

General Terms

Algorithms, Experimentation, Measurement, Verification

Keywords

Design verification, Decision procedures, Boolean satisfiability, Theorem proving.

1. INTRODUCTION

Quantifier-free fragments of first-order logic and decision procedures for them have become commonplace in many formal verification efforts. Decision procedures for the Logic of Equality with Uninterpreted Functions(EUF) have been successfully used in the automated verification of pipelined processor designs [8, 4]. Predicate abstraction methods [9] based on decision procedures have been used to verify parameterized cache coherence protocols. The logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions [6] (CLU), which generalizes EUF, is the basis for the UCLID verifier which has been used for bounded model checking and inductive invariant checking of out-of-order microprocessors with unbounded resources [11]. Decision procedures are an integral part of software verification systems including the Code Validation tool [13] and the Blast verifier [10]. Hence the importance of having efficient decision procedures can hardly be overstated.

Many decision procedures leverage off the recent advances in Boolean satisfiability (SAT) solvers. These decision procedures differ in the Boolean encoding and the degree of the coupling with the SAT solver. One can classify these procedures as being either *eager* or *lazy*. In the eager approaches [4, 6, 12, 14], the quantifier-free first-order formula is converted in a single step to an equivalent Boolean formula which is checked using the SAT solver. The lazy approaches (e.g., [3, 1]) iteratively refine the Boolean encoding based on satisfying assignments from the SAT solver that are inconsistent with the first order theory. The process is repeated until a consistent assignment is found or all assignments are explored.

We consider the problem of deciding formulas in the logic of Separation Predicates and Uninterpreted Functions (SUF), which can express properties of systems modeled in CLU logic. For decision procedures using the eager approach, two Boolean encoding methods have been proposed: *small-domain encoding* (also called *finite instantiation*) [6], and *per-constraint encoding* [14]. In small-domain encoding, each integer variable is interpreted over a finite set of values that is determined by analyzing the formula. On the other hand, in per-constraint encoding, each atomic subformula is encoded as a Boolean variable. Constraints are then added to prune off the inconsistent assignments of the Boolean variables.

In this paper, we confirm results of our previous experimental evaluation [5] of these two encoding methods, concluding that neither of the approaches are robust to variations in formula characteristics. Hence, we propose a new *hybrid* method that combines the two methods, and show that it can be more robust than either approach. Finally, we report empirical results comparing our hybrid method with other decision procedures for SUF logic.

2. BACKGROUND

Figure 1 summarizes the expression syntax for SUF logic. Expressions can be of two types: integer or Boolean. Boolean expressions are formed by combining equalities, inequalities, or applications of uninterpreted predicates using Boolean connectives. Integer expressions are formed by applying an uninterpreted function to a list of integer expressions, by applying the unary arithmetic functions

[*]This work was supported by ARO grant DAAD19-01-1-0485 and SRC contract 1029.001

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2003, June 2–6, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-688-9/03/0001 ...$5.00.

succ ("+1") and **pred** ("−1"), or by applying the *ITE* ("if-then-else") operator. 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*.

$$
\begin{array}{rcl}
\textit{bool-expr} & ::= & \textbf{true} \mid \textbf{false} \mid \neg \textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \\
 & & \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\
 & & \mid \textit{predicate-symbol}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{int-expr} & ::= & \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
 & & \mid \textbf{succ}(\textit{int-expr}) \mid \textbf{pred}(\textit{int-expr}) \\
 & & \mid \textit{function-symbol}(\textit{int-expr}, \ldots, \textit{int-expr})
\end{array}
$$

Figure 1: SUF Syntax.

Uninterpreted functions and predicates are used in hardware design verification to abstract word-level values of data and implementation details of functional blocks. These functions and predicates satisfy no particular property other than *functional consistency*, viz., that they evaluate to the same value on the same arguments. Adding equalities, inequalities, and interpreted functions **succ** and **pred** facilitates reasoning about ordering and equality of data values, and modeling ordered data structures such as queues.

2.1 Decision Procedure Overview

Assume we start with a well-formed formula F_{suf} in SUF. The decision procedure must determine whether it is *valid*, i.e., true under all possible interpretations of the function and predicate symbols. Through a sequence of validity-preserving transformations, we convert a SUF formula to a propositional formula F_{bool}. Validity of F_{suf} is then checked by checking the (un)satisfiability of $\neg F_{bool}$.

