Model Checking
and

Ant Colony Optimization

By: Matthew Stephan

For: Professor Selim Akl

Course: CISC 879

Date Submitted: Dec 9, 2009

# Abstract

As software engineering is constantly evolving so is its use of abstraction as a tool for both understanding and analyzing a system’s domain. The level of abstraction we use in the field has reached such a level that model-driven paradigms are becoming more and more prevalent as is the use of software models in general. Having software models allow us to do complex analysis sometimes referred to as model checking, which includes examining the consistency and well-formedness of these models as well as checking for property violations and change impact. In the case of property violation checking, we often run into the state explosion problem where an exhaustive search of the model/state space is necessary but our model is too complex to do such an analysis. The following paper looks at current work in research that uses the Ant Colony Optimization (ACO) algorithm to perform this type of model checking. The ACO algorithm is a graph-search and path-discovery algorithm that takes inspiration from the way ants forage for food in nature. In this paper, we provide a summary of four relevant and recent works that use the ACO algorithm for model checking. The first paper uses the ACO algorithm to discover which branch of the state space can be abstracted away based on the chances that a violated property is discovered within that branch. The second paper uses a variant of the ACO algorithm to discover correct and incorrect traces within a program and to locate the sources of the errors. The third and fourth paper use extended versions of ACO, ACOhg and ACOhg-live, to discover safety and liveness property violations, respectively. The third paper also uses partial order reduction before executing ACOhg in order to improve performance. We then compare the four papers. Firstly, we compare the papers by noting the different goals in using ACO. We then compare and summarize the various modifications made to the ACO algorithm, if any, by each paper. We also note that all four papers have favorable results in both achieving their goal and in the metrics recorded. Lastly, we propose 3 improvements to the work presented in the four selected papers. The first involves using the ACO algorithm as done in the first paper before executing either ACOhg or ACOhg-live. The second improvement entails trying a parallel version of the ACO algorithm and the final improvement is to improve the automation of the work presented in Paper 2.

Contents

[Abstract 2](#_Toc248130305)

[List of Figures 4](#_Toc248130306)

[List of Tables 4](#_Toc248130307)

[Introduction 5](#_Toc248130308)

[Background Information 7](#_Toc248130309)

[Ant Colony Optimization 7](#_Toc248130310)

[Model Checking 8](#_Toc248130311)

[Safety Checking 9](#_Toc248130312)

[Liveness Checking 9](#_Toc248130313)

[Paper Summaries 11](#_Toc248130314)

[Paper 1 11](#_Toc248130315)

[Paper 2 13](#_Toc248130316)

[Paper 3 14](#_Toc248130317)

[Paper 4 15](#_Toc248130318)

[Paper Comparisons 16](#_Toc248130319)

[Ways to Improve 18](#_Toc248130320)

[Conclusions 19](#_Toc248130321)

[Works Cited 20](#_Toc248130322)

# List of Figures

[Figure 1: Ants Discovering Shortest Path 8](#_Toc248130323)

[Figure 2: State Space Exploration Models 12](#_Toc248130324)

[Figure 3 An ant nest representation of a CFG 13](#_Toc248130325)

# List of Tables

[Table 1: Summarization of ACO Modifications Made 17](#_Toc248130326)

# Introduction

In science and engineering disciplines, the notion of abstraction is very useful. It can assist in understanding a problem by abstracting out the concepts and details within the problem domain that are not needed for comprehension. Furthermore, it can assist in reasoning and evaluating aspects of the domain in scope because it provides a narrower focus on the relevant areas. Although somewhat slower than other engineering disciplines, the use of abstraction is more and more finding its way into Software Engineering. It began with the abstraction of machine code into assembly, assembly into programming languages, and, more recently, object orientation. Most recently, software engineering is beginning to adopt the use of Software Models, that is, models that abstract away code and platform specific behavior leaving only domain specific and platform-independent information. Their use is becoming so prevalent that a new software design paradigm, entitled Model-driven development, entails having software projects that are entirely model-centric . That is, models are the main artifacts within the project used by developers and stakeholders and code is generated automatically and updated from the models. Working at this higher level of abstraction allows errors to be detected earlier within a software project’s development cycle because models are developed early, which leads to cheaper and quicker repair. Also, the existence of these high-level models allows for various kinds of model checking including consistency checks, well-formedness checks, property checks, and trade-off checks that can be accomplished by exploring a model’s state space. One problem, however, in doing model checking is that as our software projects grow more and more complex, so do our software models, which leads to a state-space explosion.

