Verifying an Elevator Scheduling Control System

Huan Zhang[1]*, Haoyang Lu[1]†, Long Cheng[2], Hao Wu✉[1]
[1]Computer Science Department, Maynooth University, Maynooth, Ireland
huan.zhang.2024@mumail.ie; haoyang.lu.2024@mumail.ie; ✉haowu@cs.nuim.ie;
[2]School of Control and Computer Engineering, North China Electric Power University, Beijing, China
lcheng@ncepu.edu.cn

***Abstract*—Elevators are crucial for modern buildings, transporting large volumes of passengers across multiple floors every day. Among their components, the scheduling control system plays a central role in determining passenger waiting times and overall efficiency. However, traditional simulation-based testing is time-consuming and costly, which limits its scalability.**

We present an approach that combines both verification and testing for elevator scheduling systems. Our technique models the scheduling logic as a state machine and translates it into Satisfiability Modulo Theories (SMT) formulas that can be efficiently solved and tested by SMT solvers. Preliminary evaluation shows that our technique can verify key properties of scheduling control logic. Our experimental results demonstrate that the approach scales reasonably well with increasing numbers of floors and requests. This can provide a potential practical and automated verification solution for industrial elevator controllers.

***Index Terms*—Formal Verification, Elevator Scheduling, SMT Solver.**

I. Introduction

Elevators are essential in modern infrastructure. They provide reliable vertical transportation in buildings ranging from small residential complexes to high rise skyscrapers. Design flaws in elevator scheduling can cause inefficiencies and long delays. For instance, during the 1999 construction of the Taipei Financial Centre (101-story skyscraper), poor scheduling algorithms led to severe elevator congestion, with passengers waiting more than 10 minutes during peak hours. Such cases show the need for verification techniques to ensure the correctness of elevator scheduling algorithms, which is an important perspective for reassuring the safety of elevator movements.

Traditional approaches to validating elevator scheduling systems rely primarily on simulation-based testing and field trials. These methods can identify design flaws, but they are usually time-consuming and limited. This is because simulation-based approaches cannot explore all possible request sequences and passenger behaviours, and field testing is expensive and time-consuming, often requiring months of deployment before critical issues surface. More importantly, neither approach provides guaranties about the correctness of key properties such as correct request process sequence and deterministic direction switching.

In this paper, we present an approach that combines both verification and testing to check elevator's core scheduling principles. This approach models scheduling control logic as a state machine along with the key properties, and casts it into a set of Satisfiability Modulo Theories (SMT) formulas that can be efficiently solved by SMT solvers. For verification, we examine two core properties: floor reachability (every request is eventually serviced) and direction switching (direction changes are deterministic). To tackle the scalability challenge from verification, we collect a set of test cases (from 20 to 50 floors) that cover a range of real-world scenarios. We then run each test case to ensure that the moving sequence matches the expected behaviours.

We evaluate our approach using the state-of-the-art SMT solvers to understand its feasibility, scalability, and limitations. We also provide the key lessons learned and insights into industrial potential. Further, our approach also shows the potential feasibility of testing extra-functional properties (EFP) through our dedicated tool [1]. In short, our contribution to this paper can be summarized as follows:

1) A scalable state machine. The state machine explicitly captures a set of conditions for scheduling control logic. This state machine can scale up to 50-floor configurations compared to the small to medium scale examples (typically 5 floors) from the literature [2]–[5]. Based on this state machine we conclude 2 key properties that capture the scheduling principles for safety assurance.
2) An automated workflow for verification and testing. We provide a tool-assisted workflow that automatically translates a state-machine specification into SMT formulas that can be checked by mainstream SMT solvers. This workflow can perform (1) bounded checking for elevator's scheduling properties up to 15 floors, and (2) scenario-based testing when the verification does not scale to high-level (50 levels) settings.
3) An additional benchmark for state-of-the-art SMT solvers. We evaluate multiple different encodings for our state machine. These encodings generated by our tool can be used as an additional benchmark for testing capabilities of multiple SMT solvers [1].

II. Background

This section introduces the fundamental concepts of elevator scheduling systems, focusing on scheduling control logic. We first explain key terminology and scheduling principles, followed by an illustrative example that demonstrates scheduling behaviours.

*Huan Zhang is supported by John & Pat Hume Scholarship.
†The author contribute equally to this paper.

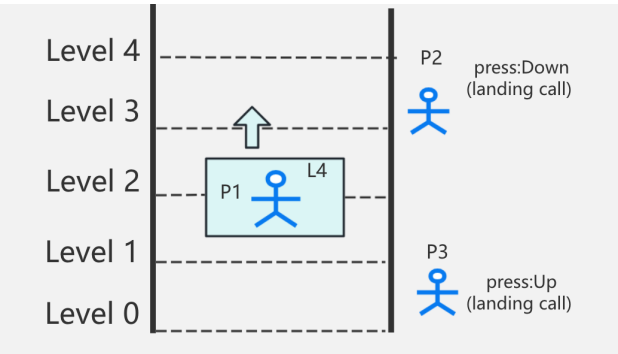

Fig. 1. A scheduling scenario that illustrates direction switching and call prioritisation.

A. The Principles of Scheduling Logic

For any elevator scheduling system, two primary types of requests need to be handled:

- **Landing calls** are requests from levels to call the elevator.
- **Car calls** are requests from inside the elevator to a destination level.

Normally, the scheduling program processes these requests and determines the elevator's movement sequence based on its current position, direction, and active calls. The scheduling algorithm aims to minimise waiting time while maintaining a fair distribution of services [6]. In this paper, we consider the ground floor as Level 0.

The control unit of an elevator determines scheduling through a state-based control system, which follows three key principles:

1) **Direction persistence:** The elevator continues in its current direction as long as there are outstanding landing calls or car calls in that direction.
2) **Direction change conditions:** The elevator changes direction only when no requests remain in its current direction of travel.
3) **Call integration:** Newly issued landing calls and car calls are incorporated into the current service sequence without disrupting ongoing movement.

B. Illustrative Example

To illustrate the core scheduling behaviour, including direction switching and request handling, we present the scenario shown in Fig. 1.

