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

Quoc-Nam Tran*1, 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 DESIGNS/CPN are used to make reachability analysis. The biggest problem with this approach is that it is impossible to generate the full occurrence 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 not happen. Usually, the system under investigation is modeled by means of a Petri net and tools dedicated to Petri nets such as DESIGNS/CPN are used to make reachability analysis [27, 35]. The biggest problem with this approach is that it is impossible to generate the full occurrence 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 \( \varphi \) of atomic propositions using

† Corresponding Author
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 $\pi : \mathbb{N} \rightarrow 2^\mathbb{N}$ 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 $\Sigma^\omega$ where $\Sigma = 2^\mathbb{N}$ as follows:

1) $\pi,i \models \varphi$ means that formula $\varphi$ holds at the time $i$ of the computation $\pi$.
2) $\pi,i \models p$ iff $p \in \pi(i)$.
3) $\pi,i \models X \varphi$ iff $\pi,i+1 \models \varphi$.
4) $\pi,i \models G \varphi$ iff $\pi,j \models \varphi \forall j \geq i$.
5) $\pi,i \models F \varphi$ iff $\pi,j \models \varphi$ for some $j \geq i$.
6) $\pi,i \models \varphi U \psi$ if $\exists j \geq i$ such that $\pi,j \models \psi$ and $\pi,k \models \varphi \forall j < k \leq i$.

Notice that operators G and F can be derived from X and U as $F \varphi \equiv true \land \psi \land (\varphi \land U \psi)$ and $G \varphi \equiv \neg F \neg \varphi$.

In LTL model checking, we assume that the specification is given in terms of properties expressed by LTL formulas. For example the formula $G(request \rightarrow F grant)$, 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:

- $\pi$ satisfies a formula $\varphi$, denoted by $\pi \models \varphi$, iff $\pi,0 \models \varphi$.
- $\models_{\text{f}}(\varphi) = \{ \pi \in (2^\mathbb{N})^\omega : \pi \models \varphi \}$.
- $\varphi$ is satisfiable iff $\models_{\text{f}}(\varphi) \neq \emptyset$ (non-emptiness).
- $\varphi$ is valid iff $\models_{\text{f}}(\varphi) = (2^\mathbb{N})^\omega$ (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, $10^{30}$ 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 $\psi$ and a bound $k$, a Boolean formula is constructed, which is satisfiable if and only if $M$ has a counterexample of length $k$ to $\psi$. 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

- $P = \{p_1,p_2,\ldots,p_n\}$ is a finite set of places,
- $T = \{t_1,t_2,\ldots,t_m\}$ 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 $pol(M) = \Pi_{p \models M} p^{M(p)}$ and each transition $t$ of Petri net can be associated with a polynomial: $pol(t) = \Pi_{p \models M} p^{M(p)} - \Pi_{p \models M} q^{M(q)}$.

Consider a set of polynomial $P \subseteq K[X^\Delta]$. We say that two polynomials $f$ and $g$ of $K[X^\Delta]$ are equivalent modulo $P$ and write $f \equiv_P g$ if their difference can be expressed in terms of $P$, i.e. $f-g = k_1 p_1 + \ldots + k_n p_n$ for some $p_1,\ldots,p_n \in P,k_1,\ldots,k_n \in K[X^\Delta]$. Deciding $f \equiv_P g$ 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 $g \rightarrow_P h$, if and only if there exists $f \in F$, $c = le(g)/le(f)$ 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 \( g \rightarrow_F^* h \). An \( h \) is said to be in reduced form if \( h \) cannot be reduced further w.r.t. \( F \), denoted by \( h_F \).

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. \( \forall g, h, k ( h_F \leftarrow F \cdot g \rightarrow F \cdot k_F \Rightarrow h = k ) \).

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 nets to solve the reachability problem [8].

Given a set of transitions, \( F = \{ pol(t) : t \in T \} \) where \( F \subseteq K[X^\Delta] \), 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 \( M_2 \) is reachable from marking \( M_0 \) if and only if \( pol(M_1) \rightarrow_G pol(M_0) \).

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 \( \text{process} \) 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).
input places \texttt{in1} and \texttt{in2}, and two output places \texttt{out1} and \texttt{out2}.

