Tikrit Journal of Engineering Sciences (2025); 32(Sp1): 2683
DOI: http://doi.org/10.25130/tjes.sp1.2025.48

IRAQI Academic Scientific Journals

ISSN: 1813-162X (Print); 2312-7589 (Online)
Tikrit Journal of Engineering Sciences
available online at: http://www.tj-es.com

TJES
Tikrit Journal of Engineering Sciences

Toward Proving the Extension of Johnson's Rule to a Three-Machine Flow Shop Scheduling Problem

Rafat Alshorman, Hashem Alrossan*, Saja Smadi
Computer Science Department, Faculty of Information Technology and Computer Sciences, Yarmouk University, Irbid, 21163, Jordan.

Keywords:
Linear-Time Logic; NuSMV; Model checking; Computational Tree Logic; Flow shop scheduling.

Highlights:

- *Formal verification framework employing CTL, LTL, and NuSMV model checker to validate the three-machine extension of Johnson's scheduling rule.*
- *Finite-state machine formulation encompassing random sequencing, dominance checks, machine reduction, and makespan evaluation for comprehensive logical analysis.*
- *NuSMV-generated counterexample demonstrating heuristic suboptimality (Johnson's makespan=26 > optimal=25), advancing formally verifiable scheduling methods.*

ARTICLE INFO
Article history:

Received	20 Jul. 2025
Received in revised form	14 Aug. 2025
Accepted	29 Sep. 2025
Final Proofreading	13 Dec. 2025
Available online	28 Dec. 2025

© THIS IS AN OPEN ACCESS ARTICLE UNDER THE CC BY LICENSE.
http://creativecommons.org/licenses/by/4.0/

Citation: Alshorman R, Alrossan H, Smadi S. **Toward Proving the Extension of Johnson's Rule to a Three-Machine Flow Shop Scheduling Problem**. *Tikrit Journal of Engineering Sciences* 2025; **32**(Sp1): 2683.
http://doi.org/10.25130/tjes.sp1.2025.48

Corresponding author:
Hashem Alrossan
Computer Science Department, Faculty of Information Technology and Computer Sciences, Yarmouk University, Irbid, 21163, Jordan.

Abstract: Manufacturing companies can achieve their goals of reducing production costs and improving productivity by using an efficient production schedule. In this paper, a formal verification procedure, supported by relevant arguments, has been used to verify the accuracy of flow-shop scheduling behaviour and to reduce total production time. We focus on the algorithm of sequence generation (Johnson's algorithm), as well as model a three-machine scheduling procedure as a finite-state machine (FSM). The correctness requirements are expressed in Computational Tree Logic (CTL) and Linear-Time Logic (LTL) and are proved by the NuSMV model checker. The associated technique enables automated scheduling of property proofs during the design phase. In our verification, a counterexample demonstrates that Johnson's rule can yield a suboptimal makespan in a three-machine setting, thereby revealing a limitation of its generalisation. The current findings indicate that using a model checker in CTL/LTL can be an effective method for establishing the logical validity of flow-shop scheduling and failure scenarios, thereby informing future heuristic development.

1.INTRODUCTION

Flow shop scheduling is a manufacturing scheduling problem with numerous applications across various fields, including economics and industry. A flow shop consists of n jobs and m machines, where each job has a fixed production sequence across the machines. Each machine is dedicated to a single operation. Because processing times differ across jobs, an appropriate job sequence is required to minimise the total completion time (makespan). Thus, this research aims to achieve the preferred sequence. Johnson's algorithm was introduced in 1954. It addresses only the two-machine flow-shop case and minimises the total completion time (make span); however, this two-machine restriction is considered a drawback [1]. In this regard, researchers have been working to solve this obstacle. The heuristics were developed to make this algorithm more practical and to operate based on the make span. In [2], Campbell, Dudek, and Smith (CDS) presented an extension of Johnson's rule that handles more than two machines by decomposing the m-machine problem into two-machine problems. This was resolved using Johnson's rule for each case. This case study examines the application of CDS in a real-world manufacturing environment to minimise make span. It uses the heuristic as a black-box optimiser, however, without verifying the logical correctness of the produced sequences. On the other hand, our work aims to formally establish that the output of a rule-based heuristic (Johnson's Rule extension) is logically suspicious, thereby providing a distinct yet complementary perspective. This company relies on client orders, in-stock product availability, and random manufacturing. As shown in the results, the CDS algorithm saved 90.9 minutes relative to the actual makespan [3]. Past studies have suggested methods like NEHLPD (NEH with the most extended processing duration), NEHLPD1 (NEH with the most extended processing duration - Variant 1), and NEHLPD2 (NEH with the most extended processing duration - Variant 2), which are based off of NEH (Nawaz, Encore, and Ham) and CDS to solve the flow shop scheduling problem. The present study does not apply such techniques, although it seeks to formally verify the extension of the Johnson Rule under dominance conditions. These approaches were used for 13 cases, and the NEHLPD method achieved the lowest makespan among the NEH and CDS methods [4]. In [5], we propose a scheduling approach that improves daily production using the Total Work (TWK) and the CDS algorithm. As the results showed, the combination of these algorithms could obtain the optimal scheduling system. This paper identifies the problems LTL systems face when planning in automated environments. It builds strategies based on the system's potential behaviour and capabilities [6]. In [7], CTL, LTL, and NuSMV models are used to develop anti-theft car protection systems. This system originated from a keypad and remote control. This paper [8] used the IICTL algorithm to verify CTL properties using three operators: SAT, IC3, and FAIR that correspond to specific sub-queries of an EX node, an EU node, and an EG node, respectively. If the query is feasible, the for-all-exists reasoning is used to generalise the returned trace. IC3 is used to improve reachability information, thereby promoting greater generalisation when the query is unattainable. This process enhances the effectiveness of the CTL model. [9] This paper presents a solution to the obstacles faced by current frameworks for proving CTL properties: they cannot directly maintain particular existential CTL formulas, are restricted to a subset of CTL, and are limited to certain program types. The goal of this paper was achieved by using an abstract of the operational trace semantics of a program. [10] discussed the application of model checkers for STCTL and synthesis strategies, while the current approach is constrained in scope and difficult to prove valid. The modulo SMT rewriting logic was used to accomplish the synthesis strategies. This paper considers a proving technique for a flow-shop scheduling system. It presents a control system that computes the makespan using Computational Tree Logic (CTL) and Linear-Time Logic (LTL) and the NuSMV Model Checker. Given the proliferation of manufacturing companies, we need to develop an effective strategic design to increase productivity and remain competitive. To this end, this research aims to answer three scientific questions to achieve our goals.

