https://doi.org/10.2738/NET.2026.0003

ORIGINAL ARTICLE

Formal verification of the correctness of elevator control systems based on the Isabelle/HOL theorem prover

Heng ZHANG[1,*], Long CAO[2,*]✉

1. Binjiang Vocational and Technical School of Changshu City, Suzhou, 215500, China
2. School of Computer Science and Engineering, Suzhou University of Technology, Suzhou, 215500, China

Received December 27, 2025; accepted February 4, 2026
E-mail: 2111468290@qq.com

* These authors contributed equally to this work.
© Higher Education Press 2026

Abstract

In the context of smart manufacturing and modern production systems, elevators function as indispensable equipment for vertical logistics and personnel transportation, serving as the vertical arteries connecting workshops, warehouses, and different floors. The reliability and intelligence of their control technology directly determine the efficiency of material and personnel flow, the safety of industrial operations, and the flexibility of production adaptation—key factors that underpin the core goals of smart manufacturing: efficiency, safety, flexibility, and sustainability. With the increasing integration of elevators into intelligent logistics networks and collaborative production systems, any failure or inaccuracy in their control logic may lead to production disruptions, safety hazards, or resource waste. Therefore, verifying the correctness of elevator control systems has become an imperative and critical issue in the field of smart manufacturing. This paper primarily adopts the refinement method, a rigorous formal technique, to construct precise formal models that encapsulate elevator states, passenger requests, and operational rules. Leveraging formal methods, the study conducts a systematic verification of three core properties of the elevator control system: safety, liveness, and fairness. The research findings demonstrate that the Isabelle/HOL theorem prover, a powerful tool for interactive theorem proving, can effectively validate the correctness of the constructed formal models and rigorously verify the aforementioned key properties. This work not only provides a robust theoretical guarantee for enhancing the reliability and safety of elevator control systems in industrial scenarios but also offers a replicable formal verification framework for the design and optimization of similar safety-critical control systems in smart manufacturing.

Keywords

Elevator Control Systems; Correctness Verification; Formal Verification; Isabelle/HOL; Theorem Prover

1 Introduction

Elevators are indispensable vertical transportation tools in modern buildings, widely used in various locations such as commercial buildings, residential complexes, hospitals, and schools, bringing significant convenience to people's daily commuting, work, and lives. Whether in high-rise-dense urban centers or developing emerging areas, elevators play a crucial role. However, against the backdrop of intelligent manufacturing and production, locations like production workshops and logistics centers have extremely high requirements for elevator operational efficiency. This necessitates that elevator control systems, through intelligent scheduling algorithms, quickly and reasonably assign elevator tasks based on call requests from different floors and the elevators' current positions. For instance, in large multi-floor smart factories, elevator control systems can simultaneously handle the operation and scheduling of multiple elevators, avoiding elevator idling and long waiting times. This ensures that the personnel, equipment, and materials can be rapidly transported between floors, significantly improving elevator utilization efficiency and reducing elevator operating costs.

In elevator control systems, precise speed regulation and acceleration control can reduce elevator operation time and enable high-speed operation control, which helps to improve production cycle time and reduce waiting time in production processes. However, once an elevator control system malfunctions during operation, it is likely to cause serious safety accidents, posing a significant threat to the safety of people's lives and property. Therefore, the correctness and reliability of elevator control systems are of crucial importance.

Traditional testing methods for elevator control systems primarily rely on simulation testing and field operation testing. These methods have inherent limitations: they struggle to cover all possible scenarios,

and the system tends to have various bugs during the operational phase. In contrast, formal methods—as rigorous mathematical approaches and verification tools—can precisely define, describe, and verify systems. They provide a more rigorous and comprehensive mathematical verification basis than traditional testing methods, helping identify potential issues and errors in the early stages of system design. This ultimately enhances the reliability and safety of elevator control systems, offering innovative verification ideas and methods for the safe and stable operation of such systems.

Isabelle/HOL is a powerful formal verification tool based on higher-order logic. It can formally model and verify various complex application scenarios in elevator control systems, ensuring their safety and reliability. Using rigorous mathematical language and logical rules, it precisely constructs mathematical models of elevator control systems, defines states such as elevator stopping floors, running direction, and door opening/closing, and describes actions and events in the system—including passenger calls, elevator responses, and door operations. It proves safety properties such as the elevator not colliding and not entering abnormal operating states, as well as liveness properties of the system, such as ensuring passenger calls are promptly responded to and the elevator operates normally between floors. By carefully designing proof strategies and steps, applying existing theorems and inference rules, and gradually decomposing complex properties and conditions, it completes logical deductions and proofs, while correcting and improving errors and incompleteness encountered in the proof process.

This paper employs Isabelle/HOL to verify the correctness of elevator control systems, providing reference ideas and experiences for the optimization and development of such systems.

2 Related Work

In the field of design and verification of elevator control systems, many scholars have conducted research on such systems, providing theoretical foundations and practical experience for the further development of this field.

In the field of formal methods and model construction, Kaynar, Lynch, and others introduced automata-theoretic techniques [1], which precisely describe the temporal constraints and state transitions of elevator operation, model and analyze real-time systems, and provide reference ideas for elevator control systems with extremely high real-time requirements. The theory of Model Checking [2]—a system checking method—verifies whether elevator control systems satisfy specific properties and identifies potential issues and errors in the system. The SPIN model checking tool [3] can be used for the design verification of elevator control systems, exploring the system's state space to ensure the system's correctness and safety.