2.1.1 Eliminating Function Applications

The first step is to eliminate applications of uninterpreted function and predicate symbols of arity greater than one. In doing so, we keep track of a property of function applications called *positive equality*. The general idea is to determine the polarity of each equality in the formula, i.e., whether it appears under an even (positive) or odd (negative) number of negations. Applications of uninterpreted functions can then be classified as either p-function applications, i.e., used only under positive equalities, or g-function applications, i.e., general function applications that appear under other equalities or under inequalities. The p-function applications can be encoded in propositional logic with fewer Boolean variables than the g-function applications, thus greatly simplifying the resulting SAT problem.

To be able to exploit positive equality, Bryant et al. eliminate function applications using "*ITE*" expressions. As an example, if function symbol f has two occurrences: $f(a_1)$ and $f(a_2)$, then we would generate 2 new symbolic constants vf_1 and vf_2. We would then replace all instances of $f(a_1)$ by vf_1 and all instances of $f(a_2)$ by $ITE(a_2 = a_1, vf_1, vf_2)$. Note that this encoding ensures functional consistency. Predicate applications are eliminated using a similar process.

2.1.2 Deciding Separation Logic

Eliminating function and predicate applications leaves us with a formula F_{sep} containing only symbolic constants, ITEs, successors, predecessors, equations, inequalities, and Boolean connectives. We will refer to this logic as *separation logic*. In addition, we will refer to equalities or inequalities involving only symbolic constants, successors, and predecessors, as *separation predicates*.

The problem of deciding the satisfiability of a formula in separation logic is NP complete. It is NP-hard since Boolean satisfiability can be trivially reduced to it. In addition, it is in NP because the logic has a "small-model" property, viz., if it is satisfiable, there exists a satisfying assignment such that the values of the integer symbolic constants in the formula are polynomially bounded in the size of the formula.

As mentioned in Section 1, there are two approaches to encoding a formula in separation logic as a Boolean formula:

1. **Small-Domain Encoding (SD):** For each symbolic constant v_i in F_{sep}, we compute the bound b_i on its value. The bounds b_i are then used to derive an encoding of each v_i as a "symbolic bit-vector" of Boolean variables. This results in a formula with only Boolean variables, along with ITEs, successors, predecessors, equations, inequalities, and Boolean connectives. Binary arithmetic is then used to transform this formula into a Boolean formula F_{bool}.

2. **Per-Constraint Encoding (EIJ):** In this method, we first eliminate the ITE construct from the formula, to get a formula that is a Boolean combination of separation predicates. Each separation predicate is then encoded with a fresh Boolean variable to get a Boolean formula F_{bvar}. To rule out assignments to these Boolean variables that do not correspond to any assignment to the integer-valued symbolic constants in F_{sep}, we construct a Boolean formula F_{trans} that imposes "transitivity constraints" on the values of the fresh Boolean variables. For example, if $e^{=,0}_{x,y}$ encodes $x = y$, and $e^{=,0}_{y,z}$ encodes $y = z$, we would impose the constraint $e^{=,0}_{x,y} \wedge e^{=,0}_{y,z} \implies e^{=,0}_{x,z}$. This process might, in general, result in new Boolean variables, such as $e^{=,0}_{x,z}$, being generated. Let F_{trans} denote the conjunction of all the imposed constraints. We finally construct F_{bool} as $F_{trans} \implies F_{bvar}$.

For example, if F_{sep} is $x \geq y \wedge y \geq z \wedge z \geq \textbf{succ}(x)$, then using SD, F_{bool} is the Boolean formula

$$[\langle 0x_1x_0\rangle \geq \langle 0y_1y_0\rangle] \wedge [\langle 0y_1y_0\rangle \geq \langle 0z_1z_0\rangle] \wedge [\langle 0z_1z_0\rangle \geq \langle 0x_1x_0\rangle + 1]$$

where 2 Boolean variables suffice to encode each symbolic constant, and the arithmetic and relational operators are re-interpreted over bit-vectors of length 3. Using EIJ, F_{bool} is

$$[(e^{\geq,0}_{x,y} \wedge e^{\geq,0}_{y,z} \implies e^{\geq,0}_{x,z}) \wedge (e^{\geq,0}_{x,z} \implies \neg e^{\geq,1}_{z,x})] \implies [e^{\geq,0}_{x,y} \wedge e^{\geq,0}_{y,z} \wedge e^{\geq,1}_{z,x}]$$

where the antecedent of the outermost implication is F_{trans} and the consequent is F_{bvar}.