One method of discrete optimization in computer science that takes inspiration from the natural world is the Ant Colony Optimization (ACO) algorithm. The ACO algorithm models itself after swarm intelligence found in nature, specifically, the behavior shown by ants as they forage for food. The algorithm allows for shortest path discovery and has been used to solve discrete optimization problems such as the traveling salesman problem and the quadratic assignment problem . At a high level, the algorithm includes ants as agents that are allowed to examine a search space and, as they do so, they affect their environment by leaving behind pheromone on the path they just traveled thereby influencing the other ants/agents. Letting the ants explore for a time eventually leads to an optimal path.

This paper looks at recent work done in research that uses the Ant Colony Optimization algorithm in order to accomplish model checking. In some cases, the algorithm is modified significantly to accomplish the specific type of model checking desired, while in others, only slight modifications are necessary. This paper begins by providing background information on both model checking and the Ant Colony Optimization algorithm that will allow the reader to understand the contents that follow. We then continue by presenting individual summaries of four papers that employ the ACO algorithm to model checking. The papers were chosen by us on the basis of their topic relevance and how recent the work was. We then continue by comparing the papers’ different goals in using the ACO algorithm, the modifications made to the algorithm by each paper, and other applicable comparisons. Following this, we discuss ways that the work done in each of the papers can be improved, possibly drawing ideas from the other 3 papers. Lastly, we draw conclusions about ACO and model checking based on what was discussed in our paper.

# Background Information

This section attempts to provide the necessary background information to the reader to facilitate understanding of the rest of the paper that follows. First, we present the Ant Colony Optimization algorithm and its high-level details. Following this, we include a section on model checking in general and a brief description of liveness and safety property checking.

## Ant Colony Optimization

The Ant Colony Optimization algorithm draws inspiration from the swarm intelligence properties shown by ants as they forage food as a colony. Swarm intelligence is the idea that many agents with limited abilities or capacities working together can form a collective intelligence that is much greater than what the agents can accomplish individually. The agents must be able to interact with one another and their environment and follow local rules that govern their behavior . In order to work as a swarm system, agents must be in proximity to facilitate interactions, be able to evaluate their interactions, be able to react to new and unknown situations while not changing behavior in all cases, and they must be adaptable to changes in their population and environment .

Ants exhibit these properties. Individually, an ant cannot asses the needs of the colony nor is their one ant assigning all others to do specific tasks. Working together as equal agents they are able to exhibit a collective intelligence. One notable example of this is the way that ants forage for food. As discussed in , ants are able to exploit food sources while still exploring and they eventually find the shortest path to the food over time relative to where the nest / colony home is. This is accomplished through local competition between the ants as each ant leaves behind them a chemical attractant called a pheromone that causes other ants to follow it. This pheromone that is left behind forms a trail that other ants follow. As shown in the experiment exhibited in Figure 1 from , ants are initially faced with a choice of two paths, one shorter than the other, to a food source. At the beginning there will be an equal chance that an ant will take either path. However, as the ants taking the shorter path will return quicker, more pheromone will be laid on the shorter path causing more and more ants to follow the shorter path. This positive feedback through the use of pheromones and the indirect communication results is termed stigmergy. Ants are able to discover the shortest path to a food source and are able to minimize their travel time.



Figure : Ants Discovering Shortest Path

This idea is emulated in conventional computing to come up with the Ant Colony Optimization algorithm. Simulated ants are allowed to travel randomly and independently throughout a search space/graph and deposit pheromone according to some quality metric. Ants follow the edges of the state space based on the pheromone present on the edge. Pheromone evaporates over time to prevent instability with more complex graphs. Specifically, when an ant is faced with multiple edges to travel, there is a probability that an ant will take a specific edge. This probability is influenced by the pheromone on the edge. More specific details on the ACO algorithm and the supporting math can be found in .

