Ant Colony Optimization Directed Program Abstraction for Software Bounded Model Checking

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

Abstract—The increasing complexity and size of software designs has made scalability a major bottleneck in software verification. Program abstraction has shown potential in alleviating this problem through selective search space reduction. In this paper, we propose an Ant Colony Optimization (ACO)-directed program structure construction to formulate a novel under-approximation based program abstraction (UAPA). By taking advantage of the resulting abstraction, a new software bounded model checking framework is built with the aim of improving the performance of property checking, especially for property falsification. Experimental results on various programs showed that the proposed ACO-directed program abstraction can dramatically improve the performance of software bounded model checking with significant speedups.

I. INTRODUCTION

Model checking has emerged as a powerful formal verification technique that proves the correctness of a design through a systematic exploration of its state space. Despite the vast technical advances in the last decade, it still suffers from the well-known state space explosion. With the recent advances in the capabilities of SAT-solvers, bounded model checking (BMC)[12,13] has become a good candidate for relieving such problem in model checking both hardware and software systems when the unrolling depth is bounded. Nevertheless, even if BMC can significantly increase the capacity and applicability of model checking in verifying large designs, scalability is still a major concern, especially in the context of software verification. This is due to the dramatically increased size and complexity of both the data and control flows in software when compared with hardware designs.

One of the primary reasons behind the state space explosion problem is the inherent nature of model checking, that all states must be traversed. Therefore, effectively reducing the burden of state space exploration may provide promise towards performance and scalability enhancement of model checking. Abstraction has been widely applied as an effective technique to reduce the size and complexity of the system under verification by removing unnecessary portions while retaining only the relevant information in the design with respect to the verification target. Two well-adopted types of abstractions are: 1) over-approximation, in which the abstract system contains a superset of the behavior of the concrete system, and 2) under-approximation, in which the abstract system contains a subset of the behavior of the concrete system. In this paper, we explore an under-approximation based program abstraction technique with the aim to improve the performance of software bounded model checking by falsifying the target property more quickly.

Intuitively, the complexity of solving a large and intractable problem usually can be mitigated through the integration of results from solving several smaller and simpler problems. In the context of software model checking, rather than exploring the original huge state space, it may be beneficial to partition the original (large) state space of a given program into smaller subspaces, i.e., building an under-approximation by imposing feasible assumptions. Then, we collectively integrate the verification results from each abstract subsystem. In particular, at the initial phase of code design, when falsification (bug finding) is the major task of model checking, it is highly possible that by intelligently identifying an under-approximate model, the corresponding reduced state space would be sufficient to falsify a target property. In this regard, we focus on property falsification as the major aim in the rest of our paper.

if \((0 < x \leq N \&\& y \leq M)\) // e.g. a cex occurs when x=N, y=0
\[ x = f1(); \]
\[ y = f2(); \]
else // \(x<y\) holds after the execution of else-branch
\[ \text{while ( cond )} \]
\[ x = f3(); \]
\[ y = f4(); \]
...... // operations on \(x\) and \(y\)
\[ l:\text{ assert}(x < y); \]

Fig. 1. Example for Property Falsification

Conventional program reduction includes static program slicing, where code portions that have no effect on the semantics of the target point of interest get deleted. However, program slicing usually results in an exact abstraction of the original program with respect to the target, and irrelevant statements are removed solely based on the program data and control flow structure. For example, in Fig. 1, program slicing cannot remove any of the shown statements before the assertion because all of them directly or indirectly contribute to the values of \(x\) and \(y\), which are the variables involved in the assertion. Apparently, the reduced program obtained via static program slicing can only provide modest simplification over the program and thus limits the potential gains it could bring to the software verification. Instead of relying only on the static program structure information, we seek for a more aggressive abstraction approach via under-approximations under the guidance of concrete state space exploration, i.e., dynamic program execution. During execution, useful information regarding the characteristics of program behaviors (e.g. variable relations, branching directions, etc.) can be extracted with carefully designed program inputs. Therefore, unlike the conservative nature of static approaches that must be sound for every single input, dynamic execution is a good candidate for identifying an approximation of promising areas and further narrowing down the size of the
resulting reduced program, so that it can be utilized by the model checking to achieve higher performance improvement.