At the beginning, an elevator is approaching to Level 3 from Level 2. There are three active requests:

- A passenger (P1) inside the elevator has pressed the button for Level 4 (a car call).
- A passenger (P2) at Level 3 pressed the down button (a landing call), requesting to go to Level 0 (ground level).
- A passenger (P3) at Level 0 presses the up button (a landing call), requesting to go to Level 2.

The scheduling system processes these requests as follows:

1) Since the elevator is moving upward and the passenger (P2) indicates he/she would like to go downstairs, which is opposite to the elevator's current direction, the elevator skips stopping at Level 3 and continues upward to serve the car call for Level 4. This follows the principle of finishing all requests in the current direction before changing direction.
2) The elevator checks for any remaining requests above Level 4 and finds none. This means it has already travelled to the furthest level in this direction, and there are two remaining requests at lower levels. Hence, it must change its direction after reaching Level 4.
3) The elevator moves downward to Level 3 and picks up passenger P2.
4) The elevator continues downward to the ground floor. Passenger P2 exits, while P3 enters and issues a new car call to Level 2. The elevator then changes direction from downward to upward and transports P3 to Level 2.

This example illustrates how an elevator scheduling program manages direction changes and request processing. The elevator completes all calls in its current direction before changing, and only changes direction when no further calls exist, consistent with common practice.

TABLE I
ALL POSSIBLE STATES THAT ARE USED IN OUR STATE MACHINE CONSTRUCTION.

State	Description
Decide	The controller decides what is the next action for the elevator.
Idle	The elevator is set to the idle mode.
MoveUp	The elevator is moving up.
MoveDown	The elevator is moving down.
SetMotionUp	Set elevator's direction to move up.
SetMotionDown	Set elevator's direction to move down.
Level_i	Represents the i-th level in a building, i ranges from 0 to $n-1$. $i=0$ represents the ground level while $i=n-1$ denotes the top level of an n-story building.

III. Modelling Scheduling Control Program

To model the scheduling logic behind the elevator, we first build a state machine, and this state machine captures the necessary conditions for reaching a specific floor and direction switching. We then translate this state machine into a graph-based specification that can be verified by using SMT solvers.

The main idea of this state machine is to model a controller that is able to decide what is the next action for an elevator based on a set of conditions. The controller issues a set of commands that can move an elevator up/down or change its current direction. This cycled-execution workflow follows the standard of PLC programs [7]. The set of states defined in this state machine is described in Table I.

Given our state machine, an elevator can make a transition from one state to another based on satisfying a set of conditions. We define these conditions as transition conditions. This means that the transitions among different states must obey these conditions. To successfully build our transition

TABLE II
A SET OF STATE VARIABLES WE USED TO BUILD OUR TRANSITION CONDITIONS.

Name	Description	Type
n	The total number of levels in a building.	Integer
f	Current level of the elevator (ranging from 0 to $n-1$).	Integer
m	The elevator's current moving direction: $\uparrow$ (Up), $\downarrow$ (Down), or $-$ (Idle).	Enum
c_i	Whether a car call button of level i is on or not.	Boolean
l_i^u	Whether an "Up" landing call button at level i is on or not.	Boolean
l_i^d	Whether a "Down" landing call button at level i is on or not.	Boolean

conditions, we first define a set of state variables. These variables are summarized in Table II.

For example, we use a set of boolean variables $c_0, \ldots, c_{n-1}$ to encode the call buttons for cars, and each c_i represents a call button at the i-th level where integers of 1 and 0 are used to denote $True$ and $False$, respectively. Hence, $c_i = 1$ indicates that the car call button at the i-th level has been pressed and is otherwise inactive. Using the state variables defined in Table 2, we can now express a transition sequence with changes in the state variables. For example, the following sequence captures the scenario of Figure 1 ($A \rightarrow B$ denotes a transition from state A to B): Initially, the elevator is at level 2 moving up towards level 3. Then requests of passenger P1, P2 and P3 are sequentially handled, and eventually elevator is idled due to no further requests.

$$\underbrace{\text{Decide}(f=2 \wedge m=\uparrow \wedge c_4=1 \wedge l_3^d=1 \wedge l_0^u=1)}_{\text{Process request from P1}} \rightarrow \text{MoveUp}(f=3) \rightarrow \text{MoveUp}(f=4) \rightarrow \underbrace{\text{Level}_4(c_4=0)}_{\text{Finish P1's request}} \rightarrow \text{SetMotionDown}(m=\downarrow) \rightarrow \text{MoveDown}(f=3) \rightarrow \underbrace{\text{Level}_3(l_3^d=0 \wedge c_0=1)}_{\text{Process request from P2}} \rightarrow \text{MoveDown}(f=2) \rightarrow \text{MoveDown}(f=1) \rightarrow \text{MoveDown}(f=0) \rightarrow \underbrace{\text{Level}_0(c_0=0)}_{\text{Finish P2's request}} \rightarrow \text{SetMotionUp}(m=\uparrow) \rightarrow \underbrace{\text{Level}_0(l_0^u=0 \wedge c_2=1)}_{\text{Process request from P3}} \rightarrow \text{MoveUp}(f=1) \rightarrow \text{MoveUp}(f=2) \rightarrow \underbrace{\text{Level}_2(c_2=0)}_{\text{Finish P3's request}} \rightarrow \text{Idle}(m=-)$$

A. Transition Conditions

The controller decides the next action for an elevator, and a specific command issued is typically based on satisfying a set of transition conditions. That is, a transition condition defines a precondition for a state to move to another. For our state machine, we define a total of 6 transition conditions. In this section, we describe our key transition conditions, particularly for direction switching.