## Model Checking

Many benefits are seen by representing software systems at the model level. Aside from the comprehension aspects of abstraction, we are also able to perform various forms of model analysis and checking that would be otherwise difficult or implausible to accomplish if the system were represented at a lower level, such as programming code, assembly-language code, or machine code. One type of model checking includes verifying the well-formedness of the model with respect to some meta model, such as done in SPIN , or verifying if an instance model, such as code, conforms to that model. Another type of model checking is checking the consistency of models, that is, are the constraints within a model consistent/inconsistent. Related to this idea is the type of model checking known as property checking, which determines if various properties on a model hold. Alloy is one tool that is able to accomplish both of these kind of checks. Lastly, there is tradeoff analysis, which is when model checking is used to analyze the impact of changes to a model. There are various ways of accomplishing model checking including, but not limited to, logic-based solvers, such as satisfiability or constraint solvers; the use of state machines; and the use of process algebras. This paper concerns itself only with the state machine approach as that is the one that can be attempted through the use of the ACO algorithm. The following sections will go into brief detail of two specific types of property checking: liveness checking and safety checking, as they are areas that are covered in two of the four papers we present.

### Safety Checking

Safety checking is the process of checking a concurrent model for safety properties. Safety properties are properties that indicate something bad that should never occur within the system. Examples of this include the property that the system should never be in deadlock, a violation of mutual exclusion of a certain region of code, or another property that is specified in Linear Temporal Logic (LTL) . So, checking for safety involves traversing the model and finding a state that violates this property. This can be done by representing the states as a finite state machine through asynchronous composition and composing it with the negation of the safety property synchronously and then searching either using breadth-first or depth-first search. It is important to note, that safety violations can have a finite execution as a counter example because once the error has occurred, it cannot be undone .

### Liveness Checking

Liveness properties are those that express that something good/desirable will eventually happen at some point throughout the system. Liveness violations, in contrast to safety violations, can be violated only by infinite executions because there are infinite counterexamples since the good thing has not occurred but still may occur. Liveness checking involves representing the concurrent system as a Buechi automaton, which is an FSM with an acceptance condition that allows for an infinitely occurring state. The checking, like with safety checking, continues by composing this Buechi automaton with the negation of the liveness property and searching it using a nested depth-first search [[8](#Chi08),[6](#Din09)].

# Paper Summaries

This section provides an individual summary of each of the papers we selected that use Ant Colony Optimization to accomplish model checking. The citation for each paper is included once in the beginning of the summary and not repeated throughout each summary. After this section, each paper shall be referred to by the paper number given in its corresponding section title.

## Paper 1

The paper entitled “Ant Colony Optimization Directed Program Abstraction for Software Bounded Model Checking”, by Cheng and Hsiao, attempts to use the Ant Colony Optimization algorithm to form a directed program abstraction by formulating an under-approximated program abstraction. This directed program abstraction can then lead to quicker model checking. The authors begin by introducing the notion of model checking and explaining the state space explosion as well as the need for models to be abstracted further to facilitate model checking. They then discuss how it should be possible to identify an under-approximate model, whose reduced state-space would be sufficient to show a property that is false. The paper then provides background information on the Ant Colony Optimization algorithm. It also provides a contrast between the work done in the paper and program abstraction based verification noting that the technique implied in the paper uses abstraction at the path-level of the program for aggressive slicing on the state space. Further related work, dynamic execution guided software model checking, is presented as it is used by the authors. However, the authors note that in other approaches the concrete states are used to give advice for abstract state space traversal but in the work the authors present the concrete states are used to guide the code-level under-approximation.

The main section of the paper contains the algorithm that the authors create to traverse the concrete state space to discover the information necessary to achieve under-approximation based program abstraction. It begins by splitting the program representation into the two graphs shown in Figure 2: the input space model and the control flow model. The input space model (ISM) represents the input variables (v) within the input domain (d) while the control flow model (CFM) is a “virtual space” used to represent the paths exercised. At each iteration a single ant is sent to the ISM model and traverses the state space to construct “promising” program structure, meaning the likely program structure that leads to the false property. It follows the trail according to the ACO we described earlier except its probability of taking a path is based upon the pheromone on both the edge in the ISM and the edge in the CFM. That same single ant also updates pheromone intensity for both models accordingly.



Figure : State Space Exploration Models

The authors go into more detail about the algorithm by first discussing the path choice that the ant takes noting that pheromone for an edge consists of intensity and probability for the ants to follow that edge. The authors also explain how they obtain the corresponding final path for the CFM from the ISM by using randomly sampled inputs based on the pheromone levels. The updating of pheromone is then discussed by the paper in which it is noted that the pheromone “fitness” function is based on the reaching of/distance from the target node in the CFM representing the property failure. Lastly, the algorithm is described by means on saying how the model is bounded using an under-approximation technique in which the CFM trails are used to determine which multi-branch conditional statements will be pruned. After the abstraction has taken place, the bounded model is sent to a satisfiability solver and it is determined if the property can be falsified within. The authors of the paper continue by presenting the results they achieved during experimentation that shows the performance of doing this type of pruning before model checking is significantly improved.