In the field of algorithm optimization and control theory, the robotic trajectory optimization algorithm proposed by Klemm, Viragh, and others [4] has its ideas migrated to elevator operation path planning and scheduling algorithm design, which can optimize elevator operation efficiency. The automata-theoretic verification method adopted by Li, Lefebvre, and others [5] provides theoretical support for the coordination verification between various components in elevator control systems, ensuring the coordination of system operation. The supervisory control theory for discrete event processes adopted by Zhang, Huang, and others [6] can be applied to the operation management of elevator control systems, ensuring the system operates safely and stably under all circumstances. The decentralized supervisory control theory adopted by Hayano, Takai, and others [7] provides theoretical support and methodological guidance for the design and verification of multi-elevator cooperative control, enhancing the operational efficiency and reliability of elevator group control systems.

In the field of logic and verification techniques, the temporal logic techniques identified by Ghari [8] provide a logical foundation for the formal specification and verification of elevator control systems, facilitating the precise expression of the system's dynamic behaviors and properties. The context-bounded model checking technique adopted by Monteiro, Alves, and others [9] enhances verification efficiency in elevator control system verification by limiting the context to reduce the state space. The research on linear temporal logic model checking conducted by Tuxi, Carvalho, and others [10] can be applied in elevator control systems to verify complex properties related to integer variables such as the number of floors and the number of passengers.

In the field of hybrid systems and timed systems analysis, the symbolic methods for verification and control of hybrid systems adopted by Liebrenz and Herber [11] have effectively modeled, analyzed, and verified elevator control systems involving both discrete and continuous behaviors. Nejati and Soudjani studied and applied timed abstract bisimulation in timed systems analysis [12], which helps simplify the verification process of time-related properties in elevator control systems.

Additionally, Lin, Zhou, and others [13] conducted formal design and verification of control systems based on Petri nets, while Wu, Lu, and others [14] utilized colored Petri nets for modeling and verification of such systems. These approaches, leveraging the characteristics of Petri nets, detailedly describe resource allocation and concurrent behaviors in elevator systems, providing new perspectives and methods for the modeling and analysis of elevator control systems and enhancing the accuracy of system verification.

Thus, most designs are methods for component-based systems that verify their compatibility. They only address the syntactic compatibility of components without analyzing and verifying runtime behaviors. Meanwhile, methods involving runtime behaviors suffer from the problem of state explosion—specifically, as the number of components increases, the number of global states grows exponentially, leading to increased complexity of analysis. Literature [15] proposes a method for component-based systems that verifies behavioral compatibility and temporal behaviors. This method is mainly completed automatically by the system, requiring little manual derivation, and analyzes two interacting components at a time, thus being free from the impact of state explosion. This pairwise method uses model checking and can automatically verify temporal behaviors in polynomial time with respect to the number and size of all components. Component interactions are captured in their behavioral automata, and the method checks whether the behaviors of a pair of interacting components conform to given properties specified in temporal logic. By applying this method

to the design and verification of component-based elevator control algorithms, the authors demonstrate its effectiveness.

This study focuses on the entire operational process of elevators, systematically analyzes the operational states of elevators, and verifies the relevant properties of elevator control systems. It provides a new perspective for the research on formal modeling and verification of elevator control systems, further improves the accuracy of such systems, and offers theoretical guidance for the modeling and verification of the full operational process of elevators. The comparative analysis is as follows (Table 1):

Table 1 The comparative analysis of related works

Comparison Dimension	Previous Research	Innovative Points of This Study
Research Focus	Component-based system-level verification	Focus on the entire operational process of elevators
Technical Method	Verifying system properties based on component-based approaches	Adopt the Isabelle/HOL interactive theorem proving framework to model and verify elevator control systems
Core Advantage	Component decomposition enables highly targeted verification of elevator subsystems	Cover blind spots in component interaction, reduce code complexity, and enhance the comprehensiveness and accuracy of system verification

3 Formal Methods and the Isabelle/HOL Theorem Prover

3.1 Overview of Formal Methods

Formal Methods serve as the "mathematical language" for system design and verification—a tool that uses precise mathematical symbols and rigorous logic to formalize the design and verification of the behaviors and characteristics of hardware and software systems. They mainly encompass formal specification languages, formal verification techniques, and formal development methods. By formally designing each behavioral state in a system through mathematical logic, potential issues in the system can be accurately identified, inconsistencies between requirements and design can be detected, and security vulnerabilities can be uncovered—providing innovative ideas and methods for the safety and reliability of systems.

Formal specification languages describe the functions and behaviors of a system clearly and accurately, providing rich mathematical structures and logical operators for the precise definition of various properties and constraints of the system. Common formal specification languages include Z notation, VDM (Vienna Development Method), and the B Method. Maximiliano C, Gianfranco R, et al. [16] employed the Z specification to verify its consistency, providing valuable insights and references for formal verification in the field of system security.

Formal verification techniques verify whether a system meets its specifications, with key approaches including model checking and theorem proving. Model checking traverses the state space of a system model to check for violations of the specifications; it is an automated technique but has certain limitations. For instance, when the system scale is large, model checking may encounter the problem of state explosion. Theorem proving, on the other hand, is a method that verifies the properties of a system through mathematical logical reasoning, step-by-step deduction, and proof.

