Simulation-Directed Invariant Mining for Software Verification*

Xueqi Cheng
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24061, USA
{xcheng, mhsiao}@vt.edu

Abstract
With the advance of SAT solvers, transforming a software program to a propositional formula has generated much interest for bounded model checking of software in recent years. However, reasoning at the Boolean level often may not be able to identify some key relations among the original high-level program variables. In this paper, we propose a novel framework that uses simulation-directed data mining in the original program to extract a set of high-level potential property invariants according to the dynamic execution data of the software. When these learned invariants are added as constraints to the bounded model checking instances of the software, they help to significantly reduce the search space. The simulation-directed invariant mining framework exhibits more flexibility compared to the conventional static program analysis approaches, and the experimental results showed that our approach can lead to up to an order of magnitude of speedup in software verification via bounded model checking.

1. Introduction
As software plays a progressively more important role in embedded systems, there has been an increased interest in applying model checking to software verification in recent years. However, the specific features of software, such as infinite-state, unbounded data types, etc., had made it a different and difficult task when compared to hardware designs.

Bounded model checking (BMC) is an automatic verification technique that checks the temporal logic property based on unfolding the system’s transition relation for a given finite depth $k$ together with a monitor that checks for violation of the given temporal property [1, 2]. By converting this problem instance into a propositional formula, conventional satisfiability solvers can be used. The advances in SAT solvers have made SAT-based BMC a scalable verification alternative to the BDD-based symbolic model checking, and it has gained wide acceptance for the automatic analysis of finite state systems, especially in the hardware community.

However, the SAT-based BMC has limitations, especially in the context of software program verification. One such limitation is during the process of conversion from a given program to one large propositional formula. When each variable is represented with a bit vector whose size is determined by its variable type, some high-level information such as variable correlations and program structure features that can be easily gained in high-level programming languages are lost. Subsequently, extra burden is placed on the SAT-solver to reason about such relations at the Boolean level. For example, in code segment (1), suppose the loop index $i$ is not changed within the loop body, it is quite obvious that property $x \geq y$ always holds at line $L_i$. However, it may not be evident when each variable has been converted into bit-vectors.

\[
\text{for } (i = 0; i < N; i++) 
\begin{align}
L_0 & : \ldots \\
L_1 & : x = y + i; \\
L_2 & : \ldots 
\end{align}
\]

In addition, since the resulting propositional formula is simply a static translation of the original program semantics, no run-time implications on the variable correlations and path feasibilities are available. For example, in code segment (2), suppose the operations between $L_0$ and $L_1$ on pointers $p$ and $q$ always result in $(p) \succ (q)$ at $L_i$, then, whenever $L_i; (i < 0)$ is evaluated to true, $x$ is always greater than 0 at $L_2$ due to the relations between $(p)$ and $(q)$, consequently, $L_i; (i > 0)$ would always be true, making the path $L_i; L_{i-1}; L_{i-2}; L_{i-3}; L_{i-4};$ infeasible. Intuitively, adding these implied properties to the propositional formula may help to further reduce the search space of model checking, thus improving the performance of SAT solving.

\[
\text{for } (i = 0; i < N; i++) 
\begin{align}
L_0 & : \text{int } x, \ y, \ z, \ \ast p, \ \ast q; \\
L_1 & : \ldots \\
L_2 & : x = (\ast p); (\ast q); \\
L_3 & : x = (\ast p) + (\ast q); \\
L_4 & : z = x - y; \\
L_5 & : \ldots \\
L_6 & : \text{if } (z > 0) \\
L_7 & : z = z + x; \\
L_8 & : \ldots 
\end{align}
\]