-- Module for transition with 1 input and 1 output
\texttt{MODULE c_1(in1, out1)}
\begin{verbatim}
ASSIGN
new(in1):= case

\end{verbatim}
\begin{verbatim}
next(out1):= case
\end{verbatim}
\begin{verbatim}
\end{verbatim}
\begin{verbatim}
FAIRNESS running
\end{verbatim}
\begin{verbatim}
.....
\end{verbatim}
\begin{verbatim}
--Module for area A
\end{verbatim}
\begin{verbatim}
MODULE prc_A(A02, A03, A05, A06, W12, W14)
\end{verbatim}
\begin{verbatim}
VAR
\end{verbatim}
\begin{verbatim}
\end{verbatim}
\begin{verbatim}
FAIRNESS running
\end{verbatim}
\begin{verbatim}
.....
\end{verbatim}

Figure 3. Module for Transitions of AVG

- Each AGV process has \textbf{FAIRNESS running} to make sure that every AGV 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).

\textbf{SPEC}

\begin{verbatim}
!EF((((A02>0 | A05>0) & (B02>0 | B13>0)) | ((B04>0 | B14>0) & (D02>0 | D05>0)) | ((B06>0 | B05>0) & (E02>0 | E09>0)) | (E04>0 | E07>0) & (F02>0 | F07>0))
\end{verbatim}

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 \texttt{Z1}. New SMV program is written adding these new places (Figure 6) and this time verification shows that its a collision free design.

In Table II six different variable orders and their corresponding running time is shown.

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>
<thead>
<tr>
<th>Order 1</th>
<th>Order 2</th>
<th>Order 3</th>
<th>Order 4</th>
<th>Order 5</th>
<th>Order 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>W12</td>
<td>A02</td>
<td>W12</td>
<td>W12</td>
<td>W12</td>
<td>W12</td>
</tr>
<tr>
<td>W14</td>
<td>A06</td>
<td>W14</td>
<td>W12</td>
<td>W12</td>
<td>W14</td>
</tr>
<tr>
<td>A02</td>
<td>A03</td>
<td>A02</td>
<td>W14</td>
<td>A02</td>
<td>A02</td>
</tr>
<tr>
<td>A03</td>
<td>A04</td>
<td>A03</td>
<td>A02</td>
<td>A03</td>
<td>A03</td>
</tr>
<tr>
<td>A05</td>
<td>A04</td>
<td>A03</td>
<td>A04</td>
<td>A04</td>
<td>A04</td>
</tr>
<tr>
<td>A06</td>
<td>B07</td>
<td>A16</td>
<td>A05</td>
<td>A06</td>
<td>A06</td>
</tr>
<tr>
<td>W31</td>
<td>W35</td>
<td>W33</td>
<td>W33</td>
<td>W33</td>
<td>W33</td>
</tr>
<tr>
<td>19.14 sec</td>
<td>21.84 sec</td>
<td>18.95 sec</td>
<td>30.22 sec</td>
<td>14.35 sec</td>
<td>13.73 sec</td>
</tr>
</tbody>
</table>

Table II: Running Time for Various Variable Order for AVG Program
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 $pol(M_1) = a_3b_7d_6e_5f_1w_12w_23w_31w_32$
   
   - Initial state $pol(M_0) = a_6b_8d_4e_1f_3w_12w_23w_32w_33$
   - Graded reverse lexicographic order: $w_33, w_35, w_32, w_31$.
   - Groebner Basis $G = \{a_2 - a_6, a_5 - a_6, b_4 - b_3, b_6 - b_5, e_1 - e_7, f_3 - f_2\}$.
   - Normal form of $\text{pol}(M_0) = a_6b_8d_4e_1f_3w_12w_23w_32w_33$

   b) **Scenario II:** : Marking with polynomial $pol(M_2) = a_3b_7d_6e_5f_8w_12w_23w_33w_35$
   
   - Normal form of $M_0$ with respect to Groebner Basis $G$ is $-w_35w_31w_2w_24w_34w_35w_3w_31f_2w_24w_34e_6f_2d_5$.
   - 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 $-w_35w_31f_2w_24w_34e_6f_2d_5$ 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.

2) **Scenario I:** : Marking with polynomial $pol(M_1) = a_2b_9d_4e_1f_3w_12w_23w_32w_33c_2c_3c_4$
   
   - Initial state $pol(M_0) = a_6b_8d_4e_1f_3w_12w_23w_32w_33c_2c_3c_4$
   - Graded reverse lexicographic order: $w_33, w_35, w_32, w_31, w_23, w_24, w_12, w_14, b_9, b_2, e_10, f_8, f_1, e_7, e_2, e_5, b_7, d_1, d_6, d_4, d_3, a_4, a_3, e_1, f_7, f_2, e_6, b_8, d_2, d_5, a_5, a_2, b_13, b_11, f_3, a_6, b_14, e_9, e_4, b_6, b_4, b_12, b_10, e_8, e_3, b_5, b_3, z_2, z_1, z_4, z_3.$

