2005

Learning to deal with COTS (commercial off the shelf)

Sreeram Bayana
West Virginia University

Learning to deal with COTS (Commercial off the Shelf)

Sreeram Bayana

Thesis submitted to the College of Engineering and Mineral Resources
at West Virginia University
In partial fulfillment of the requirements
For the degree of

Master of Science
In
Computer Science

Supratik Mukhopadhyay, Ph.D., Chair
Hany H. Ammar, Ph.D.
Tim Menzies, Ph.D.

Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2005

ABSTRACT

Learning to Deal with COTS (Commercial Off-the-Shelf)

Sreeram Bayana

With the advent of model based development technologies, dependence of COTS in software development has increased considerably. Use of COTS is considered economical and practical when it comes to integration of various software components. However COTS are trapped with some pitfalls. COTS provided are not usually accompanied by models or extensive specifications. This approach makes usage & integration of COTS components with in house developed software components a very challenging task. Conformance of the implementation with the specification forms the basis for our approach. In this thesis, we analyze an approach where the model is extracted from the COTS software that greatly aids in integration.

We developed a system that extracts the state machine model from the COTS software using Dana Angluin's L* Algorithm. We also developed a hierarchical approach of viewing the state machine model by static analysis of assembly code.

Keywords: Machine Learning, COTS, Learning Algorithms, Model Checking, Static analysis(Assembly), Type Inference, Model based development, L* algorithm, Type Based Decompilation.

DEDICATION

I feel honored to dedicate this publication to my family members, who encouraged and supported me during my career. I express my deep love and affection to my parents, Mr.Siva Ramakrishna Arya and Mrs.Kanya Kumari, for their continuous support and motivation that helped me pursue my higher education. I also express my love for my brother, Vinay, and sister, Manasa, for their affection.

ACKNOWLEDGEMENTS

I express my sincere thanks to Dr. Supratik Mukhopadhyay, for giving me the opportunity to work with him in his research at West Virginia University. I thank him for his motivation and guidance throughout my research program. I would also like to thank my other committee members Dr. Tim Menzies and Dr. Hany H. Ammar for their valuable time and contributions during the course of study.

CONTENTS

1 INTRODUCTION

1.1 Motivation:

Many existing software systems are using COTS, an acronym for "Commercial Off-the-shelf", 'as-is' in their products. COTS overlay a wide variety of software products including operating systems, system software, application programs, word processors etc. Mission critical systems are opting for COTS software as a result of their cost-effectiveness. Use of COTS in and as part of the larger system has effects on all product processes including architectural design, verification, validation and reliability. Such is the dependence of COTS in the industry. While model based development methodologies are being increasingly used within NASA, recent years have seen more and more dependence of NASA on COTS purchased from third party vendors. COTS from third parties are not usually accompanied by models or extensive specifications. This makes their use as well as their integration with in house developed software a challenging task. This thesis work describes a methodology to extract state machine models from COTS executables and give us a precise understanding of the models that these COTS products encompass. COTS software purchased from third party vendors is now being increasingly used within NASA. There are several reasons for this trend.

- NASA developers can focus on the core of a project leaving the outer skin to third party developers

- Lack of skilled personnel in certain areas
- Cost of developing certain components in house is prohibitively expensive compared to outsourcing them to third party vendors
- Increasing use of component based technologies
- Easier to meet deadlines

While most of the organizations continued to depend to a large extent on third party vendors, needs to amortize the cost of software development have led developers to follow a new trend. Model based development techniques [3] are increasingly being used in NASA projects. Examples of NASA projects include Livingstone [14], Remote Agent [4]. The main reasons behind the trend are as follows.

- Models developed in one project can be reused in another
- Validation can be done at an early stage of the software development cycle
- Model based programming has been found to provide cost benefits in the order of 50%
- Automatic production of reliable code is possible from the models
- Engineers describe physical systems in terms of models

However the two trends above seem to be contradictory to each other. COTS purchased from commercial organizations are usually proprietary and neither comes with the source code nor with models. As a result it becomes difficult to integrate such software with the inner core developed using model based techniques within NASA. Software from third

party comes with minimal specification. Often even the interface specifications that accompany such software are wrong. As a result it becomes difficult to find out whether a COTS product fits a requirement. Lack of models makes it difficult to validate such software as neither model based testing nor model based verification techniques [6] can be used. It is difficult to predict the behavior of such software. In case source code is available, a model can be extracted from software using well known techniques from compiler construction [5]. In our case unavailability of source code makes the problem of extracting models more difficult. In order to mitigate the problems listed above, we present a technique for extracting state machine models from executables of COTS components that are accompanied by minimal interface specifications. The basis of our method is Angluin's L* algorithm [2] that can learn from an unknown regular set an automaton recognizing it. A variant of Dana Angluin's algorithm [2] called as Predicate based L* algorithm developed as part of this thesis helps in understanding the extracted model in a much better way by giving the details (predicate information) about each particular state in the model.

1.2 Related Work

Model checking [6] and testing [13] are two of the main approaches in verification and validation (V&V) of systems. With information about the design of the system, formally checking the system for some properties to hold is model checking, where as testing is checking an implementation with respect to the abstract design where the structure of the

implementation is not known. A combination of these two techniques is used in cases where the software system is viewed as a black box. This methodology is called as Black Box Checking (BBC) [8]. The COTS obtained from third party vendors is proprietary and does not enclose either the source code or the model of the product procured from third party vendors and is considered as a black box. A combined approach called the BBC is used in cases where the software system is viewed as a black box. The system can be automatically verified by BBC [8] where a model is not known at the beginning and the model is obtained as verifications are done. The model is learned here through a series of experiments and checked using model checking [6]. Angluin's algorithm is used in BBC for learning and quickly detecting errors in the system.

Another methodology, based upon black box checking, is called the Adaptive model checking [9] which uses Black box checking as well as machine learning concepts for automatic verification of the model. Discrepancies exist between a system and its model. This can be due to errors or any modification done to the system. The availability of an approximate model that reflects the original system with respect to properties is typically assumed here. In Adaptive Model Checking, an inaccurate model is used to expedite the updates [9]. These updates are made possible by using Angluin's algorithm. This approach is applied in cases

- When a model consists of an error
- When a new feature is added to the existing system.
- When the system is upgraded to a newer version.

Adaptive model checking has been found to be more efficient than using only black box

checking [8]. A model that approximates the original system is present at any stage of the verification process. So building the model from scratch is avoided alleviating the problem further.

A technique called regular extrapolation [15] provides descriptions of systems or aspects of the system in a largely automatic way. Descriptions are available in the form of models. The model built by this approach gets updated in a system's life cycle. The input knowledge from many sources, i.e. observations, test protocols, specifications and knowledge of experts, is expressed as a finite automaton. Observations can be seen as traces of the system. These concrete traces are abstracted into an automaton whose language contains abstract images of the traces. The automaton yielded is refined by using machine learning algorithms and expert knowledge. Learning aims at classification of superficially similar states and also discovering new system traces. Expert knowledge probes in the form of declarative specifications (expressed as temporal logic). These are used to eliminate certain patterns in the model leading to a refined one.

In [10], behavior based model construction has been studied which views moderated regular extrapolation [15] from the view point of abstract interpretation, model checking and verification/validation of the model. Abstract interpretation serves as the key for applying known learning techniques for practical usage. Model checking is used as a guiding process for learning the model. Temporal logic formulas are used to depict the system. The formulas are verified at each hypothesis generated by the learning algorithm used to build the model. Any form of discrepancy in the learning process is used as a counterexample that helps in building the approximate model. The realization of this approach starts with a model (initially empty) and a set of observations. The observations

obtained in the form of traces are gathered from a reference system and preprocessed to build the model iteratively. The traces are obtained passively, i.e. by observing an already running inference system or actively, i.e. by running a simulation of the inference system with some test cases. This approach depends upon the automata learning technique, Angluin's algorithm, for its model construction. The worst-case time complexity for the algorithm is exponential with respect to the number of states where as the running time is polynomial with respect to number of states.

Capturing of temporal patterns, like identification of trends, cycles and common subsequences, from a time series data is shown in [16] using L* algorithm. Trends indicate the ramping up or ramping down of values in a time-series data, cycles are indicated by substring repetition. Subsequence can be expressed as dissimilar substrings of varying lengths interspersed with common substrings. These patterns are expressed as a Deterministic Finite Automata (DFA) which can be learned by Angluin's algorithm. The subsequences are learnt with a modified L* algorithm in the presence of a fallible teacher [17]. Similarly, to learn more about the model, a variant of the algorithm has been designed as part of our thesis. To discover sequences in a time-series data, a method called sequential pattern mining approach can also applied. This approach exceeds a predefined minimal support threshold. With the help of the minimum support, sequences which are of no interest to the problem at hand are pruned making the process of learning efficient. SPADE is one of the learning algorithms used in sequential pattern mining. SPADE [41] outperforms sequential pattern algorithms like AprioriAll and GSP by a factor of two. All the sequences using this approach are discovered with only three passes over the database.

Compositional verification [12] is one approach which addresses the state explosion problem associated with model checking. This approach employs the "divide and conquer" policy where the properties of the system are decomposed into the properties of the components. The properties of the system are proved by individually checking the properties of the components. The properties of each component are checked in an assume-guarantee style for proving the properties of the system. In order to check a component against a property, this approach generates assumptions that the environment needed to satisfy for the property to hold. The assumptions are generated with the help of a learning algorithm, in this case L*. Several frameworks have been proposed for the assume guarantee style of reasoning, but many suffer from requirement of human interaction. Each of the iterations may deduce whether the property holds true or not. This process is guaranteed to terminate. It converges into an assumption that is necessary and sufficient for the property to hold. The approach provided in [12] is incremental and fully automatic.