The intuitions from the simple examples above have also been suggested by studies [14]. They showed that most of the properties established during verification are either invariants or depending crucially on invariants. Therefore, how easy these invariants, even trivial ones, can be deduced becomes a key issue to the effectiveness of the automated formal verification. Motivated by the gap between the limitation of SAT-based BMC and the characteristics of automated verification, in this paper, we propose a novel framework for software verification that uses simulation-directed data mining to help uncover these hidden high-level relations among variables. Data mining is the process of analyzing and extracting useful but implicit knowledge from large amounts of data. Compared to the conventional data flow/static analysis [4, 13] approaches for constraints learning, data mining has the following advantages:

1) Straightforward. In contrast to static analysis, where approximations about the possible set of values at various points...
in a program are computed without actually executing it, data mining is directly based on the profiled data collected during actual program execution and it requires the learned property to be unconditionally true over the simulated data set. Therefore, data mining can directly reflect the actual program execution flow via more straightforward profiling and learning compared to the conservative computation in static analysis.

2) Flexible. Although in this paper, we focus on discovering both potential Single Linear Properties (SLP) and Complex Property Implications (CPI), it can be easily extended to any other formats of property without modifying the data repository being mined and with similar efforts, including complex relations between pointers, like \((^p)p\rangle(^q)g\) at \(L_1\) in code segment (2). However, in static analysis, different dataflow equations or models may need to be constructed for each different target property format, besides, not each model can be built and solved with the same level of difficulty.

In our simulation-directed invariant mining framework, we use an association rule mining [10, 11] approach to extract potential high-level property invariants according to the actual execution of the software programs. Optimization algorithms are then applied to minimize the number of properties by removing both redundant and false positives being discovered. As a result, the cost of identifying potential invariants is reduced as well. Finally, the verified invariants are added as constraints to the original program to help reduce the search space during model checking. Experimental results showed that our approach may lead to up to an order of magnitude of improvement on the performance of software bounded model checking.

The remainder of the paper is organized as follows. In Section 2, we give an overview on association rule mining. The architecture of our framework for software verification is presented in Section 3, followed by the discussion of detailed property mining and optimization algorithms in Section 4. Section 5 presents the results. A discussion of related work is given in Section 6. Finally, Section 7 concludes our findings.

2. Preliminaries

2.1 Basic Definitions of Association Rule

We applied a specific data mining technique — association rule mining for potential property discovery in our framework. Association rule mining provides a useful mechanism for discovering rules such as implication or correlations among items that co-occur frequently within a data set.

A data set usually consists of \(M\) independent transactions, each of which contains \(N_i\) items drawn from a set of all possible \(N\) items. Table 1 gives an example data set consisting of 4 transactions with 5 items \(\{I_1, I_2, I_3, I_4, I_5\}\), where the value ‘1’ means the occurrence of an item in a given transaction.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>(I_1)</th>
<th>(I_2)</th>
<th>(I_3)</th>
<th>(I_4)</th>
<th>(I_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

An association rule has the form of \(A \Rightarrow B\), where \(A\) and \(B\) are disjoint sets of items or itemsets. A \(k\)-itemset is a set containing \(k\) items. The strength of an association rule is usually measured by two metrics: support and confidence. The support is a fraction of transactions in the data set that contain both \(A\) and \(B\):

\[
support(A \Rightarrow B) = \frac{\text{Count}(A \cap B)}{M}
\]

where \(\text{Count}(x)\) represents the total number of transactions that satisfy condition \(x\). The confidence of an association rule is a measure of the reliability and accuracy of the rule.

\[
\text{confidence}(A \Rightarrow B) = \frac{\text{Count}(A \cap B)}{\text{Count}(A)}
\]

The association rule mining problem is that of computing all association rules that satisfy user-specified minimum support and minimum confidence threshold.