## Paper 2

The paper entitled “Model Checking Algorithm Based on Ant Colony Swarm Intelligence”, by Wu; Hu; and Wang, presents work that uses artificial ants to traverse a model backwards to determine both correct and incorrect traces and locate errors simultaneously. The paper begins with a brief background on model checking including the use of control flow graphs and state transition graphs. The authors then present their algorithm. They choose to view a control flow graph as a large series of ant nests that can generate ants and view the program transitions as the links between the nests, as shown in Figure 3. Each nest has a table containing the pheromones for the nests that follow it, where positive values imply correct traces. When there is an error state, for example, an error ant goes through the trace in reverse (upward in the figure) and deposits error pheromone at each vertex’s entry for the state that the ant came from. Once at the home/top-most nest, the error trace is recorded. A correct ant is generated and functions analogously by depositing correct pheromone up the trace. As we mentioned in our introduction, a benefit of using ACO is that the ants can all act simultaneously. The first phase consists of ants laying pheromone and traversing upward the control flow graph.



Figure An ant nest representation of a CFG

The second phase consists of locating the cause of errors, which is accomplished by sending search ants down the CFG and following the appropriate pheromone trail. If the next edge contains both correct and error pheromone than it not the cause of the error. Edges are traversed until there is one edge that has only error pheromone. The authors provide a concrete example that illustrates the algorithm and then present their results and conclusions noting that this approach is feasible in locating errors.

## Paper 3

In the paper “Ant Colony optimization with partial order reduction for discovering safety property violations in concurrent models”, by Chicano and Alba, the authors propose the idea of using a specialized ant colony optimization algorithm with partial order reduction in order to find safety property violations in concurrent models. The paper begins with the authors providing background information on model checking, including the SPIN analyzer we mentioned earlier, safety checking, and the state explosion problem that can result. They discuss various model checking tools but note that the work in this paper is focused on non-exhaustive algorithms much like Genetic Algorithms. However, unlike genetic algorithms, they present an algorithm, ACOhg, that is intended for discover short paths representing counter examples, in graphs. The authors begin by first providing an overview of the problem of finding a safety property similar to the one we provided earlier in our paper. The authors then present the ACOhg algorithm. They note that there are two significant differences between ACOhg and the ACO we discussed earlier. Firstly, the construction phase (when the ants are building the paths and laying trail) has ants that travel a limited length path, where the length is specified in the algorithm. The second difference is that ants start from various vertices during the search in ACOhg whereas ACO has only a single starting point. The specifics of the ACOhg are presented including the fitness function that measures the quality of each path. After the construction/exploration phase of an iteration ants are used to determine the initial starting point/vertex for the next iteration and to update the pheromones after pheromone reduction has taking place. The authors then introduce the idea of using partial order reduction before beginning the ant colony optimization. They note this is feasible because the ordering between independent concurrent instructions are equal to one other so a single partial ordering can be picked to prune the state graph. The paper then goes into detail about the experiments using promela and a variant of SPIN to test if the combination of ACOhg with partial order reduction can assist in discovering safety properties that are not upheld. They show that partial ordering reduction improves ACOhg across the board in terms of memory and CPU time required for calculations.

## Paper 4

The paper entitled “Finding Liveness Errors with ACO”, by Chicano and Alba, extends the same ACOhg algorithm described in the previous section on Paper 3 to be able to discover liveness property violations. Similar to Paper 3, the authors begin the paper by motivating the need for such an algorithm because of the state explosion problem that is encountered when dealing with larger models. Background information is then presented by the authors on safety and liveness properties and model checking. The paper continues by including background information on using heuristics such as property-specific heuristics and hamming distance, which is the distance within finite state machines, which are used in the ACOhg-live algorithm presented in the paper. The authors continue by formalizing the problem of finding violated liveness properties in a concurrent system. This is followed by a presentation of the ACOhg-live algorithm. ACOhg-live is comprised of two phases. The first phase of the algorithm uses the pre-existing ACOhg algorithm to find any accepting states within the Buchi automata that we discussed previously. For each of the accepting states discovered, the next phase involves attempting to discover a cycle that involves the corresponding accepting states. If a cycle is discovered then the path is returned (a violation), otherwise the algorithm returns to phase 1 and does not include the accepting states discovered previously. The authors then present once more the original ACOhg algorithm, which is unchanged from the one presented in Paper 3. The paper continues by presenting experiments comparing ACOhg-live against the classic solution to this problem, nested-depth-first search, showing that ACOhg-live outperforms it and has less memory and time requirements. The authors acknowledge the drawback of ACOhg-live requiring many parameters and conclude the paper and present future work.

