Deductive Verification of Discrete Control Systems

Lucky

11 June, 2019

M.Tech. Project Report

Abstract

We are exploring deductive approach for formal verification of discrete controllers for k-length properties. Discrete controller's verification is mainly done by model checking, one of the ways of formal verification. Model checking is simple but not easy in the aspect that if states of a system increases, it loses efficiency as its search space explodes. On the other hand, deductive approach is not so used in case of discrete controllers, but the part of learning invariant in deductive verification is already explored in detail. We want to use this advantage of deductive approach in case of discrete controllers in order to perform better than model checking in terms of efficiency. But in the case of learning invariant, the algorithm explores all variables used in model and possibilities of invariant becomes exponential in terms of these variables. We are trying to solve this issue by providing a reduced search space in already existing state of the art learning algorithm.

1 Introduction

A discrete system is a system which has discrete number of states. For example, a computer is a finite state machine that may be viewed as a discrete system. Discrete controllers play an important part in real world discrete systems. Their working has to satisfy certain safety specifications defined for them. For example, in a traffic light system, the controller may take the system to a state in which lights in both directions are green, which may lead to accidents. So, we should definitely verify these systems according to their safety properties, in order to avoid such dangerous scenarios.

There are many approaches for formal verification. Well-known approach for discrete controllers is model checking. In model checking, specifications about the system are expressed as temporal logic formulas and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not.

In our project, we are exploring deductive approach for discrete controllers. It is more of a manual process and that's why not preferred much in comparison to model checking. It contains a part where, we need to find an invariant, which is manual. There has been much work in automating this process via learning techniques [1] [2] , mainly decision trees. We will be using one such technique and also suggest some changes that can be done in order to make this algorithm more precise and efficient in setting of discrete control systems.

Automating the strengthening is not a trivial problem because the search space in order to come up with an invariant is exponential in the terms of number of variables used in the model. One primary goal of the project is also to reduce the search space of the learning algorithm by suggesting possible variables which constructs the invariant.

We have verified 8 programs which Horn-ICE with the original search space was unable to verify and produce invariant for the program. We have added this reduction in our implementation to verify the discrete systems successfully.

In this report, working is explained using an example of **Traffic Light Controller** which spreads over all coming sections. All important different parts of the controller are explained in different sections. In upcoming section, we will talk about discrete controller. After which two approaches of verification are described with respect to the example. Next section after approaches, deductive verification is explained in the detail with help of the example. And after that

the learning algorithm Horn-ICE is discussed which is used in the project. In later section, the proposed improvement and strengthening is explained. Related work is also discussed afterwards. Finally, we are concluding the report and explaining scope of future work.

2 Discrete Controllers

A controller is an important part of a system which controls the functioning of the system. A system can be discrete or continuous. A discrete system is a system with a countable number of states. Discrete systems may be contrasted with continuous systems, which may also be called analog systems. A final discrete system is often modeled with a directed graph and is analyzed for correctness and complexity according to computational theory. Because discrete systems have a countable number of states, they may be described in precise mathematical models like finite state automata. We are calling a controller to be discrete if it is of a discrete system.

Here we are explaining general structure of controller. We are considering systems from Simulink, and generating C code for their controller. Say C is controller, P is plant, S is the safety property that we want to hold at each sample point. Let s_c represent the state of the controller, s_p the state of the plant, i the input variable, and o the output variable (both from the controllers point of view). Let the system state be represented by $s = (s_c, s_p, i, o)$. The structure of controller is as follows for every system.

```
// Initialization
   init(sc, sp);
   o := output(sc,i);
   while(true) // invariant I(s)
   {
     // read inputs from plant
     read();
     // update state:
     sc := update(sc,i);
     // compute outputs:
     o := output(sc,i);
   }
```

Traffic light controller and its state transition flow are shown in Figure 1 and Figure 2 respectively. System is consisting of two directions, north-south and east-west. One assumption is that traffic is more in east-west direction compared to the north-south direction. So, traffic light is green in east-west direction for more time, and we have put a sensor in north-south to sense the traffic. If the traffic in north-south is considerable then the light will be changed from red to green for the traffic to clear, and it will again go back to red.