Each of SD and EIJ has its drawbacks when used in isolation. We describe these drawbacks in the next section.

3. COMPARISON OF SD AND EIJ

The method of encoding a SUF formula using SD usually results in a compact Boolean formula. As shown in [6], if there are N distinct nodes in the Directed Acyclic Graph (DAG) representation of a SUF formula F_{suf}, then the size of the DAG for the resulting Boolean formula is roughly *O(Nlg(N))*. However, since numerical constants are encoded in unary (e.g., $x+k$ as $\textbf{succ}^k(x)$), N can be very large if the formula has very large constants. However, in our experience, formulas rarely have very large constants. The EIJ method, on the other hand, can produce a Boolean formula whose size is exponential in N. This is because there is a potential of generating an exponential number of transitivity constraints using this method [14]. Note, however, that for the subclass of logic involving only equalities without arithmetic, the number of transitivity constraints grows only polynomially in the number of equalities in the original formula [7].

The SD and EIJ methods also differ in their impact on search in SAT solvers. As the SD method uses a vector of Boolean variables to encode each symbolic constant, sufficient number of these Boolean variables have to be assigned values before the search space of solutions is pruned. On the other hand, in the EIJ method, each separation predicate (e.g., $x \geq y + c$) is encoded using a single Boolean variable (e.g., $e^{\geq,c}_{x,y}$). This encoding appears to assist the DPLL-based SAT solvers, as has been observed by Velev and Bryant [15] for deciding the restricted class of EUF logic. This is probably because, in many formulas, separation predicates 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.

In previous work [5], we performed an experimental study comparing SD and EIJ along with other variants of a decision procedure for SUF. The results indicated that neither SD nor EIJ performed consistently well over all benchmarks used in the study. However, it was noted that they appear to complement each other, suggesting that they might perform well in combination. In the same paper, we presented a hybrid technique that encodes equalities without arithmetic using EIJ and the remaining separation predicates using SD; however, this combination met with limited success as the decision of whether to choose SD or EIJ is fixed, i.e., it does not depend on features of the formula being checked.

In the rest of this section, we report on new experiments that confirm our previous findings comparing SD and EIJ and identify features of input formulas that affect the relative performance of SD and EIJ.

Benchmarks. For use in all the experiments reported in this paper, we collected a set of 49 valid SUF formulas to use as benchmarks, many of which were used in our previous study [5], and are described in detail in that paper. We drew these formulas from real problems encountered in both hardware and software design verification. The hardware designs include the load-store unit of an industrial microprocessor, an out-of-order microprocessor design [11], a cache coherence protocol, and a 5-stage DLX pipeline. The software benchmarks are generated in the verification of safety properties of device driver code [10], and in translation validation [13]. The sizes of the formulas, when measured in terms of the number of nodes in a DAG representation, range from 100 nodes to 7500 nodes.

For the experiments described in this section, we selected a sample of 16 formulas at random from the above benchmark set such that there was at least 1 formula from each problem domain. Experiments were run on a Pentium-IV 2 GHz machine with 1 GB of RAM.

Impact on SAT solver. We first performed an experiment to study how the encoding method affects the performance of the SAT solver. Figure 2 shows results on 5 of the larger benchmarks from the sample of size 16. The numbers in the table support our conjecture that the EIJ encoding is more effective with SAT solvers. Although the original number of clauses are far more in the case of EIJ method (due to transitivity constraints), the number of conflict clauses needed to span the entire search space is far less for EIJ encoding. The lower number of conflict clauses added for EIJ method suggests better pruning of the search space through case splitting on the Boolean variables encoding separation constraints. This directly translates to significant improvement in the run time of the SAT solver (zChaff).

Benchmark	# of CNF Clauses		# of Conflict Clauses		SAT Time (sec)	
	SD	EIJ	SD	EIJ	SD	EIJ
22s	67699	169387	15811	150	21.63	0.56
25s	67528	104941	12304	95	9.71	0.29
elf.r8	30556	30889	1807	77	0.89	0.06
ooo.t12	85441	116482	20364	2937	31.83	5.49
dlx	27469	37033	11358	745	5.06	0.43

Figure 2: Effect of Encoding on zChaff performance. "Conflict Clauses" denotes the conflict clauses added by zChaff.