Using the code fragment of Fig. 1 again, suppose \( x \leq y \) at line \( l_k \) is the target safety property for which few counterexamples (e.g., \( x = N, y = 0 \)) can be found. Further, let us assume that the property can only be falsified via the then-branch of the if-then-else (ITE) code. Without abstraction, the entire program will be converted into one single Boolean formula to be solved in the context of software BMC [13]. Moreover, if the else-branch of the ITE contains a large bulk of code and complex program control structure, like the while-loop shown in Fig. 1, the size of the resulting formula might grow quite large, which in turn could significantly degrade the performance of model checking. On the contrary, if we are able to identify promising code areas from dynamic execution that provides hints to where counterexamples most likely reside (e.g., the then-branch) beforehand, it would be helpful to enhance the efficiency of model checking by taking advantage of the reduced state space exploration with those less useful areas removed.

In this paper, we propose an Ant Colony Optimization (ACO) [1] directed under-approximation based abstraction technique with the goal of improving the performance of software bounded model checking. ACO is a population-based meta-heuristic that can be used to find approximate solutions for optimization and search problems through the cooperation and adaption of artificial ants that simulate the food hunting behaviors of real world ants. In our approach, we leverage the power of ACO to facilitate the construction of path-level abstraction for software model checking. The basic idea is as follows: given a property to be checked, artificial ants are sent out to explore the concrete state spaces such that the program execution flows would be led towards traversing the promising state space with respect to the property falsification. The program traces gathered by the ants would then be used to construct program under-approximation by biasing the control-flow that ignores less useful statements. Subsequently, we adopt an iterative abstraction-based software bounded model checking framework on the reduced program until either a counterexample is found or the property is verified. Experimental results demonstrated the effectiveness of the approach in achieving dramatic performance enhancement in software verification, especially for property falsification.

The remainder of the paper is organized as follows. Section 2 discusses background knowledge and previous work. The ACO-directed program abstraction process is described in Section 3. Section 4 presents the new bounded model checking algorithm through iterative program abstraction. Experimental results are given in Section 5. Finally, Section 6 concludes the paper.

II. BACKGROUND AND RELATED WORK

Our work touches upon several research areas, including: program abstraction, dynamic execution-guided software model checking, and the application of ACO as an optimization methodology for handling the problems in software testing and verification. We will discuss the corresponding background and previous work in these areas in the following subsections.

A. Ant Colony Optimization

Ant Colony Optimization (ACO) is a class of meta-heuristic, bio-inspired algorithms by observing the foraging behaviors of real-world ants. Initially, ants wander randomly in search of food. Once a food source is found, they lay down pheromones along trails with appropriate amounts based on the attractiveness (i.e., quality and quantity) of the food when they return back to the colony. Pheromones evaporate as well over time at a rate proportional to the length of the path between the food and the colony. Such pheromone trails thus establish indirect communication links among ants and provide guidance for the future ants towards the food source along the most likely best (shortest) paths.

Ant System [1] was the first known ACO algorithm that applied artificial ant colonies to solve the traveling salesman problem. Since then, many variants have been proposed solving various combinatorial optimization problems. The general framework of the ACO algorithm is as follows:

\[
\text{Algorithm ACO}() \\
1. \quad \text{Initialization()} \\
2. \quad \text{while} \ (\text{end\_conditions}) \\
3. \quad \text{constructSolution()} \\
4. \quad \text{updatePheromone()} \\
\]

In the ACO algorithm, the optimization problem to be solved is usually represented as a graph, the artificial ants move around the graph iteratively to find solutions (e.g., paths in the graph). In each iteration, ants construct possible solutions stochastically: the edge leading to the next node with a higher pheromone level is more likely to be followed. Then, a heuristic measure is used to evaluate the “quality” of the solution and update the pheromones accordingly along the solution path. The above process repeats until the terminating conditions are met.

