An Integrated Formal Method Combining Labeled Transition System and Event-B for System Model Refinement

LEI RAO¹, SHAOYING LIU¹, Fellow, IEEE, AND HAN PENG²
¹Graduate School of Advanced Science and Engineering, Hiroshima University, Hiroshima, 739-8511 Japan
²School of Computer, Xi’an Aeronautical University, Xi’an, 710077 China
Corresponding author: Shaoying Liu (e-mail: sliu@hiroshima-u.ac.jp).

This work was supported by ROIS NII Open Collaborative Research 2021-(21FS02).

ABSTRACT Formal modeling and verification of a concurrent system is an essential means to ensure the security and reliability of the system. However, at present, there is no single formal method that can fully meet the modeling and verification requirements of concurrent systems. In this paper, we propose an integrated formal method that utilizes both event-based method Event-B and state-based formalism LTS to address this problem. We first analyze the difference and connection between Event-B and LTS in building system models and then propose to use the graphical front-end iUML-B of Event-B to obtain a unified representation with LTS so as to take their advantages in the integrated method. Finally, we carry out a case study to demonstrate the practicality of the proposed method. The case study shows that our method effectively models and verifies the various properties of the system, and to a large extent makes up for the shortcomings of a single formal method in the process of system modeling and verification.

INDEX TERMS Concurrent system, Event-based, State-based, Integrated Formal Method

I. INTRODUCTION Formal methods refer to various mathematical methods for formal specification and verification of software. They use formal specification languages as modeling elements and use a set of tools to support syntax checking and property verification of specifications [1]. Supported by formal methods, researchers can use rigorous mathematical models to describe the behavior and requirements of systems and verify a given system or system model against the required behavioral properties [2].

However, Clarke et al. pointed out in the 1990s that no single formal method can solve the core problems of complex system modeling and verification in a satisfactory way [3]. Moreover, Almeida et al. also believed that none of the current single formal methods can fully meet the modeling and verification requirements of complex systems [2]. For a control-intensive system such as a concurrent system, the numerous objects contained therein make it difficult to directly establish the final system model. It is necessary to adopt refinement to add system functions and details gradually until a model meeting all the requirements defined in the specification is obtained. In addition, the intricate behavioral interactions in the system also bring great obstacles to the analysis of the system. Therefore, in the process of modeling and verification of concurrent systems, it is necessary not only to ensure the consistency of data refinement, but also to pay attention to the way to display correct and easy-to-observe behavioral interactions in the system. Our experience suggests that this challenge can be alleviated by appropriately integrating two or more formal methods, each of which meets a specific requirement for system modeling.

In this paper, we discuss how the problem with the behavioral interaction expression in Event-B can be solved by integrating Event-B with labeled transition system (LTS). Due to the lack of behavioral semantics of Event-B itself, the researchers proposed to use CSP as the behavioral semantic model of Event-B to facilitate the expression of the sequence of events in the Event-B model. It can also ensure its behavior refinement while using Event-B’s own refinement mechanism to ensure action refinement [4, 5]. In addition, due to the similarity of the invalidation-divergence semantics between CSP and Event-B, CSP is also used as the behavior refinement framework of Event-B to ensure that behavioral consistency can be maintained during the refinement of the Event-B model [6]. However, this method uses CSP and Event-B to model the system separately. Without a unified model, it cannot guarantee that the two system models established meet the same requirements.
Aiming at addressing the problem with the modeling and verification requirements of concurrent systems, this paper proposes an integrated formal method combining the event-based method Event-B and its graphical front-end integrated UML-B (iUML-B), with the state-based formalism LTS. Specially, our work makes the following contributions.

(1) We work out the differences and connections between the event-based refinement process of Event-B and the state-based refinement process of LTS, as well as their respective advantages and disadvantages in modeling.

(2) We propose a way to combine Event-B with LTS to achieve a unified representation during a refinement process.

(3) We demonstrate the practicality of the integrated method by applying it to the modeling and verification of the ARINC653 specification.

The remainder of the paper is outlined as follows. Section 2 introduces some basic knowledge for understanding our method. Section 3 discusses the details of our method. Section 4 takes the ARINC653 specification as an experiment to illustrate how our method can be used to model and verify the system. Section 5 presents the experiment result and compares it with existing work. Section 6 discusses related work and its relation with our method. Section 7 concludes the paper and points out future research direction.

II. PRELIMINARY

In this section, we introduce some preliminary definitions and notations that are used in this paper.

A. LTS AND ITS COMBINATIONS

LTS belongs to a specific category of automaton, which is widely used to model and analyze the behavior of concurrent and distributed systems [7]. LTS is a state transition system in which the transitions are marked as actions. The set of actions of the LTS is called its communication alphabet [8]. The following is the formal definition of LTS and its composition.

**Definition 2.1.** LTS [9]: Let States represent a universal set of states, Acts represent a universal set of actions, and then an LTS P is defined as a quaternon $P = \langle Q, \Sigma, \Delta, q_0 \rangle$ where:
- $Q \subseteq$ States, representing the state set of P;
- $\Sigma = \alpha P (\alpha P \subseteq$ Acts), representing the action set of P;
- $\Delta \subseteq Q \times \Sigma \times Q$, representing the transition relation in P, these transitions are labeled with the elements in $\Sigma$;
- $q_0 \in Q$, representing the initial state of P.