The system can be in five states: *Stop, StopForTraffic, StopToGo, GoToStop* and *Go*. Each state has some variables. Some important variables are these following :

- `y1` and `y2` : light in north-south and east-west
- `oy1` and `oy2` : old values of light in north-south and east-west
- counter : It keeps track of how much time the system has been in current state

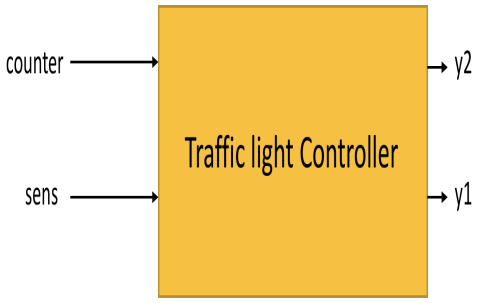

Figure 1: Traffic Light Controller

The generated code of the controller is in C and is too large to include in this report. It can be found in this link. The outline of the code is as follows :

- When *Stop* state is active, Traffic light is *RED* for North-South and *GREEN* for East-West. ie. `y1` is *RED* (North-South) and `y2` is *GREEN* (East-West)
- After 20 clock ticks, *StopForTraffic* become active state. But, `y1` is still *RED* and `y2` is still *GREEN* for next 20 ticks.
- Sensor is detected.
- If sensor is true in the north-south direction, active state becomes *StopToGo* and `y1` is set to *RED* and `y2` to *YELLOW*
- After 10 clock ticks, active state becomes *GoToStop*, thus `y1` = *YELLOW* and `y2` = *RED*.
- After 3 clock ticks, active state becomes **Stop**.

One basic safety property is that the light should be not be *green* in both directions simultaneously. This property is very critical, but we may have many properties. One such property is:

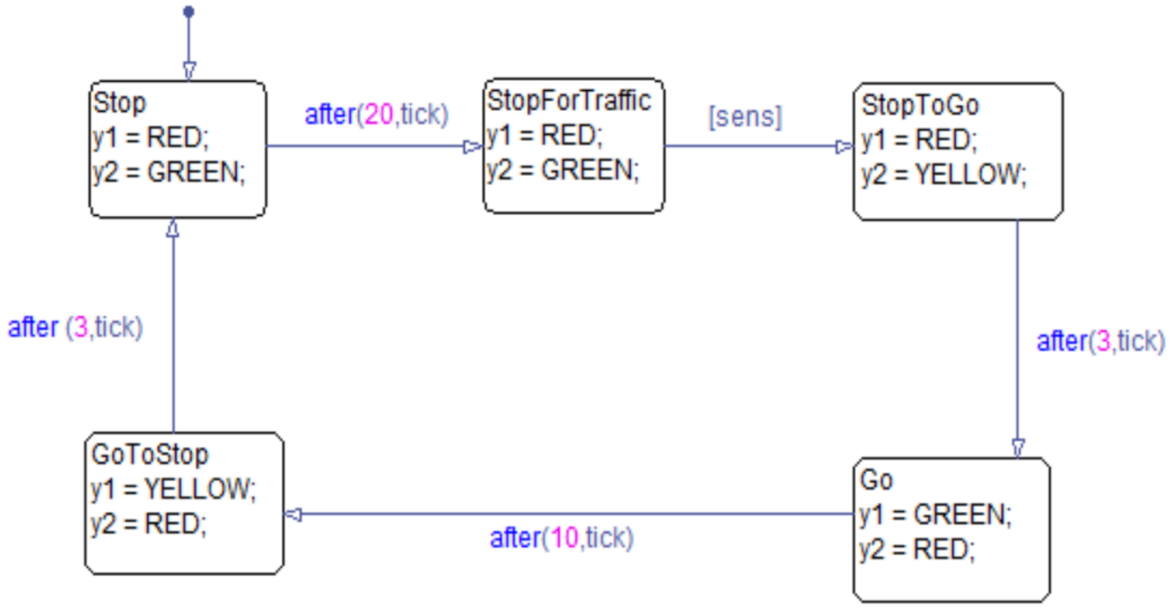

Figure 2: State flow for Light Controller

- If light is *green* in current state then in the next state it will be either *green* or *yellow* but not *red*

3 Model Checking

Model Checking is a well known technique which is used for formally verifying finite-state concurrent systems. Temporal logic formulas are used to express specifications about the system. Model is traversed using efficient symbolic algorithms in order to check the specifications are valid at each state.

In case of traffic light system, if we are following model checking approach, we have to check for every state in every possible scenario, whether a property is being satisfied or not.