- How can the proposed CTL and LTL verify the perfection of flow shop scheduling system behaviours?
- How can the system adapt to achieve the optimal makespan?
- How can CTL and LTL contribute to improving the performance of institutions?

This research aims to automatically verify the correctness of the flow-shop scheduling system behaviour to meet optimal requirements in industry. Manufacturing companies seek to minimise makespan to increase productivity, reduce energy consumption, reduce staff effort, and improve customer satisfaction. This system can identify inefficiency at the earliest stage of

production. Flow shop scheduling systems are used in various domains, including manufacturing, medical services (e.g., surgical procedures), the food industry, and power plants. These institutions implement verification systems to ensure the reliability of their application systems. Also, prevent and identify the problem at an early stage to advance and refine their systems' techniques. In this research, we aim to achieve this using CTL, LTL, and the NuSMV Model Checker.

1.1.Contribution and Novelty

This study presents a formal verification-based framework for assessing the validity of heuristic scheduling algorithms in flow shops, with particular reference to a three-machine extension of Johnson's rule. This contrasts with prior literature, which relies solely on performance-based comparisons or simulations for verification. [2,4-5], but the study uses temporal logic (CTL and LTL) and the NuSMV model checker to verify the logical behaviour of the scheduling process. Among the most significant contributions is the generation of a finite-state machine (FSM) model that encompasses the entire scheduling process, i.e., random sequencing, dominance, reduction of 3-machine schedules to 2-machine schedules, and makespan comparison. This model enables the verification of all feasible scheduling paths and outcomes; hence, completeness in evaluating schedule correctness is achieved—a dimension that has not been comprehensively addressed in scheduling studies. [7,11]. Furthermore, the paper illustrates how failures of heuristics otherwise widely adopted in practice, such as an extended version of Johnson's rule, can be revealed through formal counterexamples. The algorithm is assessed in a particular case that does not yield an optimal sequence, and this violation is traced to a temporal logic specification that is false. Not only does this present evidence of limitation, but it also provides a way forward to the construction of verifiably proper heuristic procedures. Although the current techniques, NEHLPD, CDS, and TWK-centred hybrids [4-5, aim to enhance scheduling outcomes, they do not formally establish the correctness of the scheduling logic. This paper fills that gap by employing CTL/LTL-based formal verification, thereby providing logical rather than numerical optimality. This is the only way to proceed when performance is not the only critical factor in the system. Moreover, methodologically, this procedure is generalizable to other scheduling algorithms, and verification frameworks can be developed to apply to other algorithms (metaheuristics-based or AI-driven models) [6,9-10]. It can serve as a basis for subsequent researchers to design verifiable scheduling systems, including automated manufacturing systems, cyber-physical systems, and real-time operations. Overall, this paper brings, in addition to innovating the methodological aspect of the problem, i.e., the incorporation of formal methods in scheduling, an insight into the working mode of failure of traditional heuristics. It also bridges the gap between operations research and formal verification. It suggests a line of development of scheduling strategies that are not only efficient but also formally correct by design. To clear the definition of the Makespan, see the following Subsection.

1.2.Definition of Makespan

The total time required to complete all jobs in a given schedule is known as the makespan [4]. In a multi-machine flow-shop scheduling problem, it is the completion time of the last job on the previous machine. One of the basic goals in production scheduling is to minimise makespan, as it curtails overall production efficiency, resource utilisation, and throughput. When the makespan is lower, the system can complete more jobs in a shorter period, resulting in lower operational costs and higher productivity. This paper considers makespan as the primary performance metric to evaluate the efficiency of scheduling heuristics, such as Johnson's rule, CDS, and NEH.

2.THE EXTENSION OF JOHNSON'S RULE

In this study, we propose a verification system based on CTL and LTL to extend Johnson's rule to flow-shop scheduling. Figure 1 depicts the finite-state machine of the proposed system.

1- The state S0 includes the presence of three jobs for three machines. The number of possible job orders is 3! (six possible orders). For each possible order, the makespan is calculated and stored in a random-choice variable.
2- Randomly choose one order from the random choice variable. This operation transitions to a new state S1. Suppose the optimal makespan is zero. Subsequently, check whether any machine is dominant over the other. If this operation does not achieve an optimal makespan of zero, then transits to state S2; otherwise, transits to state S3.
3- In state S2, the system ends at this state and cannot apply Johnson's rule. In state S3, Johnson's rule applies to the machines. Then, the operation is signified by converting 3M to 2M, which transits to state S4.
4- Apply Johnson's rule operation to the three resulting jobs and two machines (state 4) to transit to state S5.
5- The Johnson's rule solutions are represented in the new state S5 (optimal job order), then calculate the makespan operation to transit to state S6.
6- When calculating the makespan for the optimal order, the result will be the optimal

makespan (state S6). Then, if the optimal makespan is less than or equal to the random makespan, then return to the S0 state; otherwise, transit to the S7 state (Johnson's rule failed). This operation occurs when the optimal makespan exceeds the random makespan, indicating that Johnson's rule fails.

7- When transiting from S6 to S0 state, this operation ensures that all possibilities are considered to determine if there is any possible order better than the optimal makespan. Subsequently, the previous operations are repeated until state S1 is reached.

8- In state S1, random order (for second-order jobs) is compared with the new value of the optimal makespan. If the optimal makespan is better than the random order, return to S0; otherwise, return to S7, and this operation represents (Optimal > Random).

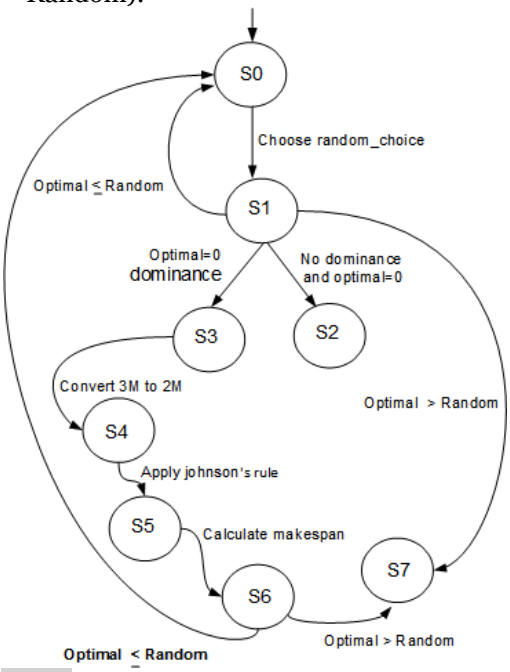

Fig. 1 Finite State Machine of the Proposed System.