# Paper Comparisons

This section investigates the similarities and differences amongst the four papers we selected.

 Paper 1 uses the Ant Colony Optimization algorithm to determine which parts of a model representing the state space is more promising in finding a false property, that is, which part of the state space can be pruned. This is quite different then the use of ACO in Paper 2, which uses ACO to discover correct/error traces and to locate errors. It is more similar however to Paper 3 and Paper 4, which use ACO as a non-exhaustive search to discover property violations. The difference is the use in Paper 1 finds a likely area of the state space that contains the error, that is, it abstracts away unlikely non-violating parts, while Paper 3 and Paper 4 use it to ascertain the specific trace/path that a violation has occurred in.

The next area we consider is the way that each paper modified the original ACO algorithm. Paper 1 leaves the ACO algorithm relatively intact, modifying only the trail composition/choice aspects and the way that the pheromone is updated. Paper 2 makes significant modifications by having 3 different types of ants and 2 different types of pheromone. Furthermore, the ants travel in only one direction directly upward or downward. Also, a somewhat minor change is that ants leave/update the pheromone at each vertex rather than each path/edge. Lastly, the search ants have a terminating condition that is a function of the edge(s) it has traversed, whereas, typically, ants are allowed to move until the end of an iteration. Paper 3 modifies the original ACO algorithm in the two ways we already mentioned in the summary. It also includes the modification of the probabilistic choice and pheromone deposit and evaporation amounts. Specifically, for the path choice, the fitness function includes a penalty when an ant does not end in a final node, that is, the path contains a cycle and where shorter cycles imply higher penalties. Paper 4 employs the same modifications as Paper 3 to the original ACO algorithm. It is just passed different parameters and used differently. Table 1, based on a table from, presents a summary of the modifications made in each of the papers.

Table : Summarization of ACO Modifications Made

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Biology |  |  |  |  and  |
| Ant | Agent used to build solution. Travels entire search space without constraints | Builds solution on ISM. Updates pheromones on two graphs. Travels downward only | 3 types of ant. Travel in one direction only. Ants “die” when reach either top or bottom. Come from multiple starting points | Ants travel a path of predetermined length. Come from multiple starting points |
| Pheromone trail and evaporation | Constant pheromone added to edge and random evaporation | Pheromone added is based on characteristics of virtual paths in CFM. Gradual (not random evaporation) | 2 types of pheromone, correct and error pheromone. No evaporation | When an ant is in a cycle penalty values are added. Also a function of heuristic associated with property. Same evaporation as original |
| Choice / Fitness Function | Based on pheromone levels only | Distance from goal node impacts pheromone. Average pheromone merit of path is used | The choice search ants take is dictated by error trace. Follow edges with more error pheromone | Best path found is given extra pheromone through best-so-far heuristic. This influences choice |
| Termination | Desired solution achieved | Stops with two possible outputs: valid input vector leading to violation or set number of iterations | 2 phases. 1st phase has correct and error ants travel upward from various states. 2nd phase has search ants go downward from top nest | Each step/iteration consists of a number of stages. Stop after a predefined number of iterations  |

While the nature of the results are different in that the papers use ACO for different purposes, the results in all four of the papers show that using the Ant Colony Optimization algorithm is able to achieve its goal and in a way that is advantageous when compared to and/or used in conjunction with existing approaches. All four of the papers consider CPU running time as a metric and all four refer to the amount of states searched, Paper 2;Paper 3; and Paper 4, or pruned, as in Paper 1. Memory is considered only in papers 3 and 4.

# Ways to Improve

The first notion we have of improving the work done in the papers is to combine the work done in Paper 1 with the ACOhg algorithm presented in Paper 3 and Paper 4. Just as Paper 3 looks to improve/reduce the search space through partial order reduction before applying ACO or a variant of it, why not use the ACO algorithm to look for promising branches to abstract out before passing the state space to the ACOhg algorithm. We would expect a performance gain similar to the one experienced in Paper 1.