In recent years, the ACO has been utilized for test input generation in various contexts: from Markov software usage model [3], based on UML state chart diagrams [5], for event sequence construction [4] and for mutation testing [6]. Unlike the above approaches, we take a step beyond the test data generation: the information gathered along program branches/paths during the ants’ incremental search process for program inputs is utilized to facilitate the search space reduction of formal software model checking.

B. Program Abstraction Based Verification

A substantial amount of research has been conducted on the application of abstraction in software model checking, ranging from model level abstraction (e.g., partial order reduction [14]) to program code level abstraction (e.g., predicate abstraction [15, 16]). Our work, instead, shares similarities with abstraction techniques based on under-approximation refinement (e.g. finite-precision bit-vector encoding[2, 18]) or exact program transformation through program slicing[17, 18]. We explore a different abstraction scheme which is performed at the path-level of the program. A reduced program is obtained by pruning away the less promising branches from conditional statements with respect to the target property checking. On the other hand, with the help of ACO-directed program structure traversal, our
approach intends to gain more aggressive slicing on the entire state space than the traditional program slicing to achieve higher performance savings.

C. Dynamic Execution Guided Software Model Checking

There has been an emerging trend that combines software testing/dynamic program execution with formal verification of software. These dynamic techniques work as heuristics to guide state space traversal towards promising areas, so as to alleviate the state space explosion problem in model checking. For example, CMC [7] applied direct program executions to guide the bug-finding. Another category of research focuses on integrating the concrete execution-directed under-approximation with over-approximation based abstraction techniques [8, 9, 10, 11], where dynamic executions or testing are used to guide the reasoning or refinement decisions made in abstract models.

The above approaches require the concrete states to be mapped to abstract states, with the concrete states providing hints for the traversal though the abstract state space. While in our work, concrete execution results are used to direct the code-level under-approximation. Furthermore, instead of using randomly generated program inputs, we feed the dynamic program execution with inputs selected by ACO so that execution flows can be effectively led down to the promising areas with respect to the property to be checked.

III. ACO-DIRECTED STATE SPACE EXPLORATION

In this section, we present in detail the proposed ACO-directed concrete state space exploration algorithm to reveal useful structural information in the code with respect to the target property. Such information will be utilized to guide the subsequent construction of under-approximation based program abstraction for property verification.

A. ACO-Directed State Space Exploration Model

Suppose the program under verification \( P \) consists of \( n \) input variables: \( l=x_1, ..., x_n \), each of which lies within an input domain \( D_l=[Di_{\text{min}}, Di_{\text{max}}] \), where \( 1 \leq l \leq n \). Let the target property to be checked, \( A \), be represented as \( \{f\} @l \), where \( f \) is a Boolean formula defined over variables in \( P \), and \( l \) is the line number in the code. In other words, property \( A \) checks whether \( f \) holds at line \( l \). The ACO-directed state space exploration problem involves the process of finding an appropriate input vector: \( T=<t_1, ..., t_n> \) for \( P \), such that \( t_i \in D_l \) (\( 1 \leq i \leq n \)) and \[ \neg f \] is true at line \( l \) when the program is run with \( T \) as its inputs. In short, ACO tries to lead the program execution to falsify \( A \). In this paper, we assume the target property is a hard corner case such that it is not so easy to find such a \( T \) by directed program execution alone within a pre-defined number of iterations.

In essence, the first step of applying ACO is to model the problem with appropriate representations so that the artificial ants can navigate toward the target. As the ultimate goal of our ACO-directed state space exploration is to identify promising structures in \( P \) that constitute state spaces that can most likely falsify the property to be checked, we are focusing more on the paths/traces that the artificial ants have explored during the solution construction rather than focusing on the program input values. Hence, we divide the problem representation into two graphical sub-models: Input Space Model (ISM) and Control Flow Model (CFM).