3.2 Introduction to the Isabelle/HOL Theorem Prover

Compared with model checking tools and formal specification and verification languages, the Isabelle/HOL [17] theorem prover boasts strong logical expressiveness and rich verification mechanisms. It supports multiple logical systems, including first-order logic, higher-order logic, set theory, and number theory, enabling more flexible and accurate system modeling and precise description of complex data structures and functions within systems. Mnacho Echenim, Mehdi Mhalla et al. [18] leveraged the Isabelle/HOL theorem prover to formalize the fundamental concepts and results in the quantum domain, proving that the local hidden variables proposed by Einstein for modeling quantum mechanics do not hold. Richard Schmoetten, Jacques D. Fleuriot et al. [19] established a formal theory of smooth vector fields, Lie groups, and Lie algebras using the Isabelle/HOL theorem prover, providing scientific theoretical support and rich practical guidance for particle physics and robotics, as well as laying a more solid theoretical foundation for advancing the further development of formal theory research. It is evident that the ability to define and prove mathematical theorems, as a prominent advantage of Isabelle/HOL, deepens the learning and research of related theories and greatly improves the speed and accuracy of formal proofs.It is widely used in defining and proving mathematical theorems, as well as in the formal verification of various system behaviors and properties.

Isabelle/HOL is equipped with rich proof tactics and automated verification tools, which help users simplify and automatically verify specific target propositions—greatly reducing the workload of proofs and lowering the difficulty of verification. Filip S, D. J F, et al. [20] adopted formal methods to verify a process composition framework oriented to designated goals based on input and output resources, and mechanically verified the correctness of this framework. In Reference [21], Jose D, C J S J, René T, et al. leveraged locales and local type definitions in Isabelle/HOL to verify the Berlekamp-Zassenhaus algorithm for the factorization of square-free polynomial factors. This theoretical research provides theoretical guidance for the formal verification of algorithms, ensures algorithmic accuracy, and offers a safe and reliable guarantee for the application of this algorithm in systems. Da X, Fei Y Z, Li S L, et al. conducted the formal design and verification of hardware systems by means of the HOL system[22], and verified fundamental hardware Trojan detection methods. This work safeguards systems against Trojan intrusions, to a certain extent, provides verification ideas and methodologies for the security and reliability of operating systems, and further promotes the advancement of formal methods in ensuring system security. Through the formal design and verification of hardware, the stability, security and performance of the target system can be basically guaranteed. Refined design based on the higher-order logic framework ensures the

design-level security of both software and hardware, laying a solid foundation for the development of the security field.Isabelle/HOL can also verify the correctness and safety of various systems in practical applications, including hardware designs, operating systems, and communication protocols. Using Isabelle/HOL for formal verification of systems enables the early detection of potential issues during the system development phase; timely optimization, modification, and verification of the system can reduce development costs, mitigate development risks, and enhance the system's safety and reliability.

Formal Methods are rigorous and reliable mathematical means in the process of system development and verification, while the Isabelle/HOL Theorem Prover is a powerful tool for implementing formal verification. The organic combination of the two provides strong support for building high-quality and highly reliable systems. In recent years, with the rapid development of artificial intelligence and big data, the security and reliability of systems are facing unprecedented challenges. Isabelle/HOL has emerged as an essential tool for the design and verification of secure and reliable operating systems in the current context. Relevant literature [23–25] has reported that researchers have leveraged Isabelle/HOL to verify relative security, state protocols, Minkowski spacetime, and weak memory programs, among other domains. In the future, formal methods are poised to become a research hotspot in the field of formal design and verification.

The symbolic language and corresponding meanings of Isabelle/HOL are as follows (Table 2):

Table 2 The symbolic language of Isabelle/HOL

Notation/Keyword	Meaning
bool	Boolean type
nat	Natural number type
datatype	User-defined data type
record s= x::nat y::nat	Definition of structure data type
fun	Global function definition
∀	Universal quantifier (For all)
∃	Existential quantifier (There exists at least one)
λ	λ-abstraction (Anonymous function)
∧	Conjunction (And)
∨	Disjunction (Or)
≡	Identically equal to
⇒	Function type designator/Material implication
lemma	Proposition or lemma to be proved
assume	Introduce proof premises
show	Prove a sub-conclusion or the final conclusion
case	Split base cases and recursive cases
have	Derive and save intermediate conclusions
by	Invoke a proof method
simp	Simplifier
auto	Automatically combines simplification and logical reasoning
induct	Proof strategy specifically for specifying "induction"
proof	Open a proof block
qend	Close a proof block

In this paper, we leverage the unique advantages of Isabelle/HOL to construct the model of the elevator control system and systematically verify its key properties during the elevator's operational process.

4 Formal Modeling of Elevator Control Systems

This paper takes a specific elevator control system as an example and elaborates on its formal modeling. The architecture of modeling and verfication is shown as Fig. 1.

4.1 Definition of Elevator States

The state of an elevator can be represented by a triple, i.e., (position, direction, door state). Among them, Position denotes the current floor where the elevator is located, Direction indicates the elevator's operation direction, and Door State represents the open/closed state of the elevator door.