1.3 Objective

The objective of this thesis is to develop a learning technique that deal with problems encountered or arising out of using COTS software within NASA projects. Models of the COTS products, purchased from third party vendors, are not available. This work suggests a methodology of extracting state machine models from COTS products where the minimal interface specifications for the model are known. This work can be used to

- Check whether the component meets the requirement.
- Develop interface specifications for the component.
- Aid in integration and reuse of the component.
- Generate tests to be used in model-based and partition-based testing.

The learning technique that we propose can also be used to learn from a component its operational profile that can help in requirement analysis. In conjunction with techniques developed for static and dynamic code analysis tools, such a learning tool should be able to provide more reliability guarantees for software that will be used in safety critical missions.

1.4 Uniqueness

Machine learning techniques have been used for generating the assumptions in assume-guarantee reasoning for compositional verification and validation of software. They have been used in requirements engineering for requirements optimization, for early life cycle quality indicators, analysis of defect data, software reuse and for testing truisms in software engineering. To the best of our knowledge no work has been reported on using learning techniques for dealing with COTS, in particular learning hierarchical state models from COTS software.

2 SURVEY ON LEARNING

2.1 Introduction

This chapter gives us insight about the background information on concepts related to our thesis. This includes a brief introduction about learning & applications of machine learning that realizes the importance in real world applications. Learning systems are developed and used in both commercial & research applications. This chapter provides a survey on learning and explains in detail some of the learning strategies.

Learning [18], according to Herbert Simon, is defined as "Any change in a System that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population."

Machine Learning is used for the following reasons cited below

- To improve the understanding capability and efficiency of human learning.
- Structures or new things unknown are discovered.
- Extracting incomplete specifications about a domain.

Machine learning is being applied in many areas including speech recognition, character recognition, routing in communication networks, automatic automobile drive, program generation etc and also in many fields including banking, web applications & Bio-

technology. The training given to the learner is an important aspect of learning. Appropriate training helps the learner to generalize well on previously unseen examples.

2.2 Components of a Learning System

There are four components that make up a learning system, which are

- Performance System
- Critic
- Learning Element
- Experiment Generator

The above components form the crux for any learning system. The learning element constructs the hypothesis. The performance element uses the learned hypothesis to solve the problem at hand. Critic gives feedback to the learning element on how well it is doing. The Learning element takes the training examples as input and in gives the hypothesis in return. The experiment generator takes the hypothesis as input and suggests new problems which help in further training of the system.

2.3 Forms of Learning

The forms in which a learner is learned are broadly classified as

- Direct Learning (Supervised).
- Indirect Learning (Reinforcement).
- Unsupervised Learning.

Each of the above learning forms is dealt in detail in the sections to follow.

2.3.1 Direct Learning

This is a form of learning where a teacher or external supervisor provides the learner with examples from which the learning can take place. The examples given will act as the training data which consists of input & output pairs. The principal task of the learner is to formulate a function representation from the small training set which will represent the input output pairs and should also work reasonably well when presented with unseen input output objects. The output of the formulated function should be able to classify the class label with respect to the input object. Hypothesis is derived based on the examples at hand. The more the training, the better is the hypothesis. The supervisor/teacher then tests the hypothesis learnt by the learner. If a counter example is found, the learner has to modify the hypothesis in order to fit the counter example in the hypothesis. This cycle is repeated until no counterexample is found. This approach of supervised learning is used in areas like handwriting recognition, pattern recognition, speech recognition, spam detection etc.

2.3.2 Unsupervised Learning

Unlike the supervised learning, there is no external supervisor or a teacher. Only set of inputs are given without the outputs. From these inputs, a model is built that fits the data provided. This model can be used for decision making, predicting and reasoning. This kind of learning is often referred to as Self Organization. Unsupervised learning facilitates natural groupings or clusters of patterns. The central goal of unsupervised learning is to find clusters in the data and also model the density in the data. K-means clustering, principal component analysis, hebbian learning & vector quantization are examples of unsupervised learning algorithms. Clustering is an algorithm which, when given a set of inputs, divides inputs into classes of inputs where every input is related to other inputs in the class in some way or the other. The purpose of unsupervised learning is to know about the hidden structure of the data, encoding of the data, compressing the data etc. This paradigm is used in areas like data compression, classification etc.

2.3.3 Reinforcement Learning

Reinforcement learning (RL) is the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment [20]. Reinforcement learning [21] is learning what to do i.e. what actions must be performed in which situations. This type of learning deals with agents in a complete unknown environment where the agent learns by taking actions. The agent gets a reinforcement signal from the

environment when an action is taken. Based upon whether the signal is positive or negative, the agent acts accordingly in the future. There is a trade-off between exploration & exploitation which exists in RL. Exploration is search for new better actions where as exploitation is selecting the best action using the past experiences. RL is applied in areas like robot navigation, elevator scheduling etc.

2.4 Types of Learning

In this survey we will be focusing on some of the different types of learning strategies which include

- Statistical Learning
- Automata Learning
- Reinforcement Learning
- Knowledge-based inductive learning.

2.4.1 Statistical Learning

Statistical Learning is learning from statistical data i.e. from a statistical model. The statistical model is constructed from the statistical data. On applying probability theory and decision theory to the statistical model, we can arrive at an algorithm to learn the

statistical model. Statistical learning can be applied to large number of statistical models such as Hidden Markov Models, Linear regression, Gaussian process, Decision tree, Linear Classifier, Logistic regression etc. Statistical learning is widely used in areas like machine learning, robotics, pattern recognition (ex: handwriting recognition) etc.

The study of statistical learning can be broadly divided into 4 parts [19].

- Theory of consistency of Learning Processes.
- Non-Asymptotic theory of rate of convergence of learning processes.
- Theory of controlling the generalization ability of learning processes.
- Theory of constructing learning machines.

In the first section, Theory of consistency of Learning Processes, the necessary and sufficient conditions for consistency of learning process based on empirical risk minimization principle are studied.

The section, Non-Asymptotic theory of rate of convergence of learning processes, deals with how fast the processes are learnt.

The section, Theory of controlling the generalization ability of learning processes, deals with controlling the rate of convergence of the learning process.

The last section, Theory of constructing learning machines, deals with constructing of algorithms that control the generalization ability.

The section, Theory of constructing learning machines, introduced a famous algorithm called the support vector machine. Support vector machines are learning machines which can perform both binary classification as well as regression estimation. Statistical

learning theory contains important concepts such as the VC dimension and structural risk minimization. This theory is foundation of a real understanding of machine learning [19]. The VC dimension (or Vapnik Chervonenkis dimension) is a measure of the capacity of a classification algorithm. It is one of the core concepts in statistical learning theory and was originally defined by Vladimir Vapnik and Alexey Chervonenkis [19].

2.4.2 Automata Learning

Learning about the finite state automaton (FSA) of the program is called automata learning. The FSA learning can be used to detect anomalous program behaviors as presented in [11]. This FSA learning is based on learning sequences of system calls made by the program. The sequences are represented using a finite state automaton. The normal sequences of system calls made are learnt and any deviation from the normal is treated as anomalous behavior. The FSA learning algorithm presented in [11] learns the automata in an efficient manner which has faster learning capabilities, better detection, reduction in false positives, compact representation & fast detection than earlier FSA learning algorithms presented earlier like N-gram algorithm [22]. As each system call is made, name of the system call as well as the program counter at the point of system call are recorded. Different values of the program counter results in a different state in the automaton. To build the transitions the pair (Syscall/Program Counter) and (PrevSysCall/Previous Program Counter) are considered.

Angluin's learning algorithm L* [2] is another type of automata learning which learns the deterministic finite automaton for any regular set from a minimal adequate teacher in a time polynomial to the number of states in the automaton. A minimally adequate teacher is assumed to answer two types of queries about the unknown regular set. One type of query is whether a particular string belongs to the regular set. The second type is a conjecture where the teacher returns with a "yes" if the unknown regular set is learned correctly or a counterexample if the regular set is not learned correctly. The learner will ask membership queries based on which, the strings in the unknown regular set are classified. At any time L* has information about finite number of strings namely members and non-members. This information is organized into an observation table (S, E, T), consisting of three entities: a nonempty finite prefix closed set S of strings, a nonempty finite suffix closed set E of string and a finite function T mapping ((S U S)**.**A) to {0, 1} where A is the alphabet set. A set is prefix closed if and only if every prefix member of the set is also a member of the set. Suffix-closed is defined analogously. Some of the applications where automata learning is used are Process control, Pattern recognition, Control of service activity, Task scheduling, optimization problems, Image processing, Diagnosis, Computer vision, Concept learning etc.

2.4.3 Reinforcement Learning

Reinforcement learning (RL) [21] is learning in an environment where no information about the environment is known. The agent learns to deal with the environment on a trial

and error basis. The agent takes actions in the environment through which it learns more about the environment. The agent receives reinforcement signal from the environment which is large negative number if the agent collides with an obstacle or a high positive reinforcement signal if the agent reaches the final destination. This type of learning is unlike supervised learning which is not sufficient to learn by interaction. In interactive problems, it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act [21]. One of the challenges faced by reinforcement learning which is not seen in other types of learning is the trade-off between exploration and exploitation. Exploration is searching for best path to reach the destination. Exploitation is using the optimum best path available to reach the final destination. The agent has to explore what it already knows to obtain reward and also exploit for finding better paths for the future. There are four elements of RL:

- Policy
- Reward function
- Value function
- Model of the environment.

The policy basically defines the behavior of the agent i.e. it basically provides us with information like which actions are to be taken from a particular state. The reward function defines the goal of the agent. It basically maps state-action pairs to rewards. The main aim is to maximize the reward in RL. The value function specifies what is good in the long run of learning while the reward function specifies what is good in the immediate sense. The value function of a state gives the total value that the agent may

accumulate over the future starting from that state. The model tries to imitate the real environment. It is basically used for planning purposes. Given a particular state and an action, the model is used predict the next outcome state and also the reward obtained from that transition. There are two types of RL algorithms: Temporal Difference [21] and Q-Learning [23]. Q-learning is a form of RL algorithm that does not require any model for its environment. The Q-learning algorithm depends on estimating the values of state-action pairs. The value Q(s,a) is defined as the expected discounted sum of future payoffs determined by taking action 'a' from state 's' and adopting an optimal policy from there on wards. The optimal action of any state is the state with the highest Q-value. The algorithm is as follows.