3.TEMPORAL LOGIC

Temporal logic is a modal logic used to interpret changes in symbols over time. It provides tools to analyse the system's performance over time. The temporal logic used for asynchronous and synchronous systems [12,13]. In this paper, we present two temporal logics: CTL and LTL.

3.1.Syntax of the CTL

CTL is used to specify system properties for several possible execution paths. It can verify whether the system satisfies the intended temporal properties and apply operators and quantifiers to express complex properties [11]. This can be gained using the following operators:

1- Logical Operators $\neg, \top, \wedge, \bot, \vee, \rightarrow$, path quantifiers $\boldsymbol{A}$, $\boldsymbol{E}$.

2- temporal operators $\boldsymbol{G}$, $\boldsymbol{U}$, $\boldsymbol{X}$, and $\boldsymbol{F}$.

In this paper, the $\Phi The\ The\ The$ equation formula contains many Atomic Propositions (AP) that are used in our system, such as calculating makespan, ordering jobs, checking dominance, and converting a 3-machine problem to a 2-machine problem. These AP will relate to p_i, $i = 1,2, \ldots$, A CTL formula is written as follows:

$\phi ::= p_i \mid \neg \phi \mid \phi_1 \wedge \phi_2 \mid$
$\phi_1 \vee \phi_2 \mid \boldsymbol{EX}\Phi \mid \boldsymbol{AG}\phi \mid \boldsymbol{AX}\Phi \mid \boldsymbol{EF}\Phi$
$\mid \boldsymbol{AF}\Phi \mid \boldsymbol{A}[\phi_1 \boldsymbol{U} \phi_2] \mid \boldsymbol{AG}\phi \mid \boldsymbol{E}[\phi_1 \boldsymbol{U} \phi_2]$

3.2.Semantics of the CTL

Now, we can say when an atomic proposition. p_i is true at a state or time s_i in the system M if: $M, s_i \vDash p_k$, for all $p_k \in p_i$. The structure on which the system M is based is an encounter with a formal description in the form of a tuple (S, I, R, AP, L) as presented in [7,14]:

S: is a finite set of states.

$I \subseteq S$: is a finite set of initial states.

R: is a total transition relation such that $R \subseteq S \times S$.

AP: is the set of atomic propositions

L: is a function that maps to each state the entire set of atomic propositions that hold in this state.

Such that $L: S \rightarrow 2^{AP}$.

The semantics of the basic operators are formally described as follows:

$$M, s_i \vDash \neg \phi \ \ iff \ \ M, s_i \nvDash \phi.$$
$$and M, s_i \vDash \phi \wedge \psi \ \ iff \ \ M, s_i \vDash \phi \ \ and \ \ M, s_i \vDash \psi.$$
$$M, \vDash \phi \vee \psi \ \ iff \ M, s_i \vDash \phi \ or \ M, s_i \vDash \psi.$$
$$M, s_i \vDash \phi \Rightarrow \psi \ iff \ \ if \ \ M, s_i \vDash \phi \ \text{then} \ M, s_i \vDash \psi.$$

Let a path λ that starts in a state. s_i, where $\lambda = s_i, s_{i+1}, \ldots$ in the system model M. The following is an interpretation of the temporal operators over M:

$$M, s_i \vDash \boldsymbol{AX}\Phi iff \ \forall \lambda = (s_i, s_{i+1}, \ldots), \ M, s_{i+1} \vDash \Phi.$$
$$M, s_i \vDash \boldsymbol{EX}\Phi \ iff \ \exists \lambda = (s_i, s_{i+1}, \ldots), M, s_{i+1} \vDash \Phi.$$
$$M, s_i \vDash \boldsymbol{AF}\Phi \ iff \ \forall \lambda = (s_i, s_{i+1}, \ldots), \exists \, j \geq i, \ M, s_j \vDash \Phi.$$
$$M, s_i \vDash \boldsymbol{EF}\Phi \ \text{iff} \ \exists \lambda = (s_i, s_{i+1}, \ldots), \exists \, j \geq i, M, s_j \vDash \Phi.$$
$$M, s_i \vDash \boldsymbol{AG}\phi \ iff, \forall \lambda = (s_i, s_{i+1}, \ldots), \text{and} \ \forall j \geq i, M, s_j \vDash \phi.$$
$$M, s_i \vDash \boldsymbol{EG}\phi \ iff \ \exists \lambda = (s_i, s_{i+1}, \ldots), \text{and} \ \forall j \geq i, M, s_j \vDash \phi \ .$$

$M, s_i \vDash \boldsymbol{A}[\phi_1 \boldsymbol{U} \phi_2]\ iff\ \forall \lambda = (s_i, s_{i+1}, \dots), \exists\, j \geq i$ such that $M, s_j \vDash \phi_2$and,$\forall k, i \leq k < j, M, s_k \vDash \phi_1$.
$M, s_i \vDash \boldsymbol{E}[\phi_1 \boldsymbol{U} \phi_2]\ iff\ \exists \lambda = (s_i, s_{i+1}, \dots)$such that $\exists j \geq i, M, s_j \vDash \phi_2$and,$\forall k, i \leq k < j, M, s_k \vDash \phi_1$.