The datatype keyword in Isabelle/HOL is used to define custom data types, which enables the precise definition of the floor type and the elevator state. The elevator state includes the current floor, the door state (open and closed), and the direction of the elevator's movement. The specific definitions are as follows:

```
datatype floor = Floor1 | Floor2 | Floor3 (*Additional floors can
be added as needed. *)

record elevator_state =
  current_floor :: floor (* Current floor *)
  door_status :: bool (* Door State: True denotes open, False
denotes closed *)
  direction :: "up | down | idle" (* Operation Direction *)
```

4.2 Definition of Passenger Requests

A passenger request can be represented by a pair, i.e., (starting floor, target floor). Passenger requests can be divided into internal requests and external requests: internal requests refer to floor buttons pressed by passengers inside the elevator, while external requests refer to floor buttons pressed by passengers outside the elevator.

Employ the datatype keyword to describe passenger requests. The specific definition is as follows:

```
datatype passenger_request = InternalRequest floor |
ExternalRequest floor
```

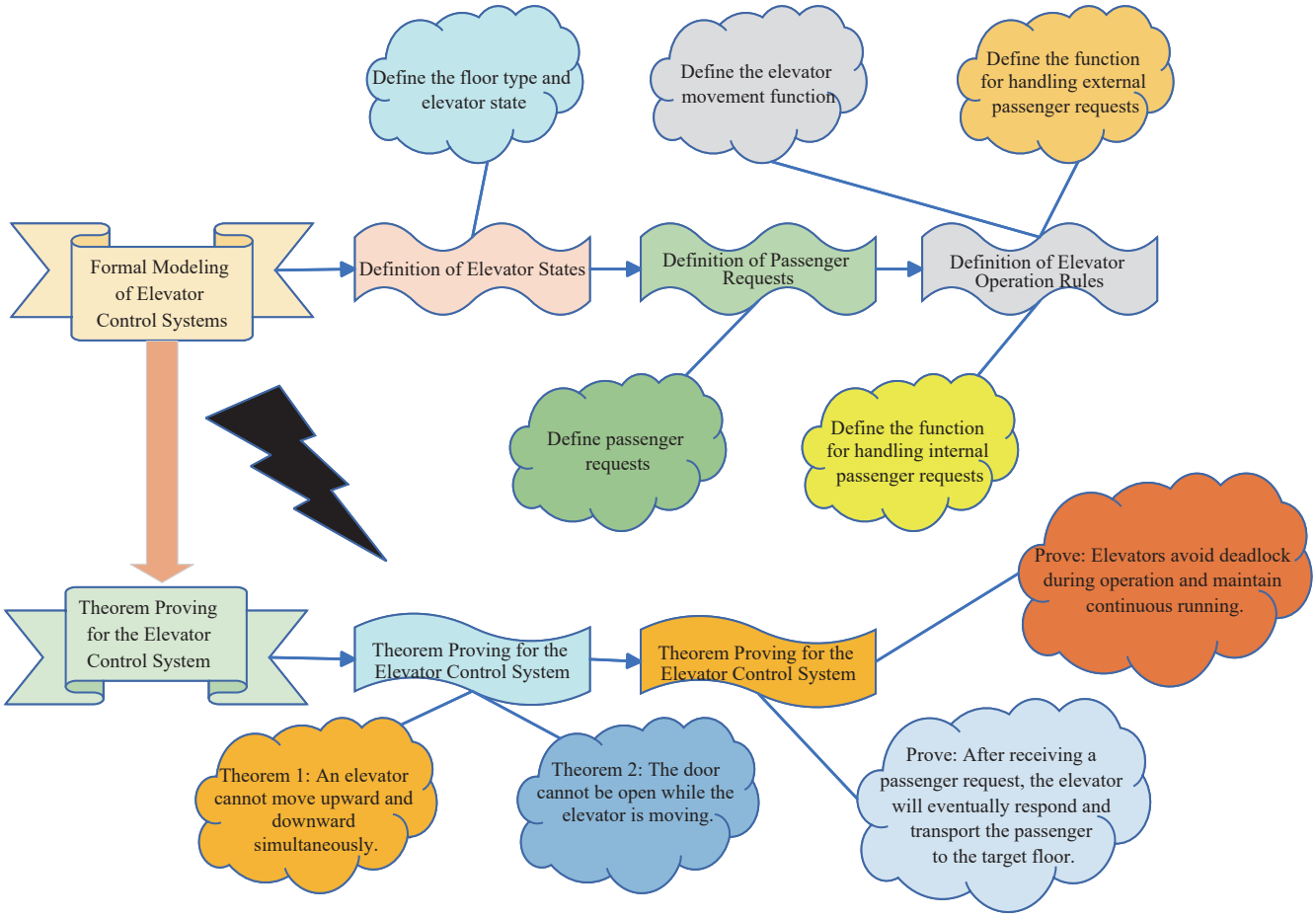

Fig. 1 The architecture of modeling and verfication

4.3 Definition of Elevator Operation Rules (Table 3)

Table 3 The elevator operation rules

Rule	Description
Elevator Responding to Requests	After an elevator receives a request sent by a passenger, it determines whether it can respond to the request based on factors such as its current position, direction, and door state. If it can respond, the elevator will move toward the requested floor.
Elevator Door Opening and Closing	When the elevator reaches the requested floor, it decides whether to open the door based on its current state and the type of request. If the door opens, it will automatically close after a certain period of time.
Elevator Direction of Operation	During operation, the elevator determines its direction of operation based on its current position and the distribution of requests. If it is moving upward and there are no requests for higher floors but there are requests from lower floors, the elevator will change its direction of operation to move downward.