2.2 A-Priori Algorithm

A-Priori is a classic algorithm for mining association rules. It consists of two major steps: the first step is the frequent itemset generation by employing a level-wise search, where the frequent \((k+1)\)-itemsets are found using the computed frequent \(k\)-itemsets. Take the data set in Table 1 as an example, suppose the support threshold \(s_a = 0.5\), starting with iteration \(k=1\), we first compute the support value of each possible 1-itemset and those 1-itemsets whose support values reach \(s_a\) generate the frequent 1-itemset \(L_1 = \{I_1, I_3, I_4, I_5\}\) as shown in Table 2. Next, in the second iteration, \(C_2\), a set of candidate 2-itemsets is constructed by listing all pairs of itemsets from \(L_1\). Then, we calculate the occurrence of each candidate 2-itemset in \(C_2\) by scanning the data set in Table 1 and compute its support. Finally, \(C_2\) is pruned using support threshold \(s_a\) and forms the frequent 2-itemset \(L_2 = \{\{I_1, I_3\}, \{I_1, I_4\}, \{I_3, I_4\}\}\) as shown in Table 3. The above process is repeated until iteration index \(k\) reaches certain pre-defined value or the candidate sets become empty.

The second step of the A-Priori algorithm is the association rule derivation. Suppose the frequent itemset generation stops at iteration \(k\), for each frequent \(k\)-itemset \(f\) in \(L_k\), all of its non-empty proper subsets can be enumerated. Then for each such subset denoted as \(f_s\), a set of candidate association rules in the format of \(R: f_s \Rightarrow (f - f_s)\) is generated, \(R\) is outputted if it fulfills the minimum confidence requirement. For example, for frequent 2-itemsets \(\{I_1, I_4\}\), all of its non-empty proper subsets are \(\{I_1\}\) and \(\{I_4\}\), thus, \(I_1 \Rightarrow I_4\) and \(I_4 \Rightarrow I_1\) are two candidate association rules. Based on the earlier definition of the confidence, \(\text{confidence}(I_1 \Rightarrow I_4) = 2/2 = 1\), \(\text{confidence}(I_4 \Rightarrow I_1) = 2/3 \approx 0.67\). Suppose the confidence threshold \(c_{th} = 0.8\), then \(I_1 \Rightarrow I_4\) is an association rule for frequent 2-itemset \(\{I_1, I_4\}\).

Table 4 listed all the candidate association rules and the corresponding support and confidence values for the above frequent 2-itemsets \(L_2\). Given \(c_{th} = 0.8\), the resulting association rules are \(I_1 \Rightarrow I_4\) and \(I_1 \Rightarrow I_3\).
3. Simulation-Directed Invariant Mining Framework for Software Verification

The key idea behind our simulation-directed invariant mining framework is to discover and add appropriate property invariants (single as well as complex relations) to the original program, so that the performance of model checking can be improved when verifying an assertion.

Figure 1 shows the overall architecture of the framework. First, the original program under verification $P$ is analyzed to identify a set of related variables $\{v_1, \ldots, v_n\}$, which could be either control/data dependent on variables in a target assertion (if known) or user defined. Then $P$ is instrumented by adding profiling statements: record_value(linelnIndex, $v_1, \ldots, v_n$) at the locations of the uses and definitions of each related variable to record its value as well as the line number where it lies. Next, the annotated program is executed with randomly fed input values and the profiled values for related variables during execution are stored into a data set.

In our work, we focus on mining two kinds of invariants: (a) single linear property (SLP) that holds at a certain line of the program execution sequence, and (b) complex property implication (CPI) that represents relations among variables at different lines of the program.
a) Single Linear Property Mining: A single linear property has the format of \((L_1 = \lambda \Rightarrow x \cdot y)\), which means \(x \cdot y\) always holds at line \(\lambda\) of the program. According to the item definition in (3), a single linear property depicts the relation between the two elements in an item tuple; hence, the mining of a SLP invariant becomes a frequent 1-itemset mining. The support and confidence are computed as follows:

\[
support(L_1 = \lambda \Rightarrow x \cdot y) = \frac{\text{Count}(L_1 = \lambda \& x \cdot y)}{\text{Count}(\text{Trans.})}
\]

\[
\text{confidence}(L_1 = \lambda \Rightarrow x \cdot y) = \frac{\text{Count}(L_1 = \lambda \& x \cdot y)}{\text{Count}(L_1 = \lambda)}
\]