3.3.LTL Syntax

The LTL model focuses on system behaviour over time and assesses whether specified temporal requirements are satisfied. 1 5]. LTL c p_i, $i = 0$,1,2, ..., logical operators $\neg, \top, \wedge, \bot, \vee, \rightarrow$ and temporal operators $\boldsymbol{G}$, $\boldsymbol{U}$, $\boldsymbol{X}$, $\boldsymbol{F}$. The LTL is clarified inductively using the following formulas:

$$\phi ::= p_i \mid \neg \phi \mid \boldsymbol{G}\phi \mid \phi_1 \vee \phi_2 \mid \boldsymbol{F}\Phi \mid \boldsymbol{X}\Phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \boldsymbol{U} \phi_2$$

3.4.LTL Semantics

Analogous to CTL, if ϕ is a path formula, then $M, s_i \vDash p_k$ for a path $\lambda = s_i, s_{i+1}, \dots$ This indicates that ϕ is satisfied along the path λ within the model structure M. The relation is inductively defined as follows (see [7]):

$$M, s_i \vDash \neg \phi \ \ iff\ \ M, s_i \nvDash \phi$$

$$and M, s_i \vDash \phi \wedge \psi\ \ iff\ M, s_i \vDash \phi\ \ and\ M, s_i \vDash \psi$$

$$M, \vDash \phi \vee \psi\ iff\ M, s_i \vDash \phi\ or\ M, s_i \vDash \psi$$

$$M, s_i \vDash \phi \Rightarrow if\ \ M, s \ \ \vDash \phi\ \text{then}\ ,\ \ \vDash \psi M, s_i \vDash \boldsymbol{X}\Phi\ \ iff\ \ M, s_{i+1} \vDash \Phi. M, s_i \ \vDash \boldsymbol{F}\phi\ if\ \exists, s_j \vDash \Phi$$

$M, s_i \vDash \boldsymbol{G}\phi\ iff\ \forall ji, M, \ \ _j \vDash \phi\ M, s_i \vDash \boldsymbol{A}[\phi_1\, \boldsymbol{U} \phi_2]\ if\ \exists\, j \geq i$ such that $M, s_j \vDash \phi_2$ an,$\forall k, i \leq k < j, M, s_k \vDash \phi_1$.

4.SPECIFICATIONS OF THE PROPOSED FLOW SHOP SCHEDULING SYSTEM

Specifications specify the standards that systems must meet to ensure correct performance. It is related to guide manufacturing [16]. Under this heading, we present the correctness conditions that the proposed model must satisfy. If all these conditions are met, our model is correct.

1- At any time, Johnson's rule eventually fails to produce the optimal order for the three tasks, so the random order chosen is better. Below is the condition encoded in CTL and LTL:

$$\boldsymbol{\delta 1 = AG\,(EF\,(state = s7))} \quad \textbf{(1)}$$
$$\boldsymbol{\beta 1 = G\,(F\,(state = s7))} \quad \textbf{(2)}$$

2- When determining the optimal order using Johnson's rule, ensure that no two tasks are duplicated within the order list at any time. Below is the condition encoded in CTL and LTL:

$$\boldsymbol{\delta 2 = AG\,(j1.job1\_position \neq j1.job2\_position) \wedge (j1.job1\_position \neq j1.job3\_position) \wedge (j1.job2\_position \neq j1.job3\_position)} \quad \textbf{(3)}$$
$$\boldsymbol{\beta 2 = G(j1.job1\_position \neq j1.job2\_position) \wedge (j1.job1\_position \neq j1.job3\_position) \wedge (j1.job2\_position \neq j1.job3\_position)} \quad \textbf{(4)}$$

3- At any time, if we are in state 1 and the first and second machines dominate the third machine, the system can apply Johnson's rule, meaning it will not reach state 2. Below is the condition encoded in CTL and LTL:

$$\boldsymbol{\delta 3 = AG((state = s1 \,\&\, d1.dominance\_machine1 - and - machine2\_over\_machine3 = TRUE) \Rightarrow \neg(AF\,(state = s2)))} \quad \textbf{(5)}$$
$$\boldsymbol{\beta 3 = G((state = s1 \,\&\, d1.dominance\_machine1 - and - machine2\_over\_machine3 = TRUE) \Rightarrow \neg(F(state = s2)))} \quad \textbf{(6)}$$

4- At any time, given the system's processing times, there will always be a dominance, so state 2 cannot be reached under any circumstances. Below is the condition encoded in CTL and LTL:

$$\boldsymbol{\delta 4 = AG\,(\neg(state = s2))} \quad \textbf{(7)}$$
$$\boldsymbol{\beta 4 = G\,(\neg(state = s2))} \quad \textbf{(8)}$$

5- At any time, if we are in state 1 and the optimal makespan is greater than zero and less than or equal to random choice, then the next state will be state 0. Below is the condition encoded in CTL and LTL:

$$\boldsymbol{\delta 5 = AG((state = s1 \wedge optimal\_makespan > 0 \wedge optimal\_makespan \leq random\_choice) \Rightarrow AX(state = s0))} \quad \textbf{(9)}$$
$$\boldsymbol{\beta 5 = G((state = s1 \wedge optimal\_makespan > 0 \wedge optimal\_makespan \leq random\_choice) \Rightarrow X(state = s0))} \quad \textbf{(10)}$$

6- At any time, if we are in state 1 and the optimal makespan is greater than zero and greater than the random choice, the next state will be state 7. Below is the condition encoded in CTL and LTL:

$$\boldsymbol{\delta 6 = AG((state = s1 \wedge optimal\_makespan > 0 \wedge optimal\_makespan > random\_choice) \Rightarrow AX(state = s7))} \quad \textbf{(11)}$$

$$\beta 6 = G((state = s1 \wedge optimal\_makespa > 0 \wedge optimal\_makespan > random\_choice) \Rightarrow X(state = s7)) \quad (12)$$

Let M be the proposed model for flow shop scheduling, and Φ be the correctness conditions expressed in LTL and/or CTL such that: $\Phi = \bigwedge (\delta i \vee \beta i)\ 1 \leq i \leq 8$. Then, for all $Sj \in S$ (set of states).

5.NUSMV MODEL CORRESPONDING TO FLOW SHOP SCHEDULING SYSTEM

NuSMV is a model that analyses system behaviour, ensures system correctness, and expresses complex and real-time system properties. [17]. It represents finite state systems, whether synchronous or asynchronous. NuSMV can verify LTL and CTL specifications to determine whether they are true or false within an FSM. [7,18-19]. The NuSMV script is used to describe our model and verify whether the proposed system satisfies the correctness conditions. Then, the NuSMV model generates all possible conditions across all states and uses the temporal logic CTL or LTL to identify properties. Once the possible conditions are generated, NuSMV checks the temporal logic formulas, yielding two possible outcomes: satisfaction of the correctness condition (True) or non-satisfaction (False). NuSMV provides a counterexample if the property is not satisfied, and the results are a refined and improved system model. This is shown in Figure 6.