The Fun keyword is used to define the elevator movement function move_elevator, which takes two parameters: elevator_state (i.e., the elevator state) and floor (i.e., the target floor), and returns the updated elevator_state. For example, when passing the parameters s and Floor1—where s denotes the elevator state and Floor1 represents the target floor—if the elevator's current floor is Floor2, the elevator will move from Floor2 to Floor1 with the downward movement direction; if the elevator's current floor is Floor3, it will move from Floor3 to Floor1 also with the downward direction. In all other cases, the original state s is returned directly. This paper takes a 3-floor elevator as an example, and the definitions for other scenarios can be formulated in accordance with the above description. The proof method proposed in this paper can be extended to elevators with multiple or any number of floors based on an analogous principle. A three-floor elevator is selected as the verification case herein because it essentially encompasses all runtime states of multi-floor elevators. By analogy, the movement function for the 3-floor elevator is thus implemented. The specific definition in Isabelle/HOL is as follows:

```
fun move_elevator :: "elevator_state => floor => elevator_state"
where
  "move_elevator s (Floor1) =(if s.current_floor = Floor2
  then (s (| current_floor := Floor1, direction := down |) )
  else if s.current_floor = Floor3
  then (s (| current_floor := Floor1, direction := down |) )
  else s)"
```

```
"move_elevator s (Floor2) =(if s.current_floor = Floor1
then (s (| current_floor := Floor2, direction := up |) )
else if s.current_floor = Floor3
then (s (| current_floor := Floor2, direction := down |) )
else s)"
"move_elevator s (Floor3) = (if s.current_floor = Floor2
then (s (| current_floor := Floor3, direction := up |) )
else if s.current_floor = Floor1
then (s (| current_floor := Floor3, direction := up |) )
   else s)"
```

We define a function for processing passengers' internal requests, which takes two parameters: the elevator state *s* and the passenger's internal request, and returns the updated elevator state. If a passenger has made an internal request and the elevator's door is currently in the closed state, the elevator state *s* is returned directly without any modification; otherwise, the elevator's door state is updated to be closed and the target floor is set to *f*. The specific definition in Isabelle/HOL is as follows:

```
fun handle_internal_request ::
 "elevator_state => passenger_request => elevator_state" where
 "handle_internal_request s (InternalRequest f) =
 (if s.door_status = False then s
 else (s (| door_status := False, current_floor := f |) ))"
```

We define a function for processing passengers' external requests, which takes two parameters: the elevator state s and the passenger's external request, and returns the updated elevator state. A nested conditional expression of the form "if {…} then {if {…}}" is adopted—when the outer conditional check is satisfied, the function does not return directly but proceeds to the inner conditional check. For the outer conditional check: the condition is satisfied if the elevator's movement direction is upward and its current floor is less than the target floor, or the elevator's movement direction is downward and its current floor is greater than the target floor, or the elevator is idle. For the inner conditional check: if the outer condition is met, the function then checks whether the elevator's current door state is closed. If it is closed, the elevator state s is returned directly without any modification; otherwise, the elevator's door state is updated to closed, and the target floor is set to the floor where the passenger made the external request. If the outer conditional check is not satisfied, the original elevator state is returned directly.

```
fun handle_external_request ::
 "elevator_state => passenger_request => elevator_state" where
 "handle_external_request s (ExternalRequest f) =
 (if (s.direction = up ∧ s.current_floor < f) ∨
 (s.direction = down ∧ s.current_floor > f) ∨ s.direction = idle
 then (if s.door_status = False then s
 else (s (| door_status := False, current_floor := f |) ))
else s)"
```

First, the floor type, elevator state, and passenger request types were defined. Next, the elevator movement function (move_elevator) and the functions for handling internal and external passenger requests (handle_internal_request and handle_external_request) were defined. These functions form the foundational part of the formal model for the elevator control system.

5 Theorem Proving for the Elevator Control System

5.1 Theorem Proving for the Elevator Control System

Theorem 1: An elevator cannot move upward and downward simultaneously.

Introduce the local assumption that "the elevator's movement direction is both upward and downward," and prove that the target proposition is False. By combining the main proof with supplementary rules, the derivation and proof of the proposition are rapidly achieved.

```
lemma "∀s. ¬(s.direction = Up ∧ s.direction = Down)"
proof (intro allI)
 fix s
 assume "s.direction = Up ∧ s.direction = Down"
 then show "False"
  by (auto simp add: elevator_direction.simps)
qed
```

Theorem 2: The door cannot be open while the elevator is moving.

Introduce the local assumption that "the elevator is not idle" to prove the target proposition: the elevator cannot open its doors while in operation. We specify the scenario where the elevator is in operation (i.e., not idle), and complete the proof of this theorem by combining the main proof with supplementary rules.

```
lemma "∀s. s.direction ≠ Idle ⟹ ¬s.door_open"
proof (intro allI)
 fix s
 assume "s.direction ≠ Idle"
 then show "¬s.door_open"
  by (cases s; auto simp add: elevator_direction.simps)
qed
```

5.2 Proof of the Liveness Theorem

(1) Passenger requests will eventually be responded to: Prove that after receiving a passenger request, the elevator will eventually respond to the request and transport the passenger to the target floor.