1. From the current state s, select an action a. This will cause a receipt of an immediate payoff r, and arrival at a next state s'.

2. Update Q(s,a) based upon this experience as follows: small changes in $Q(s,a) = x[r + y\max Q(s',b) - Q(s,a)]$ where x is the learning rate and $0 < y < 1$ is the discount factor

3. Go to 1.

A *Boltzmann distribution* strategy is usually chosen that takes care of the trade-off between exploration and exploitation. It will make sufficient exploration as well as sufficient exploitation.

Temporal Difference Learning ideas blend Monte Carlo ideas and Dynamic Programming (DP) ideas. Learning here is done directly from raw experience without the

model of the environment which is similar to Monte Carlo method. Based upon the learned estimate, Temporal Difference Learning estimates in part without awaiting for the final outcome, which is the idea behind Dynamic Programming. Some of the applications of Reinforcement Learning include game playing and robotics & control.

2.4.4 Inductive Logic Programming

Inductive logic programming (ILP) is a research area which intersects the areas of machine learning and logic programming. The essential goal of inductive logic programming is to deduce theories and inducing hypothesis from computational logic. A database contains certain facts and positive and negative examples. ILP tries to deduce a program logically which makes sure that only positive examples are proved but none of the negative examples. Generalization or specialization has to be done on the hypothesis with respect to the status of the example. The hypothesis should be generalized if a new example is not covered by the old hypothesis. If a new example contradicts the old hypothesis, then the hypothesis should be specialized. The theory of ILP is based on the proof theory and model theory for the first order predicate calculus. The techniques that are included are inverse resolution, relative least general generalizations, inverse implication and inverse entailment. ILP is used in applications like data mining, automated scientific discovery, knowledge discovery in databases, as well as automatic programming [24].

2.4.5 Knowledge-based inductive learning

Knowledge Based Inductive Learning (KBL) is set of first order logic sentences describing knowledge of system as obtained from requirements. KBL has been studied in the field of inductive logic programming. The task of inductive learning is to find a model such that $Model \wedge Background \models Description$. Logic program describes the behavior of the system. It is implemented using inductive logic programming with inverse resolution. Memorization is used to bound the search in KBL.

There two principles used are

- Generalize if the hypothesis is too specific (leaves out conclusions)
- Specialize if the hypothesis is too general (produces misconclusions)

KBL reduces the complexity in two ways. Firstly, since all the new hypotheses must be in harmony with the existing knowledge, the search space of the hypothesis is reduced. Secondly, if more knowledge is available in prior, the less is the knowledge required for hypothesis. KBL can be used to learn the behavior of a logic program. We start of with observations trying to deduce general rules.

For instance if we have observations like “Socrates was a man”, “Pluto was a man”, “Socrates was mortal”, “Pluto was mortal”, we might infer the rule “All men are mortal”. The rules learned can be used to predict information about new objects. It is noteworthy that this process might also lead to wrong inferences. The more knowledge given, the more accuracy is likely to be obtained in the framing of the rules.

3 L* ALGORITHM & ITS VARIANT

3.1 Overview

Angluin's Algorithm [2] is used to identify an unknown regular set with the help of examples in the member and non-member word concept & membership queries. The algorithm is known as L*. These member and non member words are extended over an alphabet 'A'. Examples will be important source of information. The examples are likely to be chosen in such a way that they are crucial and critical to the problem at hand. Examples should represent structural completeness of the target problem. These examples help in converging faster to the correct hypothesis. The algorithm also requires a knowledgeable teacher called the minimal adequate teacher which learns the unknown regular set in time polynomial with respect to the number of states and the maximum length of any counter example provided by the teacher. A minimal adequate teacher is one which is capable of answering two types of membership queries. The first query answers whether a string belongs to the unknown regular set. The answer to this query will be 'yes' if the string is a member of the unknown regular set or 'no' if the string is a non-member of the unknown regular set. The second type of query validates a conjecture set where a 'yes' is returned when the unknown regular set is learned correctly or a counterexample is given otherwise. The counter example returned serves in correcting the wrongly learned unknown regular set. In case of many counter examples observed by the teacher, it returns any of the counter examples generated. With the help of the counter

example, a new conjecture set is formed and checked for validity by the teacher and this cycle repeats until and unless the teacher is not able to find a counter example, which implies the learner has completely learned the unknown regular set without any discrepancies. The leaner describes regular sets by means of representing them by a deterministic finite automaton (DFA).

3.2 Angluin's Algorithm (L*)

Angluin's algorithm proved that it can learn a Deterministic Finite Automata (DFA) by using queries. This algorithm is known as L* Algorithm [2]. The L* algorithm at any given time has information in the form of strings extended over the alphabet 'A', categorizing them as members and non-members of the unknown regular set 'U'. Information about the unknown regular set is stored in an observation table which acts as a data structure. It also requires a teacher which can answer to membership queries and give counterexamples if there are any for the hypothesized DFA. An observation table consists of three things.

- A nonempty prefix closed set 'S' of strings
- A nonempty suffix closed set 'E' of strings
- 'T' is a mapping from ((S U S.A) .E) to {0, 1}

The observation table is denoted as (S, E, T). A set is considered as a prefix closed set if and only if every prefix of every member of that set is also a member of the set. A set is considered as suffix closed set if and only if every suffix of every member of that set is also the member of the set. The mapping 'T' is the membership function for the unknown regular set. The observation table initially has S and E as empty sets (S = E = {λ}) and are populated as the algorithm progresses. The observation table (S, E, T) can be envisaged as a two dimensional array. The rows are addressed by elements of the set ((S U S)**.**A) and columns are addressed by elements of the set E, where the intersection of row 's' and column 'e' can be called as T(s.e). If 's' is an element of (S U (S **.** A)), then 'row(s)' defines the finite function 'f' which ranges from E to {0, 1} defined by f(e) = T(s**.**e) where '0' indicates a non-member and '1' indicates a member. The algorithm uses the observation table to construct the deterministic finite automata for the unknown regular set. The rows addressed by elements of the set 'S' are the candidates for the states being accepted, and the columns addressed by elements of the set E are for guiding distinguished experiments and the rows addressed by elements of (S**.**A) are used for the purpose of constructing the transition function.

The furnishing of the observation table can be done in a cyclic process. First the observation table is checked to see if it is closed and consistent.

The condition to be satisfied for the observation table (S, E, T) to be closed is:

For each 't' in (S.A) there exists an 's' in 'S' such that row(t) = row(s)

The condition to be satisfied for the observation table (S, E, T) to be consistent is:

Whenever s_1 and s_2 are elements of 'S' such that row(s_1) = row(s_2), for all a in A, row(s_1.a) = row(s_2.a)

If the observation table (S, E, T) is not closed or consistent, it is made closed & consistent by the discharge of the following blocks of algorithms.

If the observation table (S, E, T) is not closed

find $s_1 \in S$ and $a \in A$ such that

row(s_1.a) is different from row(s) for all $s \in S$

add (s_1.a) to S

and extend T to (S U S. A) .E using membership queries.

If the observation table (S, E, T) is not consistent

find s_1 and s_2 in S, $a \in A$, and $e \in E$ such that

row (s_1) = row(s_2) and T(s_1.a.e) $\neq$ T(s_2.a.e)

add (a.e) to E

and extend T to (S U S. A) .E using membership queries.

The table gets populated until the observation table is closed and consistent. Once the observation table (S, E, T) is closed and consistent, the hypothesized DFA is then constructed. The DFA thus formed is tested for equivalence. If the teacher returns a counter example, a new DFA is formed and then is tested for closed ness and

consistency. The counterexample returned indicates that the learning has not completed or the unknown regular set is not completely learned. This process goes on till no counter example is returned by the minimally adequate teacher. The learning process will come to a halt when no counter example is returned by the teacher and the resultant DFA is then obtained. If (S, E, T) is a closed and consistent observation table, then we define a corresponding acceptor M(S, E, T) over the alphabet A as follows

$$Q = \{row(s): s \in S\}$$

$$q_0 = row(\lambda)$$

$$F = \{row(s): s \in S \text{ and } T(s) = 1\}$$

$$\delta(row(s), a) = row(s.a)$$

Where

Q is state set of the constructed DFA

q_0 is the starting state (initial state) of the constructed DFA

F is the set of all final states of the constructed DFA

δ defines the transition function

The next section explains the algorithm with the help of an example of learning an unknown regular set.

3.3 Illustration of L* Algorithm

We present an unknown regular set and solve it by using the L* algorithm. The example is as follows.

Unknown Regular Set: Set of all strings that contain the pattern "011" over the alphabet 'A' = {0, 1}.

Initially the observation table with sets S = {λ} & E = {λ} is shown below in a tabular format. The closed ness property of the observation table is satisfied since

row(0) = row(1) = row(λ).

		E
		Λ
S	λ	0
S. Σ	0	0
	1	0

Table 3.3.1: Initial Observation table with sets S = E = {λ}.

The observation table is also consistent since

row(0) = row(0) & row(1) = row(1)

The above observation table is closed and consistent. The hypothesis or conjecture is created. The teacher validates the DFA & generates a counter example "010". The counter example "010" and its prefixes are all added to the set S in the observation table and the set (S.A) is updated accordingly using membership queries. The new observation table is shown below.