5.1. State Variables of the Corresponding Model

CTL is used to specify system properties for several possible execution paths. It can verify whether the system satisfies the intended temporal properties and apply operators and quantifiers to express complex properties. [11]. This can be gained using the following operators: In this subsection, we introduce the state variables, modules, and correctness conditions of the proposed NuSMV model. Below is a description of our proposed model in NuSMV, which was previously illustrated as an FSM in Figure 1:

S0: start (3 jobs, 3 machines)
S1: Random choice and optimal makespan
S2: Johnson's rule cannot be applied.
S3: Johnson's rule can be applied. S4: 3 jobs, two new machines.
S5: optimal job order
S6: optimal makespan
S7: Johnson's rule failed.

As shown below, the description for the correctness condition in NuSMV:

At any time, given the system's processing times, dominance holds, so state 2 cannot be reached under any circumstances. Below is the condition encoded in CTL and LTL equations 7 and 8.

This condition is represented in the NuSMV script, and the remaining correctness conditions are encoded similarly.

$$SPEC\ AG\ (!\ (state = s2)) \quad (13)$$

$$LTLSPEC\ G\ (!\ (state = s2)) \quad (14)$$

5.2.Modules of the Corresponding Model

In this subsection, four modules are used in the NuSMV model: calculate makespan, dominance, convert three machines into two machines, and Johnson's rule. Figure 3 shows the makespan calculation module, repeated for each possible order (6 possibilities). Processing times are first assigned, and the cumulative processing times for each of the three machines are then calculated. Finally, the makespan is calculated as machine3_time2. Figure 4 shows the dominance module, which computes the minimum and maximum values for the three machines. If the minimum of one machine is greater than or equal to the maximum of another machine, the first machine will dominate the others. Based on the set processing times, Machines 1 and 2 will dominate Machine 3. Identifying the dominant machine reduces the number of machines from 3 to 2 by summing the processing times of machines 1 and 3; the resulting value represents the processing time of the new machine 1. The processing times of machine 2 are added to those of machine 3; the resulting values represent the processing times of the new machine 2. Thus, we have the processing times for two new machines, as shown in Fig. 5. The processing times for the two new machines resulting from the three-machine-to-two-machine conversion module are set within the Johnson's rule module. The minimum time for each job is then calculated, and the jobs with the minimum time are identified. The optimal order is determined based on Johnson's rule, as shown in Fig. 2.

```
MODULE Johnsons_Rule
DEFINE
  --Processing times on the two new machines
 machine1_time1 := 5;
  machine1_time2 := 6;
  machine1_time3 := 6;
  machine2_time1 := 8;
  machine2_time2 := 8;
  machine2_time3 := 10;
  -- Calculate the minimum time for all jobs in a gradual manner.
 smallest_time1 := min(machine1_time1, machine2_time1); -- Minimum time for job 1
  smallest_time2 := min(machine1_time2, machine2_time2); -- Minimum time for job 2
  smallest_time3 := min(machine1_time3, machine2_time3); -- Minimum time for job 3
  --Smallest time among all jobs
  smallest_time := min(smallest_time1, min(smallest_time2, smallest_time3));
  -- Determine the job with the shortest time
  selected_job := case
    smallest_time = machine1_time1 | smallest_time = machine2_time1 : 1;
    smallest_time = machine1_time2 | smallest_time = machine2_time2 : 2;
    smallest_time = machine1_time3 | smallest_time = machine2_time3 : 3;
    TRUE : 0;
  esac;
  --Jobs Ranking Based on Johnson's Rule
  job1_position := selected_job;
  job2_position := case
    job1_position = 1 : 2;
    job1_position = 2 : 3;
    TRUE : 1;
  esac;
  job3_position := case
    job1_position = 1 & job2_position = 2 : 3;
    job1_position = 2 & job2_position = 1 : 3;
    TRUE : 1;
  esac;
```

Fig. 2 Module for Johnson's Rule.

```
MODULE makespan1
DEFINE
  processing_time1 := [3, 6, 2];
  processing_time2 := [5, 7, 1];
  processing_time3 := [4, 8, 2];

  -- Calculate cumulative processing times for the first machine.
  machine1_time0 := processing_time1[0];
  machine1_time1 := machine1_time0 + processing_time1[1];
  machine1_time2 := machine1_time1 + processing_time1[2];

  -- Calculate cumulative processing times for the second machine.
  machine2_time0 := machine1_time0 + processing_time2[0];
  machine2_time1 := max(machine1_time1, machine2_time0) + processing_time2[1];
  machine2_time2 := max(machine1_time2, machine2_time1) + processing_time2[2];

  -- Calculate the cumulative processing times for the third machine.
  machine3_time0 := machine2_time0 + processing_time3[0];
  machine3_time1 := max(machine2_time1, machine3_time0) + processing_time3[1];
  machine3_time2 := max(machine2_time2, machine3_time1) + processing_time3[2];

  -- Final value of makespan
  makespan := machine3_time2;
```

Fig. 3 Module for Makespan.

```
MODULE dominance
DEFINE
  -- Processing times for jobs on machines
 processing_time1 := [3, 6, 2];
  processing_time2 := [5, 7, 1];
  processing_time3 := [4, 8, 2];

  -- Calculate the minimum and maximum time for each machine.
  min_time_machine1 := min(processing_time1[0], min (processing_time2[0], processing_time3[0]));
  max_time_machine1 := max(processing_time1[0], max (processing_time2[0], processing_time3[0]));

  min_time_machine2 := min(processing_time1[1], min (processing_time2[1], processing_time3[1]));
  max_time_machine2 := max(processing_time1[1], max (processing_time2[1], processing_time3[1]));

  min_time_machine3 := min(processing_time1[2], min (processing_time2[2], processing_time3[2]));
  max_time_machine3 := max(processing_time1[2], max (processing_time2[2], processing_time3[2]));

  -- Checking dominance between machines
  dominance_machine1_over_machine2 := min_time_machine1 >= max_time_machine2;
  dominance_machine2_over_machine3 := min_time_machine2 >= max_time_machine3;
  dominance_machine1_over_machine3 := min_time_machine1 >= max_time_machine3;
  dominance_machine1-and-machine2_over_machine3 := (min_time_machine1 >= max_time_machine3) & (min_time_machine2 >= max_time_machine3);
```

Fig. 4 Module for Dominance.

```
MODULE Convert_2M_to_3M
DEFINE
  -- Processing times for jobs on older machines
 processing_time1 := [3, 5, 4];
  processing_time2 := [6, 7, 8];
  processing_time3 := [2, 1, 2];

  -- Merge operations to create two new machines.
  new_machine1 := [processing_time1[0] + processing_time3[0],
                   processing_time1[1] + processing_time3[1],
                   processing_time1[2] + processing_time3[2]];

  new_machine2 := [processing_time2[0] + processing_time3[0],
                   processing_time2[1] + processing_time3[1],
                   processing_time2[2] + processing_time3[2]];
```

Fig. 5 Module for Converting 3M into 2M.