When proving properties of custom composite data types, conventional proof methods such as simp are insufficient to meet the requirements. It is necessary to initiate a structural induction proof using the induction s rule: elevator_state.induct. The proof process involves conducting case analysis, introducing local hypotheses, verifying sub-propositions, and finally completing the proof with the auto keyword.

Inductive proofs and pattern matching are utilized to verify the elevator movement logic: in Isabelle, proof acts as the keyword to initiate a proof block; for Inductive Case 1, let *f* denote the floor, *d* denote the movement direction, *b* denote the boolean flag, and *rs* denote the request list, with the hypothesis (exists *f*. *f* in text{pending_requests} (*f* # *rs*)) (i.e., there exists a floor *f* such that *f* is contained in the pending

request sequence of ((*f* # *rs*))) introduced; the keyword auto is adopted to implement automated logical deduction, the keyword moreover is used to connect multiple independent facts, and finally the combination of ultimately show ?case by blast is applied to complete the final derivation, while the have section specifies the core rules of the elevator movement function—when the elevator is idle, its state remains unchanged; when the elevator is on the 1st floor with an upward direction, its state changes with the target floor updated to the 2nd floor, the direction remaining upward and the pending request queue unchanged; when the elevator is on the 2nd floor with an upward direction, its state changes with the target floor updated to the 3rd floor, the direction remaining upward and the pending request queue unchanged; when the elevator is on the 3rd floor with an upward direction (i.e., it has reached the top floor), its state remains unchanged; when the elevator is on the 3rd floor with a downward direction, its state changes with the target floor updated to the 2nd floor, the direction remaining downward and the pending request queue unchanged; when the elevator is on the 2nd floor with a downward direction, its state changes with the target floor updated to the 1st floor, the direction remaining downward and the pending request queue unchanged; when the elevator is on the 1st floor with a downward direction (i.e., it has reached the bottom floor), its state remains unchanged—and the corresponding equality is proven through simplification rules.

```
lemma "∀s. (∃f. f ∈ s.pending_requests) ⟹
  ∃s'. s'.current_floor = hd (s.pending_requests) ∧
  s'.pending_requests = tl (s.pending_requests)"
proof (induction s rule: elevator_state.induct)
  case (1 f d b rs)
  assume "∃f. f ∈ (f # rs).pending_requests" then obtain f' where
    "f' ∈ (f # rs).pending_requests" by auto
    moreover
    have "move_elevator (s(|pending_requests := f # rs|)) =
      case s(|pending_requests := f # rs|) of (f', Idle, _, _ ) ⇒
      s(|pending_requests := f # rs|) | (f', Down, _, _ ) ⇒
      s(|pending_requests := f # rs|) | (f', Up, _, _ ) ⇒
      s(|pending_requests := f # rs|) | (Floor1, Up, _, _ ) ⇒
      s(|pending_requests := f # rs, current_floor := Floor2, direction
:= Up|)
      | (Floor2, Up, _, _ ) ⇒ s(|pending_requests := f # rs,
      current_floor := Floor3, direction := Up|)
      | (Floor3, Up, _, _ ) ⇒ s(|pending_requests := f # rs|)
      | (Floor3, Down, _, _ ) ⇒ s(|pending_requests := f # rs,
      current_floor := Floor2, direction := Down|)
      | (Floor2, Down, _, _ ) ⇒ s(|pending_requests := f # rs,
      current_floor := Floor1, direction := Down|)
      | (Floor1, Down, _, _ ) ⇒ s(|pending_requests := f # rs|)"
      by (simp add: move_up.simps move_down.simps)
    moreover
    have "case s(|pending_requests := f # rs|) of (f', Idle, _, _ ) ⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs,
current_floor :=
      f',direction := Idle|).pending_requests = tl (s.pending_requests)
      ∨ case s(|pending_requests := f # rs|) of (f', Down, _, _ ) ⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs,
current_floor :=
      f',direction := Down|).pending_requests = tl (s.pending_
requests)
      ∨ case s(|pending_requests := f # rs|) of (f', Up, _, _ ) ⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs,
current_floor :=
      f', direction := Up|).pending_requests = tl (s.pending_requests)
      ∨ case s(|pending_requests := f # rs|) of (Floor1, Up, _, _ ) ⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs,
current_floor :=
      Floor2, direction := Up|).pending_requests = tl (s.pending_
requests)
      ∨ case s(|pending_requests := f # rs|) of (Floor2, Up, _, _ ) ⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs,
current_floor :=
      Floor3, direction := Up|).pending_requests = tl (s.pending_
requests)
      ∨ case s(|pending_requests := f # rs|) of (Floor3, Up, _, _ ) ⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs|).
pending_requests =
      tl (s.pending_requests)
      ∨ case s(|pending_requests := f # rs|) of (Floor3, Down, _, _ )
⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs,
current_floor :=
      Floor2, direction := Down|).pending_requests = tl (s.pending_
requests)
      ∨ case s(|pending_requests := f # rs|) of (Floor2, Down, _, _ )
⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs,
current_floor :=
      Floor1, direction := Down|).pending_requests = tl (s.pending_
requests)
      ∨ case s(|pending_requests := f # rs|) of (Floor1, Down, _, _ )
⇒
      f' = s.current_floor ∧ s(|pending_requests := f # rs|).
pending_requests =
      tl (s.pending_requests)"
      by auto
    ultimately show ?case by blast
qed
```

This proof process employs induction and case analysis to demonstrate that if there exist pending floor requests, then there will eventually exist an elevator state where the current floor is the requested floor and the pending request list is updated.