If $P' = \langle Q, \Sigma, \Delta, q'_0 \rangle$, and $(q, \alpha, q') \in \Delta$, LTS P can be converted to LTS P' by action $\alpha (\alpha \in Acts)$, denoted as $P \xrightarrow{\alpha} P'$.

We need to use the parallel composition of LTSs to express the interaction between multiple LTSs. The following gives the definition of LTS parallel composition.

**Definition 2.2.** Parallel composition of LTSs: The parallel composition of two LTS $M = \langle Q_1, \Sigma_1, \Delta_1, q_1 \rangle$ and $N = \langle Q_2, \Sigma_2, \Delta_2, q_2 \rangle$ is expressed as LTS $(M \parallel N) = \langle Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \Delta, (q_1, q_2) \rangle$, where $\parallel$ is a commutative and associative operator, which means:

$LTS(M \parallel N) = LTS(N \parallel M)$

In addition, $\Delta$ is the minimum relation that satisfies the following constraints:

$$\frac{M \xrightarrow{a} M'}{M \parallel N \xrightarrow{a} M' \parallel N}$$

$$\frac{a \in \alpha N}{N \xrightarrow{a} N'}$$

$$\frac{M \parallel N \xrightarrow{a} M' \parallel N'}{M \xrightarrow{a}, N \xrightarrow{\tau} M' \parallel N'}$$

where $a \in \Sigma_1 \cup \Sigma_2$, $\tau$ denotes an action that is internal to a subsystem, and therefore unobservable by its environment.

B. EVENT-B, IUM-L STATE MACHINE AND ITS COMBINATION

Event-B is a formal modeling language evolved from the B method. An Event-B model consists of two parts: machine and context. The context describes the static elements of the system, including sets, constants, axioms, and theorems. The machine uses variables and events to describe the dynamic behavior of the system. In Event-B, an event consists of guards and actions, which can usually be expressed as:

$$e \equiv (\text{WHEN guards THEN actions}) \text{ END}$$

When the guards of the event are satisfied, the event can be triggered, and the expression in the actions part describes the change in the state variable when the event occurs.

However, since Event-B is based on set theory and first-order logic, there is inevitably a problem that modeling is not intuitive enough. Therefore, C. Snook invented a UML-like Event-B graphics front-end called UML-B [10]. UML-B uses common class diagrams and state diagrams to describe the state and actions of the system. The system model represented by UML-B can generate the corresponding Event-B code directly on the Rodin platform [11]. As UML-B is continuously applied and expanded, UML-B has evolved into iUML-B.

Each iUML-B state machine can automatically generate some code and embed it into the Event-B model. When there are multiple iUML-B state machines in an Event-B model, the behavior of the model is affected by all automatically generated code. In order to facilitate the analysis and verification of the behavior of these state machines, this article uses "\(\bigotimes\)" to represent the combination of iUML-B state machines. Its definition is as follows:

**Definition 2.3.** Combination of iUML-B state machines: The Event-B model generated by the combination of two iUML-B state machines $Stm_1$ and $Stm_2$ is symbolically expressed as:

$$M = Stm_1 \bigotimes Stm_2$$

Further, the Event-B model generated by the combination of $N$ iUML-B state machines $Stm_1$, $Stm_2$, ..., $Stm_N$ is symbolized as

$$M = \bigotimes_{i=1}^{N} Stm_i$$

For example, the Event-B model shown in Fig.1 is the combination of the state machine in the upper left corner and the state machine in the upper right corner.
III. METHODOLOGY

In order to combine Event-B and LTS in the process of system modeling and verification, we first work out the differences and connections between the event-based refinement process of Event-B and the state-based refinement process of LTS. Then we propose to use the graphical front-end iUML-B of Event-B to obtain a unified representation with LTS, and verify the bisimulation equivalence between them. Finally, we briefly discuss the improvements of this method.

A. REFINEMENT PROCESS IN EVENT-B AND LTS

1) EVENT-BASED REFINEMENT PROCESS

Refinement is a technology in which engineers build abstract models for software based on requirements documents in requirements analysis, and the process of modeling will build a series of more and more accurate models of software. Therefore, the refinement process is a process of increasing the function of the system and adding details. The establishment and refinement of the entire system model are completed by the decomposition and addition of events. Take Fig.2 as an example, in general, the refinement of the Event-B model follows the following process:

1) Define an abstract event (E.1.0).
2) Perform one or more of the following two operations:
   - Refine abstract events into (one or more) concrete events (E.1.0 is decomposed into E.1.1 and E.2.1) and add order constraints between concrete events (E.1.1 occurs before E.2.1); or
   - Add a new event (E.3.2) and add a constraint relationship between the new event and the original event (E.3.2 occurs before E.1.2).
3) Repeat the second step until the final refined model is obtained.

This approach is very beneficial for expressing event-based systems, but it also brings about several problems: First, the refinement process is not clear, there is no clear distinction between horizontal expansion (i.e., adding new objects) and vertical refinement (i.e., decomposition of the original object), for example, it is difficult to distinguish whether E.2.1 is decomposed from E.1.0 or a newly added event in the model.

Second, it is difficult to model and analyze concurrent objects. In Fig.2, the newly added objects and the interaction and constraints between these new objects and the original objects cannot be seen. Third, the event-based refinement process is difficult to smoothly transform into a state-based expression.