Therefore, all the items in the data set whose occurrence frequencies for both \(L_1 = \lambda\) and \(x \cdot y\) exceed a certain support threshold \(s_{th}\) would be considered as SLP candidates. However, since our goal is to first mine potential invariants before validating them, the confidence threshold \(c_{th}\) for association rule generation is set to be 1.0, which ensures the mined property occurs frequently enough during program execution, and always holds over the profiled data set. Take the code segment (1) as an example, \(\{L = L_1 \Rightarrow x \cdot y\}\) may be mined as a SLP if the item \(\{L = L_1, x \cdot y\}\) occurred frequently in the transaction database and the variable \(i\) is not changed within the for-loop body.

Due to the existence of conditional statements, loop statements, etc. in software, the execution rate of each different line in a program may vary significantly. For example, the execution flow may reach an outer-loop statement only once, while inner-loop statements could be exercised \(N\) times. As a result, the occurrence frequency of different items involving different lines in the mining database would differ based on the earlier definition of transactions. So, we usually set the \(s_{th}\) for SLP mining to a relatively low value to avoid overlooking the outer-loop properties.

b) Complex Property Implication Mining: A complex property implication is defined as:

\[
\{L = L_1 & x_1 \cdot y_1\} \Rightarrow \{L = L_2 & x_2 \cdot y_2\}, \text{ } x_1, y_1 \in \{>, \geq, <, =, \neq\}
\]

(4) where \(L_1 < L_2\), \(\{x_1, y_1\}\) and \(\{x_2, y_2\}\), respectively, are elements from the same subset in \(Q_a\), the two subsets may not necessarily be the same. The complex property implication inspects implicit knowledge among variables spanning over different program locations, that is, if \(x_1 \cdot y_1\) is true at line \(L_1\), \(x_2 \cdot y_2\) is definitely true at line \(L_2\), which makes it harder but valuable to discover.

We apply the A-Priori algorithm to mine the complex property implications with iteration \(k=2\), and association rules are generated using confidence threshold \(c_{th}\) as in the SLP mining. For example, \(\{L = L_1 & y < 0\} \Rightarrow \{L_1 = L_2 & z > 0\}\) could be a potential CPI in code segment (2) if it meets the support and confidence requirements.

Unlike in single invariant mining, the support threshold \(s_{th}\) for CPI mining usually is set to a relatively higher value to avoid mining an implication from an outer-loop property to an inner-loop property which in most cases is proven false or useless. Setting a high \(s_{th}\) value would probably filter out such implications due to the relatively large difference between the support values of their condition and consequent parts.

4.2 Optimization Algorithms

In order to remove redundant or less valuable properties learned, we applied several optimization rules on the mining results by utilizing the correlations among items and transactions as well as the program structure.

1) Optimization Rules for Single Linear Properties:

Rule 1.1: Redundant property removal: Remove those single invariants that can be implied by other learned single invariants based on the semantic relations among comparison operators. For example, if both \(P_1: L = \lambda \Rightarrow x < y\) and \(P_2: L = \lambda \Rightarrow x \leq y\) are mined, \(P_2\) will be removed because \(x < y\) can be implied by \(x \leq y\).

Rule 1.2: Equivalent property removal: If a linear relation \(R_i\) holds at both line \(i\) and \(j\), where \(i < j\), and all the variables involved in \(R_i\) are not re-defined within the program scope of \(i \leq j\), then \(R_i\) at line \(i\) and \(R_j\) at line \(j\) are considered as equivalent properties. Rule 1.2 will remove all the equivalent properties except the one at the minimum line number \(i\).