Another somewhat obvious but significant way to improve the work done in the four papers is to attempt to design a parallel version of ACO algorithm or to use an existing parallel version such as the one in. As discussed in class, many of the natural methods discussed in class and the textbook lend themselves to a parallel solution. This would improve the amount of processing time and amount of memory required, as noted in Paper 3. In regards to all four papers, having a parallel version would allow the ants’ traversal to be truly simultaneous. For cases where the ants are traveling in one direction, if we had at least one processor per ant, the run time would be O(D) where D is the depth of the state space being searched. Designing such a solution should not be onerous as each ant has the same behavior, although one consideration/constraint would be to ensure that only one ant is on an edge at a time and that each ant is getting the latest view of the pheromone of the respective edges it currently has available to choose from.

In terms of specifics with one paper, we believe that the work in Paper 2 is somewhat immature and can be advanced significantly. Firstly, much of the work appears to be manual or, if it is automatic, these details are left out of the paper and unclear. As such it is difficult to ascertain when the second (search phase) begins. According to the paper, we generate a correct or error ant at each node when it is determined (by us) if the node is a correct or error state, respectively. It seems as if combining this work with work that can automatically ascertain if a state is correct or in error would be beneficial and would make the algorithm fully automated instead of requiring manual interaction.

# Conclusions

The main conclusion we can make from reading all four papers is that the Ant Colony Optimization algorithm appears to be a very good approach to help tackle the area of model checking. All four papers were able to use the ACO algorithm for the intended purposes and all four papers recorded very promising results in regards to the CPU time and memory required when comparing it to or using it with other prominent approaches. It may be fruitful to attempt to use ACO algorithms in a similar manner on other domains that work with finite state machines to see if similar results can be received.

By drawing comparisons among the four papers, we see the flexibility of the Ant Colony Optimization approach in that different modifications of the algorithm can be used to achieve different purposes including improvement of the algorithm in various contexts. Much like in class how we were taught that an algorithm inspired from nature need not uphold the same constraints/rules of the algorithm it is mimicking, algorithms based on the original ACO algorithm need not follow the same rules or heuristics of the original ACO. In fact, by changing/specialization the algorithm, we are able to achieve the desired/better results for the specific problem being examined. Of course, sometimes this includes tradeoffs, such as the increase in parameters seen in the ACOhg algorithm.

We presented a few ways of improving the work contained in the four papers. Firstly, we noted that the work in Paper 1 could be used before running the ACOhg algorithm the same way that Paper 3 used partial order reduction before executing the ACOhg algorithm. Next, we noted that the work in all four papers would benefit from either implementing or using an existing parallel implementation of the ACO algorithm. Lastly, we presented a suggestion that the work in Paper 2 be updated to be more automatic so that a user need not indicate which states are correct and which are error states before the second/search phase of that algorithm executes.

# Works Cited

x

|  |  |
| --- | --- |
| [1] | D.C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25-31, 2006. |
| [2] | X. Cheng and M.S. Hsiao, "Ant Colony Optimization directed program abstraction for software bounded model checking," in *IEEE International Conference on Computer Design*, 2008, pp. 46-51. |
| [3] | L.N. de Castro, *Fundamentals of Natural Computing*. Boca Raton, United States of America: Chapman & Hall/CRC, 2006. |
| [4] | G.J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279-295, 1997. |
| [5] | Jackson D, "Alloy: A Lightweight Object Modelling Notation," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256-290, Aprill 2002. |
| [6] | Dingel J. (2009, November) CISC836: Models in Software Development. [Online]. <http://research.cs.queensu.ca/~dingel/cisc836_F09/readings/slides/analyzingSWModelsSmallSet1_2up.pdf> |
| [7] | F. Chicano and E. Alba, "Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models," *Information Processing Letters*, vol. 106, no. 6, pp. 221-231, 2008. |
| [8] | F. Chicano and E. Alba, "Finding liveness errors with ACO," in *IEEE Congress on Evolutionary Computation*, 2008, pp. 2997-3004. |
| [9] | X. Wu, C. Hu, and Y. Wang, "Model Checking Algorithm Based on Ant Colony Swarm Intelligence," in *Computational Intelligence and Intelligent Systems: 4th International Symposium on Intelligence Computation and Applications*, Isica, 2009, p. 361. |
| [10] | M. Randall and A. Lewis, "A parallel implementation of ant colony optimization," *Journal of Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1421-1432, 2002. |

x