![FIGURE 1. Combination of iUML-B state machines](image)

2) STATE-BASED REFINEMENT PROCESS

The refinement of an LTS refers to a transformation from an abstract state machine into a concrete machine. An LTS model can be constructed by means of a number of refinement steps. Specifically, such a refinement process includes the following actions:

1) Establish an abstract behavior model of the system, usually an object containing a few states (an object O.1 containing states A and B).
2) Perform one or more of the following two operations:
   - Decompose the state (transition) in the object into several concrete states (transitions) (state A is decomposed into D and C, and event e is decomposed into e1, e, e2), and/or
   - Add new states, which is to add new objects (an object O.2 composed of state G, I and K) and add behavioral constraints between new and existing objects (some constraints between objects O.1 and O.2).
3) Repeat the second step until the final refined model is obtained.

![FIGURE 2. Event-based refinement process](image)

![FIGURE 3. State-based behavior refinement process](image)
The state-based refinement process is depicted in Fig. 3. It can be seen that this refinement process is actually the decomposition and addition of the state in the object. The advantage of this expression is the ability to clearly express the objects that make up the system and the interactions between the objects. The weakness is that the refinement relationship between the abstract model and the refined model is not clear. People cannot understand the correspondence between transitions or states in the refined model, and corresponding elements in the abstract model even by analyzing the model. For example, it is difficult to distinguish the correspondence between the states C, D, E, F in M and A, B in M in the model.

It can be seen that event-based refinement and state-based refinement have their own advantages and disadvantages. The former is strong in maintaining the vertical refinement relationship between the refined model and the abstract model, while the expressive ability in the addition of concurrent objects is weak. The latter is just the opposite. It can clearly express the behavioral interactions between objects, but lacks the ability to express the refinement relationship of the model. Although studies have been conducted on how to transform an Event-B model to LTS, neither has been able to fundamentally resolve the syntactic and semantic gap between the two expressions, making it difficult to achieve a unified representation [12].

B. UNIFIED REPRESENTATION
This section uses the graphical front-end iUML-B of Event-B to obtain a unified representation with LTS, so as to reduce the syntactic and semantic gap between LTS and Event-B, thus realizing the purpose of achieving an appropriate combination. We achieve the combination by taking the following actions:

1) The iUML-B state machine is used as a bridge between Event-B and LTS. We use the iUML-B state machine to express the state transition and time-lapse of an object, and use the combination of these state machines to express the behavioral interaction of concurrent objects. Here, we use the slightly modified BRP protocol model in Abrial's article as a case for an explanation [13]. As shown in Fig. 4, this model contains three objects: sender, receiver, and timer. The sender is responsible for sending data to the channel, the receiver is responsible for receiving the data in the channel, and the timer is used to record the time. We model them separately. When they are combined, some behavioral interactions occur. For example, after the sender sends the data (send_data), at this time, each time the tick_tock event is triggered, the timer t will increase by one time unit. If the receiver receives the data within five time units, it means the reception is successful (rcv_success), so the guard of this event is "t <= 5", which is the prerequisite for the event to occur. However, if there is no reception within five time units, it means the reception failed (rcv_failure). Therefore, the precondition for this event to occur is "t > 5". Regardless of whether the reception is successful or unsuccessful, the timer will be reset, so the action of rcv_success and rcv_failure is "t := 0", which is the result of the event.

In this way, we use the iUML-B state machine to simulate the expression of LTS in system modeling, that is, to separate the objects in the system and model them separately, which
provides a basis for completing the smooth transition from Event-B to LTS.

(2) In the vertical refinement direction, the node refinement of the iUML-B state machine is used to represent the node refinement and edge refinement in the state transition system [14], that is, state decomposition and transition decomposition. Fig.5(a) is an initial state transition model. If we want to refine the event part into an event sequence entr → loop∗ → part (* means this event can occur from 0 to countless times), then we can adopt node refinement and edge refinement respectively to complete.

Node refinement is to replace a state in the abstract model with a super state in the refined model, then add new states within the super state, and add events between the states. As shown in Fig.5(b), the state I in the abstract model is replaced with the super state I, in which a state I with the same name and a new state J are added, two opposite edges are added between state I and J to represent the events entr and part, and add a reflexive edge to the state J to get the required sequence of events and complete the refinement.

Edge refinement is to replace the edge between the source state and the target state with a super state, and add some intermediate states to satisfy the target event sequence. As shown in Fig.5(c), first replace the edge part with the super state I, then add an intermediate state J, and finally use the edge marked by the event to connect the states to complete the refinement.

For a state transition system like LTS, both state-based refinement methods can be uniformly represented by the node refinement of the iUML-B state machine. After node refinement of the abstract model in Fig.6(a), the refined model shown in Fig.6(b) is obtained. We retained the original I state and made it a super state. Since the state with the same name is not allowed in the iUML-B state machine, we add an I node to replace the original I node, and the rest of the states and transitions are constructed as shown in Fig.5(b), that is, the refined model is obtained.

This means that all vertical refinement of LTS can also be achieved by the iUML-B state machine. Since the iUML-B state machine model is consistent with its automatically generated Event-B model, it can ensure the consistency between the event-based vertical refinement process of the Event-B model and the state-based vertical refinement process of the LTS model.