6.RESULTS AND DISCUSSION

In this section, we present the results of running the model in NuSMV, which determine the correctness of the CTL and LTL specifications. When NuSMV detects a falsity, it provides a counterexample, i.e., a path in the FSM that led to the falsity of the property; if the property is correct, NuSMV returns true. When running, the specifications stated in Section 4 hold in all possible situations, as shown in Fig. 7. To demonstrate NuSMV's ability to detect false specifications in our proposed model, we ran the following assumption. It is impossible to reach state 7; that is, no arrangement can improve on Johnson's rule. This condition can be represented by CTL as follows:

$$\delta 7 = AG\neg(state = s7) \quad (15)$$

When this condition is run in NuSMV, it returns false and generates a counterexample, as shown in Fig. 6. We will demonstrate the counterexample produced by NuSMV by analysing a 3-job, 3-machine flow shop configuration. Table 1 lists all the required processing times. According to the dominance condition used in the three-machine extension of Johnson's rule, Machine 3 can be merged with Machines 1 and 2 if

$\min(M1) \geq \max(M3) \rightarrow \min(3, 5, 4) = 3 \geq \max(2, 1, 2) = 2$

$\min(M2) \geq \max(M3) \rightarrow \min(6, 7, 8) = 6 \geq \max(2, 1, 2) = 2$

Since both conditions hold, the instance qualifies for two-machine reduction and the use of Johnson's rule. The transformed work

sequence becomes J1 → J2 → J3. The makespan thus calculated is as follows:
M1: [3, 3+ 5 = 8, 8+ 4 = 12]
M2: [3+ 6 = 9, max(8,9) + 7 = 16, max(12,16) + 8 = 24]
M3: [9+ 2 = 11, max(11,16) + 1 = 17, max(24,17) + 2 = 26]
Total makespan = 26
But with some other sequence of operations J1 → J3 → J2, the makespan is:
M1: [3, 3+ 4 = 7, 7+ 5 = 12]
M2: [3+ 6 = 9, max(7,9) + 8 = 17, max(12,17) + 7 = 24]
M3: [9+ 2 = 11, max(11,17) + 2 = 19, max(24,19) + 2 = 25]
Makespan = 25
Which proves superior to the 26 generated by Johnson's algorithm. Extended Johnson's rule produced suboptimal scheduling in this particular example. A counterexample is generated by NuSMV to demonstrate that the heuristic yields suboptimal results, as the temporal logic specification AG(johnson_result <= optimal_makespan) is invalid.

```
-- specification AG !(state = s7)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    state = s0
    optimal_makespan = 0
    ms1_value = 0
    ms2_value = 0
    ms3_value = 0
    ms4_value = 0
    ms5_value = 0
    ms6_value = 0
    random_choice = 0
    ms1_module.makespan = 26
    ms1_module.machine3_time2 = 26
    ms1_module.machine3_time1 = 24
    ms1_module.machine3_time0 = 12
    ms1_module.machine2_time2 = 17
    ms1_module.machine2_time1 = 16
    ms1_module.machine2_time0 = 8
    ms1_module.machine1_time2 = 11
    ms1_module.machine1_time1 = 9
    ms1_module.machine1_time0 = 3
    ms1_module.processing_time3[0] = 4
    ms1_module.processing_time3[1] = 8
    ms1_module.processing_time3[2] = 2
    ms1_module.processing_time2[0] = 5
    ms1_module.processing_time2[1] = 7
    ms1_module.processing_time2[2] = 1
    ms1_module.processing_time1[0] = 3
    ms1_module.processing_time1[1] = 6
    ms1_module.processing_time1[2] = 2
    ms2_module.makespan = 25
    ms2_module.machine3_time2 = 25
    ms2_module.machine3_time1 = 24
    ms2_module.machine3_time0 = 12
    ms2_module.machine2_time2 = 19
    ms2_module.machine2_time1 = 17
    ms2_module.machine2_time0 = 7
    ms2_module.machine1_time2 = 11
    ms2_module.machine1_time1 = 9
    ms2_module.machine1_time0 = 3
```

```
    ms2_module.processing_time3[1] = 7
    ms2_module.processing_time3[2] = 1
    ms2_module.processing_time2[0] = 4
    ms2_module.processing_time2[1] = 8
    ms2_module.processing_time2[2] = 2
    ms2_module.processing_time1[0] = 3
    ms2_module.processing_time1[1] = 6
    ms2_module.processing_time1[2] = 2
    ms3_module.makespan = 28
    ms3_module.machine3_time2 = 28
    ms3_module.machine3_time1 = 26
    ms3_module.machine3_time0 = 12
    ms3_module.machine2_time2 = 20
    ms3_module.machine2_time1 = 18
    ms3_module.machine2_time0 = 8
    ms3_module.machine1_time2 = 13
    ms3_module.machine1_time1 = 12
    ms3_module.machine1_time0 = 5
    ms3_module.processing_time3[0] = 4
    ms3_module.processing_time3[1] = 8
    ms3_module.processing_time3[2] = 2
    ms3_module.processing_time2[0] = 3
    ms3_module.processing_time2[1] = 6
    ms3_module.processing_time2[2] = 2
    ms3_module.processing_time1[0] = 5
    ms3_module.processing_time1[1] = 7
    ms3_module.processing_time1[2] = 1
    ms4_module.makespan = 28
    ms4_module.machine3_time2 = 28
    ms4_module.machine3_time1 = 26
    ms4_module.machine3_time0 = 12
    ms4_module.machine2_time2 = 22
    ms4_module.machine2_time1 = 20
    ms4_module.machine2_time0 = 9
    ms4_module.machine1_time2 = 13
    ms4_module.machine1_time1 = 12
    ms4_module.machine1_time0 = 5
    ms4_module.processing_time3[0] = 3
    ms4_module.processing_time3[1] = 6
    ms4_module.processing_time3[2] = 2
    ms4_module.processing_time2[0] = 4
    ms4_module.processing_time2[1] = 8
    ms4_module.processing_time2[2] = 2
    ms4_module.processing_time1[0] = 5
    ms4_module.processing_time1[1] = 7
    ms4_module.processing_time1[2] = 1
    ms5_module.makespan = 26
    ms5_module.machine3_time2 = 26
    ms5_module.machine3_time1 = 25
    ms5_module.machine3_time0 = 12
    ms5_module.machine2_time2 = 20
    ms5_module.machine2_time1 = 18
    ms5_module.machine2_time0 = 7
    ms5_module.machine1_time2 = 14
    ms5_module.machine1_time1 = 12
    ms5_module.machine1_time0 = 4
    ms5_module.processing_time3[0] = 5
    ms5_module.processing_time3[1] = 7
    ms5_module.processing_time3[2] = 1
    ms5_module.processing_time2[0] = 3
    ms5_module.processing_time2[1] = 6
    ms5_module.processing_time2[2] = 2
    ms5_module.processing_time1[0] = 4
    ms5_module.processing_time1[1] = 8
    ms5_module.processing_time1[2] = 2
    ms6_module.makespan = 27
    ms6_module.machine3_time2 = 27
    ms6_module.machine3_time1 = 25
    ms6_module.machine3_time0 = 12
    ms6_module.machine2_time2 = 20
    ms6_module.machine2_time1 = 19
    ms6_module.machine2_time0 = 9
    ms6_module.machine1_time2 = 14
    ms6_module.machine1_time1 = 12
    ms6_module.machine1_time0 = 4
    ms6_module.processing_time3[0] = 3
    ms6_module.processing_time3[1] = 6
    ms6_module.processing_time3[2] = 2
    ms6_module.processing_time2[0] = 5
    ms6_module.processing_time2[1] = 7
    ms6_module.processing_time2[2] = 1
    ms6_module.processing_time1[0] = 4
    ms6_module.processing_time1[1] = 8
    ms6_module.processing_time1[2] = 2
    j1.job3_position = 3
    j1.job2_position = 2
```

