International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:4, 2009

Validation of Automation Systems using Temporal Logic Model Checking and Groebner Bases

Quoc-Nam Tran[†], Anjib Mulepati
Lamar University, U.S.A.

***Abstract*—**Validation of an automation system is an important issue. The goal is to check if the system under investigation, modeled by a Petri net, never enters the undesired states. Usually, tools dedicated to Petri nets such as DESIGN/CPN are used to make reachability analysis. The biggest problem with this approach is that it is impossible to generate the full occurence graph of the system because it is too large. In this paper, we show how computational methods such as temporal logic model checking and Groebner bases can be used to verify the correctness of the design of an automation system. We report our experimental results with two automation systems: the Automated Guided Vehicle (AGV) system and the traffic light system. Validation of these two systems ranged from 10 to 30 seconds on a PC depending on the optimizing parameters.

***Keywords*—**Computational Intelligence, Temporal Logic Reasoning, Model Checking, Groebner Bases.

I. INTRODUCTION

VERIFYING the correctness of a design for an automation system is an important issue. The goal is to check if the system never enters the undesired states, i.e. the situations that should never happen. Usually, the system under investigation is modeled by means of a Petri net and tools dedicated to Petri nets such as DESIGN/CPN are used to make reachability analysis [27, 35]. The biggest problem with this approach is that it is impossible to generate the full occurence graph of the system because it is too large.

In this paper, we show how computational methods such as temporal logic model checking and Groebner bases can be used to verify the correctness of the design of an automation system.

Temporal logic model checking is one of the formal verification techniques which can be used to verify the correctness of a finite state concurrent system. Model checking [1, 2, 9, 31] with temporal logic [36] and automata-theoretic techniques [28, 29, 30, 45, 46, 47, 48, 49] have the ability to discover subtle flaws resulting from improbable events. Model checking has several attractive features. Once the Kripke model of the system and the temporal logic specifications have been defined, the process is fully automated. If a computation that violates the specification is found in the Kripke model, it can be displayed to the designer to aid the debugging process. In addition, the model checker can formally verify if the specification holds [28, 32, 33]. Popularity of model checking has been increased due to being a fast and automatic method. Unlike other techniques, model checking can produce a counter example when there is fault in the system design. This automatic generation of counter examples can be used to pinpoint the bug in the design. In the paper we also show how the behavior of a system can be expressed in terms of algebraic equations and use Groebner bases to validate the system design.

We report our results with validating the correctness of two different systems: the Automated Guided Vehicle (AGV) system and the traffic light system using SMV model checker and reachability analysis on the Petri net using Groebner basis.

The paper is organized as follows. Section 2 present the formal logic, the model checking and Groebner basis verification methodology. Section 3 and 4 describe the AGV and traffic light verification problem respectively.

II. VERIFICATION METHODOLOGY

The methodology that we use to verify the correctness of automation systems are known as temporal logic model checking and Groebner bases. Temporal logic [36] is a logic for expressing the ordering of events in time without introducing time explicitly. Model checking is an automatic verification approach used for concurrent systems [1, 2, 9, 31]. Model checking provides a counter-example if it finds an error during verification. By studying such counter-examples, we can pinpoint the error, correct it, and re-verify.

A. Temporal Logics

The first step in formal verification is the representation of formal specification of the design consisting of a description of the desired behavior. As sequential systems capture time-variant behavior, it is not possible to describe their properties completely in the framework of conventional propositional formulas. In a temporal logic, temporal operators are additionally provided, which can be used to express time-variant dependencies. A widely used specification language for designs is Temporal Logic [36], which is a modal logic (i.e. logic to reason about many possible worlds with their semantics based on Kripke structures).

In linear temporal logic (LTL), time is treated as if each moment has a unique possible future. (An alternative approach is to use branching time (CTL) [10].) Linear temporal formulas are constructed from a set $\wp$ of atomic propositions using

† Corresponding Author

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:4, 2009

usual Boolean connectives as well as temporal connectives **X** ("next"); **G** ("always"); **F** ("eventually"); and **U** ("until").

The semantics of LTL formulas are based on an appropriate Kripke structure of the form $K = (\mathbb{N}, \leq, \pi)$, where $\mathbb{N}$ is the set of natural numbers, $\leq \subseteq \mathbb{N}^2$ is the standard linear order, and $\pi : \mathbb{N} \rightarrow 2^{\wp}$ is a function that defines what propositions are true at a certain time instant.

An LTL formula is interpreted over *computations* viewed as infinite sequences of truth assignments to the atomic propositions in Σ^{ω} where $\Sigma = 2^{\wp}$ as follows:

1) $\pi, i \vDash \varphi$ means that formula φ holds at the time i of the computation π,
2) $\pi, i \vDash p$ iff $p \in \pi(i)$,
3) $\pi, i \vDash \mathbf{X}\varphi$ iff $\pi, i+1 \vDash \varphi$,
4) $\pi, i \vDash \mathbf{G}\varphi$ iff $\pi, j \vDash \varphi \ \forall j \geq i$,
5) $\pi, i \vDash \mathbf{F}\varphi$ iff $\pi, j \vDash \varphi$ for some $j \geq i$,
6) $\pi, i \vDash \varphi \mathbf{U} \psi$ iff $\exists j \geq i$ such that $\pi, j \vDash \psi$ and $\pi, k \vDash \varphi$ $\forall j > k \geq i$.

Notice that operators **G** and **F** can be derived from **X** and **U** as $\mathbf{F}\varphi \equiv true\,\mathbf{U}\varphi (true \equiv p \vee \neg p)$ and $\mathbf{G}\varphi \equiv \neg \mathbf{F} \neg \varphi$.