The problem with this approach is that once the number of states increase in the model, it becomes inefficient. The state space is exploded exponentially with the number of variables in the model. Also it cannot handle infinite state spaces. Therefore, Model Checking won't be an efficient way of property proving in Simulink models, as most of the real world models may contain large number of states.

4 Deductive Approach

We are considering systems which have two parts, one where it initializes variables V and second where system can make transitions. One simple example is as below:

```
int x;
init()
{
    x:=1;
}

trans()
{
    if(1<=x && x<=10)
    {
        x++;
    }
    else
    {
        x--;
    }
}
```

The property which we want to verify is in the form of a predicate on V. Like $0 <= x$ is a property and

$V = x$ for above example .

One has to follow a two step process in order to verify the above property using deductive approach. Let $init(S)$ denotes the initial state of the system, $P(S)$ denotes that property P is satisfied by state S, $trans(S, S')$ denotes transition of system from a state S to S'. In first step we prove that initial state satisfies property P. In the second we prove that if system is in a state satisfying P and then moves to a new state then the new state will also hold P. These steps can be written as following constraints:

1. $init(S) \Rightarrow P(S)$
2. $P(S) \wedge \text{trans(S,S')} \Rightarrow P(S')$

If we try to prove the above example, we can note that second constraint fails if the starting state have $x = 0$. But if we add this condition: $x > 0$ then the conditions (1) and (2) above are satisfied. This is called strengthening P.

We use Boogie, which is an intermediate verification language, and all verification in our project is done through this. Boogie works on the basis of Hoare logic [3]. It converts a program with annotations like "assume" and "assert" to logical formulas called *verification conditions*, which are then passed to an SMT solver like Z3 to check for validity.

One disadvantage of deductive approach is when the property is not get satisfied and we have to come up with an invariant which implies the desired property and inductive. We aim to apply learning techniques to make it automatic.

5 Model transformation for k-length property

Let S be the initial code for the controller and P be the property to be proved true for each state of system. First S will converted into Boogie code Q. Then Q will be further modified according to property. If the property is considering k states simultaneously, then we need to form a window of k states in Q_k also, which is not present in original C and Boogie code. We are denoting that to be Q_k. Then we need to phrase the property also in Boogie. That program we are calling as Q_{kp}.

For both initialization and induction part, we have to prove the property P. In initialization part, we will initialize one state and do one step transition in order to modify other states in the window, if property is followed by the k-length window, then we say the system satisfies the property in the initial state. In the update state, we are looking the property to be satisfied over two window of states. So, by *requires* we guarantee that the staring window of states follow the property and by *ensures* we can say after one transition the next window is also satisfying the property.

Below is shown the structure of proving, where P is the original property and q is the strengthening in order to help the proving.

```
init(){
....
}
ensures(p and q)
//------------------------------
requires(p and q);
update(){
....
}
ensures(p and q )
```

Initial step is to simulate traffic light system in simulink. The possible states of the **controller** are shown in Figure 2 and **controller** is shown in Figure 1. Second step, is to generate the C code of the controller.

Third step is to convert this C code into Boogie code. We are using Smack tool for this conversion. This initiates to define a new structure for the Boogie code. The code in Boogie will have initialization part and an induction part. Now properties are added to the Boogie program and for both the parts, we need to prove properties as explained above.

In the next section, we explain the machine learning algorithm we used for verification purpose of safety properties for discrete controllers.

6 Horn-ICE learning algorithm

This is the machine learning algorithm used by us for learning invariant and finally verifying programs according to the given specification. It is a learning algorithm for synthesizing invariants using Horn implication counter examples (HornICE), extending the ICE learning model. In particular, it is a decision tree learning algorithm that learns from non-linear Horn-ICE samples and uses statistical heuristics to learn trees that satisfy the samples.

In order to understand Horn-ICE algorithm, let us take a system with variables x. $Init(x)$ denotes the initial state of the system and $trans\,(x, x')$ denotes the transition from state x to x'. And the property which we want to satisfy is that the system does not reach a

set of bad/unsafe states denoted by $Bad(x)$. We can say that this property is satisfied if we come up with an inductive invariant $I(x)$ which also satisfies three constraints:

$$(1)\ \forall x.\, \text{Init}(x) \Rightarrow I(x)$$

It says that in initial state invariant which is basically a boolean combination of variables is satisfied.

$$(2)\ \forall x. \neg(I(x) \wedge Bad(x));\ \text{and}$$

This constraint says that invariant can never be true in a bad state.

$$(3)\ \forall x, x'. I(x) \wedge \text{Trans}\,(x, x') \Rightarrow I\,(x')$$

This constraint helps in proving induction of the invariant. It says that if the system is in a state where invariant is true and it makes a transition to a new state then invariant will also be satisfies in the new state.