		E
		Λ
S	λ	0
	0	0
	01	0
	010	0
S. Σ	0	0
	1	0
	00	0
	01	0
	010	0
	011	1
	0100	0
	0101	0

Table 3.3.2: Observation table after adding '010' & its prefixes to set S

The observation table represented by the *table 2.3.2* is not closed since

row(011) is distinct from all rows in S.

The observation table has to be updated in order to make it closed. So '011' is added to 'S' and (S.A) is updated using membership queries. The new observation table is shown below.

		E
		Λ
S	λ	0
	0	0
	01	0
	010	0
	011	1
S. Σ	0	0
	1	0
	00	0
	01	0
	010	0
	011	1
	0100	0
	0101	0
	0110	1
	0111	1

Table 3.3.3: Observation table after adding '011' & its prefixes to set S

The new observation table in *table 2.3.3* is closed since

every row in S. Σ has a matching row in S.

The observation table is not consistent since

row(λ) = row(01) but row(1) ≠ row(011).

String ‘1’ which is the alphabet ‘a.e’ is added to E and the observation table is extended. The resulting table is shown below.

		E	
		Λ	1
S	λ	0	0
	0	0	0
	01	0	1
	010	0	0
	011	1	1
S. Σ	0	0	0
	1	0	0
	00	0	0
	01	0	1
	010	0	0
	011	1	1
	0100	0	0
	0101	0	1
	0110	1	1
	0111	1	1

Table 3.3.4: Observation table after adding ‘1’ to set E

The new observation table in *table 2.3.4* is closed since

every row in S. Σ has a matching row in S.

The observation table is not consistent since row(λ) = row(0) but row(1) ≠ row(01).

String '11' which is the alphabet 'a.e' is added to E and the observation table is extended.

		E		
		λ	1	11
S	λ	0	0	0
	0	0	0	1
	01	0	1	1
	010	0	0	1
	011	1	1	1
S. Σ	0	0	0	1
	1	0	0	0
	00	0	0	1
	01	0	1	1
	010	0	0	1
	011	1	1	1
	0100	0	0	1
	0101	0	1	1
	0110	1	1	1
	0111	1	1	1

Table 3.3.5: Observation table after adding '11' to set E

The observation table in *table 2.3.5* is closed and consistent. Now the hypothesis can be generated to check if the unknown regular set is equal to the generated hypothesis. If a counterexample is returned, the observation table must be extended to fit the counterexample. The hypothesis is generated. The teacher doesn't turn up with a counter example which states that the unknown regular set is learned completely. Thus the algorithm converges until there are no counter examples. In some cases the teacher might converge at a hypothesis which different from the expected hypothesis. This only happens when the teacher is not given enough or crucial examples required for the teacher to build the automaton. Since examples are a very important source, care must be taken to see that crucial examples for the problem are given to the teacher.

3.4 Case Study

We consider a circuit taken from [1] as a case study for studying the L* algorithm. The circuit is simulated and then learned by the algorithm with minimal interface specifications (input/output specifications). We then construct the DFA of the learned circuit. The circuit is an electrical circuit which is shown below

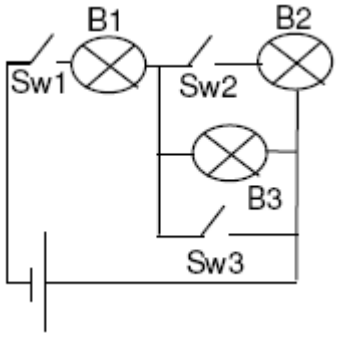

Fig 3.4

3.4.1 Description of the circuit

The circuit shown in the above *figure 2.4* has three switches namely Sw1 (Switch 1), Sw2 (Switch 2) and Sw3 (Switch 3). Three bulbs are also the part of the circuit with nomenclature B1 for Bulb 1, B2 for Bulb 2 and B3 for Bulb 3. The bulbs glow when the switches are put 'on' depending upon the design of the circuit. The current flows from positive terminal of the battery to the negative terminal. For the bulbs to glow, the circuit should be closed. The combinations of switches make the circuit either closed or open. The switches shown in the above figure are all open. Initially when all the switches are open, current tries to flow from positive terminal to the negative terminal. It has three paths via Sw3, Sw2 and B3. Since all the three switches are open, the current doesn't flow and none of the bulbs glow. For any of the bulb to glow irrespective of the state of the switches for Sw2 & Sw3, Sw1 has to be closed. From the circuit we could see that if only Sw1 is closed, the current tries to flow through the above mentioned three paths of which the current flows through only the path having B3. The other paths are closed as a result of Sw2 and Sw3 being open. Thus B1 and B3 glow if only Sw1 is closed. When Sw1 and Sw3 are closed, all the current is bypassed through the path via Sw3 because there is no resistance offered via that path. As a result only B1 glows. Similarly different combination of the switches being closed results in different combination of bulbs glowing. The complete list of combination of the switches where at least one bulb glows is given below. In all the other combinations no bulb glows since Sw1, which is the key for the circuit to be closed, is open.

If Sw1 is closed, B1 and B3 glow.

If Sw1 and Sw2 are closed, all the three bulbs B1, B2 & B3 glow.

If Sw1 and Sw3 are closed, only B1 glows.

If Sw1, Sw2 and Sw3 are closed, only B1 glows.

3.4.2 Implementation with L* Algorithm

The circuit shown above in *Fig 2.4* is simulated with the help of a program. The propositions of the circuit are incorporated into the simulation. The program gives information about the state of the three bulbs, when the state of the switches is given as input. The program is a C program compiled using gcc. The type of the executable file used for simulation is shown below.

a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), not stripped

The algorithm questions the executable regarding membership queries. These membership queries are answered by using the propositions fed into the executable. Our aim is to build a DFA which accepts the strings (that represent states) where all the three bulbs glow. This forms our member set. The strings where at least one of three bulbs does not glow form the non-member set. The teacher categorizes the strings as members and non-members depending upon the answer given by the executable. The alphabets used here for the construction of the DFA are six, 3 for closing the switches and 3 for opening the switches. Membership queries asked by the teacher are propagated to the executable

with the help of a driver program. This driver program is fabricated in java which links the output stream of the learner program to the input stream of the executable file & vice versa. The driver forwards the membership queries generated by the algorithm to the circuit simulator and in turn returns the predicate to the algorithm which makes the process automated. Running the algorithm with the above mentioned specification results in the construction of the DFA that accepts strings where all the three bulbs B1, B2 & B3 glow.

The DFA obtained from the learner is described as follows.

Q = {‘0000’, ‘0001’, ‘0011’, ‘0101’, ‘1111’} forms the state set.

q_0 = ‘0000’ is the initial state.

F = {‘1111’} forms the final state set.

Σ = {1, 2, 3, a, b, c} is the alphabet set.

Where {1, 2, 3} of the alphabet set represents switching ‘on’ and {a, b, c} of the alphabet set represents switching ‘off’ the switches Sw1, Sw2 & Sw3 respectively.

3.4.3 Why Angluin’s Algorithm is not suitable

As we can observe from the above example, that we have very little information about the states. The states for which we know enough data or propositions are the initial and the final states. The initial state (‘0000’) has all the three bulbs B1, B2 and B3 in the ‘off’ state while the final state (‘1111’) has all the bulbs glowing. The automata constructed by

Angluin's L* algorithm [2] does not give any information about the intermediate states; for example on parsing the automata with the string 'a, b, 1', we reach at a state for which we have no information of bulbs glowing. In particular, no information is provided about the propositions that hold true in these intermediate states. The automata don't give information about the state i.e. given any name of the state except for the initial and the final states, there is nothing much to interpret about it. This can be achieved by making some modifications to the algorithm resulting in its variant. This variant can be useful in giving more information about the deterministic finite automata. The next section describes about the modified L* algorithm called the Predicate Based L* Algorithm.

3.5 Predicate Based L* Algorithm

The modified L* algorithm gives information about the propositions that hold true for any particular state. This is called the Predicate Based L* algorithm where predicates give information about states in the state machine. The following are the changes made to the conventional L* algorithm [2] in order for the required improvements to reflect. In the L* algorithm, the mapping function T maps to {0, 1} which classifies all the strings of the unknown regular set into member set and non-member set. In the modified algorithm, examples are classified into predicates rather than members and non-members which make the function T map to the available predicates that conform to the automata. For example, if there are four states in the automata namely '0', '1', '2' & '3', the function T maps to {'0', '1', '2', '3'}. So the strings will be classified based on the

predicates unlike members & non-members in L* algorithm [2]. Therefore there will be strings that belong to predicate 0, predicate 1, predicate 2, and predicate 3 & combinations of predicates. When a membership query is asked for a particular string, the name of the predicate to which the string belongs is returned unlike member/non-member returned in the traditional algorithm. There were only two files (member/non-member) where the strings are stored in the former approach. In the modified approach, each predicate has a file that store strings which belong to that particular predicate.

If (S, E, T) is a closed and consistent observation table, then we define a corresponding acceptor M(S, E, T) over the alphabet A as follows

$$Q = \{row(s): s \in S\}$$

$$q_0 = row\ (\lambda)$$

$$\delta\ (row(s),a) = row(s.a)$$

where

Q contains all the states that are formed from the predicates and their combinations.

q_o is the initial state & the transition function δ are left unchanged.

The final state for the automata does not exist as the final state for each string differs. Each string belongs to a particular predicate. So after the hypothesis has been created, each string in the predicate file is tested to see whether the string belongs to that particular predicate. If this test satisfies for all the strings in all the predicate classes, there is no counter example, otherwise the string is returned for which the above condition doesn't hold and its acts like a counter example. The automaton is again build to incorporate the changes which is free of any counter examples.