In LTL model checking, we assume that the specification is given in terms of properties expressed by LTL formulas. For example the formula $\mathbf{G}(request \rightarrow \mathbf{F}\ grant)$, which refers to the atomic propositions *request* and *grant*, specifies that every state in the computation in which *request* holds is followed by some state in the future in which *grant* holds.

B. Validity in Model Checking

Using temporal logic as a specification language for systems quite naturally leads to the idea of model checking. In fact, model checking has become one of the most actively studied automated formal verification techniques [9, 10, 37]. In model checking, an LTL specification is checked against a Kripke model of the finite state system (i.e. the implementation). Validity and satisfiability are defined as follows:

- π satisfies a formula φ, denoted by $\pi \vDash \varphi$, iff $\pi, 0 \vDash \varphi$,
- $\text{models}(\varphi) = \{\pi \in (2^{\wp})^{\omega} : \pi \vDash \varphi\}$,
- φ is satisfiable iff $\text{models}(\varphi) \neq \emptyset$ (non-emptiness),
- φ is valid iff $\text{models}(\varphi) = (2^{\wp})^{\omega}$ (universality).

C. Model Checking

In model checking, the specification is expressed in temporal logic and the system is modeled as a finite state machine. For realistic designs, the number of states of the system can be very large, 10^{20} or higher, and hence explicit traversal of the state space becomes infeasible. Perhaps the biggest hurdle for the practical use of model checking is the state explosion problem [44]. To relieve the state explosion problem, many approaches have been proposed.

Symbolic model checking using ordered binary decision diagrams (BDD) is a successful and widely used techniques for verifying properties of concurrent hardware and software systems. In symbolic model checking, the state space is represented implicitly using symbolic means, and the propositional logic formulas are manipulated using BDDs [23]. Symbolic model checking succeeded in checking systems with unprecedented large state spaces

Bounded model checking uses the same basic idea as symbolic model checking using BDD in that the state space of the system is represented implicitly by Boolean formulas. However, instead of manipulating the Boolean formulas using BDDs, the bounded model checker transforms the model and property specifications into a propositional satisfiability (SAT) problem. Given a system M, a temporal logic formula ψ and a bound k, a Boolean formula is constructed, which is satisfiable if and only if M has a counterexample of length k to ψ. A SAT solver such as SATO or zChaff is used to perform the query. In BMC, the growth of the size of the Boolean formulas can be known in advanced, but predicting the running times of the SAT solver is difficult. BMC is particularly good at finding shallow counterexamples, and some industrial applications have been successful of BMC [3, 4, 13].

Other approaches to relieve the state explosion problem include the partial order methods [17, 26, 43, 50]; complete finite prefixes [21, 31]; compositional methods [12, 19, 25]; symmetry reduction [16, 20, 22, 24, 38]; and abstraction [11, 14, 15, 18].

Petri nets (PN) provide a graphical and mathematical representation of many systems [34]. Formally we can define a *Petri net* as a bipartite directed graph represented by a quadruple $PN = (P, T, I, O)$, where

- $P = \{p_1, p_2, ..., p_n\}$ is a finite set of places,
- $T = \{t_1, t_2, ..., t_m\}$ is a finite set of transitions,
- $I : P \times T \rightarrow \{0, 1\}$ is an input function that defines the set of directed arcs from P to T,
- $O : T \times P \rightarrow \{0, 1\}$ is an output function that defines the set of directed arcs from T to P,
- $P \cup T \neq \phi$, $P \cap T = \phi$

Furthermore, a marking M of the Petri net assigns to each place a non-negative integer $M(p_i)$, the number of tokens on that place. In the Petri net every marking M of Petri net can be associated with a polynomial $pol(M) = \prod_P p^{M(p)}$ and each transition t of Petri net can be associated with a polynomial: $pol(t) = \prod_P p^{w(p,t)} - \prod_P q^{w(t,q)}$.

Consider a set of polynomial $P \subseteq K[X^{\triangle}]$. We say that two polynomials f and g of $K[X^{\triangle}]$ are equivalent modulo P and write $f =_P g$ if their difference can be expressed in terms of P, i.e. $f - g = k_1 p_1 + ... + k_n p_n$ for some $p_1, ..., p_n \in P, k_1, ..., k_n \in K[X^{\triangle}]$. Deciding $f =_P g$ can be made easy with Groebner Basis [5, 6, 7, 39, 40, 41, 42].

A polynomial g reduces to another polynomial h modulo some polynomial set F, written as $g \rightarrow_F h$, if and only if there exists $f \in F$, $c = lc(g)/lc(f)$ and $u = lm(g)/lm(f)$ such that $h = g - c \cdot u \cdot f$ [8]. The new polynomial h is equivalent to g with respect to the ideal generated by F. If a polynomial g can

be reduced to h by finitely many reduction steps w.r.t. F we write $g \to_F^* h$. An h is said to be in *reduced form* if h cannot be reduced further w.r.t. F, denoted by $\underline{h}_F$.

A given a finite set of polynomials G is a *Groebner basis* if and only if $\forall g,h,k$ if h and k are normal form of g modulo G then $h = k$, i.e.$\forall g,h,k$ $(\underline{h}_F \leftarrow_{F^*} g \to_{F^*} \underline{k}_F) \Rightarrow (h = k)$.

The reachability test will answer whether we can reach some state in the *Petri net*. Formally, the *reachability problem* for a Petri net is given two markings, M_1 and M_2 of the same *Petri net M*, to answer if M_2 is reachable from M_1. A *Petri net* is called *reversible* if a marking M_2 is reachable from another marking M_1 which implies that M_1 is reachable from M_2. We can apply the *Groebner basis* procedure to reversible *Petri net*s to solve the reachability problem [8].