![ACO-Directed State Space Exploration Model](image)

For the input space, we apply a similar model as in [6]. The ISM of \( P \) is represented as a directed graph \( G_\text{ISM}(V, E) \) consisting of \( n+1 \) vertices \( <v_0, ..., v_n> \) (Fig. 2(a)), where \( v_0 \) is the starting node, and any other vertex \( v_l (1 \leq l \leq n) \) in the graph corresponds to an input variable \( x_l \) in \( I \). In order to model the value selection activity for input variables during test generation, for every input variable \( x_l \) in \( I \), we further partition its input domain \( D_l \) evenly into \( k_l \) sub-domains: \( <d_{l1}, ..., d_{lk_l}> \), where \( D_{\text{min}}=d_{l1} \) and \( D_{\text{max}}=d_{lk_l} \). Correspondingly, every vertex \( v_{li} (1 \leq l \leq n) \) in \( G_\text{ISM} \) is connected with its next vertex \( v_l \) through \( k_l \) edges, representing \( k_l \) possible paths that can be selected from and traveled along by ant agents. If the \( j^{th} \) \((1 \leq j \leq k_l)\) edge is taken by an ant from \( v_{li} \) to \( v_{lj} \), it implies that the current value of input variable \( x_l \) is ranged within its \( j^{th} \) sub-domain: \( d_{lj} \). Therefore, a valid input vector of \( P \) is determined when a directed path from \( v_0 \) to \( v_n \) is traversed by an ant, with each node to be visited only once and only one edge chosen between every pair of adjacent nodes.

While the ISM is used as the real site for ants to navigate through, the CFM works as a virtual space that captures the paths exercised by the ants virtually when the program is executed with a valid input vector consistent with a real path in the ISM. As shown in Fig. 2(b), the CFM \( G_\text{CFM}(B, C) \) is built directly on the control flow graph (CFG) of \( P \). Each node \( b_l \) represents a basic block in \( P \) and the edge \( c_{lj} \) may be labeled with conditional guards that control the transition from node \( b_i \) to \( b_j \). A virtual path in CFM starts from the entry node \( b_0 \) and ends at a leaf node which has no descendants. Given a target property \( \{f\} @l \), we first instrument \( P \) with an ITE statement inserted at line \( l \): if \( (f) \) then \( \{b_{goal}\} \); else \{original code blocks in \( P \) when \( f \) is not violated\}. The task of ants is to lead the program execution towards the target block \( b_{goal} \) in CFM during the input selection process when they walk around ISM.

The basic idea of the ACO-directed state space exploration algorithm is as follows: At every iteration, an ant is sent to the ISM and performs the following two tasks: 1) Construct promising program structure by traversing the state space with an appropriate input vector, which is obtained using a probabilistic transition rule based on the amount of synthetic pheromone along the trails in both real and virtual models in Fig. 2; 2) Update pheromone intensities on both trails based on a fitness evaluation. The above process is repeated until
the target block is reached or it exceeds a pre-defined number of iterations.

B. Stochastic Promising Structure Construction

In order to construct promising program structures with respect to the target \( b_{\text{goal}} \), a new ant enters ISM and tries to select an appropriate input vector (i.e. a path in ISM) that would most likely trigger the program execution flow down to \( b_{\text{goal}} \). As the pheromone levels indicate the quality of a route, the path selection of the new ant is guided by the pheromone trails left by preceding ants. We associate a 2-tuple pheromone vector: \(<\text{phrm}, \text{prob}>\) with every edge \( e \) in the ACO model (ISM and CFM), where the \( \text{phrm} \) represents the pheromone intensity accumulated on the edge and \( \text{prob} \) indicates the probability for the incoming ants to follow \( e \). Suppose a vertex \( v_i \) has \( l_i \) outgoing edges, the probability \( \text{prob}_{b_j} \) that an ant will take the \( j^{th} \) (\( 1 \leq j \leq l_i \)) edge out of \( l_i \) possible paths is calculated as: \( \text{prob}_{b_j} = \text{phrm}_{b_j} / \Sigma(\text{phrm}_{b_j}) \). The pheromone vectors define a stochastic transition rule for edge selection between two adjacent nodes in ISM and branch taking in CFM, such that edges with higher pheromone levels (i.e. quality) are more likely to be followed. According to the relationship between the edges in ISM and the sub-domains of input variables, we can thus create a valid input vector after the ant chooses a path in ISM from \( v_0 \) to \( v_n \) by assuming that all the possible input values within a sub-range defined along each edge is uniformly distributed.