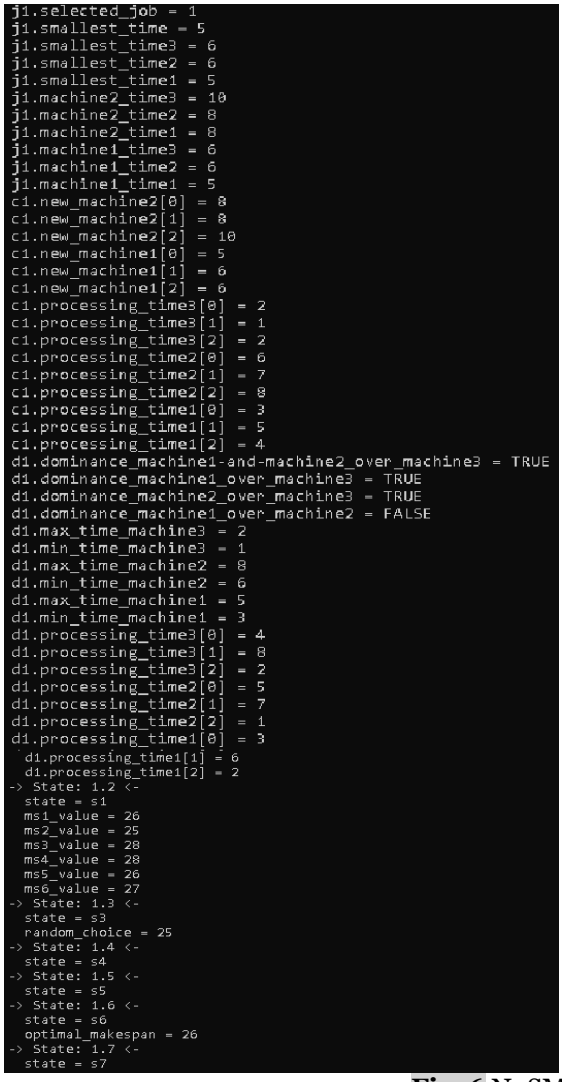

Fig. 6 NuSMV Counter Example.

```
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification AG !(state = s2)  is true
-- specification EF state = s7  is true
-- specification  G !(state = s2)  is true
-- specification  F state = s7  is true
```

Fig. 7 NuSMV Run Script.

Table 1 M1, M2, and M3: Processing Times for Job J on Machines 1, 2, and 3, Respectively.

	M1	M2	M3
J1	3	6	2
J2	5	7	1
J3	4	8	2

J1, J2, J3: Job identifiers.

Makespan: Total time required to complete all jobs in the sequence.

7.CONCLUSIONS

In this study, we applied CTL, LTL, and the NuSMV model checker to formally evaluate the extended Johnson's rule for scheduling three-machine flow-shop problems. Unlike in the standard two-machine case, where Johnson's algorithm yields the optimal schedule, our verification shows that the three-machine case can yield non-optimal schedules. This was clearly demonstrated by an automatically generated counterexample in NuSMV, which employed a Kripke structure to model the scheduling process. According to the findings, FM helps reveal unexpected flaws in widely used scheduling methods. Our method introduces a new analytical tool by uncovering cases in which the extended rule is suboptimal. Other researchers may strengthen the extension, propose new heuristics with formal warranties, or employ analysis methods similar to those used in other metaheuristic scheduling routines.