Decide $\rightarrow$ **Level**$_i$: This transition indicates that the controller decides whether the elevator has reached[1] a particular level after processing either a landing or car call. The condition for this transition is shown in Condition 1. This condition indicates that the elevator is not in idle mode and has reached the floor ith. Meanwhile, if there exists a car call for level i (c_i) or a landing call (at level i) that matches the elevator's current direction, then the controller decides the elevator has processed this request and stopped at level i.

$$(f=i) \wedge (m \neq -) \wedge (c_i \vee m=\uparrow \wedge l_i^u \vee m=\downarrow \wedge l_i^d) \quad (1)$$

Decide $\rightarrow$ **MoveUp**: This transition indicates the elevator is moving up. In order to ensure this transition is successful, Condition 2 encodes its transition condition. That is, if (1) the elevator's current direction is moving up, (2) there are no landing (up) or car call requests at the current level, and (3) there are some requests from floors l_j ($j > f$), the elevator should keep its current moving up direction.

$$\bigvee_{0 \leq i < n-1} (f=i \wedge m=\uparrow) \wedge \neg(l_i^u \vee c_i) \wedge \bigvee_{i<j<n} c_j \vee l_j^u \vee l_j^d \quad (2)$$

Decide $\rightarrow$ **SetMotionUp**: This transition represents the controller decides to perform a moving-up direction switch for the elevator. Condition 3 states the three conditions to be fulfilled before a moving-up direction switch.

$$\underbrace{(m \neq \uparrow)}_{①} \wedge \bigvee_{0 \leq i < n-1} (f=i) \wedge \underbrace{\left(((m=-) \wedge c_i) \vee l_i^u \vee \bigvee_{i<j<n} l_j^u \vee l_j^d \vee c_j\right)}_{②} \wedge \underbrace{\left((m=\downarrow) \Rightarrow \neg(l_i^d \vee c_i) \wedge \neg \bigvee_{0 \leq x < i} c_x \vee l_x^u \vee l_x^d\right)}_{③} \quad (3)$$

The three conditions are:

- ① The elevator's current direction is not moving-up.
- ② The elevator is currently idle and a car call at level i is issued (c_i), there exists a (up) landing call request at the current level, or there are requests above the current level.
- ③ If the elevator's current direction is moving-down, then no down call or car call exists at current level, and there exists no call of any kind below current level.

These three conditions impose a constraint that the elevator would never switch direction if it were already moving in that direction. The direction switch only happens when all the requests in that direction have been processed, and there are some outstanding requests in the opposite direction.

[1]For simplicity, we omit the process that the elevator opens and closes its door.

Decide $\rightarrow$ **Idle**: This transition indicates that the elevator is going to enter idle mode. This only happens if the elevator is not idle and all the requests have been completed. The condition is shown in Condition 4.

$$(m \neq -) \wedge \neg \bigvee_{0 \leq i < n} l_i^u \vee l_i^d \vee c_i \tag{4}$$

B. Properties

The core of an elevator's scheduling control logic can be captured using two properties: floor reachability and direction switching.

Floor Reachability (FR). This property simply says that once a request (a car call or a landing call) has been made, the elevator will eventually process this request. That is, the elevator will finally reach a destination level based on a particular request.

Direction Switch (DS). The controller must deterministically issue a command, and this includes direction switch. In other words, the elevator should never have an option of moving-up or moving-down at the same time if it is not idled. The process of determining an elevator's direction must be deterministic.

To be able to verify these two properties, we first define the configuration of an elevator. A configuration captures the *state* of an elevator and each transition in our state machine yields a new configuration. Formally, a configuration is defined as: $C = \langle L, T, F, M \rangle$, where L is a set of active landing call requests, T a set of active car call requests, F denotes elevator's current level and M denotes elevator's current moving direction. For example, the configuration $\langle \{l_1^u, l_3^d\}, \{c_2\}, 0, - \rangle$ represents that the elevator is currently idle[2] and at ground level, and there are 3 active requests (2 landing calls and 1 car call). Hence, verifying properties can now be considered as checking if an elevator can reach one or more configurations from its initial configuration. However, this is challenging as in the real-world scenario each passenger could make a request, and a new configuration is yielded each time a request is completed. Thus, we categorize configurations into *fixed* and *dynamic* configurations and verify the two properties, where:

- A *fixed* configuration represents that an elevator starts with an initial configuration and no other requests can be made during the process. Hence, we can check if the scheduling logic will yield an expected configuration.
- A *dynamic* configuration represents that an elevator starts with an initial configuration and any possible requests can be made during the process. This includes landing calls or car calls at any possible level. Thus, verifying properties using dynamic configuration is much more challenging.

To tackle two properties defined (FR and DS) under both *fixed* and *dynamic* configuration, we encode our properties as follows:

[2]We use $\uparrow$ to denote moving-up, $\downarrow$ to denote moving-down and $-$ to denote idle.

- **(FR 1)** Condition 5 states that at any time when the elevator enters idle mode at any level i, all requests must be already processed. We use a set of auxiliary variables (a_i^u, a_i^d, a_i^c) to record whether a corresponding request (landing call or car call) at level i has been processed. If that is the case, then it is safe to say no requests for level i are outstanding. This is denoted by using another auxiliary variable h_i.

$$\begin{aligned}&(state = \text{Idle}) \Rightarrow \\ &\bigwedge_{0 \leq i < n} \neg(c_i \vee l_i^d \vee l_i^u) \wedge ((a_i^u \vee a_i^d \vee a_i^c) \Leftrightarrow h_i)\end{aligned} \tag{5}$$

- **(FR 2)** Similarly, when the elevator reaches level i, and this means that the requests that match elevator's current direction at level i must already be processed. This is captured by using Condition 6.

$$\begin{aligned}&(state = \text{Level}_i) \Rightarrow \\ &\begin{pmatrix}\neg c_i \wedge (m = \uparrow) \wedge \neg l_i^u \wedge (a_i^u \vee a_i^c) \\ \vee (m = \downarrow) \wedge \neg l_i^d \wedge (a_i^d \vee a_i^c)\end{pmatrix}\end{aligned} \tag{6}$$