Given a set of transitions, $F = \{pol(t) : t \in T\}$ where $F \subseteq K[X^{\triangle}]$, we can compute G, a Groebner Basis for F. The new generated polynomials after the *Groebner basis* computation still corresponds to a *Petri net* because the generated polynomials are of the same form as the polynomials arising from the translation . Now, marking M_1 is reachable from marking M_0 if and only if $pol(M_1) \to_G pol(M_0)$ **.**

III. Automatic Guided Vehicle System

A. Problem Formulation

The objective of the problem is to check whether the paths followed by the five *Automated Guided Vehicles (AGVs)* modeled by the Petri Net as in Figure 1 [35] are collision free or not. The model consist of five different paths followed by vehicle A, B, D, E and F. Additionally there are three different workstations represented by W1,W2 and W3. Shaded rectangular depict the collision area for two vehicles i.e. two vehicles can't be in this area simultaneously.

Further we have used model checking technique to verify whether the adding of new places as mentioned in [35] makes the design collision free or not.

To solve the problem using model checking, we use transitions as SMV processes. For each transition, one process was instantiated. Moreover, processes are run asynchronously, which means among all the modules instantiated, one is non-deterministically chosen, and the assignment statements declared in the process are executed in parallel. This is done by instantiating process with using *process* keyword. This approach seems elegant since only one process can run at a time and is determined non-deterministically.

- Places are represented by the variables and ASSIGN declaration specifies their initial values (Figure 2).

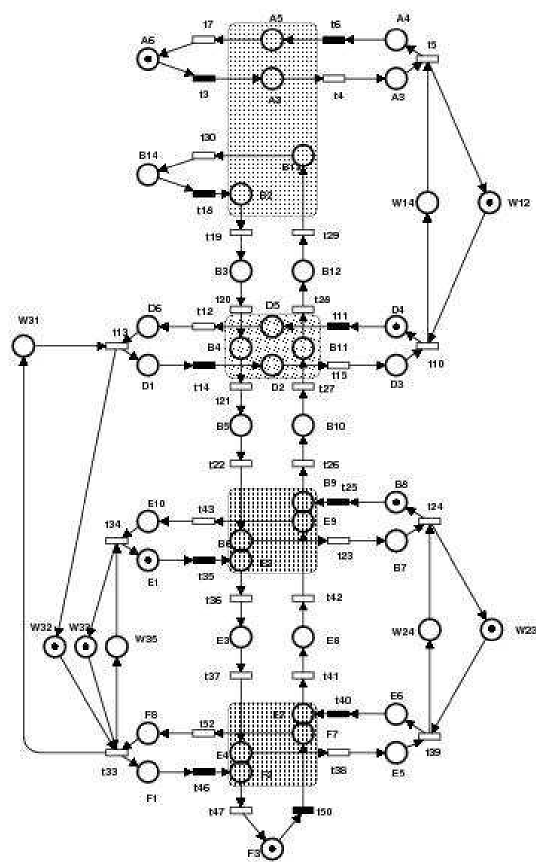

Figure 1. Reduced Version of AGV [35]

```
---------------------- MAIN ----------------------
MODULE main
 VAR
  -- Place variable
  A02:0..10;
  A03:0..10;
  A04:0..10;
  A05:0..10;
  A06:0..10;

  .....

  -- Transition process
  p1: process prc_A(A02,A03,A04,A05,A06,W12,W14);
  p2: process prc_B(B02,B03,B04,B05,B06,B07,B08,B09,B10,B11,B12,B13,B14,W23,W24);
  p3: process prc_D(D01,D02,D03,D04,D05,D06,W12,W14,W31,W32);
  p4: process prc_E(E01,E02,E03,E04,E05,E06,E07,E08,E09,E10,W23,W24,W33,W35);
  p5: process prc_F(F01,F02,F03,F07,F08,W31,W32,W33,W35);

  .....

 ASSIGN
  init(A02):=0;
  init(A03):=0;
  init(A04):=0;
  init(A05):=0;
  init(A06):=1;

  .....
```

Figure 2. AVG Main Module

- Processes are nested, meaning five process modules for five AGVs, and in each AVG's process there are processes for transitions. These transition processes are actually the instantiation of the process, differentiated by the number of parameters used which actually represent the number of input and output places (Figure 3). For example, *t_2(in1,in2,out1,ou2)* represents a transition with two

New Place	Input Transitions	Output Transition
Z1	t4, t7, t19, t30	t3, t6, t18, t25
Z2	t12, t15, t21, t28	t11, t14, t18, t25
Z3	t23, t26, t36, t43	t18, t25, t35, t40
Z4	t38, t41, t47, t52	t35, t40, t46, t50

Table I
FOUR NEW PLACES ON AVG MODEL TO AVOID COLLISION[35]

input places *in1* and *in2*, and two output places *out1* and *out2*.

```
-- Module for transition with with 1 input and 1 output
MODULE t_1(in1,out1)
 ASSIGN
   next(in1):= case
    in1>0: in1 - 1;
    1: in1;
   esac;

   next(out1):= case
    in1>0: (out1 + 1) mod 11;
    1: out1;
   esac;

   .....

--Module for area A
MODULE prc_A(A02,A03,A04,A05,A06,W12,W14)
 VAR
  t3: process t_1(A06,A02);
  t4: process t_1(A02,A03);
  t5: process t_2(A03,W14,A04,W12);
  t6: process t_1(A04,A05);
  t7: process t_1(A05,A06);

 FAIRNESS running

 .....
```

Figure 3. Module for Transitions of AVG

- Each AGV process has *FAIRNESS running* to make sure that every AVG process is executed infinitely often , so the system must make progress when possible.
- Finally, we have specification to check existence of the collision state. We check for the occurrence of one or more tokens in the places within collision area (Figure 4).

```
SPEC
 !EF(((A02>0 | A05>0) & (B02>0 | B13>0))|
     ((B04>0 | B11>0) & (D02>0 | D05>0))|
     ((B06>0 | B09>0) & (E02>0 | E09>0))|
     ((E04>0 | E07>0) & (F02>0 | F07>0)))
```

Figure 4. Specification for AVG Model

Verification done with SMV shows that there is a sequence of steps which leads to collision. This model can be made collision free by adding new places as shown in Table I and Figure 5 shows adding one of the new place Z1. New SMV program is written adding these new places (Figure 6) and this time verification shows that its a collision free design.