When a conjectured invariant fails to satisfy the first two conditions, Boogie can come up with configurations labeled positive and negative to indicate ways to correct the invariant. However, when the third constraint fails, a single configuration labeled positive/negative is not enough. The correct counter example is a pair of configurations c and c' , such that if $I(c)$ is true, then $I\,(c')$ must also satisfy. They call these as implication counter examples.The first ICE (Implication Counter-Example) learning framework developed by Garg et al.[2].

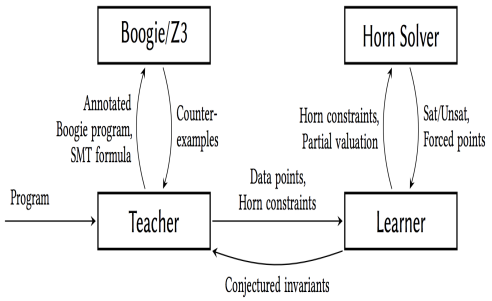

Figure 3: Architecture of the Horn-ICE invariant synthesis tool

In this algorithm, there are two parts, the Learner and the Teacher as shown in Figure 3. Learner conjectures an invariant for the program, and the Teacher, with help of Boogie verifier, checks that the invariant proves the program correct or not. It returns a counter example to the learner if the invariant is not adequate or inductive. The learner proposes a new invariant using the counter example. In this black box technique, algorithm is fully unaware of the program.

Although, Horn-ICE algorithm is a state of the art work but it is also lagging behind in many cases and still we have to improve it further. It still times out for some programs because of a huge search space needed for the invariant. It is way better than other learning algorithms like PDR [4] but it still needs improvement.

6.1 Proposed improvement

Invariant is a boolean combination of predicates. In Horn-ICE, form of predicate is fixed and which is $\pm x \pm y \leq c$, where $x, y \in V$ and V is the set of variables present in model. Even if we consider c to be fixed, then also the number of possible predicates is of order $|V|^2$. Lets call the set of all possible predicates P.

Just for understanding, if we say that $|P| = 5$, then we can visualize the distribution of search space as follows by P.

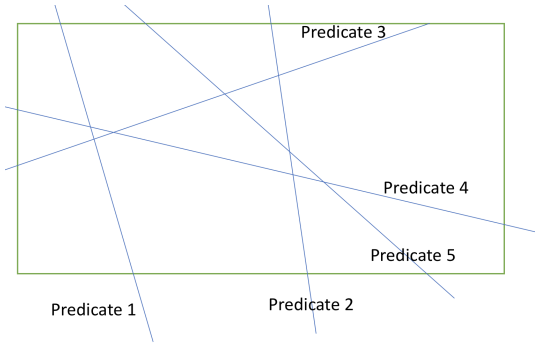

Figure 4: Divison of V by P

And suppose the decision tree we got, is as follows.

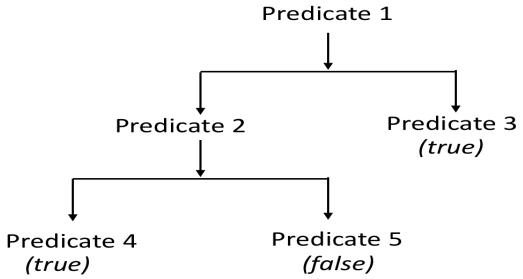

Figure 5: Example decision tree

So, invariant given by the tree is $(predicate1 \wedge predicate1 \wedge predicate4) \vee (predicate1 \wedge predicate3)$.