(3) In the horizontal expansion direction, we achieve the purpose of adding new objects in the state transition system by adding new iUML-B state machines, and adding transition edges between states to add constraints between new objects and existing objects. The Press system [13] includes multiple components such as controller and motor. In the abstract model, there is only the controller LTS, and correspondingly only the controller iUML-B state machine, as shown in Fig.7(a0) and Fig.8(a0) respectively. When we want to add a motor object, the first is to add the motor iUML-B state machine, then consider adding the constraints between the motor and the controller. For example, if the controller issues a start command, the motor must respond. To put it bluntly, after the treat_start_motor event occurs, the motor_start event must occur immediately. In order to meet this constraint, we can add a reflexive edge "motor_start" in the "working" state in the controller iUML-B state machine. Other constraints can be added one by one using similar operations, and finally we get an iUML-B model with the same behavior as the LTS model.

The horizontal expansion of LTS can be realized by adding new states and events in iUML-B. Similarly, because the iUML-B state machine model and the Event-B model are consistent in behavior, it also means that the horizontal expansion of the Event-B model is supported, thereby ensuring the consistency of the event-based horizontal expansion process of the Event-B model and the state-based horizontal expansion process of the LTS model.
In the introduction, we mentioned that one of the main problems of the current integrated formal method is that there are gaps in the syntax and semantics of different formal methods, and that the system is modeled from different perspectives by modeling objects separately. There is no guarantee that the final system models will meet the same system requirements. Therefore, we consider first establishing the iUML-B model of the system, and then transforming it to the LTS model.

Since the iUML-B state machine itself is only an expression of the Event-B, which cannot directly obtain the corresponding LTS model. Therefore, we need to transform the iUML-B state machine model into the LTS model. When constructing the LTS behavioral semantic model, the central idea is to treat the Event-B model as a combination of all state variables, and at the same time treat each state variable of the Event-B model as an atomic LTS. Therefore, the LTS behavioral semantics model of the Event-B model is a parallel combination of all atomic LTSs.

According to the description in our previous work [15], we use the following transformation rules:

**Rule 1:** Each variable in the Event-B model is modeled as an atomic LTS, and all possible values for this variable form the state space of this atomic LTS.

**Rule 2:** For each atomic LTS P, if an event e in the Event-B model changes the value of its corresponding variable from \( s_1 \) to \( s_2 \), then we add an element \( (s_1 \rightarrow e \rightarrow s_2) \) to the transition set of this atomic LTS.

The application of these rules can be illustrated by a simple example as shown in Fig.9. When we model and verify the actual system, we first establish its iUML-B state machine model according to the system requirements, secondly use the Rodin tool to automatically generate its Event-B model, then convert it to the corresponding LTS model according to the transformation rules. Finally, the properties of the other two models are verified indirectly by verifying the properties of the LTS model using linear temporal logic (LTL) property expressions constructed according to the system requirements.

### FIGURE 8. Combination of iUML-B state machines

![Combination of iUML-B state machines](image)

### FIGURE 9. System modeling and verification process

![System modeling and verification process](image)

### C. PROOF OF EQUIVALENCE

In order to prove that our method is correct, we give a proof of the bisimulation equivalence between the LTS model and the Event-B model, so as to ensure that the Event-B model generated from iUML-B model is consistent with the LTS model translated from the same iUML-B model. First, we give the definition of bisimulation [16].

**Definition 2.4** (bisimulation equivalence). Let \( LTS_i = (Q_i, \Sigma_i, \Delta_i, q_i^0), \) \( i = 1, 2 \), be labeled transition systems over the actions set \( \Sigma \). A bisimulation for \( (LTS_1, LTS_2) \) is a binary relation \( R \subseteq Q_1 \times Q_2 \) such that

1. for the initial state \( q_1^0 \) and \( q_2^0 \) \((q_1^0, q_2^0) \in R \)
2. for any \((s_1, s_2) \in R \), it holds that
   a. if \( s_1 \xrightarrow{a} s_1' \), then \( s_2 \xrightarrow{a} s_2' \) with \( (s_1', s_2') \in R \) for some \( s_2' \in Q_2 \)
   b. if \( s_2 \xrightarrow{a} s_2' \), then \( s_1 \xrightarrow{a} s_1' \) with \( (s_1', s_2') \in R \) for some \( s_1' \in Q_1 \)

LTS_1 and LTS_2 are bisimulation equivalent, denoted as \( LTS_1 \sim LTS_2 \), if there exists a bisimulation \( R \) for \( (LTS_1, LTS_2) \). The bisimulation equivalence relationship is transmitted, i.e., \( LTS_1 \sim LTS_2 \wedge LTS_2 \sim LTS_3 \Rightarrow LTS_1 \sim LTS_3 \).

In the previous section, we mentioned that a system LTS is a composition of multiple atomic LTSs:

\[
LTS(system) = \bigoplus_{i=1}^{n} AtomicLTS_i
\]
where \( i \) is the sequence number of the atomic LTS, and \( n \) is the total number of atomic LTSs.