2) Optimization Rules for Complex Property Implications:

Rule 2.1: Potential fact implication removal: Given a property implication \(P_i: \{L_1 = \lambda & x_1 \cdot y_1\} \Rightarrow \{L_2 = \lambda & x_2 \cdot y_2\}\), if the consequent part \(\{L_2 = \lambda & x_2 \cdot y_2\}\) is mined as a potential single linear property invariant, \(P_i\) can be removed because implying a fact would not be interesting. Since at the mining phase, we only have confidence on the correctness of \(\{L_2 = \lambda & x_2 \cdot y_2\}\) to some extent, but can’t guarantee it, Rule 2.1 is used as a conservative but effective heuristic when too many outputs are returned by the mining algorithm.

Rule 2.2: Equivalent implication removal: Similar to Rule 1.2, this rule removes the property implication \(P_i: \{L_1 = \lambda & x_1 \cdot y_1\} \Rightarrow \{L_2 = \lambda & x_2 \cdot y_2\}\) if both properties involve the same relation (\(\cdot\)) between the same elements (i.e., \(x\) and \(y\)) whose values are not changed between line \(L_1\) and \(L_2\).

Rule 2.3: Transitive implication removal: When the same linear property is implied by different properties, i.e., \(P_1: \{L_1 = \lambda & x_1 \cdot y_1\} \Rightarrow \{L_2 = \lambda & x_2 \cdot y_2\}\) and \(P_2: \{L_2 = \lambda & x_2 \cdot y_2\} \Rightarrow \{L_3 = \lambda & x_3 \cdot y_3\}\), where \(L_1 < L_2 < L_3\), if \(P_1: \{L_1 = \lambda & x_1 \cdot y_1\} \Rightarrow \{L_2 = \lambda & x_2 \cdot y_2\}\) is also mined, then remove \(P_2\), since \(P_1\) can be implied from \(P_2\) and \(P_3\), according to transitive feature of implication.

4.3 Ranking Scheme

As explained in Section 3, it is necessary to make tradeoffs between the number of constraints to be added to the program and the resulting propositional formula size. In this paper, we apply a ranking scheme by assigning a weight to each verified invariant and only those most valuable invariants would be added to the program based on their weights.

The basic idea is as follows: Suppose \(A_i\) is the target assertion to be checked at line \(L_{\sigma_i}\), which consists of variables \(\{x_1, \ldots, x_n\}\), and a single property \(P_j: \{L = \lambda & x \cdot y\}\) is a verified invariant learned from data mining, where \(L \geq \sigma\). Intuitively, we assume the following three parameters are major factors that contribute to the value of an invariant \(P_j\) in the verification of the assertion \(A_i: 1)\) the statement distance between \(P_j\) and \(A_i; 2)\) the variable distance between \(P_j\) and \(A_i; 3)\) the occurrence frequency of variables in \(P_j\). Particularly, the closer the \(P_j\) is to the \(A_i\), the more closely related the variables in \(P_j\) are to variables in \(A_i\), and the more locations the variables in \(P_j\) are involved in the code ahead of \(A_i\), the more crucial is the \(P_j\) in assertion verification. Therefore, we define the weight of the \(P_j\) as:

\[
\begin{align*}
\text{weight}(P_j) &= c_1 \cdot (H - (L - \lambda)) + c_2 \cdot \Sigma(\text{dist}(x, z_i) + \text{dist}(y, z_i)) \\
&+ c_3 \cdot (\text{freq}(x) + \text{freq}(y))
\end{align*}
\]

(5)

where \(\text{dist}(a, b) = \begin{cases} H, & \text{if } (a=b) \text{ or } (a \text{ or } b \text{ is a constant}) \\
H \cdot (\text{level distance between } a \\
\text{and } b \text{ in the UD chain of } b), & \text{if } a \neq b
\end{cases}\)

and \(\text{freq}(a) = \text{number of lines that } a \text{ appears between } \lambda \text{ and } L_{\sigma_i}\).