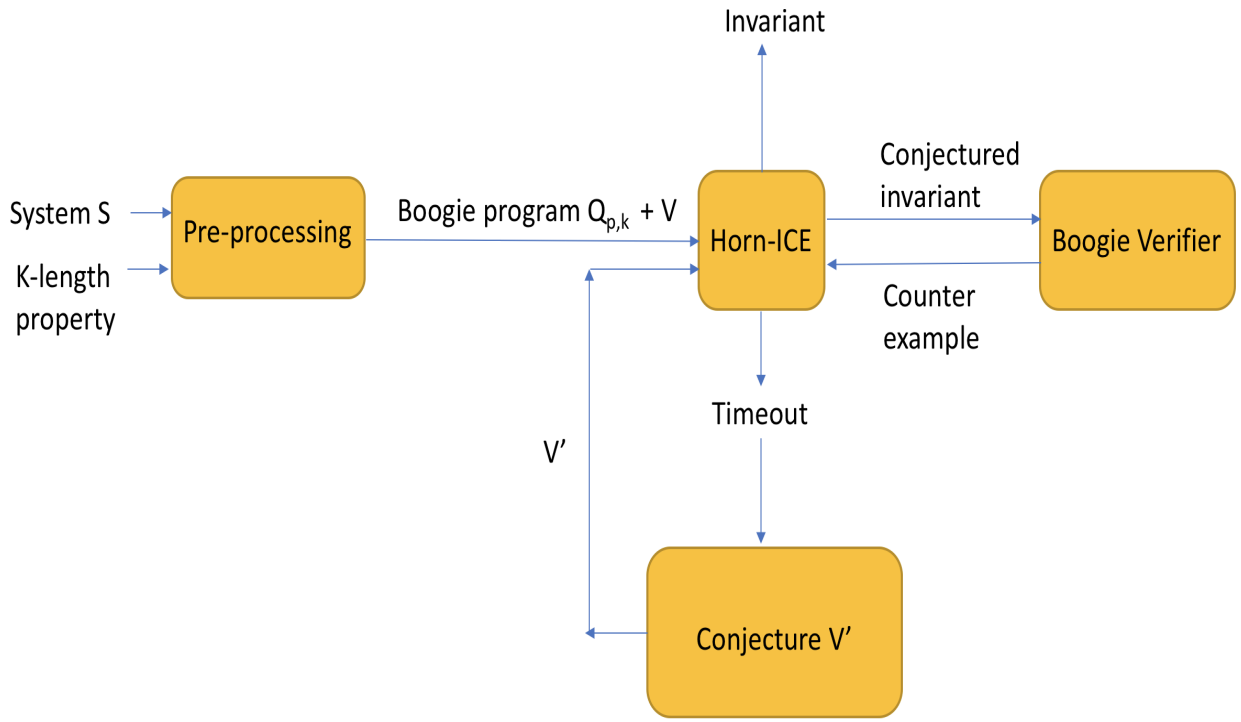

Figure 6: Architecture for proving property using Horn-ICE with the help V'

This invariant is nothing but the dis-junction of true regions and a true region is a conjunction of these predicates. So, we can clearly see that possible invariants are 2^{2^P}, where P is of order $|V|^2$ ie. exponential in terms of V.

We can drastically reduce this space if we use V' in place of V, where $V' \subset V$. In this project we have come up with an appropriate V' in case of discrete control systems. Horn-ICE successfully comes up with an invariant in case of V' whereas it times out with V.

7 Implementation

Horn ICE takes a Boogie program as input. We model any discrete system which needs to be verified for given specifications to a Boogie program with proper phrasing of specifications in the format Horn-ICE needs. This part is dealt in pre-processing section as shown in architecture in Figure 6.

Now this program is given to Horn-ICE, where Horn-ICE tries to come up with an invariant and prove the property with the help of invariant. Horn-ICE conjectures an invariant and sends it to Boogie, Boogie sends back a counter example where the invariant is failing. This process may go in a big loop when the search space for invariant is large.

This is the current state of Horn-ICE. There is no modification for the search space and for a large search space it loops forever. We have come up with a technique to reduce search space after experimenting with various examples.

If Horn-ICE loops for a certain time, we declare it a timeout and then send the Boogie code to strengthening section. In the strengthening section, we reduce the search space and modify the invariant function in the boogie code. We send back this code with reduced search space back to Horn-ICE. If Horn-ICE still loops we give it a different search space until it gets verified. If the code and properties are correct then strengthening section can always help in coming up with a better search space and hence avoiding the timeout.

There is a structure in reducing the search space. As in the case of discrete systems, it is in some state based on `V` (set of variables).

1. First suggestion by strengthening section is to pass **only state and its related output values**. We came up with this technique when we got an error trace by Boogie related to traffic light example. This trace indicated that some-

times values are not according to a state and which cause this issue.

2. Mostly the first reduction works, but if it fails then we can also pass the variables which affects the transition. Sometimes, there might be some error in state-flow, which can be avoided.