The following is the proof process:

1. We first establish an atomic LTS \( \text{AtomicLTS} = \langle Q, \Sigma, \Delta, q, \rangle \) based on the atomic iUML-B state machine. The construction process is as follows.

   a. An “atomic iUML-B state machine” is defined as \( \text{AtomicStm} = \langle \text{Node}, E, \text{Edge}, \text{InitNode} \rangle \), where \( \text{Node} \) represents a set of nodes in the iUML-B state machine; \( E \) represents the set of events that linked on the edges of the iUML-B state machine; \( \text{Edge} \subseteq \text{Node} \times E \times \text{Node} \) represents the set of edges in the iUML-B state machine; \( \text{InitNode} \) represents the initial node of the iUML-B state machine, which is the target node of the edge that is linked to the \text{Initialization} event.

   b. In the process of establishing the atomic LTS, let \( Q = \text{Node} \), \( \Sigma = E \), \( \Delta = \text{Edge} \), \( q = \text{InitNode} \). For example, if there is a node \( s_i \) (edge \( e \)) in atomic iUML-B state machine, a state \( s_j \) (transition \( t \)) is also added in the corresponding atomic LTS. This mapping process is very easy to operate, and we will not explain it further.

   c. Generate the Event-B code from the iUML-B state machine using the automatic code generation tool Rodin.

2. In the following, we prove that the Event-B model generated by \text{AtomicStm} is bisimulation equivalent to the AtomicLTS translated from the same \text{AtomicStm}.

   We define an Event-B model as \( M = \langle V, \text{Event}, \text{Guard}, \text{Action}, V_{\text{init}} \rangle \), where \( V \) represents variables set of \( M \), \( \text{Event} \) represents event set of \( M \), \( \text{Guard} \) represents guard set of \( M \), \( \text{Action} \) represents action set of \( M \), and \( V_{\text{init}} \) represents the initial value set for each element in \( V \). We define the LTS corresponding to the \( M \) as \( \text{LTS}(M) = (Q_M, \Sigma_M, \Delta_M, q_M) \).

   We named an Event-B model generated by \text{AtomicStm} as \( M_A = (V_A, \text{Event}_A, \text{Guard}_A, \text{Action}_A, V_{\text{init}_A}) \). It should be emphasized that at this time there is only one element \( \text{var} \) in \( V_A \), and \( V_{\text{init}_A} \) is the initial value of this element, because an \text{AtomicStm} only describes the change of one variable. \text{Event}_A represents events that modify the value of \( \text{var} \), and \( \text{Guard}_A \) represents those guards that contain \( \text{var} \) in the \text{when} clause of an event. Similarly, \( \text{Action}_A \) represents actions that modify the value of \( \text{var} \). We assume that the type of \( \text{var} \) is \( D \), that is, \( \text{var} \in D \), then the state space of \( \text{var} \) is \( D \).

   Since the code of \( M_A \) is generated by \text{AtomicStm}, we have \( \text{LTS}(\text{AtomicStm}) \sim \text{LTS}(M_A) = (Q_{\text{MA}}, \Sigma_{\text{MA}}, \Delta_{\text{MA}}, q_{\text{MA}}) \). If an Event-B model \( M \) is generated by a combination of multiple atomic iUML-B state machines, then we have

   \[
   \text{LTS}(M) = \bigsqcup_{i=1}^{n} \text{LTS}(M_A_i) \tag{5}
   \]

3. To explain it further.

4. We focus on the following four points for the discussion:

   (a) First, Rodin will generate a variable based on one \text{AtomicStm} and automatically generate a \text{SET} which contains all possible values of this variable. For example, an \text{AtomicStm} named node which contains \( n \) nodes (e.g., \( s_1, s_2, \ldots, s_n \)) will generate

   \[
   \text{partition}(\text{Node}, \{s_1\}, \{s_2\}, \ldots, \{s_n\}) \tag{6}
   \]

   which means \( \text{node} \in \text{Node} \). As we mentioned in (1) (b), \( Q = \text{Node} \). Therefore, the state space \( D \) of variable \text{node} of Event-B model \( M_A \) is equal to \( Q \), and then \( Q_{\text{MA}} = Q \).

   (b) Similarly, \( \Delta_{\text{MA}} = \Delta \).

   (c) Rodin will generate an event named “event_i” in the Event-B model \( M_A \) according to the event “event_i” which has been linked on the edge of iUML-B state machine, and will generate the following code according to each edge from the node \( s_i \) to \( s_j \) (where \( s_i \) and \( s_j \) are the node name):

   \[
   \text{event}_i \equiv \text{when} \quad \text{node} = s_i \quad \text{then} \quad \text{node} = s_j \tag{8}
   \]

   Therefore, for a transition \( s_i \xrightarrow{a} s_j \) in \text{AtomicLTS}, there will be a corresponding transition \( s_i \xrightarrow{\text{event}_i} s_j \) in LTS (\( M_A \)). At the same time, for each execution \( a \) in AtomicLTS, there will be an event \( \text{event}_i \) corresponding to it in the Event-B model \( M_A \). So, we have \( \Delta_{\text{MA}} = \Delta \).

   (d) The reverse mapping process from \( M_A \) to AtomicLTS is similar, and we will not repeat them here.

   (e) We can define a mapping relationship \( R \) so that AtomicLTS and LTS (\( M_A \)) comply with the requirement of bisimulation equivalence. In fact, this \( R \) can be a renamed function, such as \( R(\text{Sender}) = \text{Sender} \).

   \[
   \text{LTS} (\text{AtomicStm}) \sim \text{LTS}(M_A) \sim \text{AtomicLTS} \tag{9}
   \]

Finally, we use the theorem in the literature [16].