Appendix A

The NuSMV script for the proposed model is shown as follows:

```
1   MODULE main
2   VAR
3   State: s0, s1, s2, s3, s4, s5, s6, s7
4   optimal_makespan : 0..100;
5   ms1_module : makespan1()
6   ms2_module : makespan2()
7   ms3_module : makespan3()
8   ms4_module : makespan4()
9   ms5_module : makespan5()
10  ms6_module : makespan6()
11  ms1_value : 0..100;
12  ms2_value : 0..100;
13  ms3_value : 0..100;
14  ms4_value : 0..100;
15  ms5_value : 0..100;
```

```
        ms6_value : 0..100;
        random_choice : 0..100;
        j1 : Johnsons_Rule()
        c1 : Convert_2M_to_3M();
        d1 : dominance()
    ASSIGN
        init(state):= s0
        init(optimal_makespan) := 0;
        init(ms1_value) := 0;
        init(ms2_value) := 0;
        init(ms3_value) := 0;
        init(ms4_value) := 0;
        init(ms5_value) := 0;
        init(ms6_value) := 0;
        init(random_choice) := 0;
        next(ms1_value):= ms1_module.makespan;
        next(ms2_value):=
ms2_module.makespan;
        next(ms3_value):=
ms3_module.makespan;
        next(ms4_value):=
ms4_module.makespan;
        next(ms5_value):=
ms5_module.makespan;
        next(ms6_value):=
ms6_module.makespan;
        -- Define random choice
        next(random_choice) :=
        case
        TRUE: ms2_value
        esac;
        -- State transitions
        next(state) :=
        case
        (state = s0) : s1
        -- State s1: Check conditions
        (state = s1) & (optimal_makespan = 0) &
(d1.dominance_machine1-and-
machine2_over_machine3 = FALSE) : s2;
        (state = s1) & (optimal_makespan = 0) &
(d1.dominance_machine1-and-
machine2_over_machine3 = TRUE) : s3;
        (state = s1) & (optimal_makespan > 0) &
(optimal_makespan <= random_choice) : s0;
        (state = s1) & (optimal_makespan > 0) &
(optimal_makespan > random_choice) : s7;
        -- State s2: Johnson's rule cannot be applied
        (state = s2) : s2
        -- State s3: Check machine dominance
        (state = s3) & (c1.new_machine1[0] = 5) &
(c1.new_machine1[1] = 6)   &
(c1.new_machine1[2]=6)    &
(c1.new_machine2[0]=8)   & (c1.new_machine2[1] =
8) & (c1.new_machine2[2] = 10) : s4;
        -- State s4: Apply Johnson's rule
        (state = s4) & (j1.job1_position = 1) &
(j1.job2_position = 2) & (j1.job3_position = 3) : s5;
        -- State s5: Calculate the optimal makespan
        (state = s5) : s6
        -- State s6: Compare makespans
        (state = s6) & (optimal_makespan > 0) &
(optimal_makespan <= random_choice) : s0;
        (state = s6) & (optimal_makespan > 0) &
(optimal_makespan > random_choice) : s7;
        -- State s7: Johnson's rule failed
        (state = s7) : s7
        TRUE: state
        esac;
        next(optimal_makespan) :=
        case
        (state = s5) : 26;
        TRUE: optimal_makespan
        esac;
```

REFERENCES

[1] Della Croce F, Tadei R, Volta G. **A Genetic Algorithm for the Job Shop Problem.** *Computers and Operations Research* 1995; **22**(1): 15–24.

[2] Mashuri C, Mujianto AH, Sucipto H, Arsam RY, Permadi GS. **Production Time Optimization Using the Campbell Dudek Smith Algorithm for Production Scheduling.** *The 4th International Conference on Energy, Environment, Epidemiology and Information System (ICENIS 2019)* 2019; 1-5.

[3] Alfuad T, Dwijayanti K. **Flowshop Production Scheduling Using Campbell, Dudek, Smith, Palmer, and Dannenbring Methods to Minimize the Total Production Time (Case Study: PT. Naturindo Fresh Kulon Progo, Indonesia).** *6th International Conference on Science and Engineering* 2023; 279–290.

[4] Kurniawan LA, Farizal F. **Development of Flow Shop Scheduling Method to Minimize Makespan Based on Nawaz Enscore Ham (NEH) and Campbell Dudek and Smith (CDS) Method.** *3rd African International Conference on Industrial Engineering and Operations Management* 2022; 1224–1231.

[5] Setiawan D, Ramadhani A, Cahyo WN. **Production Scheduling to Minimize Makespan Using Sequencing Total Work Method and Campbell Dudek Smith Algorithm.** *IOP Conference Series: Materials Science and Engineering* 2020; **598**: 1-7.

[6] Aminof B, De Giacomo G, Murano A, Rubin S. **Planning Under LTL Environment Specifications.** *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS)* 2019; 31–39.

[7] Alshorman R. **Proving the Car Security System Model Using CTL and LTL.** *Journal of Theoretical and Applied Information Technology* 2024; **102**(3): 1112–1119.

[8] Hassan Z, Bradley AR, Somenzi F. **Incremental, Inductive CTL Model Checking.** *International Conference on Computer Aided Verification (CAV 2012)* 2012; 532–547.

[9] Urban C, Ueltschi S, Müller P. **Abstract Interpretation of CTL Properties.** *Static Analysis: 25th International Symposium (SAS 2018)* 2018; 402–422.

[10] Arias J, Olarte C, Penczek W, Petrucci L, Sidoruk T. **Model Checking and Synthesizing for Strategic Timed CTL Using Strategies in Rewriting Logic.** *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming (PPDP 2024)* 2024; 1–14.

[11] Huisman M, Wijs A. **Model-Checking Algorithms.** *Concise Guide to Software Verification: From Model Checking to Annotation Checking* 2023.

[12] Baumeister J, Coenen N, Bonakdarpour B, Finkbeiner B, Sánchez C. **A Temporal Logic for Asynchronous Hyperproperties.** *International Conference on Computer Aided Verification (CAV 2021)* 2021; 694–717.

[13] Krebs A, Meier A, Virtema J, Zimmermann M. **Synchronous Team Semantics for Temporal Logics.** *arXiv preprint* 2024; 1–32.

[14] Alshorman R. **Toward Proving the Correctness of the TCP Protocol Using CTL.** *The International Arab Journal of Information Technology* 2019; **16**(3): 407–414.

[15] Pucella R. **The Finite and the Infinite in Temporal Logic.** *ACM SIGACT News* 2005; **36**(1): 86–99.

[16] Bartocci E, Mateis C, Nesterini E, Nickovic D. **Survey on Mining Signal Temporal Logic Specifications.** *Information and Computation* 2022; **289**: 104957.

[17] Bozzano M, Cavada R, Cimatti A, Dorigatti M, Griggio A, Mariotti A, Micheli A, Mover S, Roveri M, Tonetta S. **nuXmv 2.0.0 User Manual.** *Fondazione Bruno Kessler Technical Report* 2019; 1–192.

[18] Xu N, Ma Z, Jiang J, Zhang P. **Model Checking Instance Based on NuSMV.** *2018 IEEE SmartWorld, Ubiquitous Intelligence and Computing* 2018; 2052–2056.

[19] Alomari A, Alshorman R. **Proving the Correctness Conditions of the Three-Way Handshake Protocol Using Computational Tree Logic.** *Journal of Theoretical and Applied Information Technology* 2021; **99**(15): 3725–3735.