We use a Monte Carlo method to simulate the ants’ path selection activity. It randomly samples inputs based on the probabilistic pheromone distribution over the trails in ISM, until the corresponding virtual path obtained (after applying the sampled inputs to the program) also satisfies the pheromone distribution over the trails in CFM. However, since the ISM and CFM indirectly correlate with each other, it might take too many iterations before satisfactory inputs can be found. In order to reduce the overhead of random sampling, we relax the requirement of satisfying the distribution of every branch along a virtual path in CFM to only satisfying the distribution of the last branch in the path that deviates from the target. In this way, we can significantly improve the efficiency of input selection, while simultaneously still able to guide the ants to make good decisions with respect to the goal.

C. Pheromone Update

During the incremental solution construction process in the ACO-based algorithm, each individual ant contributes its own partial solution as well as communicates with other ants by means of updating pheromones along the path it was traversing. The amount of pheromone laid down by an ant usually directly reflects the quality of this path with respect to certain goal. Since our goal is to reach a target node in the CFM, we perform quality assessment using a fitness function based on the characteristics of virtual paths being traveled in the CFM.

Given a virtual path \( p_r=<b_{\text{goal}}, b_{1}\text{cfm}, ..., b_{\text{goal}}> \) in CFM(\( x_0=0 \)), suppose CFM consists of \( M \) nodes, \( p_r \) misses the target node \( b_{\text{goal}} \) and \( b_{\text{cfm}} \) (\( 1 \leq q < h \)) is the last node in \( p_r \), that is an ancestor of \( b_{\text{goal}} \). We compute the fitness value of \( p_r \) with the following formula:

\[
\text{fitness}(p_r) = w_1 \times (M - \text{levDis}(b_{\text{cfm}}, b_{\text{goal}})) \\
+ w_2 \times \left( \sum_{j=0}^{q} \frac{\text{phrm}(c_{x_{i,j+1}})}{(q+1)} \right)
\]  

(1)

In Formula (1), two factors are considered to influence the quality of a virtual path with respect to the target \( b_{\text{goal}} \): 1) \text{level distance} between \( b_{\text{cfm}} \) and \( b_{\text{goal}} \), which indicates the length of the shortest path from \( b_{\text{cfm}} \) to \( b_{\text{goal}} \). The lower the level distance, the closer the ant is to the target and the higher the quality of the path. 2) \text{average pheromone merit} of \( p_r \), a measurement of how well the ant follows the value of the routes granted by preceding ants in terms of the average pheromone intensity along the path segment in \( p_r \) till the first deviation from the goal (i.e., from \( b_{\text{cfm}} \) to \( b_{\text{goal}+1} \)). In other words, a virtual path that can match more promising branches intends to have higher quality. We integrate the effect of both factors to fitness evaluation by associating each with a different weighting coefficient. In order to keep an updated correlation between the two submodels, the computed fitness value of \( p_r \) will then be used to update the pheromone trails along the segment from \( b_{\text{cfm}} \) to \( b_{\text{goal}+1} \) in \( p_r \) as well as the corresponding path in ISM that triggers \( p_r \).

Besides pheromone accumulation, we also gradually reduce the pheromone intensities along the trails to help avoid unwanted convergence and explore new regions in the search space. Pheromones evaporate over time. The quality of a path would be degraded if it is not selected by enough number of ants and accumulated with enough amount of pheromone.

The ACO-directed state space exploration stops with two possible outputs: either a valid input vector is found such that the target block has been reached or ants fail to reach the target within some pre-defined number of iterations. In this work, we focus on those hard properties where the latter situation occurs predominantly. Note that in such cases, even though the target block is not reached, the eventual pheromone trails left by ants in the model still track down all their efforts made for property falsification while building up partial solutions incrementally. In particular, the pheromone trails would reveal promising areas in the search space where the solution is located.