3.5.1 Implementation with Predicate Based L* Algorithm

The modified algorithm is tested with the same circuit taken from [1] as shown below. This circuit is simulated with a C program specifying which bulbs glow, when given the state of the switches. The states in this case with respect to circuit are

{0}	-	All the bulbs (B1, B2 & B3) are in the 'off' state.
{1}	-	Only B1 is glowing ('on' state).
{2}	-	Only B2 is glowing ('on' state).
{3}	-	Only B3 is glowing ('on' state).
{1, 2}	-	Bulbs B1 & B2 are glowing ('on' state).
{2, 3}	-	Bulbs B2 & B3 are glowing ('on' state).
{1, 3}	-	Bulbs B1 & B3 are glowing ('on' state).
{1, 2, 3}	-	All the three bulbs (B1, B2 & B3) are in the 'on' state.

Of the entire above present states/predicates, some of these states are not reachable for this particular circuit because of design restrictions incorporated. So states like {2}, {3} & {2, 3} are not appeared in the state diagram since Sw1 (acts like an "and" gate) has to be closed for the circuit to be closed. Given the state of the switches in the circuit, i.e. for example if switch 1 and switch 2 are on, the predicate returned is {1, 2, 3} (all the three bulbs are glowing) instead of member/non-member getting returned. The predicate gives information like what bulbs are glowing in a particular state unlike the tradition L*

algorithm. The finite automaton constructed is viewed with the help of Autograph [7], which is used for displaying state diagrams graphically. Autograph is one of the graphical editors available is used for displaying automata. The automaton is converted into a format called the 'fc2' format. Autograph builds the state diagram by understanding the automaton specified in the 'fc2' format with the help of the 'fc2' grammar. The automaton constructed using the predicates algorithm for the above circuit is shown below with the help of an "atg" snapshot.

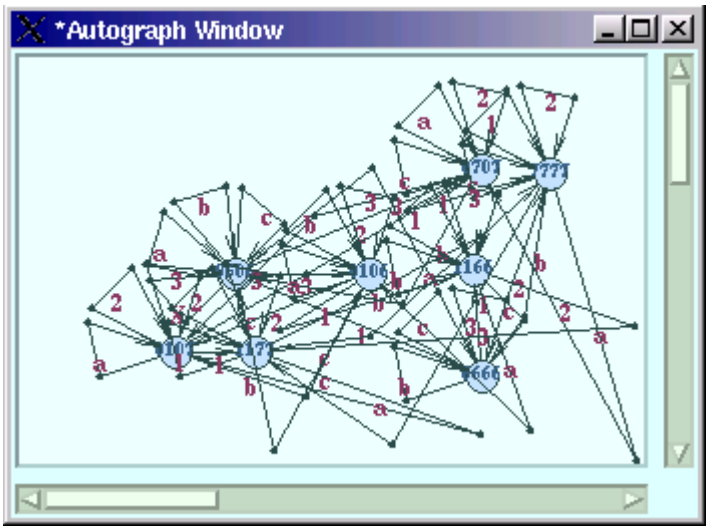

Fig 3.5.1: DFA generated with Predicate Based L* algorithm (Atg Snapshot)

'1' indicates that switch 1 is in the "on" state, '2' indicates that switch 2 is in the "on" state and '3' indicates that switch 3 is in the "on" state. Similarly 'a', 'b' and 'c' indicates that switch 1, 2 and 3 are in the "off" state respectively. For example in the above circuit, if switches 1 & 2 are in the "on" state the three bulbs glow i.e. {1, 2, 3}. The states are named in such a way that it has significant meaning. The states extracted from this circuit are the following {'0606', '0707', '0106', '0107', '1166', '1177', '6666', '7777'} eight states.

The state whose name start with a

‘0’ implies that no bulbs are glowing.

‘1’ implies that bulb B1 is glowing.

‘6’ implies that bulbs B1 & B3 are glowing.

‘7’ implies that all the bulbs B1, B2 & B3 are glowing.

Thus the propositions that hold true for the intermediate states are also obtained. These propositions help in better understanding of the model when dealing with COTS.

The circuit as shown below is also learned by this algorithm which is also simulated like the above circuit.

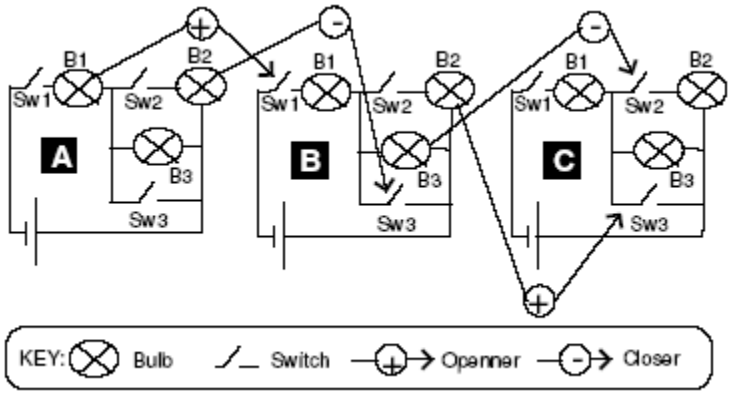

Fig 3. 5.2

The deterministic finite automata constructed using the algorithm and a circuit simulator has 36 reachable states. As the number of states increases, we are challenged by the state space explosion problem. So a hierarchical approach of constructing state diagrams is chosen to get bypassed with the state space explosion problem. The next chapter explains about the approach that was adopted to overcome the blowing up of states.

4 LEARNING HIERARCHICAL STATE MACHINES

4.1 Motivation

This chapter describes a hierarchical way out from state space explosion problem using static analysis of executable's assembly code. One of the approaches in verifying the correctness of a system is by using state spaces. The state space constitutes of all the states a system can reach and all the transitions possible between the different states. The construction of the state space by the learning algorithm [2] is fully automated. They sound as the ideal verification technique, but they suffer from a big and fundamental problem known as the state space explosion. Numerous systems have very large unmanageable state spaces. The states grow exponentially with respect to number of processes and variables. The exponential base depends upon the number of values a variable can take or the number of local states available. Though many approaches are proposed to pacify the state explosion problem, but they suffer from restriction of the verification questions that can be answered. So we deviate from the above problem with the construction of hierarchical state machines. In this approach static analysis is done on the COTS executable at the assembly level to extract the hierarchy embedded in the COTS executable. The hierarchical approach can be illustrated from the following valves example.

4.2 Valves Example

As shown in the following *fig 4.2*, there are three valves named as V1, V2 & V3 which facilitate the inflow through the vertical pipes. Similarly there are three other valves named as X1, X2 & X3 which allow the inflow through the horizontal pipes. All the six valves can be in one of the two states {'open', 'close'}. Different acids flow through the vertical pipes & water flows through the horizontal pipes. Each vertical pipe is connected to exactly one horizontal pipe. The pipe with valve V1 is connected to pipe with valve X1. Similarly pipe with valves V2 & V3 are connected to pipes with valves X2 & X3 respectively. The objective is to collect acidified water at the end of the pipes bearing the valves X1, X2 & X3. In order to collect acidified water, the valves X1, X2 & X3 should be open when V1, V2 & V3 are open respectively and valves X1, X2 & X3 should be closed when V1, V2 & V3 are closed respectively. Thus we can guarantee the acidified water can be collected. Sulphuric acid, Nitric acid & Hydrochloric acid flow through the pipes with valves V1, V2 & V3 respectively. The number of states for the valves example is 2^6. In real time many system state space explodes exponentially. We simplify this by static analyzing source code of the COTS executable at the assembly level to get the target state when an executable runs with the given input. When the input for the above valves system is given as V1 = 'open', V2 = 'close' & V3 = 'open' and there is a function 'setvalve(X1, X2, X3)' that assigns the value of these valves, we denote the final state with values F1, F2 & F3 having the values i.e. F1 = X1, F2 = X2 & F3 = X3 for the above example. The trace of the program along with function signatures is required to specify the final state in this approach. This realizes the role of static analysis in

extracting type information. Type information for the function parameters is not available at the assembly level. The typed information is retrieved by using an approach called type based decompilation which is explained in detail in the coming sections.

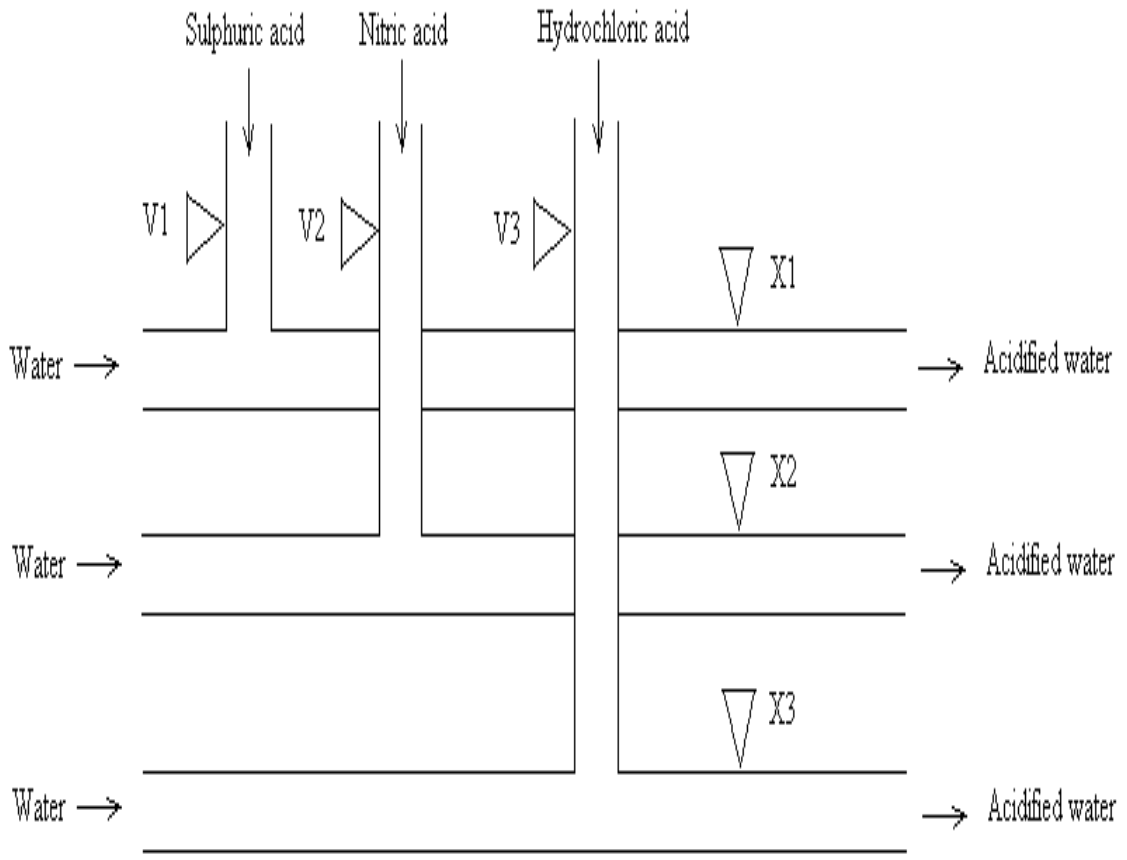

Fig 4.2: Valves Example.