\(c_1, c_2, c_3\) and \(c_4\) are weighting coefficients that satisfy \(0 \leq c_1, c_2, c_3 \leq 1\) and \(c_1 + c_2 + c_3 = 1.0\). \(H\) is a constant which is large enough to guarantee both the first part of the weight and the \(\text{dist}\) function to be non-negative. The three addends from left to right in the
expression (5) reflect the above three factors, respectively, associated with weighting coefficients. For the first factor, we consider the statement distance as simply the difference in the line numbers between the \( P_i \) and \( A_c \). Next, in order to measure the variable distance between two variables \( a \) and \( b \), we build the Use-Definition (UD) chains recursively that link the use of each variable to its definition until variable type definition is reached. If \( a \) directly or indirectly contributes to the value of \( b \), we calculate the level distance between \( a \) and \( b \) in terms of \( b \) as the number of steps needed to reach \( a \) from \( b \) by tracing backward along the UD chain of \( b \); otherwise, the level distance is set to be \( H \). Take code segment (2) as an example, the level distance between \( z \) at line 5 and \( x \) at line 2 or line 3 is 1. Finally, the occurrence frequency of a variable \( a \) is evaluated by the number of lines between \( \lambda \) and \( L_a \) that involve \( a \), i.e., either used or defined, which indirectly measures how big influences that \( a \) has to \( A_c \).

Similarly, for property implication \( P_i: \{ L=\lambda_1 & x_j \cdot y_j \} \Rightarrow \{ L=\lambda_2 & x_k \cdot y_k \} \), the weight is computed as:

\[
\text{weight}(P_i) = c_1 \times \text{weight}(L=\lambda_1 & x_j \cdot y_j) + c_2 \times \text{weight}(L=\lambda_2 & x_k \cdot y_k)
\]

where \( c_1 \) and \( c_2 \) are weighting coefficients such that \( 0 \leq c_1, c_2 \leq 1 \) and \( c_1 + c_2 = 1.0 \). In other words, our ranking scheme is an offline evaluation on invariant importance through weight computation, which plays an essential role in balancing the gains and overhead of invariant-based model checking.

5. Experimental Results

We use CBMC (C Bounded Model Checking) [6] as the base model checker with the underlying SAT-solver using MiniSat [12]. Several C programs on data structure and algorithms were selected as benchmarks, including: bubble and shell sorting and binary and Knuth-Morris-Pratt (KMP) searching. Profiling of these benchmarks was performed via dynamic program execution either for \( N=100 \) times on randomly generated inputs or when transaction count is over 5000 in the mining database, whichever comes first. In these experiments, we assume the related variable set is determined by programmers.

5.1 Data Mining and Optimization

Table 5 shows the number of properties learned purely through association rule mining and after optimization algorithm is applied for both Single Linear Properties (SLP) and Complex Property Implications (CPI). For SLP mining, the support threshold \( s_{th} \) is set as 0.3, for CPI mining, \( s_{th} \) equals to 0.8, confidence threshold \( c_{th} \) for both property mining is 1.0.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>SLP</th>
<th>CPI</th>
<th>Mining+Opt</th>
<th>Verified Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mine</td>
<td>Opt</td>
<td>Mine+Opt</td>
<td>Time (min)</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>16</td>
<td>5</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>ShellSort</td>
<td>39</td>
<td>11</td>
<td>164</td>
<td>0</td>
</tr>
<tr>
<td>BinarySrch</td>
<td>28</td>
<td>11</td>
<td>66</td>
<td>4</td>
</tr>
<tr>
<td>KMPSrch</td>
<td>57</td>
<td>14</td>
<td>262</td>
<td>13</td>
</tr>
</tbody>
</table>

As shown in the table, with our optimization algorithms, the number of properties learned by data mining can be reduced significantly in both SLP and CPI, some can even achieve more than 95% of reduction, like the CPIs in ShellSort and KMPSrch. It further proves that without the knowledge of the correlations among items and transactions, data mining would generate quite many redundant properties for software mining database, of which a significant part can be covered by the categories of our optimization rules.