IV. BOUNDED MODEL CHECKING THROUGH ITERATIVE UNDER-APPROXIMATION BASED PROGRAM ABSTRACTION

The new BMC algorithm based on the ACO-directed Under-Approximation based Program Abstraction (UAPA) is shown in Fig. 3:

Algorithm BMC_UAPA (Program \( P_{\text{abs}} \), Assertion \( \text{ast} \), CFM \( cfm \))
1. \( P_{\text{abs}}=\text{constructAbstraction}(P_{\text{org}}, cfm); \)
2. do
3. \( \text{res} = \text{BMC}_{\text{org}}(P_{\text{abs}}, \text{ast}); \)
4. if (\( \text{res} = \text{UNSAT} \))
5. //\text{absRefinement} returns NULL if the input \( P_{\text{abs}}=P_{\text{org}} \)
6. \( P_{\text{abs}}=\text{absRefinement}(P_{\text{abs}}, cfm); \)
7. while (\( \text{res} = \text{UNSAT} \) and \( P_{\text{abs}}=\text{NULL} \))
8. return \( \text{res} \);

Fig. 3. BMC Algorithm through Iterative UAPA

The first step of the algorithm is to construct an under-approximate program abstraction according to the pheromone trails left by ants in CFM. We choose multi-branch conditional statements (e.g., ITE, switch, while, etc.) as targets for abstraction since the semantics of these statements inherently divide the state space into multiple disjoint
subspaces by each branch, making them natural candidates for search space reduction. Given a multi-branch conditional statement with $R$ branches, correspondingly, it is represented as a multi-child node $b_i$ in CFM with $R$ outgoing edges $c_{i,j_1}, \ldots, c_{i,j_R}$, each of which is associated with a pheromone vector $<phrim_{i,j_k}$, prob\_{i,j_k}> (j\in\{j_1,\ldots, j_R\}) by the end of the ACO process. As described in Section III, the pheromone vector indicates how good an edge is with respect to reaching the target node. Hence, we use some pre-defined abstraction threshold $T_{abs}$ to suggest if a path should be included or excluded: if any prob\_{i,j_k} is lower than $T_{abs}$, the corresponding branch in the statement linking from node $b_i$ to node $b_j$ will be abstracted away. For example, suppose $T_{abs}=0.3$, given an ITE statement in the code:
\[
\text{if}(\text{cond}) \text{ then } \{\text{then-body}\}; \text{ else } \{\text{else-body}\}
\]
where the else-branch has a probability of 85%, and then-branch is 15%. After abstraction, the ITE is transformed into:
\[
\text{assume}(\text{cond}); \{\text{else-body}\}
\]
where the then-branch is removed and the assume statement restricts the search space at the ITE statement to only those states that satisfy the !\text{cond}, which are suggested by ACO as more promising to find solutions.

For the case of nested conditional statements, the abstraction decision on the outer-most layer statement would override that of the inner-ones. For instance, if the ITE example in (2) is located within a given branch of an outer conditional statement eligible to be removed, the entire inner ITE can be removed.

After the under-abstraction is constructed, the abstract program $P_{abs}$ is sent to a bounded model checker for assertion checking. If the BMC returns SAT, which means the property can be falsified in the reduced search space, the algorithm exits with the corresponding counterexample. Note that because the counterexample simultaneously satisfies all inserted assume() conditions along its path, this counterexample is also feasible in the original concrete program. If the BMC returns UNSAT, it may be caused by three possible reasons: 1) the property holds in the original program; 2) the $P_{abs}$ is over-abstracted such that the property cannot be violated within the reduced search space; 3) the $P_{abs}$ contains infeasible paths introduced by the conflicting assumption combinations from different abstract conditional statements. In any of the three cases, we need to perform abstraction refinement by restoring back some/all of the abstracted branches.