**Lemma 1** (congruence w.r.t LTS composition). For labeled transition systems \( \text{LTS}_1 \) and \( \text{LTS}_1' \) over \( \Sigma_1 \), \( \text{LTS}_2 \) and \( \text{LTS}_2' \) over \( \Sigma_2 \), and \( H \subseteq \Sigma_1 \cap \Sigma_2 \), it holds that

\[
\text{LTS}_1 \sim \text{LTS}_1' \quad \text{LTS}_2 \sim \text{LTS}_2'
\]

implies 

\[
\text{LTS}_1 ||_{H} \text{LTS}_1' \sim \text{LTS}_2 ||_{H} \text{LTS}_2'
\]

According to expression (4) and expression (5) and Lemma 1, we have

\[
\bigsqcup_{i=1}^{n} \text{LTS}(M_A) \sim \bigsqcup_{i=1}^{n} \text{AtomicLTS}_i \tag{11}
\]

That is, the LTS model \( \text{LTS}(\text{System}) \) of the system and the LTS model \( \text{LTS}(M) \) of Event-B model obtained according to the mapping rule of (1) are bisimulation equivalent:

\[
\text{LTS}(M) \sim \text{LTS}(\text{system}) \tag{12}
\]

**D. DISCUSSION ABOUT THE ABILITY OF THE METHOD**

Having presented the proposed method, we need to discuss about the ability of the model in modeling and verification. We focus on the following four points for the discussion:

1. LTS does not support model refinement, but the combination with Event-B gives it the ability for refinement.
(2) Use iUML-B state machines to model the behavior of concurrent objects and make up for the defects of Event-B in the expression of control flow.

(3) The iUML-B state machine is used to simulate the process of modeling concurrent objects in LTS. In the simulation, the combination/decomposition of the Event-B model is transformed into the combination/decomposition of the iUML-B state machine to avoid the learning of some complex methods in Event-B, such as Decomposition [17].

(4) iUML-B expresses the interaction between concurrent objects by means of constantly adding new state machines, which solves the problem that Event-B is difficult to express the interaction semantics of concurrent objects.

IV. CASE STUDY

To demonstrate the practicality of the method proposed in the previous section, we use an ARINC653 specification example to describe how it is used in practical applications in this section.

A. INTRODUCTION TO THE ARINC 653 SPECIFICATION

The ARINC653 software specification describes the kernel and related services of the standard APEX (Application Executive) [18]. These services are supported by a variety of safety-critical real-time operating systems (RTOS) for use in avionics. The APEX services specified in ARINC653 Part1 mainly include partition management, process management, time management, intra-partition (inter-process) communication, inter-partition communication, health monitoring, etc. These services determine the highly concurrent features of a Partition Operating System.

- Partition management services: In the ARINC653 specification, only the mode of the partition and the service of obtaining/setting the partition mode are specified. The partition mode includes IDLE, COLD START, WARM START, and NORMAL four working modes. The specification does not specify how the partition mode switching is implemented. Therefore, in the specification level modeling, only the phenomenon needs to be concerned, and there is no need to consider the object that triggers the phenomenon.

- Process management services: these services include creation, suspend, resume, stop, start, get process identifier and state, disable/allow process scheduling, and other functions. When modeling the state transition of the process, it is necessary to consider the mode in which the partition is located. For example, a process can start only when the partition is in NORMAL mode.

- Time management services: these services include services such as delayed waiting, periodic waiting, increasing process time budget, and obtaining current time value, etc.
• Inter-partition communication services: these services are mainly composed of queue port services and sampling port services, including services such as creation, read/write, and status acquisition of these two types of ports.
• Intra-partition communication services: the intra-partition communication specifies various services that can be used for synchronization and mutual exclusion between processes within a partition. It is mainly designed around four objects, including buffers, blackboards, semaphores, and events.
• Health monitoring services: the health monitoring specifies the fault response and processing mechanism of the partition operating system, including three monitoring levels: system level, module level, and partition level.

B. ARINC653 SPECIFICATION MODELING
In this section, we use our method to model the ARINC653 specification. Separate the concurrent objects that produce phenomena in the specification in the horizontal direction and model the interaction between these objects. In the vertical direction, we gradually refine the behavior of concurrent objects to ensure the behavioral consistency between the concrete model and the abstract model.

In order to reach the goal of making concurrent objects separated, we divide the objects involved in the specification into partitions, processes, inter-partition communication objects (queue port and sampling port), intra-partition communication objects (buffer, blackboard, event, semaphore), and use the method proposed in the previous section to complete the modeling of ARINC653 specification model, as shown in Fig.10.

We start with building the abstract model that only contains partitions, and then take four refinement steps to complete the modeling of all the required services except the health monitoring service. Fig.10 shows the final specification containing the models of partitions, processes, intra-partition communication objects and inter-partition communication objects.

Next, in this paper, we use partitions and processes as an example to describe our modeling process.

1) PARTITION
In the abstract model M0 of the system, we introduce the initial partition model, including the established partition iUML-B state machine model, the automatically generated partition Event-B model, and the transformed partition LTS model, as shown in Fig.11. It should be mentioned that in order to more easily and effectively simulate the model, we define two partitions in the model.