The invariant for traffic light system turned out to be, " If `y1` is GREEN than system state is GO". With original set V, Horn ICE looped for a long time and was unable to give any invariant.

8 Experiments and Results

We have worked with 8 examples. Using strengthening we are able to verify all of them. Horn-ICE alone was able to prove only 3. And with those 3 programs also, we verified them in lesser time compared to Horn-ICE.

In Figure 7, we show a comparison in time taken in both cases as strengthening some times become complex and heavy, but the time is nearly same in all cases.

In the below table, we compare the time taken for Horn-ICE to verify with original set of variables V versus a reduced set V' over various discrete systems.

Comparison over programs		
Programs	Horn ICE (V)	Horn ICE (V')
arbiter.bpl	2.03	1.13
traffic.bpl	TO	1.41
cruise.bpl	TO	1.66
tank.bpl	TO	0.92
boiler.bpl	TO	1.87
thermal.bpl	TO	1.43
discrete4.bpl	1.25	0.98
discrete5.bpl	1.13	0.79

Numbers in the table show the time taken in proving properties in seconds. TO refers to time out(more than 10 minutes). programs "discrete4" and "discrete5" are variants of traffic controller program with 4 and 5 states. Arbiter program is taken from a tutorial [5]. And all other programs are from simulink.

9 Related Work

Automatic deductive verification with invisible invariants [6] is one of the early works in automating deductive verification. This paper presents a method for the automatic verification of a certain class of parameterized systems. These are bounded-data systems consisting of N processes (N being the parameter), where each process is finite-state.

They have shown that if standard deductive inv rule is used for proving invariance properties, then all the generated verification conditions can be automatically resolved by finite-state (bdd-based) methods with no need for interactive theorem proving. Next, they also show how to use model-checking techniques over finite (and small) instances of the parameterized system in order to derive candidates for invariant assertions. Combining this automatic computation of invariants with the previously mentioned resolution of the VCs (verification conditions) yields a (necessarily) incomplete but fully automatic sound method for verifying bounded data parameterized systems.

The generated invariants can be transferred to the VC-validation phase without ever been examined by the user, which explains why one can refer to them as "invisible". They illustrate the method on a nontrivial example of a cache protocol, provided by Steve German.

In deductive verification, the correctness of program is expressed as a set of mathematical statements. The concept of Hoare's [3] assertions stated as relevant proofs for the correctness of the programs. The use of new languages and tools has helped in the verification process. The paper [7] reviews the challenges faced by deductive software verification and the tools used. The specification language includes the mathematical language specification for properties and its integration with a programming language. Thus the specification can be set as verification conditions and once this is done they are integrated in a programming language with *require* and *ensures* statements for its functions. Thus mixing the programs and logical statements. The paper introduces a methodology for the proof of specification language. The most common methodology used is Hoare's logic where the precondition is strengthened and the postcondition is weakened. Meaningful assertions are added at key places in the program and rest of assertions are figured out by the system.
These intermediate assertion can be automatically derived by the intermediate verification languages like Boogie, Why etc. They use proof assistant tools such as Coq, Isabelle to discharge the logical statements that can even do deep embedding. Once these are done the next task is to prove these verification conditions. This paper uses SMT solvers to prove these verification conditions automatically. They also say that