- **(DS 1)** Direction switch considers two cases: (1) Switching from moving-up to moving-down. (2) Switching from moving-down to moving-up. We use two variables f_{sw}^u and f_{sw}^d to record the level where the elevator switches to moving-up and moving-down direction, respectively. For both cases, $f_{sw}^u \leq f_{sw}^d$ always holds. To see why, consider case (1), elevator was moving-up from an initial level, then at the level when the elevator started to change direction to down (f_{sw}^d) must be greater or equal to its initial level of moving up (f_{sw}^u). Case (2) is the same. Since the elevator is moving down, the level where the elevator starts to switch to moving-up (f_{sw}^u) must be less than or equal to its initial moving-down level (f_{sw}^d). Hence, Condition 7 ensures this relation hold and meanwhile makes sure the elevator's current direction is different from its previous moving direction (m_{prev}) after the elevator switching its state (SetMotionUp, SetMotionDown and Idle).

$$\begin{aligned}&state \in \{\text{SetMotionUp}, \text{SetMotionDown}, \text{Idle}\} \\ &\wedge f_{sw}^u \leq f_{sw}^d \Rightarrow m \neq m_{prev}\end{aligned} \tag{7}$$

- **(DS 2)** When the elevator is moving down, the current level must be less than or equal to the level elevator starts to switch to moving-down direction. Because the elevator could accept new requests in the moving process (dynamic configurations), we use a set of variables s_i^d and s_i^c to record all the down-landing and car car calls (between each level downward and the level where the elevator starts to switch to move downward) before current direction switch and make sure no outstanding requests. The same principle applies to the scenario when the elevator switches in a moving-up direction. This property is captured by Condition 8.

$$
\begin{aligned}
(m = \downarrow) &\Rightarrow f \leq f^d_{sw} \wedge \neg \bigvee_{f < i \leq f^d_{sw}} s^d_i \vee s^c_i \\
(m = \uparrow) &\Rightarrow f \geq f^u_{sw} \wedge \neg \bigvee_{f^u_{sw} \leq j < f} s^u_j \vee s^c_j
\end{aligned} \tag{8}
$$

- **(DS 3)** When the elevator is moving downward, there must exist some active requests (down landing calls or car calls require the elevator to move downward) from the levels below or equal to current level. This means that the elevator must keep moving down in order to process these requests. This is the same when elevator is moving upward. Condition 9 captures both cases.

$$
\begin{aligned}
&(state \in \{\text{SetMotionDown}, \text{MoveDown}\}) \Rightarrow \\
&(m = \downarrow) \wedge \bigvee_{0 \leq i \leq f} l^d_i \vee l^u_i \vee c_i \\
&(state \in \{\text{SetMotionUp}, \text{MoveUp}\}) \Rightarrow \\
&(m = \uparrow) \wedge \bigvee_{f \leq j < n} l^d_j \vee l^u_j \vee c_j
\end{aligned} \tag{9}
$$

IV. Evaluation

In order to effectively evaluate our approach, we perform two types of evaluations. (1) We first perform verification evaluation to show the correctness (counter-example discovery) of our model by proving defined properties. (2) To tackle the limitation of verification, we also perform a series of testing on a set of industrial-scale test cases. These test cases describe real-world scenarios that show how an elevator is operated in rush hours in a high-level building. All evaluations are performed on a Linux PC with Intel i9-13950HX (2.20 GHz) CPU, 24GB RAM, 8GB SWAP and a 2-hour time-out setting.

In our evaluation, we use the verification tool *Cyclone* [1] to model our state machine and translate our models into Satisfiability Modulo Theories (SMT) formulas [8]. *Cyclone* uses a graph representation for verification tasks [1], [9], translating a state machine into a specification is straightforward. For example, Listing 1 shows the portion of a Cyclone specification that describes Condition 1 for level 2 (Line 7-8). The invariant (Line 13-15) describes Property FR2. All our Cyclone specifications are available in our repository[3]. In fact, we have implemented two versions of Cyclone (Cyclone$_J$ and Cyclone$_R$) to evaluate different encodings. Cyclone$_J$ is a Java implementation that uses integers and booleans to capture the search space, while Cyclone$_R$ is a Rust implementation that unrolls a specification into bit-vectors and integers.

Listing 1. A piece of sample Cyclone code shows the declarations of n, f (Table II), the transition conditions for Decide $\rightarrow$ Level$_2$ (Condition 1) and the unconditional transition back to the node Decide. Condition 6 (FR2) for a 3-level building is represented as an invariant. If a specification contains at least one invariant, Cyclone will swith to counter-example discovery mode.

```
1  const int n = 3;
2  int f where f >= 0 && f < n;
3
4  // ... other variables and states ...
5
6  edge { Decide -> Level2 where
7      f == 2 && m != #NA
8      && (c2 || m == #UP && l2_u || m == #DOWN &&
           l2_d);}
9  edge { Level2 -> Decide }
10
11 // ... other transition conditions ...
12
13 invariant FloorReachabilityLevel2 {
14   !c2 && (m == #UP && !l2_u && (a2_u || a2_c)
15   || m == #DOWN && !l2_d && (a2_d || a2_c)); } in
         (Level2)
```

[3]URL: https://github.com/lucid-brndmg/elevator-cyclone-smt

A. Property Verification

For property verification, we conduct two experiments: one for fixed configurations and one for dynamic configurations. Table III and Table IV summarize the results, respectively. In both experiments, we use n to denote the number of levels. We first ask Cyclone (both versions) to generate a set of SMT formulas and then verify them (finding counter-examples) using a list of 7 mainstream SMT solvers: z3 (4.15.2) [10], cvc5 (1.3.1) [11], yices2 (2.7) [12], opensmt (2.9.2) [13], mathsat (5.6.14) [14], stp (2.3.4) [15] and bitwuzla (0.8.2) [16]. The main reason we select these solvers is based on their rankings from SMTCOMP [17].

For each fixed configuration, we calculate a maximum elevator operation sequence, represent it into a Cyclone specification and verify it along with defined properties using mainstream SMT solvers. It can be seen that when $n \leq 10$, most of the mainstream SMT solvers can successfully prove our defined properties (defined in Section III-B) by finding no counter-example. However, when $n \geq 15$ (Table III) most of the solvers are unable to finish verification and time out. The properties typically grow exponentially when n increases. This makes most of the solvers time out. However, we find that bit-vector based encoding generally outperforms integer and boolean based encoding. In particular, solvers such as bitwuzla and stp that specialize in bit-vector solving can quickly determine whether there is a counter-example for our defined properties or not.