After a partition is created (create_partition), the partition performs mode transition (partition_mode_transition). However, in the initial model, the mode of the partition is only modeled as one state (Partition_Mode), paving the way for being split into multiple partition mode states in subsequent refinements.

The initial model is refined layer by layer to obtain the final partition iUML-B state machine model and the transformed LTS model, as shown in Fig.12 and Fig.13, respectively. Since the code of the Event-B model is too large, this paper will not present it. The model consists of two parts, Partition_Schedule controls the scheduling of the partition, and Partition_Mode shows the mode transition of a single partition.

At any one time, only one partition can be scheduled. As shown in Fig.12 (a), when a partition is scheduled for the first time, the scheduler schedules the partition to the running state by the partition_start event. When the time window allocated to the partition ends, the scheduler sets it to the suspended state by activating the partition_suspend event, and waits for the next time window of the partition to arrive and then schedules the partition again by the partition_resume event.

In Fig.12 (b), one partition has four modes: WARM_START, COLD_START, NORMAL, and IDLE.
After a partition is scheduled in a specified time window, it can be transformed between four modes in different ways. For example, when the partition is in COLD_START or WARM_START mode, it can be transformed into NORMAL mode by the `set_partition_mode_to_normal` event.

2) PROCESS
The operating system views the execution of a process as a transition between a series of continuous process states. In order to complete the modeling of the process, we introduce the initial model of the process in the first-level refined model $M_1$ and obtain the following iUML-B model, Event-B model, and LTS model, as shown in Fig.14. For the same reason, we only define two processes in the model.

The final process iUML-B state machine model and LTS model obtained by refinement describe the more specific process state and state transitions conforming to the partition mode, as shown in Fig.15 and Fig.16, respectively.

3) COMBINATION OF PARTITION AND PROCESS
After completing the construction of the partition model and the process model respectively, we need to consider how to add the constraints brought about by their combination in the model. For example, only when the partition is in Normal mode, can the process perform the startup operation. In other words, all events related to process startup in the process model can only occur when the partition is in Normal mode.

In order to meet this condition, we need to add the reflexive edges of these events to the Normal state in the partition model, as shown in Fig.17. The remaining constraints can be added one by one using similar operations until the correct behavior model is constructed.
C. PROPERTY VERIFICATION OF THE ARINC 653 SPECIFICATION MODEL

Labeled Transition Systems Analyzer (LTSA)\(^1\) is a tool that can be used to check whether the specification of a concurrent system satisfies the required behavioral properties. It models LTS and properties as state machines, and then performs compositional reachability analysis between them to exhaustively search for violations of the desired properties. In this section, we use LTSA to perform the written LTL property expressions for property verification on the established system LTS model, so as to ensure that the model we built satisfies the requirements of the system.

Table 1 gives the informal description of some properties that need to be verified, and their formalization is given in Table 2. The symbols "□" and "◇" in Table 2 represent "Always" and "Final" in LTL, respectively, while the symbols "¬" and "→" represent "negative" and "implication" in propositional logic, respectively.

We use LTSA to combine these property expressions with the constructed system LTS model to verify the behavior of the system. If there is a behavior in the model that violates these properties, LTSA will give the sequence of events corresponding to the behavior. On the contrary, it means that the model satisfies these properties.

\(^1\) https://www.doc.ic.ac.uk/ltsa/
TABLE 2. The formal description of properties to be verified in the system model

| The serial number of requirements | The formal description of requirements |
|-----------------------------------|---------------------------------------|
| SAF_1                             | □((Process[1].send_buffer → \hspace{2pt} \hspace{2pt} \hspace{2pt} Process[2].send_buffer)) |
| SAF_2                             | □((Partition[1].schedule ∧ \hspace{2pt} \hspace{2pt} \hspace{2pt} Partition[2].schedule)) |
| LIVE_3                            | □(Process[i].req_busy_resource→ \hspace{2pt} \hspace{2pt} \hspace{2pt} (Process[i].resource_become_available ∨ \hspace{2pt} \hspace{2pt} \hspace{2pt} Process[i].time_out)) |
| LIVE_4                            | □(Process[i].send_buffer_withfull→ \hspace{2pt} \hspace{2pt} \hspace{2pt} (Process[i].receive_buffer_needwakeupsendproc ∨ \hspace{2pt} \hspace{2pt} \hspace{2pt} Process[i].time_out)) |
| LIVE_5                            | □(Process[i].start→ \hspace{2pt} \hspace{2pt} \hspace{2pt} (Process[i].periodicprocess_finished ∨ \hspace{2pt} \hspace{2pt} \hspace{2pt} Process[i].aperiodicprocess_finished)) |
| LIVE_6                            | □(Process[i].timed_wait→ \hspace{2pt} \hspace{2pt} \hspace{2pt} (Process[i].time_out)) |

After verification, we do not get any information that violates the properties. In other words, as far as the current situation is concerned, the model we have established is no error, but it does not mean that it is correct. However, what we can confirm is that as long as we write more property expressions and no errors are reported after verification, the reliability of the system model will be higher.