The static analysis of COTS at the assembly level yields to hierarchical state machine of the valve example, when executed with a specified input. The state machine obtained with the input X1, X2 and X3 in the function setvalve(X1, X2, X3) is shown below.

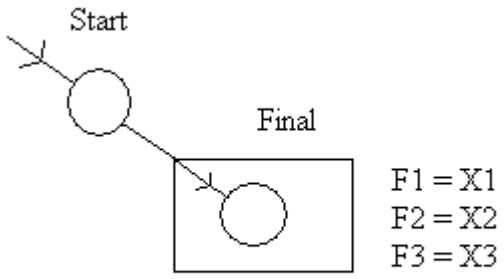

Fig 4.2.1: Hierarchical State Machine

Instead of spawning the entire state space of 2^6 states, we disembark at the above hierarchical state machine specifying the final state reached. The final state is expressed as F1 = X1, F2 = X2 and F3 = X3 where X1, X2 and X3 are passed as parameters into the function 'setvalve' that decide on the final state being reached. Thus our approach deals with the explosion of states avoiding all the state space except the final state. The next section gives concise introduction about static analysis.

4.3 Static Analysis

Static Analysis is a form of program analysis where the analysis of the program is done without executing the program. There exist several tools which are used for static analysis. CodeSurfer [25], C Global Surveyor [26] & Vault [27] are some of the static analysis tools used in today's world. All the above stated tools require source code to do the analysis. Assuming that the COTS are presented without the access to the source

code, many of the static analysis tools won't be of much help to the problem at hand. Although substantial research has been done in static analysis with respect to assembly level code, most of the research is focused on security issues. [28] is one such example where the aliasing analysis is done at the machine code level i.e. it checks whether two instructions access the same memory. Assembly level static analysis is done in [29] which checks for the compliance of rules. The COTS model is tested for the compliance with these rules. Static analysis is used as an important tool to preserve quality assurance [30]. We have done static analysis at the assembly level to obtain the type information of the function parameters required to represent in this process of constructing hierarchical state machine. The type information is obtained by evaluating type constraints imposed on the function parameters. In order to impose constraints on the parameters of the function, a necessary requirement is to know the function call conventions and the stack. In particular we here are dealing with the C language using the compiler gcc on Linux, running on Intel x86 architecture.

4.4 C Function Call Conventions and Stack Organization

The C function call conventions and the organization of the runtime stack is discussed in detail in [31]. The following conventions are subjected to a system with Intel x86 architecture operating Linux with a gcc compiler on it. The caller and the called function come to a consensus on how parameters are passed between the two and the organization of the stack. These conventions may vary depending upon the compiler, architecture or

the operating system. The portion of the stack used for invocation of a function is called stack frame or activation record. When ever a function is called, a new activation record is created on the runtime stack and all the necessary information regarding the function is stored in that activation record. The runtime stack grows upward i.e. smaller number memory locations are located on the top. The caller's activation record stays at the bottom and the callee's activation record is at the top of the caller's record. Some of the registers that are involved in the function call frame and their usage are described below.

ESP – Stack Pointer

This 32-bit pointer is manipulated by several instructions such as PUSH, POP, CALL, RET etc. This pointer is used to point the top of the stack i.e. last element residing on the runtime stack. The PUSH and the POP operations result in incrementing and decrementing the ESP register respectively.

EBP – Base Pointer

This 32-bit pointer is used as a reference pointer to point to the local variables and all the function call parameters in the current activation record. The base pointer is also known as "Frame Pointer".

EIP – Instruction Pointer

This 32-bit pointer points to the address of the next instruction which is to be executed. This pointer is saved on the runtime stack whenever a function is called so that the next

instruction after the function call is executed as soon as the control returns from the function.

The stack organization will be demonstrated with an example where the main function makes a call to a function named 'foo' with three arguments a, b & c.

$$x = foo\ (a, b, c)$$

In our example the caller is the 'main' function and the callee is the 'foo' function. Before the function 'foo' is called the main function is using ESP and EBP for its activation record. One of the conventions used here is that the callee is allowed to use the registers EAX, ECX and EDX. So these registers are stored by the caller on the stack and retrieved on function exit. On the contrary, the callee must save the registers EBX, ESI and EDI registers. If the values of these registers are changed by the callee, it is the responsibility of the callee to restore the original values before the function returns. Parameters passed from the main function to the foo function are stored on the stack. The last argument is pushed first which apparently makes the first argument on top. The EAX register is used to store the return value of 4 bytes. If the return value is more than 4 bytes, the caller passes an extra parameter which specifies the address of the location where the return value is stored at. This is only the case where the return value is greater than 4 bytes.

Caller's action before function call

'Main' function pushes the contents of the registers EAX, ECX & EDX on the stack as needed. The next step is to push the parameters of the 'foo (a, b, c)' function call. The last argument is pushed first and the rest are followed in the order.

push dword 18

push dword 15

push dword 12

Now ‘main’ can issue the call instruction.

call foo

After the call instruction has been executed, the contents of the instruction pointer are pushed on to the top of the stack. This instruction pointer contains the address of the next instruction to be executed after the function call in the ‘main’, which results in the return address being on the top. After the function ‘foo’ has been executed and returned, the stack returns back to this position. The above *fig 4.4.1* shows the stack contents just before a function call.

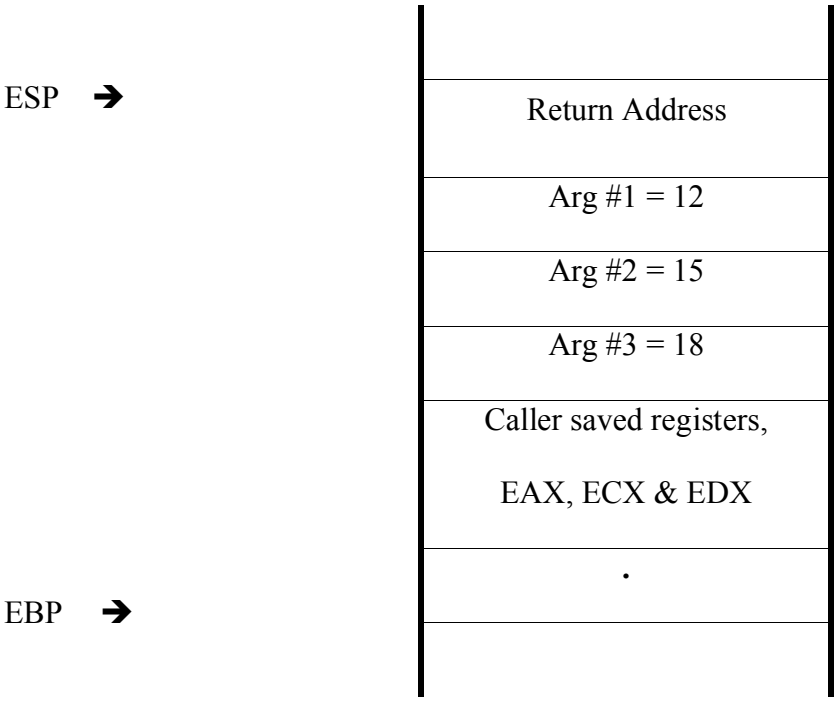

Fig 4.4.1: Runtime Stack Frame before calling function ‘foo’.

Callee's actions after the function call

After the call instruction, the function 'foo' will get the control of the program. The function has to do the following three things:

- The function has to set up its own activation record.
- Enough space should be allocated to store local information such as local variables.
- Necessary registers must be saved.

Before setting up the stack, the EBP pointer is pointing to a location in the main's stack frame. So the main's EBP is saved by pushing on to the stack & the contents of ESP are stored into the EBP pointer. This arrangement facilitates us to access the function parameters and local variables to be referenced with respect to EBP. The function call results in the following two instructions being executed.

push %ebp

mov %esp, %ebp

The next step is to leave enough space for the storage of local variables and some temporary information. Some complicated expressions involve storing the value of sub expressions in some temporary location. The instruction which requires 12 bytes of space can be showed as

sub $12, %esp

The last step implemented by the callee before returning the control to the caller is saving any registers as needed. If the callee requires the use of EBX, ESI & EDI, they are stored on the activation record. The callee now can push and pop things from the stack, but the EBP pointer is fixed which makes the access to the local variables and parameters easy. The first parameter of the function is located at a positive offset of 8 bytes from EBP i.e. EBP+8. The function might result in other functions being called or the same function being called recursively. As long as the EBP is saved on the stack, the references to local variables and parameters can be made without any discrepancies. The location of the local variables and parameters at offsets from EBP can be shown in *fig 4.4.2*.