The learned properties (after optimization) reported in Table 5 are just potential invariants; they have to be verified before being used as constraints. We asserted them into the program and used CBMC for correctness verification with the unwinding depth no lower than that of verifying the target assertion. The last column (%) of Table 5 gives the resulting percentage of verified invariants; all the benchmarks reached 100% of accuracy rate except the KMPSrch, which demonstrates that the threshold settings in our association rule mining are effective in filtering out the false positives. Although the needs for validating potential properties introduce additional overheads to software model checking, the advantage is that this is a one-pass mining and invariant verification process that can satisfy the requirements for checking different assertions. This is especially helpful when critical variables are chosen as related variable set, no iterative learning process is required. In practice, this overhead for potential invariants verification can be further reduced by applying ranking first and then focusing on verifying those top ranked potential invariants when the number of constraints to be added is limited as in our experiments, which is discussed next.

5.2 Model Checking

Before adding the proved invariants to the original program as constraints, we utilized the ranking scheme on the invariants in order to evaluate the quality with respect to the assertion to be checked. Take BubbleSort as an example, we compute the weight value for each of the six verified invariants using formula (5) and (6), where we selected \( c_1=0.2, c_2=0.3, c_3=0.5, c_4=0.4, c_5=0.6 \) and \( H=5 \). Then, we insert each of them as a constraint separately to the program under verification and run model checking. The results are shown in Table 6. We can see that this simple ranking scheme does provide a rough estimate on the value of each invariant in terms of assertion verification, although it may not always be accurate for individual cases. For example, \( P_5, P_1 \) and \( P_3 \) which achieve the best results were indeed ranked among the top positions; \( P_6 \) and \( P_4 \) which are ranked at the bottom did demonstrate much worse performance than others, especially when the unwinding depth \( N \) increases.

<table>
<thead>
<tr>
<th>Property Invariants</th>
<th>Rank</th>
<th>WeightI</th>
<th>Verification Time(sec) for N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>P.(SLP)</td>
<td>1</td>
<td>6.6</td>
<td>5.2</td>
</tr>
<tr>
<td>P.(SLP)</td>
<td>2</td>
<td>6.1</td>
<td>1.3</td>
</tr>
<tr>
<td>P.(SLP)</td>
<td>3</td>
<td>6.1</td>
<td>3.2</td>
</tr>
<tr>
<td>P.(SLP)</td>
<td>4</td>
<td>5.2</td>
<td>2.8</td>
</tr>
<tr>
<td>P.(CPI)</td>
<td>5</td>
<td>4.0</td>
<td>6.1</td>
</tr>
<tr>
<td>P.(CPI)</td>
<td>6</td>
<td>3.2</td>
<td>16</td>
</tr>
</tbody>
</table>

TABLE 6 PROPERTY RANKINGS FOR BUBBLESORT

The verified invariants are added to the original program with the target assertion to be checked using _CPROVER_assume provided by CBMC to restrict the search space and apply assume-guarantee reasoning. Considering the program size, we limit no more than 2 invariants will be added at the same time to the program. \( P_i \) and \( P_i \) are among the top ranked invariants for each benchmark. \( P_i \) and \( P_i \) means both \( P_i \) and \( P_i \) are assumed at the same time. Since all the benchmarks are loop-based algorithms, Table 7 also lists the results of model checking under different unwinding depth settings.

As shown in the results, with the added mined invariant(s), in most cases (in bold), the verification time of the real target assertion (either Sat or UnSat) can be improved significantly, some can reach up to an order of magnitude speedup (e.g., \( P_i \)-only case for BubbleSort with unwinding depth at 20). These results not only demonstrate the effectiveness of our
simulation-directed mining framework in learning and selecting valuable invariants that are crucial to the assertion verification, but also prove the importance of high-level invariant extraction for SAT-based software model checking. On the other hand, it can also be observed that adding multiple invariants at the same time does not always guarantee better performance than adding individual ones separately, especially when the unwinding depth gets larger. For example, in BinarySrch when unwinding depth is 20, $P_1+P_2$ turns out to be much slower than $P_1$-only, $P_2$-only and the original cases. This is probably because $P_1$ are $P_2$ are both invariants within loops, they will be replicated within each unraveled loop iteration, which may lead to a much larger propositional formula size especially in the case of $P_1+P_2$, such that the extra cost incurred on SAT-solver exceed the potential savings that may brought by the invariants. Therefore, it is important to balance between the added invariants and the formula complexity.