For each dynamic configuration for a particular n-level, we perform verification on all properties but with an increasing bound to ensure termination. The reason is that dynamic configuration is much more challenging as there are "infinite" possible actions a passenger could do. Hence, to ensure some level of correctness, we check the combinations of any 2, 4, 8 and 16 consecutive elevator operations. For example, the transition sequence of

$$\text{Decide} \rightarrow \text{MoveUp} \rightarrow \text{Decide} \rightarrow \text{Level}_1 \rightarrow \text{Decide}$$

represents a combination of 2 operations (MoveUp and Level$_1$), where new calls might be issued arbitrarily within each Decide state. The solvers check for all possible passenger actions and look for a property violation within these bounds. We set up these bounds based on the *small scope hypothesis*, which states that a large proportion of incorrectness can be

discovered within a small state space [18]. Here, we assume any property violation can be discovered within a sequence of maximum 16 operations. However, we acknowledge the fact that this might not guarantee the absence of bugs inside deep edge-cases. The results show that the performance of each solver varies by encoding. Again, bit-vector based encoding outperforms integer and boolean based encoding. It is not surprising that specialized solvers can handle a more complex configuration than generalized solvers.

B. Scenario Testing

Although our verification technique can ensure the properties hold for $n < 15$, the scalability issue ($n \geq 15$) imposes a significant challenge for the state-of-the-art SMT solvers. To raise confidence in the high-level settings ($n \geq 15$), we collect a set of industrial scenarios (Table V) and test each of them under fixed configurations. In fact, we contacted our industrial partner Siemens about these scenarios. Though these scenarios do not cover full cases, technicians at Siemens suggest that they at least serve as a baseline and can be easily extended for testing more advanced scenarios. Under this context, we select these scenarios to cover a variety of scheduling settings such as idle transitions, high-level call simulations and unserviceable request directions.

Each scenario begins with an initial configuration, and we then test it to check if it is possible to yield a valid configuration. This means that a moving sequence strictly matches with the behaviours defined by the scheduling control logic. For example, scenario S2 expects the elevator to first move downwards to handle the car call of the 5-th floor based on its initial down-direction and then switches to up-direction for handling the 18-th floor, although it is closer to the initial (19-th) floor.

Table VI summarizes the testing results. Similar to the verification, we use two versions of Cyclone to generate SMT formulas then feed them to the mainstream SMT solvers. Because checking/testing whether an initial configuration can yield a valid moving sequence (under fixed configurations) is much less challenging, all mainstream SMT solvers are able to finish the checking within minutes. However, the compiling time from Cyclone to SMT formulas takes much longer than we expected. This is because the compilers must unroll all possible moving sequences from an initial configuration. One possible optimization is to slice the unroll process into multiple ones. The configuration yielded from previous unroll can be used as an initial configuration for the next one.

C. Comparison

We compare our technique with nine representative approaches from the literature (Table VII), evaluating each along four dimensions: model size, formalism, level of automation, and industrial applicability. We select these approaches because they mainly focus on elevator scheduling control. Most of the approaches can only handle a small number of floors, whereas our technique combining both verification and testing can handle up to 50 floors. Compared to Petri-Net and alternative FSM-based techniques [3], [19], [20], our verification workflow is fully automated and shows strong potential for adoption by our industry partner. In addition, our cycle-based modelling follows the principle of PLC programs. Therefore, our technique can be potentially integrated with PLC-based development workflows. The approach can also be extended to include real-time constraints and delay control by introducing additional variables into the state machine, which would further increase its attractiveness to our industry partner.

D. Lessons Learned

Throughout the development of this approach in collaboration with our industry partner, we identified three key lessons:

- **Modelling.** To correctly model real-world elevator scheduling control logic is much more challenging than we expected. In fact, we have built many models in the past months and none of them really capture the principles of scheduling. Unfortunately, most of the prior work only use small to medium scale examples [2]–[5] and we could not learn much from the literature. Hence, we believe our model not only supplies safety assurance for elevator controllers but also facilitates testing important quality characteristics and EFPs such as energy efficiency: with our dedicated tool, one can generate test cases from our state machines to facilitate such testing.
- **Tool Assistance.** Tool assistance is a key in building formal models. It is impossible to manually encode a model into a formalism such as SMT formulas even for the simplest scenario. Fortunately, *Cyclone* provides an easy way to construct state machines and generate SMT formulas. Hence, it is essential to have access to automated tools and at least the tool itself should allow engineers to easily build a model without a steep learning curve. Many of them are already overloaded with their daily work. Other than usability, Cyclone also shows feasibility in modelling complex systems such as hybrid systems [9], making it possible to incorporate continuous movements and real-time constraints into the scheduling logics for stronger safety assurances.
- **Verification & Testing.** Certainly, verification ensures the highest level of confidence in a design. However, the scalability often prevents verification technique from being deployed for handling large-scale settings. Hence, combining testing and verification is a necessary step to build confidence in a complex design. In our case, it suggests a practical workflow. That is using bounded verification to establish correctness for small-to-medium sized models and introducing the testing scenarios for larger sized models. This workflow could be potentially adapted by elevator manufacturers.
- **Encodings & Solvers.** The selection of encodings and solvers has impact on the verification performance that cannot be neglected. In our modelling, the integer variables of Table II either have a fixed value (e.g., n) or

TABLE III

THE VERIFICATION TIME SPENT ON PROVING PROPERTIES UNDER FIXED CONFIGURATIONS. ALL DENOTES ALL PROPERTIES DESCRIBED IN SECTION III-B. THE UNIT HERE IS SECOND. TO DENOTES TIME OUT.