Combining SD and EIJ based on Formula Features. From the preceding discussion and experimental results, it appears that EIJ would be a better encoding method except in cases where the number of transitivity constraints grows too large, slowing down the translation procedure. Thus, a decision rule that chooses between the EIJ and SD methods of encoding would ideally be based on an estimate of the the number of transitivity constraints. Unfortunately, this number is hard to estimate in polynomial time, since it corresponds to the problem of estimating the number of cycles in a directed graph. We therefore performed a set of experiments to identify other formula features that are closely correlated with the run-time of the SD and EIJ methods. We started with a candidate set of several features, including (1) the number of separation predicates in the formula obtained after eliminating function applications, (2) the maximum over the sizes of the small-model required for each symbolic constant, (3) the fraction of function-applications that were p-function applications, and (4) the sum of the sizes of the small-models. The intuition was, for example, to use the number of separation predicates as an estimate of complexity of the EIJ method, and the small-model size as an estimate for the SD method.

For each of the 16 benchmarks, we measured CPU run-time, normalized by the size of the original SUF formula. The normalization allows us to study how each formula feature affects the run-time by removing the effect of formula size, yielding similar normalized run-times for formulas with similar features. The CPU run-times have two components: the time taken to translate the formula to a Boolean formula, and the time taken by the SAT solver to solve that resulting Boolean formula.

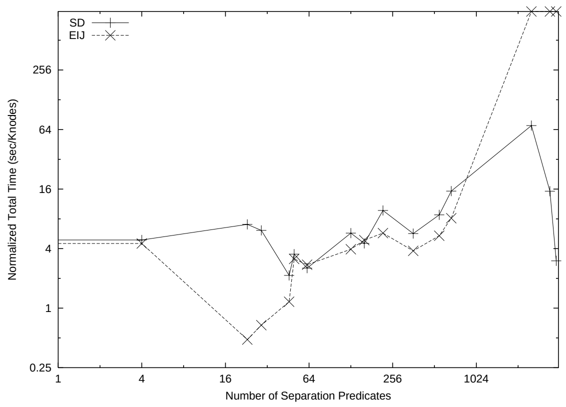

Figure 3: Effect of number of separation predicates on EIJ. Notice that both axes are on a log scale.

We found that the only feature that showed any correlation with run-time was the number of separation predicates. Specifically, the run-times of the EIJ method show good correlation with the number of separation predicates, as shown in Figure 3. When the number of separation predicates is low, the EIJ method outperforms the SD method, due to faster SAT times. But as the number increases, its performance gets worse, until beyond a certain threshold, the EIJ method fails to go beyond the formula translation stage (and was timed out after an hour). In Figure 3, this occurs for the three benchmarks with the largest number of separation predicates. On the contrary, the small-domain method completes within 6 minutes on all these three benchmarks.

The results shown in Figure 3 indicate a natural heuristic for choosing between SD and EIJ, viz., checking if the number of separation predicates is greater than a threshold value SEP_THOLD. If the number exceeds the threshold, we can use the SD encoding method, otherwise, we use EIJ. In the next section, we show how we can use this heuristic to develop a new *hybrid* method of encoding the separation logic formula as a Boolean formula.

4. HYBRID METHOD

A naive scheme based on the results of the previous section might use the number of separation predicates to select either SD or EIJ for encoding the entire formula F_{sep}. However, we can exploit the formula structure to encode different parts of the formula using different methods. The main idea is to group symbolic constants into classes, and use a different encoding method for each class. To do this, we define two symbolic constants to be in the same class if they are compared to each other using an equality or an inequality. Symbolic constants appearing in F_{sep} are thus partitioned into equivalence classes, where those in one equivalence class can be encoded completely independent of symbolic constants in other equivalence classes. For each class, the choice of encoding is based on the number of separation predicates which involve two symbolic constants from that class. In the rest of this section, we describe the steps in the hybrid encoding method, viz., how to compute the equivalence classes, estimate the number of separation predicates for each class, and then finally use the estimate to encode the formula using a combination of both schemes.