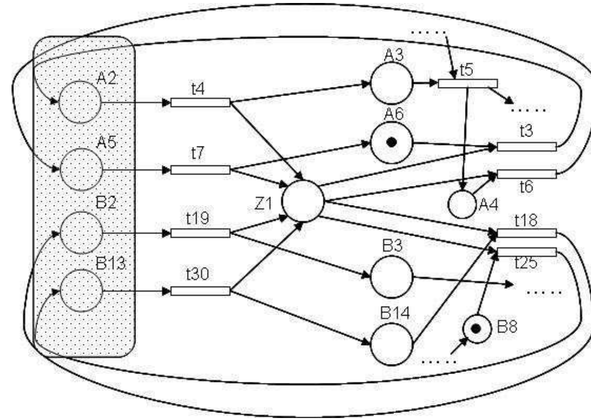

Figure 5. AGV with place Z1

```
--------------------- MAIN ---------------------
MODULE main
 VAR
  -- Place variable
  A2:0..10;
  A5:0..10;
  A3:0..10;
  A4:0..10;
  A6:0..10;

  .........

  Z1:0..10;
  Z2:0..10;
  Z3:0..10;
  Z4:0..10;

  prc1: process prc_A(A2,A3,A4,A5,A6,W14,W12,Z1);
  prc2: process prc_B(B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,W24,W23,Z1,Z2,
                    Z3,Z4);
  prc3: process prc_D(D1,D2,D3,D4,D5,D6,W12,W14,W31,W32,Z2);
  prc4: process prc_E(E1,E2,E3,E4,E5,E6,E7,E8,E9,E10,W35,W33,W23,W24,Z3,Z4);
  prc5: process prc_F(F1,F2,F3,F7,F8,W32,W33,W31,W35,Z4);

ASSIGN
  init(Z1):=0;
  init(Z2):=0;
  init(Z3):=0;
  init(Z4):=0;
```

Figure 6. SMV Code for AGV with New Places

This result can be improved if we can find some optimal variable order. Since this things can be done in NuSMV, different variable orderings for this program is tested. Various combination order and their running time is shown in Table II.

Table II
RUNNING TIME FOR VARIOUS VARIABLE ORDER FOR AVG PROGRAM

Order 1	Order 2	Order 3	Order 4	Order 5	Order 6
W12	A02	W12	W32	W12	W12
W14	A06	W14	W12	W14	W14
A02	A03	A02	W14	A02	A02
A03	A04	A03	A02	A03	A03
A04	A05	A04	A03	A04	A04
A05	B06	A05	A04	A05	A05
A06	B07	A06	A05	A06	A06
...	...	...	...	...	...
W31	W35	W33	W33	W33	W33
19.14 sec	21.64 sec	18.95 sec	30.22 sec	14.35 sec	13.73 sec

In Table II six different variable orders and their corresponding

running times are given. Among these Order 6 has least running time. An important notice in this result is that running time is greatly increased when the two variables W32 and W33 are not placed consecutively.

B. Verification with Groebner basis

1) AVG without places Z:

a) Scenario I: : Marking with polynomial $pol(M_1) = a3b7d6e5f1w12w23w31w32$

- Initial state $pol(M_0) = a6b8d4e1f3w12w23w32w33$
- Graded reverse lexicographic order: w33, w35, w32, w31, w23, w24, w12, w14, e10, f8, e5, b7, d6, d3, a3, f1, e1, e6, b8, d1, d4, a4, b13, b12, b11, b10, b14, e9, b9, b2, f7, f3, f2, e8, e7, e4, e3, e2, b6, b5, b4, b3, d2, d5, a5, a2, a6.
- Groebner Basis G = [a2 - a6, a5 - a6, b4 - b3, b5 - b3, b6 - b3, e3 - e2, e4 - e2, e8 - e7, f3 - f2, f7 - f2, b2 - b3, b9 - b3, e9 - e7, b14 - b3, b10 - b3, b11 - b3, b12 - b3, b13 - b3, a4 - a6, d4 - d5, d1 - d2, b8 - b3, e6 - e7, e1 - e2, f1 - f2, a3 - a6, -d2 + d3, d6 - d5, b7 - b3, e5 - e2, f8 - f2, e10 - e7, w12 a6 - w14 a6, w12 d2 - w14 d5, w32 d2 - w31 d5, w23 b3 - w24 b3, w24 e7 - w23 e2, w35 e7 - w33 e2, w14 d2 a6 - w14 d5 a6, w31 w12 d5 - w32 w14 d5, w35 w23 e2 - w33 w24 e2, w33 w32 f2 - w35 w31 f2, w32 w14 d5 a6 - w31 w14 d5 a6, w35 w31 f2 d2 - w33 w31 f2 d5, w33 w24 e2 b3 - w35 w24 e2 b3, w33 w31 f2 e2 d2 - w33 w31 f2 e7 d5, w33 w31 w14 f2 d5 a6 - w35 w31 w14 f2 d5 a6, w35 w32 w24 f2 e2 b3 - w35 w31 w24 f2 e2 b3].
- Normal form of $M_1 - M_0$ with respect to Groebner Basis G is $w31^2w14d5a6w24e2b3f2 - w35w31f2w24b3w14a6e2d5$ which is not equal to zero. Thus this state is not reachable.

b) Scenario II: : Marking with polynomial $pol(M_2) = a3b7d6e5f8w12w23w33w35$

Normal form of $M_2 - M_0$ with respect to Groebner Basis G is $-w35w31f2w24b3w14a6e2d5 + w35^2w24e2b3w14a6f2d5$ which is not equal to zero. Thus this state is also not reachable.