	$n=3$			$n=5$			$n=10$			$n=15$		
	FR	**DS**	**ALL**	**FR**	**DS**	**ALL**	**FR**	**DS**	**ALL**	**FR**	**DS**	**ALL**
Solver	**Cyclone$_J$ (integer and boolean encoding)**											
cvc5	0.55	0.95	1.01	11	17	30	5208	1977	TO	TO	TO	TO
mathsat	0.23	0.58	0.51	15	16	25	1584	3031	5006	TO	TO	TO
opensmt	0.44	0.71	0.81	12	14	24	5539	2086	TO	TO	TO	TO
yices2	0.34	1	0.77	12	8.13	32	TO	3299	TO	TO	TO	TO
z3	1.43	2.67	3.55	37	36	47	2309	2063	4834	TO	TO	TO
Cyclone$_R$ (integer encoding)												
cvc5	0.45	0.75	0.96	8.91	15	21	TO	1861	TO	TO	TO	TO
mathsat	0.2	0.48	0.41	14	10	23	3284	1826	4437	TO	TO	TO
opensmt	0.39	0.56	0.56	10	11	19	TO	1439	TO	TO	TO	TO
yices2	0.29	0.82	0.79	10	10	19	TO	5597	TO	TO	TO	TO
z3	0.27	0.45	0.45	11	12	21	1732	1252	5223	TO	TO	TO
Cyclone$_R$ (bit-vector encoding)												
bitwuzla	0.44	0.52	0.59	3	6.04	7.49	57	288	344	433	3484	4790
stp	0.34	0.25	0.44	3.44	4.42	5.44	113	226	366	703	2124	3300
yices2	0.28	0.41	0.39	11	9.57	26	TO	1237	TO	TO	4553	TO
z3	0.18	0.23	0.24	4.32	6.39	12	105	652	1139	334	TO	TO

TABLE IV

THE VERIFICATION TIME SPENT FOR PROVING ALL PROPERTIES UNDER DYNAMIC CONFIGURATIONS WITH BOUND INCREMENT. THE TIME UNIT HERE IS SECOND. TO DENOTES TIME OUT.

	$n=3$				$n=5$				$n=10$				$n=15$			
Solver	**Cyclone$_J$ (integer and boolean encoding)**															
cvc5	0.09	0.3	3.42	58	0.2	0.78	19	944	0.76	3.14	134	TO	1.78	8.38	370	TO
mathsat	0.03	0.15	3.06	72	0.08	0.43	12	2486	0.33	2.42	144	TO	0.84	8.28	504	TO
opensmt	0.04	0.27	6.1	78	0.1	0.78	21	612	0.49	4.44	216	TO	1.3	10	596	TO
yices2	0.02	0.14	3.32	135	0.06	0.78	32	TO	0.17	1.91	395	TO	0.41	5.44	999	TO
z3	0.09	0.55	25	86	0.16	1.86	49	896	0.59	18	232	TO	1.22	30	648	TO
Cyclone$_R$ (integer encoding)																
cvc5	0.11	0.27	3.19	96	0.14	0.71	15	1712	0.43	2.56	123	TO	0.89	5.84	286	TO
mathsat	0.02	0.1	2.18	81	0.04	0.32	10	6169	0.18	2.3	118	TO	0.55	5.39	393	TO
opensmt	0.03	0.22	3.93	101	0.08	0.63	17	1590	0.34	3.69	159	TO	0.76	7.94	418	TO
yices2	0.01	0.09	3.69	278	0.04	0.43	29	TO	0.09	1.33	282	TO	0.2	3.02	808	TO
z3	0.05	0.15	2.76	52	0.09	0.46	18	1362	0.32	2.4	159	TO	0.54	6.32	519	TO
Cyclone$_R$ (bit-vector encoding)																
bitwuzla	0.1	0.36	2.08	13	0.18	1.12	6.75	116	0.71	3.36	44	1160	1.17	7.53	118	3954
stp	0.05	0.2	1.69	13	0.1	0.68	6.07	94	0.44	2.4	42	1015	1.84	9.04	129	4439
yices2	0.01	0.09	2.36	51	0.02	0.23	12	4156	0.08	1.36	159	TO	0.19	3.15	500	TO
z3	0.04	0.14	2.14	74	0.1	0.4	10	533	0.39	2.16	132	TO	0.73	4.35	445	TO

an upper bound (e.g., $0 \leq f < n$). Using 2 implementations of Cyclone, we generate SMT formulas of different encodings from each specification: the integer encodings map integers into the SMT Int type, while the bit-vector encoding maps them into small, fixed-size bit-vectors[4]. Our results show that the solvers struggle figuring out the upper bounds of Int variables through the imposed constraints (e.g., Line 2 of Listing 1), as the Int type has an unlimited range by definition [8]. Replacing Int with bit-vectors generally improves solving efficiency, and specialized bit-vector solvers (stp and bitwuzla) provide further improvements. This is because (1) bit-vectors restrict their search range at the type level, and (2) our conditions involve only Linear Integer Arithmetic (LIA) operators such as addition and equality that are less difficult for solving. We believe this generalizes to similar LIA-based modelling of other domains but may not apply to non-linear constraints and real variables. However, neither encoding nor solvers can stop the exponential increment of verification time caused by the growth of state-space. Unfortunately, we use these SMT solvers as *black boxes*. The sophisticated designs behind their decision procedures obstruct us from giving deeper diagnoses of their performance. Nevertheless, our experimental results do provide referential meanings for future studies.

- **Limitations & Future Directions.** Our approach focuses on the scheduling logic behind the elevator. For simplicity, our modelling heavily abstracts the presence of physical movements, weight limits, timings, or door dynamics. In real-world elevators, these crucial factors can lead to various kinds of unsafety. Based on our experiments, we believe modelling these factors would make

[4]The sizes of bit-vectors are computed based on n (see our repository).

TABLE V

A COLLECTION OF SCENARIOS (COVERING UP TO 50 LEVELS) THAT CAPTURE MULTIPLE SETTINGS INCLUDE IDLE TRANSITIONS, HIGH-LEVEL CALL SIMULATIONS AND UNSERVICEABLE REQUEST DIRECTION. THE INITIAL CONFIGURATION DESCRIBES THE INITIAL STATE OF AN ELEVATOR.