Callee's action before returning

The callee must return value to the caller. This can be done by storing the return value in the EAX register. The registers EBX, ESI & EDI are restored by the values stored on the stack and are popped off later. The temporary memory and the memory allocated for the local variables are no longer required. So they are also popped off. This can be done by reducing the stack frame with the following instructions.

mov %ebp, %esp

pop %ebp

This results in the same stack arrangement as shown in *Fig 4.4.1*. The return instruction is executed next which stores the return address in the instruction pointer (EIP) & pops the return address. I386 instruction set has the *leave* instruction which performs the task done

above by the *mov* and *pop* instructions. So it is very typical to have a C function which ends with a *leave* instruction followed by the *ret* instruction.

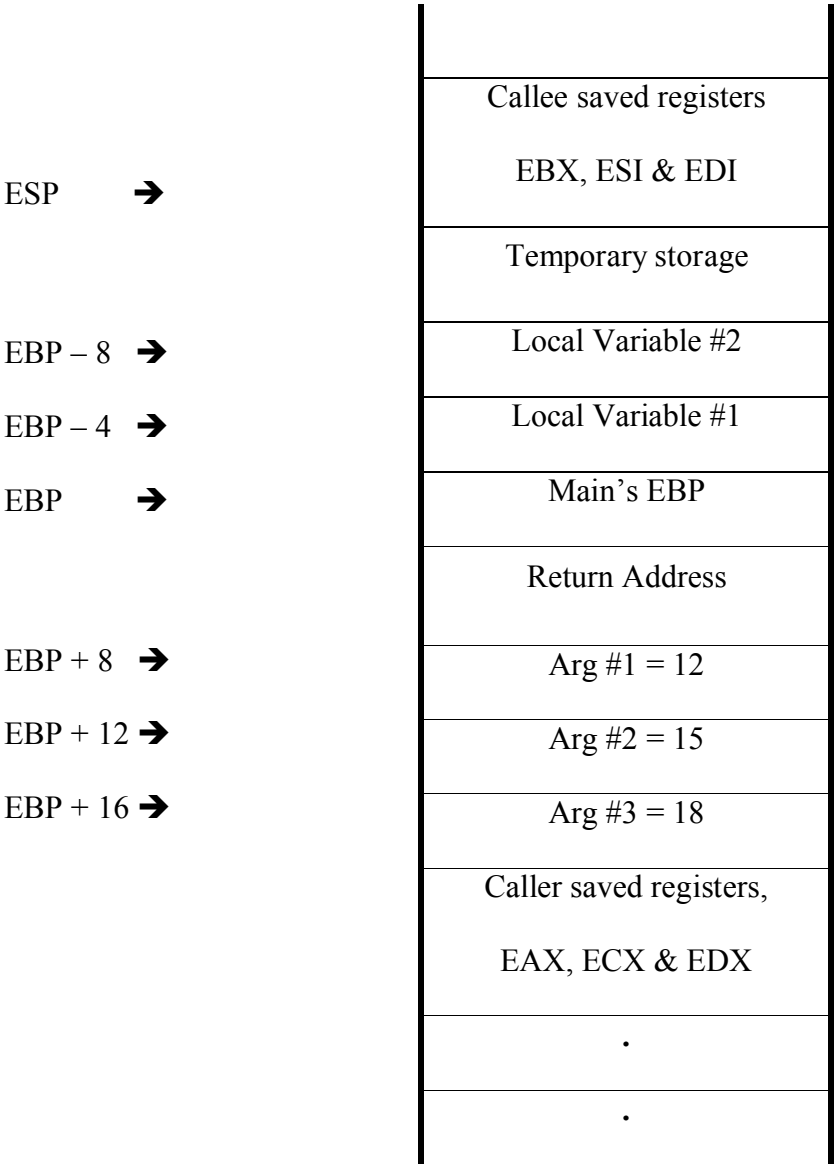

Fig 4.4.2: Offsets of parameters & local variables from EBP in Stack Frame

Callee's action after returning

After the caller regains the control from the callee, the arguments passed by the caller are

no longer required by the caller. So they are popped and the esp is adjusted accordingly. In this case there are 3 arguments and the esp is added by 12(4 bytes for each variable). The last thing done by a caller is restore the caller-saved registers EAX, ECX & EDX.

4.5 Type Inference

Type inference is the converse process of constructing type information that was omitted in compilation phase of the program. Inference is a process of constraint satisfaction. Equations are deduced based on the relationships between the types. These equations are solved by a process known as unification [32]. The equations can be classified as follows

- *Overconstrained*: There is no solution which means it relates to a type checking error
- *Underconstrained*: There is more than one solution which means it has ambiguous solutions.
- *Uniquely determined*: There is exactly one solution without any ambiguity.

Examples of type inferring languages include Haskell [33], ML, and MUMPS. Program construction is done through type reconstruction which decompiles C programs from target machine code by type inference techniques which uses Milner's algorithm [34] as the basis [35]. A type system used in a disassembler to produce enhanced assembly output by using type library files was discussed in [36]. Another variant is the use of proof carrying code in typed assembly language used by Morrisett et al [37]. Proof

carrying code facilitates the safe usage in a security domain. Preserving the type information increases the reliability of the compiler. We take as reference the method specified in [35]. We construct the signature of the functions by generating type constraints on the function parameters & thereby unifying the constraints leads us to the type information of the parameters. Target code from RTL (Register transfer language) is used to infer the types. RTL code can be obtained by disassembly of assembler files, object files, directly from compiler output or from Dynamically Linked Libraries. The disassembler used in our case for type inference is 'objdump'. A type inference engine is developed which creates constraints on the basis of the context. After all the constraints are obtained, these are solved leading to the parameter types. The type inference algorithm, steps involving the deduction of parameter types, is explained in the next section.

4.6 Type Inference Algorithm

This algorithm infers function signature concerning the parameters. Parameters are referred as stack references. Inference is done by algorithm explained in the following steps.

- Identification is done for all stack references that reflect the parameters.
- Constraints are generated based on the context in which the reference of the parameter is involved.

- All the constraints are obtained and are evaluated. This evaluation of the constraints is a process known as unification.
- If the unification process yields to under-constrained solution, then backward analysis is applied. Backward analysis is the course of action of analyzing the code in the caller's stack frame for the stack reference being inferred. The caller's code is analyzed and constraints are formed which assists in reducing to an unambiguous solution.

The unification process fosters some rules. The rules followed in the unification of constraints are

- If all access via a pointer are of the same size, then the unified type is an array.
- If the constraints to a given pointed type are all 'struct' types, then the resulting unified type is also a 'struct'.

Unification of constraints is done after all the constraints are generated. We currently implemented an inference engine which infers all the types except the C structure (struct type. Some light can be thrown on the algorithm by the subsequent example.

4.7 Intuitive Example

Consider the following function written in C. This example illustrates the constraint formation and solving of constraints.

```
int Foo( int i, float k, char *d)
{
        return (int)(k+i*d[5]);
}
```

Disassembling is done on the binary format of the above function with a disassembler- in this case 'objdump' [38].

Disassemble is done using the following objdump options.

```
murdock:~> objdump –dr Foo.o
```

'-dr' are options given to 'objdump' which indicates disassemble and relocatable entries respectively. The option 'd' displays assembler contents of executable sections, where as option 'r' displays the relocatable entries in the file. The 'Foo' function has three parameters which are i – integer, k – floating point & d – pointer to a character. Since we are considering only the assembly code in the function itself, the constraints generated might not be sufficient to infer a type. So we employ the *backward analysis* where we also examine the caller for generating constraints. The assembly code for the 'Foo' function is shown below.

```
Foo.o:     file format elf32-i386

Disassembly of section .text:

00000000 <foo>:
  0:   55                 push   %ebp
  1:   89 e5              mov    %esp,%ebp
  3:   83 ec 08           sub    $0x8,%esp
  6:   8b 45 10           mov    0x10(%ebp),%eax
  9:   83 c0 05           add    $0x5,%eax
  c:   0f be 00           movsbl (%eax),%eax
  f:   0f af 45 08        imul   0x8(%ebp),%eax
 13:   50                 push   %eax
 14:   db 04 24           fildl  (%esp,1)
 17:   8d 64 24 04        lea    0x4(%esp,1),%esp
 1b:   d8 45 0c           fadds  0xc(%ebp)
 1e:   d9 7d fe           fnstcw 0xfffffffe(%ebp)
 21:   0f b7 45 fe        movzwl 0xfffffffe(%ebp),%eax
 25:   66 0d 00 0c        or     $0xc00,%ax
 29:   66 89 45 fc        mov    %ax,0xfffffffc(%ebp)
 2d:   d9 6d fc           fldcw  0xfffffffc(%ebp)
 30:   db 5d f8           fistpl 0xfffffff8(%ebp)
 33:   d9 6d fe           fldcw  0xfffffffe(%ebp)
 36:   8b 45 f8           mov    0xfffffff8(%ebp),%eax
 39:   c9                 leave
 3a:   c3                 ret
```

The parameters referenced here are 3 – ebp(0x8), ebp(0x10) & ebp(0xC). The function may have 4 or more parameters of which it uses only three. For the complete function signature, we also examine the number of parameters pushed on to the stack by the caller. The first parameter, ebp (0x8), can be perceived as an integer because of the multiply instruction '*imul*'. If there was an addition operation, it would have generated two constraints – pointer (pointer arithmetic) or an integer. In this occasion, backward analysis is applied to generate more constraints that serve in eliminating the ambiguity. The second parameter is a floating point since all floating point operations are done separately on a co-processor 80x87 FPU with special floating point instructions. All floating point instructions start with the letter 'f'. The 80x87 provides eight 80-bit data registers organized as a stack. The third parameter can be a pointer to a character or

perhaps a ‘Struct S*’ where S is a struct(or union) with a char at offset 5 or even a char array at offset less than 5.