1. Generate Classes. We first collect the set of symbolic constants occurring in F_{sep} into two sets V_p and V_g. V_p consists of those symbolic constants occurring in F_{suf} 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 start by assigning each symbolic constant in V_g to its own class. We then compute the *dependency* set for each term in F_{sep}, denoting some subset of symbolic constants in V_g to which this term could evaluate. While doing this, we merge some of the classes so that each dependency set is a subset of some class. For term $T \doteq v$, its dependency set is $\emptyset$ if $v \in V_p$ and is $\{v\}$ if $v \in V_g$. For term $T \doteq \mathbf{succ}(T_1)$, its dependency set is the same as that of T_1. Similarly for $T \doteq \mathbf{pred}(T_1)$. For $T \doteq ITE(F, T_1, T_2)$, its dependency set is the union of those of T_1 and T_2. If the dependency sets of T_1 and T_2 are subsets of two distinct classes, then we merge those classes. For each equation $T_1 = T_2$ and each inequality $T_1 < T_2$, we perform a similar merging if the dependency sets of T_1 and T_2 are subsets of distinct classes. Let $V_1, \ldots, V_K$ be the K different symbolic constant classes generated by this procedure. It is easy to see that none of the symbolic constants in V_p belong to any of the classes $V_1, \ldots, V_K$.

2. Generate ground terms. A *ground term* is an expression of the form $\mathbf{succ}^k(v)$ or $\mathbf{pred}^k(v)$, where k is a non-negative integer and v is a symbolic constant. We transform the formula to generate ground terms by repeatedly applying the following rewrite rules until a fixed point is reached.

$$\mathbf{succ}(\mathbf{pred}(T)) \rightarrow T \qquad \mathbf{pred}(\mathbf{succ}(T)) \rightarrow T$$
$$\mathbf{succ}(ITE(F, T_1, T_2)) \rightarrow ITE(F, \mathbf{succ}(T_1), \mathbf{succ}(T_2))$$
$$\mathbf{pred}(ITE(F, T_1, T_2)) \rightarrow ITE(F, \mathbf{pred}(T_1), \mathbf{pred}(T_2))$$

At this point, the terms at the leaves consist of only ground terms. For brevity, from now on we will simply write $v + k$ (respectively $v - k$) for $\mathbf{succ}^k(v)$ (resp. $\mathbf{pred}^k(v)$).

3. Compute Domain Sizes. For each symbolic constant v, we compute $u(v)$, the maximum amount it can be incremented and $l(v)$, the maximum amount it can be decremented. $u(v)$ represents the maximum offset of v in any of the ground terms present in the formula. Similarly, $l(v)$ represents the minimum offset. For example, if $\{v-4, v-2, v, v+3, v+7\}$ is the set of ground terms in which v appears, then $u(v) = 7$, and $l(v) = -4$. For each class V_i we compute its range as:

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

This determines the size of the finite instantiation we need to consider for each symbolic constant in V_i for SD encoding. Interpreting the symbolic constants over bit vectors of this size is sufficient to determine the validity of the separation logic formula (due to the small-model property).

4. Compute an upper bound on the number of separation predicates for each class. For each of the classes V_i, estimate $SepCnt(V_i)$, the maximum number of separation predicates which would involve two symbolic constants from the class V_i. This is done as follows. Initially, $SepCnt(V_i) = 0$, for each class V_i. Then, for each equation $T_1 = T_2$ and each inequality $T_1 < T_2$, we find the set of ground terms $G(T_1)$ and $G(T_2)$ that T_1 and T_2 can evaluate to, respectively. This is done by modifying the algorithm for computing the *dependency set* (described above) to include ground terms in addition to symbolic constants. Let V_l be the class containing symbolic constants from V_g that appear in either of $G(T_1)$ or $G(T_2)$. For every pair (t_1, t_2) in the cross product $G(T_1) \times G(T_2)$ that has not been encountered yet, and where t_1 and t_2 are distinct from each other, we increment $SepCnt(V_l)$ by 1.

At the end of the process $SepCnt(V_i)$ gives an *upper bound* on the number of separation predicates that would involve two symbolic constants from the class V_i.[1]

5. Perform hybrid encoding. At this point, we have all the information we need to encode the separation logic formula into a Boolean formula. As discussed in Section 3, we use the threshold value SEP_THOLD to avoid encoding classes that have a very large number of separation predicates with the EIJ method.

The algorithm proceeds by recursing on the formula structure. A symbolic Boolean constant retains the same encoding. For a node $f_1 \wedge f_2$, we recursively encode the subexpressions f_1, f_2 and conjoin the results. Similarly, $\neg f_1$ is evaluated by encoding f_1 and negating the result. The more interesting cases involve equation or inequalities.