(2) Elevators will not enter a deadlock state: It is proven that elevators will not enter a deadlock state during operation and can always continue running.

First, we present a function that determines whether the transition between two states is valid, which is defined as follows:

```
consts valid_transition :: "elevator_state ⇒ elevator_state ⇒ bool"
```

Define valid_transition as a function that determines whether the transition between two states is valid. Below is a simple example of its definition:

Use the definition keyword to define valid_transition as a function that determines whether the transition between two states is legitimate. The definition keyword supports the definition of predicate judgment logic. An elevator state transition is legitimate in each of the following scenarios: the original state of the elevator is non-idle while the current state is idle; the elevator was originally on the first floor with an upward direction and is currently on the second floor; the elevator was originally on the second floor with a downward direction and is currently on the first floor; or the original elevator door was closed, the current elevator door is open, and the original floor is consistent with the current floor. The specific definition is as follows:

```
definition "valid_transition s s' ≡
  ((s'.direction = Idle ∧ s.direction ≠ Idle) ∨
  ((s'.current_floor = Floor2 ∧ s.current_floor = Floor1 ∧
s.direction = Up) ∨
  (s'.current_floor = Floor1 ∧ s.current_floor = Floor2 ∧
s.direction = Down)) ∨
  ((s'.door_open = True ∧ s.door_open = False) ∧ s'.current_
floor = s.current_floor))"
```

On this basis, we present a theorem stating that the elevator control system is deadlock-free, i.e., there always exists a valid state transition—the proof objective is to verify the existence of such a valid state transition for the proposition, which involves analyzing possible next states under different scenarios; for example, to prove the validity of the updated elevator state transition triggered by the first request, we can employ the simp keyword for proof simplification and invoke by blast for automated proof simplification, followed by a case analysis to demonstrate that the elevator control system avoids deadlocks under various conditions, particularly focusing on the validity of state transitions when the elevator is in the idle state: specifically, if the first requested floor is the 1st floor, the elevator's new state will be set to upward if it is currently on the 1st floor, otherwise it remains idle; if the first requested floor is the 2nd floor, the elevator's new state will be upward if it is on the 2nd floor, idle if it is on the 1st floor, and downward otherwise; if the first requested floor is the 3rd floor, the elevator's new state will be downward if it is currently on the 3rd floor, otherwise it stays idle. The detailed proof implemented in Isabelle/HOL is provided as follows:

```
lemma "∀s. ∃s'. valid_transition s s'"
proof (intro allI)
  fix s
  show "∃s'. valid_transition s s'"
  cases s
  case (f d b rs)
    (* Analyze the possible next state(s) based on different
scenarios *)
    if d = Idle then
      have "valid_transition s (s(|direction := case (hd rs) of Floor1
⇒
          if f = Floor1 then Up
            else Idle
              | Floor2 ⇒
          if f = Floor2 then Up
            else if f = Floor1 then Idle
            else Down
              | Floor3 ⇒
          if f = Floor3 then Down
            else Idle|))"
        by (simp add: valid_transition_def)
      then show "∃s'. valid_transition s s'" by blast
    else if d = Up then
      if f = Floor1 then
      have "valid_transition s (s(|current_floor := Floor2, direction
:= Up|))"
        by (simp add: valid_transition_def)
        then show "∃s'. valid_transition s s'" by blast
      else if f = Floor2 then
        have "valid_transition s (s(|current_floor := Floor3, direction
:= Up|))"
          by (simp add: valid_transition_def)
        then show "∃s'. valid_transition s s'" by blast
      else
        show "∃s'. valid_transition s s'" by (metis valid_transition_
def)
    else if d = Down then
      if f = Floor3 then
        have "valid_transition s (s(|current_floor := Floor2, direction
:= Down|))"
          by (simp add: valid_transition_def)
        then show "∃s'. valid_transition s s'" by blast
      else if f = Floor2 then
        have "valid_transition s (s(|current_floor := Floor1, direction
:= Down|))"
          by (simp add: valid_transition_def)
        then show "∃s'. valid_transition s s'" by blast
      else
        show "∃s'. valid_transition s s'" by (metis valid_transition_
def)
    qed
qed
```

Studies have shown that formal verification of elevator control systems can not only uncover potential issues in the system's operation, accurately verify the system's correctness and reliability, but also enhance the system's safety and reliability, reduce system development time, and lower the costs incurred during the development, operation, and maintenance phases.

6 Conclusion and Future Work

This paper employs the Isabelle/HOL theorem prover to conduct formal modeling and theorem proving for elevator control systems, verifying their key properties such as safety and liveness. Findings from the study indicate that the Isabelle/HOL theorem prover can effectively perform formal verification on the correctness of elevator control systems, providing safety guarantees for the stable operation of such systems.

In the future, we will further study and research the application of formal methods in elevator control systems: using them to analyze and optimize the performance of elevator control systems and improve operational efficiency; integrating formal methods with other software tools to achieve the automation of formal verification; and thereby enhancing system development efficiency while ensuring system quality and safety.

References

[1] Kaynar D K, Lynch N, Segala R, Vaandrager F. Timed I/O automata: a mathematical framework for modeling and analyzing real-time systems. In: Proceedings of the 24th IEEE Real-Time Systems Symposium, RTSS '03. 2003, 166–177.