Scenario	**Levels** (n)	**Initial Configuration**
S1	20	$\langle\{l^u_{18}\},\{c_{10},c_{15},c_{18}\},12,\uparrow\rangle$
S2	20	$\langle\{l^u_{18}\},\{c_5\},19,\downarrow\rangle$
S3	30	$\langle\{l^u_9\dots l^u_{28},l^d_{10}\dots l^d_{29}\},\{c_{10}\dots c_{29}\},9,\uparrow\rangle$
S4	30	$\langle\{l^u_{28},l^d_{25}\},\{\},26,-\rangle$
S5	30	$\langle\{l^d_{11}\},\{c_{10},c_{20},c_{25}\},22,\downarrow\rangle$
S6	35	$\langle\{l^u_8\dots l^u_{15}\},\{c_0\dots c_{11}\},0,\uparrow\rangle$
S7	40	$\langle\{l^u_{25}\dots l^u_{29},l^d_{28},l^d_{32},l^d_{35}\},\{c_{21},c_{27},c_{28},c_{30},c_{33}\},24,\downarrow\rangle$
S8	45	$\langle\{l^u_{35}\dots l^u_{39},l^d_{34}\dots l^d_{39}\},\{c_{44}\},42,\uparrow\rangle$
S9	50	$\langle\{l^u_{37},l^u_{38},l^u_{39},l^d_{36}\dots l^d_{40}\},\{c_{33}\dots c_{40}\},35,\downarrow\rangle$
S10	50	$\langle\{l^d_{38}\dots l^d_{44}\},\{c_{49}\},42,\uparrow\rangle$

TABLE VI

THE TIME SPENT ON TESTING EACH SCENARIO IN TABLE V UNDER FIXED CONFIGURATIONS. THE TIME UNIT HERE IS SECOND.

Solver	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
Cyclone$_J$ (integer and boolean encoding)										
cvc5	5.54	9.46	69	7	25	36	60	70	85	77
mathsat	2.53	4.68	35	3	13	19	34	37	46	42
opensmt	4.34	8.26	97	5.77	23	33	57	76	91	84
yices2	1.14	1.89	16	1.47	6.03	8.64	14	15	21	20
z3	0.72	1.04	5.2	0.62	1.99	2.44	3.34	3.68	3.96	3.93
Cyclone$_R$ (integer encoding)										
cvc5	1.39	2.18	15	1.34	5.59	6.93	12	13	15	15
mathsat	0.4	0.87	14	0.35	2.09	2.59	5.19	4.81	4.89	4.48
opensmt	0.49	0.88	6.61	0.77	2.26	2.66	4.24	4.34	4.68	4.64
yices2	0.26	0.32	1.83	0.16	0.59	0.82	1.36	1.08	1.2	1.51
z3	0.61	1.07	6.02	0.44	2.11	2.88	4.42	3.98	4.14	3.97
Cyclone$_R$ (bit-vector encoding)										
bitwuzla	1.81	2.94	19	1.28	6.33	9.02	13	14	15	15
stp	0.9	1.63	11	1.56	4.2	5.43	8.84	9.42	10	9.76
yices2	0.19	0.3	1.57	0.15	0.59	0.79	1.18	1.17	1.25	1.18
z3	0.47	1.39	4.92	0.36	3	3.52	3.69	3.67	3.52	3.31

verification more time-consuming and non-termination in extreme cases. In the future, we intend to address scaling issues by slicing our model into different smaller ones (concurrent verification).

These insights highlight both the practicality and the challenges of applying formal methods in real elevator systems. Overall, these lessons highlight the practical considerations that must be addressed when transferring formal verification from academic examples to industrial-scale elevator systems.

V. RELATED WORK

Extensive research has been conducted in both elevator scheduling and formal verification, each addressing critical aspects of control and correctness [6], [25]–[27]. We briefly review existing work in three main areas.

AI-Based Scheduling. AI-driven methods, such as reinforcement learning and deep learning, have shown improvements in average waiting time and dispatching efficiency [6], [28]. These approaches adapt to dynamic passenger demand and can optimize energy consumption. However, they depend heavily on training data, struggle with rare cases, and lack interpretability and correctness guarantees, which make them less suitable for safety-critical systems such as elevators.

Formal Methods. Formal verification is essential for ensuring the safety and correctness of elevator control [25], [29]. Promela/SPIN, SMV, and UPPAAL have been applied to elevator control systems in previous studies [5], [23], [24]. In contrast, our approach directly targets industrial-scale systems, automatically handling up to 50 floors, and shows strong potential for integration into real-world development pipelines.

Optimisation Strategies. Other optimisation methods, including constraint programming, real-time scheduling, and heuristic approaches such as genetic algorithms and particle swarm optimisation, have been used to reduce wait times and energy consumption [30], [31]. While effective in improving efficiency, these techniques cannot guarantee correctness and often incur significant computational costs in large-scale settings. Our work complements these approaches by focusing on scalable, automated formal verification.

In summary, while AI and optimisation methods improve performance, they lack strong guarantees of correctness. Prior formal methods ensure correctness but are limited to small-scale examples. Our approach directly models a real-world elevator scheduling system widely deployed in modern buildings, making it a promising candidate for industrial adoption by elevator manufacturers.

VI. CONCLUSION

We presented an SMT-based verification and testing approach for elevator scheduling systems. This approach allows

TABLE VII
COMPARISON OF VERIFICATION METHODS FOR ELEVATOR SCHEDULING.
NOTE: DEVS (DISCRETE EVENT SYSTEM SPECIFICATION) IS A FORMAL MODELLING FRAMEWORK FOR DISCRETE-EVENT DYNAMIC SYSTEMS.

Method	Model Size	Formalism	Automation	Industrial Potential
Cyclone + SMT (Our technique)	15 to 50 floors	Finite State Machine (FSM)	Fully automated	High (deployable)
DFA [3]	3 floors	Finite State Machine	Semi-automated	Low
Petri Nets [19]	4 floors	Transition Nets	Manual analysis	Low
UPPAAL+DEVS [5]	3 floors	DEVS+Timed Automata	Semi-automated	Medium (formal logic)
CPN [21]	20 floors	Coloured Petri Nets Programming	Semi-automated	Medium (formal logic)
GNP [22]	45 floors	Genetic Network Programming	Learning-based	High (real-world)
SMV (TWIN elevator) [23]	11 floors	Temporal Logic	Manual model	High (real-world)
COCOLOG [20]	5 floors	Finite State Machine	Manual model	Medium (PLC logic)
Event-B [4]	3 floors	Event-B	Semi-automated	Medium (early-stage)
SPIN+UPPAAL +NuSMV [24]	8 floors	Temporal Logic	Semi-automated	Medium (prototyping)

us to combine formal verification (proving the correctness of the defined properties) with scenario-based testing (validating the correct moving behaviours) to tackle unexpectedly complex scheduling control logic for an elevator. This work demonstrates the feasibility of integrating formal methods into real-world scheduling systems and paves the way toward verifying/testing complex multi-elevator settings and extra-functional properties.