For each equation $T_1 = T_2$ or an inequality $T_1 < T_2$, we find the class V_{T_1} which contains the symbolic constants which appear in $G(T_1)$ and $G(T_2)$.

If $SepCnt(V_{T_1}) >$ SEP_THOLD, then we encode T_1, T_2 using SD method. The encodings of T_1 and T_2 are symbolic bit-vectors. Bitwise equality or comparison is used to translate these bit-vectors to a Boolean expression. Each symbolic constant is encoded with

[1]$SepCnt(V_i)$ is an upper bound because we count predicates that disappear after eliminating ITEs, e.g., counting (v_1, v_2) at the node $ITE(F, v_1, v_2) = ITE(\neg F, v_2, v_1)$.

a vector of symbolic Boolean constants. The size of the vector is determined by the domain size computed previously. Note that symbolic constants in V_p are assigned distinct bit-string values. The arithmetic operations **succ** and **pred** are encoded using binary arithmetic, and ITE expressions are encoded as multiplexors.

Otherwise, we use the EIJ method to encode T_1 and T_2, using the technique proposed by Bryant et al. [4]. Suppose T_1 evaluates to a ground term g_i under the condition c_i^1, and T_2 evaluates to g_j under c_j^2. For example, the term $ITE(F, v_1, v_2)$ evaluates to v_2 under $\neg F$. The encoding of the predicate $T_1 \bowtie T_2$, where $\bowtie \in \{=, <\}$, is given by $\bigvee_{i,j} c_i^1 \wedge c_j^2 \wedge e^{\bowtie}_{g_i,g_j}$, where $e^{\bowtie}_{g_i,g_j}$ is a symbolic Boolean constant to encode the constraint $g_i \bowtie g_j$. Note that if either of g_i or g_j contains a symbolic constant from V_p, then $e^{\bowtie}_{g_i,g_j} = \textbf{false}$.

6. Generate F_{bool}. Let F_{bvar} denote the formula obtained by performing the hybrid encoding on F_{sep}. We generate the conjunction F_{trans} of all transitivity constraints for predicates in F_{sep} encoded using the EIJ method. The final Boolean formula F_{bool} is then generated as ($F_{trans} \implies F_{bvar}$).

Hereafter we will denote our hybrid encoding method by HYBRID.

4.1 Automatically Selecting a Threshold

The value of the threshold parameter SEP_THOLD determines the mix of SD and EIJ in HYBRID, and hence determines its overall efficiency. For example, when SEP_THOLD = 0, HYBRID is the same as SD. While it might be acceptable to manually set the value of SEP_THOLD for some formulas, it is highly desirable to find a good default value.

We select a default value of SEP_THOLD by statistical analysis of the results of running EIJ on a sample of the benchmarks. As a sample, we selected the same 16 benchmarks that we used in Section 3 to find the correlation of EIJ with the number of separation predicates. Given the sorted sequence $T_1, \ldots, T_{16}$ of the normalized run-times of EIJ on these benchmarks, we calculated the value of k, $1 \le k \le 16$, that minimized the sum of the variances of each the two subsets $\{T_1, \ldots, T_k\}$ and $\{T_{k+1}, \ldots, T_{16}\}$. Dividing points into subsets in this manner is a well-known technique for clustering points in one dimension, using squared distance as a measure of similarity. Having performed this clustering, we chose the default SEP_THOLD as the smallest multiple of 100 greater than n_k, the number of separation predicates corresponding to the run-time T_k. For our sample set, we calculated n_k to be 676, so we set the default value of SEP_THOLD to be 700 for all experiments reported in Section 5.

In general, a user can determine a default SEP_THOLD by using a similar statistical technique on all formulas from a relevant domain that the tool has been run on in the past.

5. RESULTS & DISCUSSION

We implemented a decision procedure for SUF based on HYBRID in Moscow ML, a dialect of the Standard ML programming language. In this section, we describe empirical results comparing HYBRID with the SD and EIJ encoding methods. We also report comparisons with other decision procedures.