[2] Alur R, Courcoubetis C, Dill D. Model-checking for probabilistic real-time systems. In: Proceedings of the International Colloquium on Automata, Languages, and Programming. 1991, 115–126.

[3] Holzmann G J. The SPIN model checker: primer and reference manual. Reading: Addison-Wesley, 2004.

[4] Klemm V, de Viragh Y, Rohr D, Siegwart R, Tognon M. Nonsmooth trajectory optimization for wheeled balancing robots with contact switches and impacts. IEEE Transactions on Robotics, 2025, 41: 497–517.

[5] Li J, Lefebvre D, Hadjicostis C N, Li Z. Verification of state- based timed opacity for constant-time labeled automata. IEEE Transactions on Automatic Control, 2025, 70(1): 503–509.

[6] Zhang H, Huang L, Huang W, Feng L, Li X. Synthesis of opacity-enforcing supervisory strategies using reinforcement learning. IEEE Transactions on Automation Science and Engineering, 2025, 22: 6896–6906.

[7] Hayano A, Takai S. A general Architecture for intersection-based decentralized supervisory control of discrete event systems. IEEE Transactions on Automatic Control, 2024, 69(1): 674–680.

[8] Ghari M. Linear temporal justification logics with past and future time modalities. Logic Journal of the IGPL, 2023, 31(1): 1–38.

[9] Monteiro F R, Alves E H S, Silva I S, Ismail H I, Cordeiro L C, Filho E B L. ESBMC-GPU: a context-bounded model checking tool to verify CUDA programs. Science of Computer Programming, 2018, 152: 63–69.

[10] Tuxi T M, Carvalho L K, Nunes E V L, Cunha A E C. Diagnosability verification using LTL model checking. Discrete Event Dynamic Systems-Theory and Applications, 2022, 32(3): 399–433.

[11] Liebrenz T, Herber P, Glesner S. Service-oriented decomposition and verification of hybrid system models using feature models and contracts. Science of Computer Programming, 2021, 152: 63–69.

[12] Nejati A, Soudjani S, Zamani M. Compositional abstraction-based synthesis for continuous-time stochastic hybrid systems. European Journal of Control, 2021, 57: 82–94.

[13] Lin Z, Zhou J, Sun S, Luo J, Zhang J. Petri net model predictive control method for batch chemical systems. Processes, 2024, 12(3): 1–15.

[14] Wu D, Lu D, Tang T. Qualitative and quantitative safety evaluation of train control systems (CTCS) with stochastic colored petri nets. IEEE Transactions on Intelligent Transportation Systems, 2022, 23(8): 10223–10238.

[15] Attie P C, Lorenz D H, Portnova A, Chockler H. Behavioral compatibility without state explosion: design and verification of a component-based elevator control system. In: Proceedings of the International Symposium on Component-Based Software Engineering. 2006, 33–49.

[16] Cristia M, Rossi G. An automatically verified prototype of the tokeneer ID station specification. Journal of Automated Reasoning, 2021, 65(8): 1125–1151.

[17] Nipkow T, Paulson L C, Wenzel M. Isabelle/HOL: a proof assistant for higher-order logic, 2283. Berlin, Germany: Springer, 2002.

[18] Echenim M, Mhalla M. A formalization of the CHSH inequality and tsirelson's upper-bound in Isabelle/HOL. Journal of Automated Reasoning, 2024, 68(1): 1–26.

[19] Schmoetten R, Fleuriot J D. Constructing the lie algebra of smooth vector fields on a lie group in Isabelle/HOL. Journal of Automated Reasoning, 2025, 69(3): 1–29.

[20] Smola F, Fleuriot J D. Linear resources in Isabelle/HOL. Journal of Automated Reasoning, 2024, 68(2): 1–49.

[21] Divason J, Joosten S J C, Thiemann R, Yamada A. A verified implementation of the berlekamp-zassenhaus factorization algorithm. Journal of Automated Reasoning, 2020, 64(4): 699–735.

[22] Xiao D, Zhu Y F, Liu S L, Wang D X, Luo Y Q. Digital hardware design formal verification based on HOL system. Applied Mechanics and Materials, 2014, 716–717: 1382–1386.

[23] Derrick J, Dongol B, Edmonds C, Griffin M, Popescu A, Wright J. Relative security: (dis)proving resilience against semantic optimization vulnerabilities in Isabelle/HOL. Journal of Automated Reasoning, 2025, 69: 1–64.

[24] Hess V A, Mödersheim A S, Brucker D A, Schlichtkrull A. PSPSP: a tool for automated verification of stateful protocols in Isabelle/HOL. Journal of Computer Security, 2025, 33(6): 425–469.

[25] Dalvandi S, Dongol B, Wehrheim H. Integrating owicki–gries for C11-style memory models into Isabelle/HOL. Journal of Automated Reasoning, 2022, 66: 141–171.

Brief Introduction to the Authors:

Heng ZHANG (b. 1982), male, a native of Suzhou, Jiangsu Province, China, holds a Master's degree, is a Senior Lecturer and Senior Technician, with research interests in Mechanical Manufacturing and Automation.

Long CAO (b. 2001), male, a native of Xuzhou, Jiangsu Province, China, is an undergraduate student, with research interests in Formal Modeling and Verification.