Interestingly these are the only two states which are not reachable in this original AGV Petri net.

C. AGV with places Z

c) Scenario I: : Marking with polynomial $pol(M_1) = a2b9d4e1f3w12w23w32w33z2z3z4$

- Initial state $pol(M_0) = a6b8d4e1f3w12w23w32w33z1z2z3z4$
- Graded reverse lexicographic order: w33, w35, w32, w31, w23, w24, w12, w14, b9, b2, e10, f8, f1, e7, e2, e5, b7, d1, d6, d4, d3, a4, a3, e1, f7, f2, e6, b8, d2, d5, a5, a2, b13, b11, f3, a6, b14, e9, e4, b6, b4, b12, b10, e8, e3, b5, b3, z2, z1, z4, z3.
- Groebner Basis G = [b4 - b3, b6 - b5, e4 - e3, e9 - e8, b11 - b10, b13 - b12, a5 - a2, f7 - f2, b9 - b2, -e2 + e3 z3, b10 z3 - b2, b7 z3 - b5, e10 z3 - e8, -e7 + e8 z4, -f2 + f3 z4, e5 z4 - e3, f1 z4 - f2, f8 z4 - f2, b3 z1 - b2, b14 z1 - b12, a6 z1 - a2, -a2 + a3 z1, a4 z1 - a2, b5 z2 - b3, b12 z2 - b10, -d2 + d3 z2, d4 z2 - d5, -d5 + d6 z2, d1 z2 - d2, d6 b3 - d4 b3, d1 b3 - d3 b3, b10 b5 - b12 b3, d5 b5 - d4 b3, d2 b5 - d3 b3, f1 e3 - f3 e3, f8 e3 - f3 e3, b7 e8 - e10 b5, a3 b10 - a6 b10, a4 b10 - a6 b10, ...].
- Normal form of $M_1 - M_0$ with respect to Groebner Basis G is -w35 w31 f3 e3 b3 w14 a2 d3 w24 + w31 w14 d5 a2 w35 w24 b2 f2 e8 which is not equal to zero.
- Thus this state is not reachable. This is true because B vehicle can move to collision area if and only if there is token at z1, z2 and z3 and also remove token from these places if it moves to collision area i.e. z1, z2 and z3 control the vehicle B.

d) Scenario II: : Marking with polynomial $pol(M_2)$: $a6b8d4e4f7w12w23w32w33z1z2z3z4$

This marking is also not reachable since normal form of $M_2 - M_0$ with respect to Groebner Basis G and with same graded reverse lexicographic order as above is -w35 w31 f3 e3 b3 w14 a2 d3 w24 + w35 w31f2b3w14a2d3w24e3 which is not equal to zero. This is true because token of route E can't be in collision area if both places Z3 and Z4 do not have the token.

IV. Traffic Light System

A. Problem Formulation

The objective of this problem is to verify that traffic light system at a busy intersection works correctly. The complete structure of this intersection is shown in Figure 7. On each direction we have five lanes, right two lanes are for incoming traffics and left three for outgoing traffic from that direction. Among three lanes on the left hand side, rightmost lane is to turn left, middle one to go straight and leftmost can be used either to go straight or turn right. Incoming vehicles are sensed by the sensor and controlled by the traffic lights. Vehicles are allowed to continue at the junction if it is safe to do so i.e. there is no possibility of collision with the vehicle from opposite direction. Further we also have four buttons for pedestrians on each of the four directions. When a pedestrian presses the button, lights must be set in a way such that it is safe for the pedestrian to cross the road.

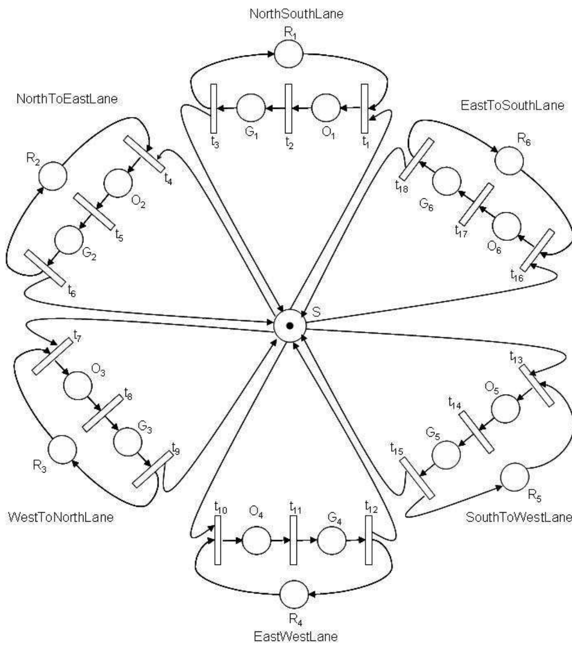

Figure 8. Traffic Light Petri Net

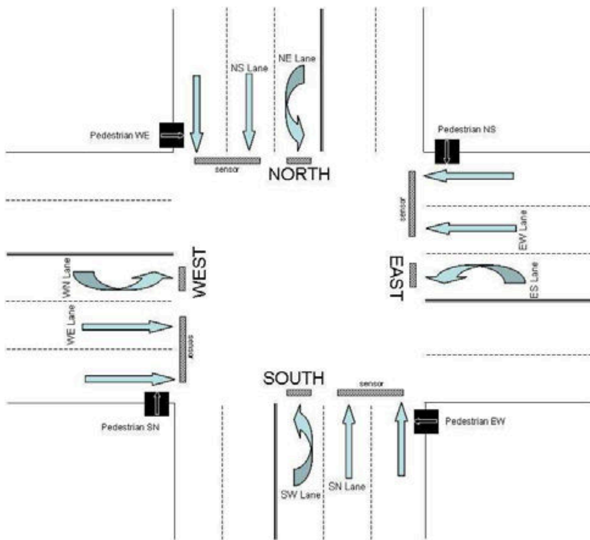

Figure 7. Traffic System

To solve this problem the whole program is divided into three modules, for straight lane (Figure 9), for left turn lane and for pedestrian. The following approach is taken to simulate lane activities (Figure 9):