Given $K$ multi-branch conditional statements that have been partially abstracted in the current $P_{abs}$, each of which is composed of two parts: abstracted portion ($abs$) and retained portion ($rp$). Every time we pick one statement $s$ for refinement according to the following two rules: 1) no other abstracted conditional statements are nested within the body of $s_{rp}$; 2) $s_{abs}$ is located closest to the target block among $abs$ portions of $K$ statements. Then, for any selected conditional statement $s$, we perform a two-step refinement process:

Step 1: Restore $s_{abs}$ back while removing $s_{rp}$, then run BMC again on the resulting program;
Step 2: If BMC in Step 1 still returns UNSAT, restore the entire statement $s$, and pick the next eligible conditional statement for refinement.

For instance, the two-step refinement for the ITE in (2) is as follows:

step1: assume(\text{cond}); \{\text{then-body}\}
step2: if(\text{cond}) then \{\text{then-body}\}; else \{\text{else-body}\}

where step 1 negates the assumption condition in (3) and step 2 restores the original ITE statement.

Partial abstraction may also be interleaved with the restoration process. At Step 1, if any conditional statements nested within $s_{abs}$ have never been restored (i.e., they were fully abstracted) but satisfy the partial abstraction condition, instead of fully restoring $s_{abs}$, we attempt to restore such inner-statements partially only on their promising branches. Then, before moving to Step 2 for statement $s$, we apply the same two-step refinement for all the inner abstract conditional statements within $s_{abs}$ until all of them have been fully restored or the BMC returns SAT. In other words, we carry out the refinement iteratively on each abstracted conditional statement in an inside-to-outside, bottom-up fashion, with BMC applied at each iteration. The refinement process stops either when the program is restored to its original status or a counterexample is found.

V. EXPERIMENTAL RESULTS

We evaluated the proposed ACO-directed under-approximation based abstraction in bounded model checking with different software programs. As shown in Table 1, several sorting and selection algorithms as well as software applications were selected as benchmarks. The last four rows are benchmarks constructed from the policy engine module in a software defined radio system PSCR [19], which perform an equivalence checking between the policy engine with a mutated version of the code. For each benchmark, given the assertion to be checked, we applied our ACO-directed concrete state space exploration with the goal of negating the assertion through 150 ants (i.e., iterations). The runtime of ACO algorithm is always under 5 seconds, thus is negligible.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Abs. Rate</th>
<th>Depth</th>
<th>Orig. BMC</th>
<th>Iterative UAPA</th>
<th>UAPA Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>3/16</td>
<td>5</td>
<td>73</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1683</td>
<td>743</td>
<td></td>
<td>743</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>TO</td>
<td>2235</td>
<td></td>
<td>2235</td>
</tr>
<tr>
<td>Shellsort</td>
<td>1/5</td>
<td>25</td>
<td>530</td>
<td>187</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>2439</td>
<td>2458</td>
<td></td>
<td>2458</td>
</tr>
<tr>
<td>Selection</td>
<td>2/15</td>
<td>5</td>
<td>34</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1480</td>
<td>165</td>
<td></td>
<td>165</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>5792</td>
<td>1435</td>
<td></td>
<td>1435</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>59</td>
<td>17</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>483</td>
<td>109</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>Heapsort</td>
<td>1/9</td>
<td>35</td>
<td>125</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>483</td>
<td>109</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>pe_mut1</td>
<td>10/16</td>
<td>25</td>
<td>2195</td>
<td>(1, 148)</td>
<td>149</td>
</tr>
<tr>
<td>pe_mut2</td>
<td>28/40</td>
<td>5612</td>
<td>(5, 135)</td>
<td></td>
<td>134</td>
</tr>
<tr>
<td>pe_mut3</td>
<td>6/16</td>
<td>5155</td>
<td>134</td>
<td></td>
<td>134</td>
</tr>
<tr>
<td>pe_mut4</td>
<td>7/36</td>
<td>TO</td>
<td>(87,44,1884,120)</td>
<td></td>
<td>2135</td>
</tr>
</tbody>
</table>

Based on the ACO results, an under-approximation of program is constructed with the abstraction threshold $T_{abs}=0.6/R$, where $R$ is the number of branches of a conditional statement ($R \geq 2$ for switch statements). Table 1 compares the performance of the original BMC algorithm with the BMC through iterative UAPA for each benchmark.
We use the C-Bounded Model Checker (CBMC) [13] as the underlying BMC engine, and all the selected properties are hard false properties. The second column in Table 1 shows the ratio of the number of abstracted conditional statements to the total number of conditional statements in the program at the last iteration of UAPA. As some of the benchmarks are loop intensive, the results of property checking under different unwinding depths are also reported.