### Table 7: Model Checking Results

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Unwind depth</th>
<th>Verification Time (sec)</th>
<th>$P_1$-only</th>
<th>$P_2$-only</th>
<th>$P_1+P_2$</th>
<th>Sat?</th>
</tr>
</thead>
<tbody>
<tr>
<td>BubbleSort</td>
<td>20</td>
<td>59</td>
<td>19</td>
<td>6</td>
<td>188</td>
<td>Sat</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>325</td>
<td>194</td>
<td>164</td>
<td>287</td>
<td>Sat</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>1245</td>
<td>353</td>
<td>1504</td>
<td>1973</td>
<td></td>
</tr>
<tr>
<td>ShellSort</td>
<td>20</td>
<td>158</td>
<td>154</td>
<td>76</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>545</td>
<td>452</td>
<td>573</td>
<td>607</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>1473</td>
<td>1314</td>
<td>1071</td>
<td>2055</td>
<td></td>
</tr>
<tr>
<td>BinarySrch</td>
<td>10</td>
<td>42</td>
<td>37</td>
<td>140</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>KMPsrch</td>
<td>10</td>
<td>16</td>
<td>10</td>
<td>23</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>154</td>
<td>136</td>
<td>149</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>805</td>
<td>693</td>
<td>615</td>
<td>523</td>
<td></td>
</tr>
</tbody>
</table>

### 6. Related Work

A large amount of research work has been done in the area of model checking and data mining separately, but not much was engaged in the combination of both. In terms of mining, a machine learning approach was used in [17] to mine formal specifications by observing the program executions, assuming the program is generally correct. For hardware, in [5], data mining was applied for global constraints mining to improve the performance of SAT-based bounded equivalence checking of sequential circuits.

Some previous work was proposed to assist software verification by adding invariants learned through static analysis. Two different types of learning were applied: 1) offline learning: invariants are computed statically and relatively independent from verification process, such as invariants in the form of $\exists x \forall y \leq c$ in [7] and high-level design information (both within and between basic blocks) extracted from EFSM models in [3]. 2) online learning: invariants are deduced and strengthened during the verification process iteratively to help reduce the search space [8, 9, 14, 15].

The offline approaches only need to run once but suffer the problems of effectiveness since the invariants learned may not fit the verification requirements of an assertion, while the online approaches have the advantage of deducing appropriate invariants when it needs to but require being conducted iteratively which might incur extra cost. Our light-weight simulation-directed property mining is an enhanced version of offline method. It is a one-pass process before verification takes place, which compared to the most offline learning approaches, offers more flexibility on the property formats that can be learned. In addition, the learned invariants will be filtered and ranked according to the specific assertion being checked and the program structure before it is used.

Besides static analysis, there are also other invariants discovery techniques. Daikon [16] is such a simulation-based tool for likely program property and conditional implication learning, but none of them are used as additional knowledge to improve performance of model checking as our work did.

### 7. Conclusion

We have presented a novel framework for software verification that combines data mining mechanism with the bounded model checking. High-level single property invariants and complex property implications are extracted via an association rule mining technique according to the profiling data of the dynamic program execution. Optimization algorithms and ranking scheme help to further filter out those not-so-interesting properties to control the quality as well as the quantity of the property learned. The simulation-directed invariant mining framework provides straightforward and flexible mechanism for knowledge discovery. Experimental results showed that by adding the extracted and verified properties as constraints to the program, it can help improve the performance of software bounded model checking significantly through knowledge and search space reduction.

### References