- For each lane four states are defined: *idle* (no vehicle), *entering* (vehicle sensed), *critical* (request is pending) and *exiting* (successfully crossed).
- Initially lane is in *idle* state
- If vehicle arrive sensor will be activate , set the request and change state to *entering*.
- After being in *entering* state try to get lock on the lane if
  – none of the opposite lanes have set it.
  – none of opposite lanes have requested and
  – none of pedestrian on the way have requested.
- After gaining lock check if all other opposite lane has not requested (same condition as for gaining lock). If so enter the *critical* state and set the go status.
- Once done, free the lock if pair lanes are also done and set off the go status.

```
-- Module for the straight lanes.
MODULE s_lane(myLane-Lock,oppLane-Lock, my-Req, oppPair-Req, my-Go, my-Sense, pair1-Go,
          pair2-Go, ped1-Req, ped2-Req)
 VAR
   -- Possible state of the lane
  state: {idle, entering, critical, exiting};

 ASSIGN
  --Initially no vehicle so state is idel
  init(state):=idle;

  -- Set next state
  next(state):=
   case
     state = idle & my-Req = 1 : entering;
     state = entering & !oppLane-Lock & !oppPair-Req & !ped1-Req & !ped2-Req: critical;
     state = critical & !my-Sense : exiting;
     state = exiting : idle;
     1 : state;
   esac;

 --Set lock to avoid collision
  next(myLane-Lock):=
   case
    state = entering & !oppLane-Lock & !oppPair-Req & !ped1-Req & !ped2-Req: 1;
    state = exiting & !pair1-Go & !pair2-Go : 0;
    1 : myLane-Lock;
   esac;

  -- Set request on if vehicle sensed
  next(my-Req) :=
   case
    !my-Req & my-Sense : 1;
    state = exiting : 0;
    1 : my-Req;
   esac;

  -- Set green light
  next(my-Go) :=
   case
    state = entering & !oppLane-Lock & !oppPair-Req & !ped1-Req & !ped2-Req: 1;
    state = exiting : 0;
    1 : my-Go;
   esac;

  -- Randomly sense vehicle.
  next(my-Sense) := {0,1};

  FAIRNESS
   running & !(my-Go & my-Sense)
```

Figure 9. SMV Code for Straight Lane

```
-- Module for left turn lanes
MODULE t_lane(myLane-Lock,oppLane-Lock, my-Req, oppPair1-Req, oppPair2-Req, my-Go, my-Sense, pair-Go,
          ped1-Req, ped2-Req)
 VAR
  -- Possible state of the lane
  state: {idle, entering, critical, exiting};

 ASSIGN
  -- Initially no vehicle  so state is idle
  init(state):=idle;

  -- Set value of next state
  next(state):=
   case
     state = idle & my-Req = 1 : entering;
     state = entering & !oppLane-Lock & !(oppPair1-Req | oppPair2-Req) & !ped1-Req & !ped2-Req: critical;
     state = critical & !my-Sense : exiting;
     state = exiting : idle;
     1 : state;
   esac;

  -- Try to set lock
  next(myLane-Lock):=
   case
    state = entering & !oppLane-Lock & !(oppPair1-Req | oppPair2-Req) & !ped1-Req & !ped2-Req: 1;
    state = exiting & !pair-Go : 0;
    1 : myLane-Lock;
   esac;

  -- If vehicle sensed set request
  next(my-Req) :=
   case
    !my-Req & my-Sense : 1;
    state = exiting : 0;
    1 : my-Req;
   esac;

  -- Set the green light
  next(my-Go) :=
   case
    state = entering & !oppLane-Lock & !(oppPair1-Req | oppPair2-Req) & !ped1-Req & !ped2-Req: 1;
    state = exiting : 0;
    1 : my-Go;
   esac;

  -- Nondeterministically set vehicle presence
  next(my-Sense) := {0,1};

  FAIRNESS
   running & !(my-Go & my-Sense)
```

Figure 10. SMV Code for Left Turn Lane

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:4, 2009

Similarly, pedestrian module as shown in Figure 11 is written to simulate pedestrian activity. Following is an outline of steps done to do so (Figure 11):

- Three states of pedestrian are defined as follows: *idle* (no request), *press* (requested) and *exiting* (successfully granted).
- The initially state is *idle*.
- State changes to *press* as soon as a sensor senses the request.
- Crossing is allowed if all incoming lanes (there will be four lanes to check each time) had not requested.

```
-- Module for Pedestrian
MODULE ped(my-Sen, my-Req, my-Go, lt1-Req, lt2-Req, lt3-Req, lt4-Req)
 VAR
  -- State for Pedestrain
  state : {idle, press, exiting};

 ASSIGN
  -- Initially no request so idle
  init(state) := idle;

  -- Nondeterministically set pedestrian request
  next(my-Sen) := {0,1};

  -- Set next state
  next(state) :=
        case
                state = idle & my-Req = 1 : press;
                state = press & !my-Sen : exiting;
                state = exiting : idle;
                1: state;
        esac;

  -- If button press set request
  next(my-Req) :=
        case
                !my-Req & my-Sen : 1;
                state = exiting : 0;
                1 : my-Req;
        esac;
  -- Set green light for crossing
  next(my-Go) :=
        case
                state = press & (!lt1-Req & !lt2-Req & !lt3-Req & !lt4-Req) : 1;
                state = exiting : 0;
                1 : my-Go;
        esac;
```