All experiments were run on a Pentium-IV 2 GHz machine with 1 GB of RAM running Linux. As a SAT engine, we used the zChaff SAT solver (version 2001.2.17) with the default options. A limit of 30 minutes was imposed on each run of a decision procedure. We used the entire set of 49 benchmarks described in Section 3, comparing HYBRID to SD, EIJ, and other decision procedures on the total time taken to decide each formula. For HYBRID, SD, and EIJ, this is the sum of the encoding time and the SAT time. Our results are depicted in Figures 4, 5, and 6. In each plot, the x-coordinate of each point is the time taken by HYBRID, and the y-coordinate is the time taken by the decision procedure we compare it against. We also plot the diagonal line $y = x$ in each plot. Thus, points above the diagonal correspond to benchmarks on which HYBRID outperforms the other procedures, while points below it correspond to benchmarks on which HYBRID is outperformed.

Comparing HYBRID to SD and EIJ. The results obtained from comparing HYBRID against SD and EIJ fall into two categories. The first set of results are on 39 benchmarks on which HYBRID with the default value of SEP_THOLD outperformed both SD and EIJ, and are depicted in Figure 4. These 39 benchmarks exclude formulas generated in proving correctness of hardware designs using invariant checking. On the remaining 10 benchmarks, SD outperformed both EIJ and HYBRID, for various values of SEP_THOLD greater than 100 including the default value; the results obtained by setting SEP_THOLD to 100 are shown in Figure 5.

Figure 4 shows that HYBRID scales better than EIJ and SD. We notice that SD and EIJ timed out after 30 minutes on some benchmarks, while HYBRID completed on all benchmarks. On several benchmarks that SD and EIJ completed on, HYBRID is a factor of 4 to 8 times faster. The only benchmarks on which SD and EIJ are marginally faster than HYBRID are small ones, all of which are decided within 16 seconds. The improvement of HYBRID over EIJ is due to reduction in the number of transitivity constraints, while the improvement over SD is due to reduced SAT time. On one instance, zChaff fails to decide the SAT problem generated by SD within the timeout limit of 30 minutes, while it finishes on the HYBRID encoding in about 90 seconds.

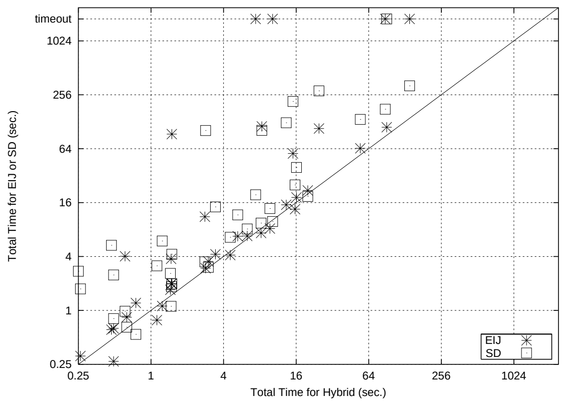

Figure 4: Comparing HYBRID against EIJ and SD. Non-invariant checking benchmarks were used. Note the log scales on both axes.

However, the results on the invariant checking formulas are very different. These formulas are characterized by many inequalities and a very large number of applications of uninterpreted functions, almost none of which are p-function applications. This results in a relatively small number of large-sized symbolic constant classes. Thus, even if the original number of separation predicates in each class is relatively low (e.g., less than SEP_THOLD), the number of symbolic constants involved in those predicates is large, and this leads to a large number of transitivity constraints. Hence, using the EIJ encoding method results in poor performance, whether in pure form or in HYBRID. We found that SD performs the best on these benchmarks. EIJ and HYBRID using the default value of SEP_THOLD fail to complete on any of them. Setting SEP_THOLD to 100, we found that HYBRID manages to complete on some benchmarks, but it is still outperformed by SD, as shown in Figure 5.

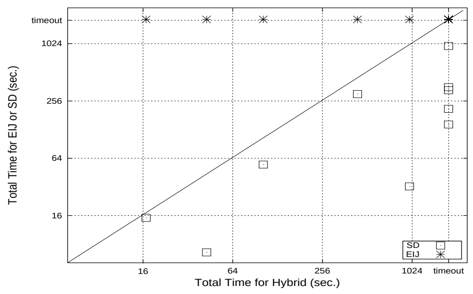

Figure 5: Comparing HYBRID **against SD and EIJ on invariant checking benchmarks.** SEP_THOLD **was set to 100.**

Comparison with Other Decision Procedures. We compared our decision procedure against existing decision procedures that can decide SUF formulas. One such procedure is the Stanford Validity Checker (SVC) [2]. SVC (version 1.1) can decide a superset of SUF, including, in addition, linear arithmetic and bit-vector arithmetic. We also compared against the more recent Cooperating Validity Checker (CVC) [3], which is SAT-based. In all experiments, we used HYBRID with the default value of 700 for SEP_THOLD. Since both SVC and CVC interpret function applications over the rational numbers, we could not run them on the invariant checking benchmarks which involve inequalities, and whose validity depends on the property that the integers are not dense. We therefore restricted our performance comparisons to the 39 non-invariant checking benchmarks.