One of the main problems with C is the expressiveness of C’s struct, union or array types compared to those of Java. Arrays or structs containing other arrays cannot in general be uniquely decoded [34]. Consider the situation where an array lives inside a struct, which can be viewed as an array of structs or just an array. These are the points of interest which should be addressed by proof carrying code. Given a proof carrying code, inference of types in an efficient way is quite possible.

GNU GCC [39] is the compiler used in this case. GNU Compiler Collection is abbreviated definition of GCC. GCC is an integrated distribution of compilers for several major programming languages including C, C++, Objective-C, Java, Fortran, and Ada. GCC is one of the most optimizing compilers (i.e. produce optimized code) in the available lot. Since we have extracted the type information for all types (except structures) quite comfortably, our method can be extended to any other compiler and is bound to extract substantial type information.

4.8 Experimental Setup

Once the function signatures with respect to type information are available using the approach specified above, we use GDB to debug the binary COTS executable and find the values of the parameters. GDB stands for “GNU Project Debugger”. GDB [40] allows

seeing what is going inside other program while it executes or what happened when the program crashes. The binary executable COTS file is debugged using GDB. The command for debugging using gdb is

gdb <filename>

The above command lands at 'gdb' prompt. The names of all the functions in the binary executable are obtained from 'objdump'. Breakpoints are set at the start of all user defined functions which facilitates seeing the exact values of the function parameters before any modifications done in the course of execution of the program.

gdb>break <function-name>

The debugger is started by using 'start' or 'run' commands.

gdb> start

The debugger starts executing and stops at the first function call made. The inference engine developed as part of this thesis gives the number of arguments including their types for a particular function. With this information, memory is examined at that particular breakpoint which reveals the values of the parameters. The command for examining memory at a given address is examined by

gdb>x/nfu <address>

n,*f* and *u* are optional parameters which specify how much memory to examine. '*u*' specifies the unit of memory to examine. The different sizes of memory are specified by

b – specifies byte

h – examine halfwords (two bytes)

w – words (four bytes).

g – gaint words (eight bytes)

For example

gdb>x/w <address>

examines a word(four bytes) at the specified address.

The values examined are used to display the state machine in the hierarchical approach.

5. CONCLUSIONS & FUTURE WORK

This thesis presents the way of dealing with COTS software. We implemented a method of extracting the model from a COTS executable. We used L* Algorithm, an algorithm used to learn regular sets from queries and counterexamples, in our approach of extracting the model. The learning of the model is incremental and fully automated. The most important thing that differentiates our research from most of the related work is the construction of model from COTS executable. The model is learned by learning algorithm L* by querying the executable. This realizes the knowledge of the input and output specifications of the COTS executable. The learning algorithm is assumed to have additional information available to the learner. Thus the queries from L* may include strings that do not belong to the set of original episode strings. The learner learns from the teacher. A minimally adequate teacher is assumed to answer two types of queries. Firstly membership query, the answer is yes or no depending upon whether the string is a member of the unknown set or not. The conjecture contains description of the regular set. The answer is yes if the unknown regular set is equal to the current description or a counterexample which specifies the symmetric difference between the two regular sets.

The learning algorithm learns the model of the executable once the specifications are obtained. A variant of the L* algorithm called the Predicate Based L* algorithm is obtained by modifying the basic algorithm. This modified algorithm aids in understanding the model better by providing with further information about the model extracted. The model extracted may suffer from state explosion problem. So a

hierarchical approach is employed that deviates from this problem. The hierarchical approach is obtained by static analysis of the assembly code of the executable. Assumed is a function in the executable which results in the final state being decided. The values of the function parameters realize the final state. These values are recovered by debugging the executable after finding the type information which was lost during compilation. The inference of types is done using static analysis at the assembly level. Constraints are formed on the parameter types which when unified gives the type of the parameter. Backward analysis (i.e. analysis at the caller function) is also performed in case the constraints are not solved in the current stack frame. Thus the hierarchical approach results in the final state being obtained based upon the input. As part of the future work, the model thus obtained can be represented in temporal logic formulas to be fed into a model checker for conformance of the model. Conformance of the abstract design is done with the original design. We would like to analyze on the scalability of our work by extracting the state machine model from a real time COTS product available in the industry. We also like to investigate whether the learning algorithm can be made more efficient in our milieu.

REFERENCES

1. I. Bratko, Prolog Programming for Artificial Intelligence, (*third edition*). *Addison-Wesley*, 2001.

2. Dana Angluin, Learning Regular Sets from Queries and Counterexamples, *Information and Computation*, 1987, Vol-75.

3. Brian C. Williams & P. Pandurang Nayak, A Model-based Approach to Reactive Self-Configuring Systems. In *Proceedings of AAAI-96, 1996.*

4. Barney Pell, Douglas E. Bernard, Steven A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams, An Autonomous Spacecraft Agent Prototype, In *Proceedings of the First International Conference on Autonomous Agents,* Marina del Rey, CA 1997

5. A. V. Aho, R. Sethi & J. D Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, June 1987.

6. Edmund Clarke, Orna Grumberg & Doron Peled, Model Checking, The MIT Press, Cambridge, Massachusetts, 1999.

7. A Quick Introduction to Autograph (Atg),
http://www-sop.inria.fr/meije/verification/index.html

8. Doron Peled, Moshe Y. Vardi & Mihalis Yannakakis, Black Box Checking, Journal of Automata, Languages and Combinatorics, Volume 7 , 2001

9. Alex Groce, Doron Peled & Mihalis Yannakakis, Adaptive Model Checking, In *Computer Aided Verification (CAV)*, pages 521--525, Copenhagen, Denmark, July 2002

10. Bernhard Steffen & Hardi Hungar, Behavior-Based Model Construction, *VMCAI 2003*, LNCS 2575, pp 5-19, 2003.

11. R. Sekar , M. Bendre , D. Dhurjati , P. Bollineni, A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors, *Proceedings of the IEEE Symposium on Security and Privacy*, p.144, May 14-16, 2001

12. Jamieson M. Cobleigh, Dimitra Giannakopoulou, Corina S. Pasareanu, Learning Assumptions for Compositional Verification, *In Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

13. G. J. Myers, The Art of Software Testing, *Wiley International*, 1979.

14. Barney Pell, Douglas E. Bernard, Steven A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams, A Remote Agent Prototype for Spacecraft Autonomy, In *Proceedings of the SPIE Conference on Optical Science, Engineering, and Instrumentation, 1996.*

15. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R. Kutsche and H. Weber, editors, *Proc. of the 5th Int. Conference on Fundamental Approaches to Software Engineering (FASE '02)*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

16. Lynne Vettel, Raj Bhatnagar, Learning Automata to Capture Temporal Patterns, *Proceedings of the 2002 International Conference on Machine Learning and Applications,* ICMLA 2002.

17. Ron D, Rubinfeld, Learning fallible finite state automata. *Machine Learning, 18:149-185*, 1995.

18. G. F. Luger and W. A. Stubblefield, Artificial Intelligence: Structures and Strategies for Complex Problem Solving, The Benjamin/Cummings Publishing Company, Inc. 1989.

19. The Nature of Statistical Learning Theory by Vladimir N. Vapnik.

20. Leslie Kaelbling, Michael Littman, Andrew Moore. Reinforcement Learning: A Survey, *Journal of Artificial Intelligence Research 4* (1996).

21. Richard Sutton & Andrew Barto, Reinforcement Learning, MIT Press, 1998.

22. S.Forrest , S.A.Hofmeyr, A.Somayaji, Intrusion Detection using sequences of system calls, *Journal of Computer Security Vol 6(1998) pg 151-180*.

23. Watkins, Christopher J.C.H. (1989), Learning from delayed rewards, PhD thesis, University of Cambridge, Psychology Department.

24. Advances in Inductive Logic Programming (Frontiers in Artificial Intelligence and Applications, 32) by L. De Raedt.

25. Anderson, P., Reps, T., and Teitelbaum, T., Design and implementation of a fine-grained software inspection tool. In *IEEE Trans. on Software Engineering 29* 8 (Aug. 2003), 721-733.

26. Brat, G., and, Klemm, R. "Static Analysis of the Mars Exploration Rover flight software." *In Proceedings of the 1st International Space Mission Challenge for Information Technology, pp. 321-326. Pasadena*, California, 2003.

27. Robert DeLine and Manuel Fähndrich. "Enforcing high-level protocols in low-level software." In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2001, pages 59-69.

28. W. Amme, P. Braun, E. Zehendner, F. Thomasset. Data Dependence Analysis of Assembly Code. *Proc.PACT* 1998.

29. R Venkitaraman and Gopal Gupta, Static Program Analysis of Embedded Executable Assembly Code. *Compilers, Architecture, and Synthesis for Embedded Systems (ACM CASES)*, September 2004.

30. David A. Wagner. Static analysis and computer security: New techniques for Software Assurance. University of California at Berkley, PhD Dissertation, Dec. 2000.

31. © Richard Chang, 2001, http://www.cs.umbc.edu/~chang/cs313.s02/stack.shtml.

32. http://www-2.cs.cmu.edu/~rwh/introsml/core/typeinf.htm.

33. Glasgow Haskell Compiler, http://www.haskell.org/ghc/.

34. Mycroft Alan, Type-Based Decompilation. *Lecture Notes in Computer Science: Proc. ESOP'99, vol. 1576, Springer-Verlag*, 1999.

35. Milner R, A Theory of Polymorphism in Programming, *JCSS 1978*.

36. Guilfanov, A Simple Type System for Program Reengineering. *WCRE 2001: 357-361.*

37. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *In TwentyFifth ACM Symposium on Principles of Programming Languages, pages 85--97, San Diego*, January 1998.

38. http://www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html.

39. GCC – GNU Compiler Collection, http://gcc.gnu.org/.

40. GDB – GNU Project Debugger, http://www.gnu.org/software/gdb/gdb.html

41. Zaki, M. J. 2001. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning 42, 1/2, 31-60.*