   - Groebner Basis $G = \{b_4 - b_3, b_6 - b_5, e_4 - e_3, e_9 - e_8, b_11 - b_10, b_13 - b_12, a_5 - a_2, f_7 - f_2, b_9 - b_2, e_2 + e_3, b_10 - b_3, b_7 - b_5, e_10 - e_8, e_7 + e_8, z_4\}$.
   - B. Verification with Groebner basis
     
     a) **Scenario I:** : Marking with polynomial $pol(M_1) = a_3b_7d_6e_5f_1w_12w_23w_31w_32$
     
     - Initial state $pol(M_0) = a_6b_8d_4e_1f_3w_12w_23w_32w_33$
     - Graded reverse lexicographic order: $w_33, w_35, w_32, w_31$.
     - Groebner Basis $G = \{a_2 - a_6, a_5 - a_6, b_4 - b_3, b_6 - b_5, e_1 - e_7, f_3 - f_2\}$.
     - Normal form of $M_0$ with respect to Groebner Basis $G$ is $-w_35w_31f_2w_24w_34w_35w_31f_2w_24w_34e_6f_2d_5$ which is not equal to zero. Thus this state is reachable. This is true because B vehicle can move to collision area if and only if there is token at z_1, z_2 and z_3 and also remove token from these places if it moves to collision area i.e. z_1, z_2 and z_3 control the vehicle B.

   d) **Scenario II:** : Marking with polynomial $pol(M_2) = a_6b_8d_4e_4f_7w_12w_23w_32w_33c_2c_3c_4$
   
   - 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 $-w_35w_31f_2w_24w_34e_6f_2d_5$ 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.
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.

```plaintext
-- Modules for the straight lanes,
MODULE l_lane(lock: lock, opp: lock, my: lock, opp: lock, my: lock, opp: lock, my: lock, opp: lock)
VAR
  -- Possible state of the lane
  state : (idle, entering, critical, exiting);
  lock : lock;

-- Initially no vehicle so state is idle
INIT
  state := idle;

-- Set next state
NEXT
  case
  when idle
    -- set sensor
    state := entering;
    lock := lock;
    request := request + 1;
    if request == 2
      exit
    end
  when entering
    -- try to get lock
    lock := lock;
    if lock
      state := critical;
    end
  when critical
    -- Set green if both lanes
    -- w/o request
    if lock
      state := exiting;
      if lock
        lock := lock;
        exit
      end
    end
  when exiting
    -- If lock set
    lock := lock;
    if lock
      exit
    end
  end

-- Set request if vehicle sensed
request(x: req) :
  case
  when idle
    -- set sensor
    lock := lock;
    state := entering;
    request := request + 1;
    if request == 2
      exit
    end
  when entering
    -- try to get lock
    lock := lock;
    if lock
      state := critical;
    end
  when critical
    -- Set green if both lanes
    -- w/o request
    if lock
      state := exiting;
      if lock
        lock := lock;
        exit
      end
    end
  when exiting
    -- If lock set
    lock := lock;
    if lock
      exit
    end
  end
```

![Figure 9. SMV Code for Straight Lane](image)

---

`World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:3, No:4, 2009`

---

International Scholarly and Scientific Research & Innovation 3(4) 2009 scholar.waset.org/1999.4/3046
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.

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.

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.

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(M1) = R1G2R3R4G5R6

- Initial state pol(M0) = 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 = \{ G_5 - O_5, G_6 - O_6, G_3 - O_3, G_2 - O_2, O_1 - G_1, G_4 - O_4, S R_6 - O_6, S R_5 - O_5, -O_4 + S R_4, -O_3 + S R_3, -O_2 + S R_2, S R_1 - G_1, R_5 G_1 - R_1 O_5, R_5 O_6 - R_6 O_5, R_1 O_6 - R_6 G_1, R_6 O_3 - R_3 O_6, R_5 O_3 - R_3 O_5, R_1 O_3 - R_3 G_1, R_6 O_4 - R_4 O_6, R_5 O_4 - R_4 O_5, R_3 O_4 - R_4 O_3, R_1 O_4 - R_4 G_1, R_6 O_2 - R_2 O_6, R_5 O_2 - R_2 O_5, R_4 O_2 - R_2 O_4, R_3 O_2 - R_2 O_3, R_1 O_2 - R_2 G_1 \}

• Normal form of \( M_1 - M_0 \) with respect to Groebner Basis \( G \) is \( R_6 G_1 R_2 O_5 R_3 O_4 - R_6 O_5 R_1 R_2 R_3 R_4 \) 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 must be true to ensure safety at the crossroad.

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


[39] Q.-N. Tran. A fast algorithm for Groebner basis conv-