Figure 6 compares HYBRID against both CVC and SVC. Let us first consider the comparison with SVC. We see that the formulas on which SVC outperforms HYBRID are relatively small, decided within 32 seconds. This is mainly due to the overhead of translating to SAT in HYBRID. Also many of these formulas are conjunctions of atomic predicates on which SVC performs well, since deciding a conjunction of separation predicates can be reduced to a shortest-path problem. However, for larger formulas involving several disjunctions, SVC's run-time quickly blows up, and it fails to finish within 30 minutes. HYBRID, on the other hand, completes within 3 minutes on all formulas.

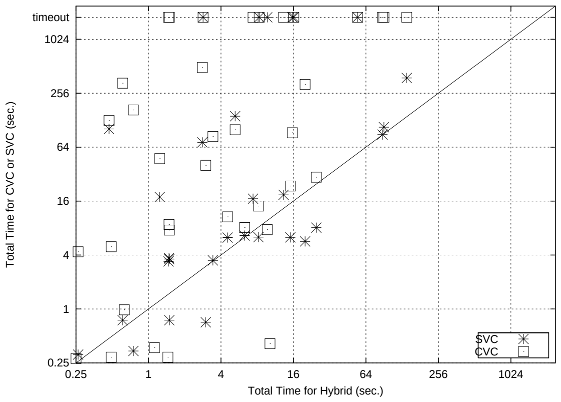

Figure 6: Comparing HYBRID **against SVC and CVC.**

Next, consider the comparison with CVC. Like EIJ, CVC works by replacing every equality or inequality with a fresh Boolean variable, and calling a SAT solver (a customized version of Chaff) on the Boolean encoding. However, unlike EIJ, transitivity constraints are enforced lazily, based on adding conflict clauses to rule out spurious satisfying assignments from the SAT solver. Each iteration of adding conflict clauses requires a call to a first-order decision procedure. Our experiments indicate that for many benchmarks, the overhead of lazy enforcement of transitivity constraints is too high, and HYBRID outperforms CVC by orders of magnitude. The exceptional cases where CVC outperforms HYBRID are conjunctions that are decided by both methods within 16 seconds. We believe this is because CVC tries to add conflict clauses that involve the smallest possible subset of literals from the satisfying assignment, and for a conjunction, this clause enforces transitivity constraints without need for further iteration.

Conclusions. The hybrid encoding method presented in this paper combines the complementary strengths of the small-domain and per-constraint encodings. Even with the default value of threshold, the method is robust to variations in formula characteristics, performing well on the majority of the benchmarks, and scaling better than other decision procedures.

Acknowledgments

We are grateful to Steven German, Ranjit Jhala, and Ofer Strichman for providing us with benchmarks.

6. REFERENCES

[1] G. Audemard, P. Bertoli, A. Cimatti, A. Korniowicz, and R. Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *CADE'02*, volume LNCS 2392, pages 195–210.

[2] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *FMCAD'96*, LNCS 1166, pages 187–201.

[3] C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *CAV'02*, LNCS 2404, pages 236–249.

[4] 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.

[5] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Deciding CLU logic formulas via Boolean and pseudo-Boolean encodings. In *Constraints in Formal Verification CFV'02*.

[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 *CAV'02*, LNCS 2404, pages 78–92.

[7] R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. In *CAV '00*, LNCS 1855, pages 85–98.

[8] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *CAV '94*, LNCS 818, pages 68–80.

[9] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99*, LNCS 1633, pages 160–171.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV'02*, LNCS 2404, pages 526–538.

[11] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *FMCAD'02*, LNCS 2517, pages 142–160.

[12] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domain instantiations. In *CAV'99*, LNCS 1633, pages 455–469.

[13] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT) - automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2, 1998, pages 192–201.

[14] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *CAV'02*, LNCS 2404, pages 209–222.

[15] M. N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *DAC '01*, pages 226–231.