Figure 11. SMV Code for Pedestrian

Now let us explain the specification give for the verification. Specification is shown in Figure 12. Two sets of specification are given, one to check for mutual exclusion i.e. when any one lane or one side pedestrian is in go state no other opposite lane be in go state too. This ensures that no collisions or hit will ever occur. Other set of specifications ensure that no process can block other process from getting the request.

```
--------------------------------------------------------------------------------
-----                           SPECIFICATION                             -----
--------------------------------------------------------------------------------

--------------------------- For Mutual Exclusion -------------------------------
  SPEC AG!(NE-Go & (WN-Go | WE-Go | SW-Go | SN-Go | ES-Go | EW-Go))
  SPEC AG!(NS-Go & (WN-Go | WE-Go | SW-Go | ES-Go | EW-Go))
  SPEC AG!(WN-Go & (SW-Go | SN-Go | ES-Go | EW-Go | NE-Go | NS-Go))
  SPEC AG!(WE-Go & (SW-Go | SN-Go | ES-Go | NE-Go | NS-Go))
  SPEC AG!(SW-Go & (ES-Go | EW-Go | NE-Go | NS-Go | WN-Go | WE-Go))
  SPEC AG!(SN-Go & (ES-Go | EW-Go | NE-Go | WN-Go | WE-Go))
  SPEC AG!(ES-Go & (NE-Go | NS-Go | WN-Go | WE-Go | SW-Go | SN-Go))
  SPEC AG!(EW-Go & (NE-Go | NS-Go | WN-Go | SW-Go | SN-Go))
  SPEC AG !(PedNS-Go & (NE-Go | WE-Go | ES-Go | EW-Go))
  SPEC AG !(PedEW-Go & (NS-Go | SN-Go | SW-Go | ES-Go))
  SPEC AG !(PedSN-Go & (WN-Go | WE-Go | SW-Go | EW-Go))
  SPEC AG !(PedWE-Go & (NE-Go | NS-Go | WN-Go | SN-Go))
--------------------------------------------------------------------------------

---------------------------- For Non Blocking ----------------------------------
  SPEC AG EF(NE-Req)
  SPEC AG EF(NS-Req)
  SPEC AG EF(SW-Req)
  SPEC AG EF(SN-Req)
  SPEC AG EF(WN-Req)
  SPEC AG EF(WE-Req)
  SPEC AG EF(ES-Req)
  SPEC AG EF(EW-Req)
  SPEC AG EF(PedNS-Req)
  SPEC AG EF(PedSN-Req)
  SPEC AG EF(PedWE-Req)
  SPEC AG EF(PedEW-Req)
--------------------------------------------------------------------------------
```

Figure 12. Specification for Traffic Light Verification

1) Verification with Model Checking: Following observation (Figure 13) is made while verifying the model. All the specification comes to be true so no counter example is given which means that the desire properties are fulfilled by the model.

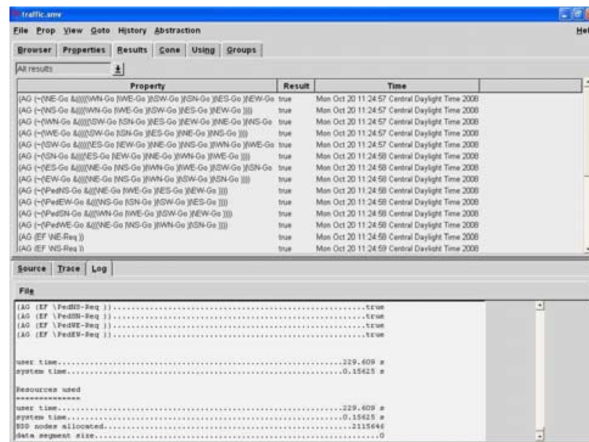

Figure 13. Traffic Light Problem Verification Result

2) Verification with Groebner Basis: Now, further verification can be done with the help of Petri Net and Groebner basis. Petri Net for this traffic problem can be defined in terms of traffic lights. Here six traffic lights will be enough to simulate, four lights for each left turn and one light for each horizontal and vertical lane. Pedestrian lights can share same transition as of horizontal and vertical lanes. For example North to South pedestrian light will be identical to light for North-South lane.

Scenario I: Lights for both North-East turn and South-West turn is green $pol(M_1) = R1G2R3R4G5R6$

- Initial state $pol(M_0) = SR1R2R3R4R5R6$
- Graded reverse lexicographic order: R1, R2, R3, R4, R5, R6, G4, O1, G2, O2, O4, G3, O3, G6, O6, G5, G1, O5, S.

- Groebner Basis G = [G5 - O5, G6 - O6, G3 - O3, G2 - O2, O1 - G1, G4 - O4, S R6 - O6, S R5 - O5, -O4 + S R4, -O3 + S R3, -O2 + S R2, S R1 - G1, R5 G1 - R1 O5, R5 O6 - R6 O5, R1 O6 - R6 G1, R6 O3 - R3 O6, R5 O3 - R3 O5, R1 O3 - R3 G1, R6 O4 - R4 O6, R5 O4 - R4 O5, R3 O4 - R4 O3, R1 O4 - R4 G1, R6 O2 - R2 O6, R5 O2 - R2 O5, R4 O2 - R2 O4, R3 O2 - R2 O3, R1 O2 - R2 G1]
- Normal form of $M_1 - M_0$ with respect to Groebner Basis G is R6 G1 R2 O5 R3 R4 - R6 O5 R1 R2 R3 R4 which is not equal to zero. Thus this state is not reachable which must be true to ensure safety at the crossroad.