REFERENCES

[1] H. Wu, F. Thomas, and M. Dominique, "Cyclone: A new tool for verifying/testing graph-based structures," in *18th International Conference on Tests and Proofs*. Springer, 2024.

[2] W. Fokkink, A. Kakebeen, and J. Pang, "Adapting the uppaal model of a distributed lift system," in *International Conference on Fundamentals of Software Engineering*. Springer, 2007, pp. 81–97.

[3] A. Sivakumar, G. B. Reddy, Y. Tanvi, S. Sidhu, and R. D, "A collaborative elevator system optimization: A finite automata approach," in *2024 5th IEEE Global Conference for Advancement in Technology (GCAT)*, 2024, pp. 1–6.

[4] J. yi Yao, S. rong Zou, and X. Geng, "Formal modelling and visualization of elevator system based on event-b," in *2022 IEEE 2nd International Conference on Computer Systems (ICCS)*. IEEE, 2022, pp. 123–130.

[5] H. Saadawi and G. Wainer, "Verification of real-time devs models," in *Proceedings of the 2009 Spring Simulation Multiconference*. Society for Computer Simulation International, 2009.

[6] J. Zhang and Q. Zong, "Energy-saving scheduling optimization under up-peak traffic for group elevator system in building," *Energy and Buildings*, vol. 66, pp. 495–504, 2013.

[7] K.-H. John and M. Tiegelkamp, *IEC 61131–3: Programming industrial automation systems*, 2001st ed. Berlin, Germany: Springer, Jun. 2013.

[8] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[9] H. Wu and Z. Cheng, "Verifying Event-B hybrid models using Cyclone," in *Proc. Int. Conf. Rigorous State-Based Methods*. Springer, 2023, pp. 179–184.

[10] L. D. Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[11] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, "cvc5: A versatile and industrial-strength smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 415–442.

[12] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, July 2014, pp. 737–744.

[13] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, "Opensmt2: An smt solver for multi-core and cloud computing," 2016.

[14] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *Proceedings of TACAS*, ser. LNCS, N. Piterman and S. Smolka, Eds., vol. 7795. Springer, 2013.

[15] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 519–531.

[16] A. Niemetz and M. Preiner, "Bitwuzla," in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, ser. Lecture Notes in Computer Science, C. Enea and A. Lal, Eds., vol. 13965. Springer, 2023, pp. 3–17.

[17] T. Weber, S. Conchon, D. Déharbe, M. Heizmann, A. Niemetz, and G. Reger, "The SMT competition 2015-2018," *J. Satisf. Boolean Model. Comput.*, vol. 11, no. 1, pp. 221–259, 2019.

[18] D. Jackson and C. A. Damon, "Elements of style: Analyzing a software design feature with a counterexample detector," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 239–249, 1996.

[19] F. Ahmad, I. Fakhir, S. A. Khan, and Y. D. Khan, "Petri net-based modeling and control of the multi-elevator systems," *Neural Computing and Applications*, vol. 24, no. 7, pp. 1601–1612, 2014.

[20] D. N. Dyck and P. E. Caines, "The logical control of an elevator," *IEEE transactions on automatic control*, vol. 40, no. 3, pp. 480–486, 2002.

[21] M. Assiri, "Modeling elevator system with coloured petri nets," M.A.Sc. Thesis, McMaster University, 2015.

[22] L. Yu, S. Mabu, and K. Hirasawa, "Multi-car elevator system using genetic network programming for high-rise building," in *2010 IEEE International Conference on Systems, Man and Cybernetics*. IEEE, 2010, pp. 1216–1222.

[23] F. Kammüller and S. Preibusch, "An industrial application of symbolic model checking," *Computer Science - Research and Development*, vol. 22, pp. 95–108, 02 2008.

[24] Z. Qian, X. Li, and X. Wang, "Modeling distributed real-time elevator system by three model checkers," *International Journal of Online Engineering (iJOE)*, vol. 14, no. 4, pp. 94–110, 2018.

[25] A. Anta, R. Majumdar, I. Saha, and P. Tabuada, "Automatic verification of control system implementations," in *Proceedings of the tenth ACM international conference on Embedded software*, 2010, pp. 9–18.

[26] R. Crites and A. Barto, "Improving elevator performance using reinforcement learning," *Advances in neural information processing systems*, vol. 8, 1995.

[27] A. N. Z. Rashed, M. Yarrarapu, R. T. Prabu, G. S. Raj Antony, L. Edeswaran, E. S. Kumar, K. Aswitha, N. Snehith, and S. H. Ahammad, "Connected smart elevator systems for smart power and time saving," *Scientific Reports*, vol. 14, no. 1, p. 19330, 2024.

[28] A. Evangelidis, N. Dimitriou, P. Charalampous, T. D. Mastos, and D. Tzovaras, "Efficient deep q-learning for industrial equipment calibration in elevator manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 20, no. 10, pp. 12 220–12 230, 2024.

[29] D. Basile, F. Mazzanti, and A. Ferrari, "Experimenting with formal verification and model-based development in railways: the case of umc and sparx enterprise architect," in *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 2023, pp. 1–21.

[30] P. Friese and J. Rambau, "Online-optimization of multi-elevator transport systems with reoptimization algorithms based on set-partitioning models," *Discrete Applied Mathematics*, vol. 154, no. 13, pp. 1908–1931, 2006.

[31] S. Ramalingam, A. Raghunathan, and D. Nikovski, "Submodular function maximization for group elevator scheduling," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 27, 2017, pp. 233–241.