From the results, the UAPA-based BMC resulted in dramatic performance improvement for almost all the cases (in bold) over the original BMC algorithm. For instance, in pe_mut1, the original BMC run took 2195 seconds. Our approach took two iterations (each value within the parenthesis represents the runtime at different refinement iterations). In the first iteration, only 1 second was needed due to the over-abstraction. After refining the model, 148 seconds were needed to find the counterexample. In pe_mut3, a speedup of 40 was achieved. For all benchmarks except for the policy engine (pe_mut*) cases, our approach was able to disprove the property with no need of refinement. Even when iterative abstraction-refinement was needed, the total runtime of all the iterations was still much less than the original. These results show that the ACO-directed program abstraction is able to direct the UAPA towards promising areas in the code. The proposed iterative UAPA framework, thus, saves the time spent in exploring the unnecessary search spaces. The strength of ACO is further demonstrated in Table 2, where we compare the ACO-directed abstraction with a random-vector-directed abstraction. The same parameter settings and iterative refinement framework are used in both techniques.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Abs. Rate</th>
<th>Depth</th>
<th>Original</th>
<th>ACO</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>3/16</td>
<td>7</td>
<td>2238</td>
<td>2568</td>
<td></td>
</tr>
<tr>
<td>Shellsort</td>
<td>3/15</td>
<td>10</td>
<td>5792</td>
<td>1438</td>
<td>7162</td>
</tr>
<tr>
<td>Selection</td>
<td>7/9</td>
<td>45</td>
<td>483</td>
<td>109</td>
<td>483</td>
</tr>
<tr>
<td>Heapsort</td>
<td>1/9</td>
<td>25</td>
<td>2195</td>
<td>149</td>
<td>4614</td>
</tr>
<tr>
<td>pe_mut1</td>
<td>6/16</td>
<td>25</td>
<td>5612</td>
<td>141</td>
<td>569</td>
</tr>
<tr>
<td>pe_mut3</td>
<td>6/16</td>
<td>25</td>
<td>5155</td>
<td>134</td>
<td>(1,1304)</td>
</tr>
<tr>
<td>pe_mut4</td>
<td>7/16</td>
<td>25</td>
<td>2135</td>
<td>(1,3545)</td>
<td></td>
</tr>
</tbody>
</table>

It can be observed that ACO outperforms the random-vector-based execution in identifying good portions of the code for abstraction. Although running the program with random inputs was also able to find and prune away some highly biased branches in conditional statements according to the occurrence frequency, these branches do not necessarily link to the solution space. For example, consider benchmark selection, the original BMC time was 5792 seconds. With our ACO-directed abstraction, the run time was reduced to 1435 seconds; on the other hand, with the random-directed abstraction, the BMC performance was even worse than the original. This shows that, with the help of pheromone trails, ACO is able to narrow down the search to locations where the target property may be negated. The amount of pheromones indicates the quality of branches with respect to the verification target. Hence, they also provide useful hints during the iterative refinement than the pure execution frequency of branches triggered by random vectors.

VI. CONCLUSIONS

In this paper, we proposed an ACO-directed under-approximation based strategy to improve the performance of software model checking through search space reduction, especially for property falsification. An ACO-directed state space exploration was applied trying to construct a solution that falsifies the property through the coordination among ants. The ants leave pheromone trails along program structure to indicate the promising areas in the code from which an under-approximate model is built. A new BMC algorithm through the proposed UAPA framework was presented to search for solutions over the reduced state space, which was iteratively refined until it contains enough details to verify the property. Experimental results showed that with the proposed technique, the performance of software BMC for property falsification can be improved significantly, reaching up to 40 times of speedup. The ACO also demonstrated effectiveness in guiding the code abstraction over random-vector-directed abstraction.

REFERENCES