Scenario II: Lights for both North-South and East-West lane are green $pol(M_2) = G1R2R3G4R5R6$

The normal form of $M_2 - M_0$ with respect to Groebner Basis G and with same graded reverse lexicographic order as above is R6 G1 R2 O5 R3 R4 - R6 O5 R1 R2 R3 R4 which is not equal to zero. Thus this state is also unreachable.

V. Conclusion

We have shown that computational methods like model checking and Groebner bases can be used to verify the correctness of an automation system by specifying the desired properties in the CTL form and verifying the corresponding temporal formula. The benefit of using SMV for model checking is its feature of giving counter example in the case of fault in the system. This feature not only helps to catch the bug but also helps to rectify the fault in the design. We also use algorithms from the field of computer algebra particularly theory of Groebner basis is used on Petri Net to test the reachability problem.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of TACAS*, volume 1579 of *LNCS*. Springer, 1999.

[3] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifiying safety properties of a power pc microprocessor using symbolic model checking without bdds. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification, CAV'99*, volume 1633 of *LNCS*, pages 60–71, Trento, Italy, 1999.

[4] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference on Computer Aided Verification, CAV 2001*, number 2102 in LNCS, pages 454–464, Paris, France, 2001. Springer.

[5] B. Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-dimensional Polynomial Ideal (in German)*. PhD thesis, Institute of Mathematics, Univ. Innsbruck, Innsbruck, Austria, 1965.

[6] B. Buchberger. An algorithmic criterion for the solvability of algebraic systems of equations (german). *Aequationes Mathematicae*, 4:374–383, 1970.

[7] B. Buchberger. Groebner Bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, chapter 6, pages 184–232. Reidel Publishing Company, Dodrecht, 1985.

[8] O. Caprotti, A. Ferscha, and H. Hong. eachability test in petri nets by groebner bases. Technical report, RISC-Linz, 1995.

[9] E. Clarke and E. Emerson. Design and synthesis of syncronization of skeletons using branching time temporal logic. In *Proceedings of the IBM workshop on logics of programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[11] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[12] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*. IEEE Press, 1989.

[13] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of the 13th International Conference on Computer Aided Verification, CAV 2001*, number 2102 in LNCS, pages 454–464, Paris, France, 2001. Springer.

[14] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.

[15] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transaction on Programming Languages and Systems*, 19(2):253–291, 1997.

[16] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 1996.

[17] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 1993.

[18] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[19] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[20] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design*, 15(3):217–238, 1999.

[21] K. Heljanko. *Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets*. PhD thesis, Department of Computer Science and Engineering, Helsinki University of Technology, February 2002.

[22] C. N. Ip and D. L. Dill. Better verification through

symmetry. *Formal Methods in System Design*, 9(1/2):41–76, 1996.

[23] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[24] T. Junttila. *On the symmetry reduction method for Petri nets and similar formalisms*. PhD thesis, Laboratory for Theoretical Computer Science, Helsinki University of Technology, 2003.

[25] R. Kaivola. *Equivalences, preorders and compositional verification for linear time temporal logic and concurrent systems*. PhD thesis, Department of Computer Science, University of Helsinki, 1996.

[26] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992.

[27] B. H. Krogh and L. E. Holloway. Synthesis of feedback control logic for discrete manufacturing systems. *Automatica*, 27(4):641–651, 1991.

[28] O. Kupferman and M. Vardi. From complementation to certification. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 591–606. Springer, 2004.

[29] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[30] R. P. Kurshan. *Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[31] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[32] K. Namjoshi. Certifying model checker. In *Proceedings of the 13th International Conference on Computer-Aided Verification*, volume 2102 of *LNCS*, pages 2–13. Springer, 2001.

[33] D. Peled and L. Zuck. From model checking to a temporal proof. In *Proceedings of the 8th International SPIN Workshop*, volume 2057 of *LNCS*, pages 1–14, Toronto, Canada, 2001. Springer.

[34] J. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[35] L. Petrucci. Design and validation of a controller. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI)*, volume 8, 2000.

[36] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[37] J. Quille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th International Symposium on Programming*, page 1981, 337–350.

[38] K. Schmidt. How to calculate symmetries of petri nets. *Acta Informatica*, 36(7):545–590, 2000.

[39] Q.-N. Tran. A fast algorithm for Groebner basis conversion and its applications. *Journal of Symbolic Computation*, 30:451–468, 2000.

[40] Q.-N. Tran. Efficient algorithms for implicitization using groebner walk. *Journal of Computer Aided Geometric Design*, 21:837–857, 2004.

[41] Q.-N. Tran. A new class of term orders for elimination and applications. *Journal of Symbolic Computation*, 42, 2007.

[42] Q.-N. Tran. A p-space algorithm for groebner bases computation in boolean rings. In *Proceedings of World Academy of Science, Engineering and Technology*, volume 35, pages 495–501, Paris, France, 2008.

[43] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[44] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.

[45] M. Vardi. Alternating automata and program verification. In *Computer Science Today - Recent Trends and Developments*, volume 1000 of *LNCS*, pages 471–485. Springer-Verlag, 1995.

[46] M. Vardi. Automated verification = graphs, logic, and automata. In *Proceedings of IJCAI*, 2003.

[47] M. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, 1986.

[48] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

[49] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[50] K. Varpaaniemi. On stubborn sets in the verification of linear time temporal properties. *Formal Methods in System Design*, 26(1):45–67, 2005.