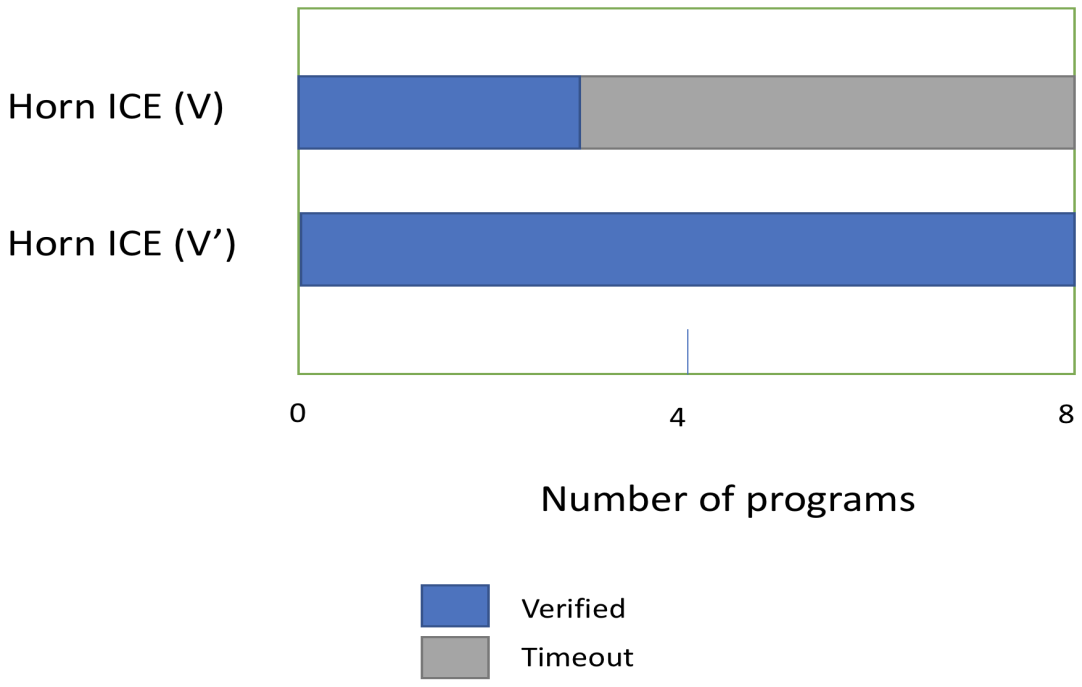

Figure 7: Comparison between Horn-ICE with V versus Horn-ICE with V'

if it is not able to prove the verification conditions automatically then adding intermediate assertions is an option that will reduce the cost of proof. Other technique used here is dedicated provers that motivate the need for close connection between verification conditions and input code and specifications.

Learning invariants using decision trees and implication counterexamples in also done very recently. Our work is based on this paper. This paper [2] explains that inductive invariants can be robustly synthesized using a learning model where the teacher is a program verifier like Boogie which instructs the learner through concrete program configurations. Configurations are classified as positive, negative, and implications. Author proposes the first learning algorithm in this model with implication counter-examples that are based on decision tree learning algorithm.
In particular, They extend classical decision-tree learning algorithms in machine learning to handle implication samples, building new scalable ways to construct small decision trees using statistical measures. They also develop a decision-tree learning algorithm in this model that is guaranteed to converge to the right concept (invariant) if one exists. implementation of the learners and an appropriate teacher is done, and it is shown that resulting invariant synthesis is efficient and convergent for a large suite of programs. This is done for continuous programs and not discrete programs, our focus will be on discrete programs.

Checking safety properties using induction and a SAT-solver [8] is also been explored in recent years. Author is looking at problem of how to check safety properties of finite state machines. Here, interest is particularly in checking safety properties with the help of a SAT-solver. They describe some novel induction-based methods, and show how they are related to more standard fix-point algorithms for invariance checking. They also present preliminary experimental results in the verification of FPGA (Field Programmable Gate Array) cores. This demonstrates the practicality of combining a SAT-solver with induction for safety property checking of hardware in a real design flow. This paper was referred in a hope that it can be used in deductive approach but it is clearly close to model checking approach.

10 Conclusion and Future Work

Deductive verification is preferred approach for formal verification, specially in systems having large number

of states. Finding inductive and adequate invariant is an important part of the proof technique. Correctness of programs was proved to satisfy by the model using Boogie verifier. The structure introduced for the Boogie code is proper and efficient for this approach.
We have tested strengthening with a bunch of examples. And strengthening surely helped us to show that using it with Horn-ICE learning algorithm is a better combination. We have successfully verified all the programs we considered. In future one can explore it for more general system rather than just discrete systems.

References

[1] Deepak D'Souza, P. Ezudheen, Pranav Garg, P. Madhusudan, and Daniel Neider. Horn-ice learning for synthesizing invariants and contracts. *CoRR*, abs/1712.09418, 2017.

[2] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. 2016.

[3] Deepak D'Souza and K. V. Raghavan. Hoare logic. 2017.

[4] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 157–171, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[5] Getting started with smv.

[6] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 82–97, London, UK, UK, 2001. Springer-Verlag.

[7] Amir Pnueli and Elad Shahar. A platform for combining deductive with algorithmic verification. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 184–195, London, UK, UK, 1996. Springer-Verlag.

[8] Mary Sheeran, Satnam Singh, and Gunnar Stalmarck. Checking safety properties using induction and a sat-solver. 2016.