V. DISCUSSION

In this section, we compare our work with the work of Zhao et al., who used Event-B alone to establish the system model of the ARINC653 specification in Section 4 [19]. The simulation results on the Rodin platform show that in each layer, the event traces of the model obtained using our method are consistent with theirs. In their work, all Event-B code is hand-written by the research team, which involves a large number of variables, invariants, guards, actions and other elements. Such a huge workload requires a lot of time and energy, and text-based programming also makes the code extremely prone to errors.

In our work, we first use the states and edges linked with events in the iUML-B to build system models, and use the pattern state machine proposed in our previous work to improve reuse rate and programming efficiency [20], and then use Rodin tools to automatically generate the Event-B model. The growth trend of the main elements in the iUML-B model with the increase of the refinement layer is shown in Figure 19. Since one edge can link multiple events, the number of edges is greatly reduced, and modeling becomes easier [20]. Although the variables and invariants in the code still need to be written manually, the guards and actions with the highest proportion can be automatically generated. In other words, the percentage of manual coding we can save is roughly (833+390)/1374 = 89%.

Another benefit of our method is that the control flow and behavioral interaction become visible. Compared with Event-B’s event-based textual representation, iUML-B state machine can express and analyze the event order of the system easily.

Finally, we pointed out in [21] that we choose LTS as the behavioral semantic model of Event-B, and convert the Event-B model to the corresponding LTS model, so that the behavioral properties of the Event-B model can be analyzed and verified. From the point of view of behavioral semantic verification of event-based method, this is an advantage.

VI. RELATED WORK

In order to solve specific problems of specific systems, researchers have combined Event-B with multiple methods. To analyze and model the behavior of the system formally, Kenza et al. combined UML and Event-B [22]. In their work, UML is used to model the system from an informal specification. Then, the UML model is transformed into Event-B model, and proceed with the formal verification and numerical analysis.
validation using B tools. However, from the basic knowledge in the second section, iUML-B plugin, which is a UML-like and graphical front-end for Event-B, keeps most of the features in UML. Moreover, the model established using iUML-B can automatically generate the corresponding Event-B model, which is more efficient than the manual combination of UML and Event-B.

Dghaym proposed an approach to workflow modelling that addresses both control flow and data handling [23]. ERS (Event refinement structure) is used for control flow, while Event-B mathematical notation supports the data handling. This combination not only benefits from the verification capabilities of Event-B and the graphical nature of ERS, but also benefits from the refinement and hierarchy to achieve incremental modelling. However, the combination between them lacks the means to verify the behavioral properties of the system. In contrast, we use LTS and iUML-B state machines to express the control flow and behavioral interaction of the Event-B model intuitively, and verify its behavioral properties easily.

On the premise of comprehensively considering the advantages and disadvantages of LTS and Event-B, we combine the two in the system model refinement process to give full play to the advantages of both, that is, LTS has strong expressive ability for control flow and behavioral interaction, and Event-B supports gradual refinement and so on. The method given in this paper converts the Event-B model to the LTS model, which to a certain extent achieves the goal of expressing the behavioral interaction of concurrent systems. However, it can be seen that there is no distinction between input actions and output actions between concurrent objects, which brings us certain difficulties in analyzing the behavioral interaction relationship of concurrent objects.

VII. CONCLUSIONS AND FUTURE WORK

This paper proposes a combination of Event-B and LTS to provide an integrated method for modeling and verifying concurrent systems. In the process of combination, we analyze in detail the difference and association between the event-based refinement process of the Event-B model and the state-based refinement process of the LTS model during the construction of the system model. Then, we propose to use the graphical front-end iUML-B of Event-B to obtain a unified representation with LTS to reduce the syntactic and semantic gap between LTS and Event-B, thus realizing the purpose of achieving an appropriate combination. This paper takes the ARINC653 specification as an example to demonstrate the process of using this method to model and verify the system. The case study shows that our integrated method can be effectively used to verify the various properties of the system and improves the reliability of the system model. To a large extent, it has overcome the shortcomings of the single formal method in the concurrent system modeling and verification process.

Despite the progress we have made in our work, there are still some issues remaining to be addressed. The transformation from Event-B model to LTS model is based on manual transformation, which only uses the transformation rules we have defined. Further research needs to be carried out to investigate how the transformation can be done automatically. The next step is to study the automatic transformation mechanism and technology, and to develop the corresponding automatic transformation tools.

REFERENCES

[1] N. Plat, J. van Katwijk, and H. Toetenel, "Application and benefits of formal methods in software development," Software Engineering Journal, vol. 7, no. 5, pp. 335-346, 1992.
[2] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. Melo de Sousa, "An Overview of Formal Methods Tools and Techniques," Rigorous Software Development, pp. 15-44, 2011.
[3] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," ACM Computing Surveys (CSUR), vol. 28, pp. 626-643, 1996.
[4] S. Schneider, H. Treharne, and H. Wehrheim, "The behavioural semantics of Event-B refinement," Formal Aspects of Computing, vol. 26, pp. 251-280, 2014.
[5] M. Farrell, R. Monahan, and J. F. Power, "Combining Event-B and CSP: An Institution Theoretic approach to Interoperability," in International Conference on Formal Engineering Methods, 2017, pp. 140-156.
[6] S. Schneider, H. Treharne, and H. Wehrheim, "Stepwise Refinement in Event-B CSP: Part 1: Safety," Department of Computing University of Surrey, 2011.
[7] D. Ramchandani, A. Russo, and D. Alrajeh, "Refining Labelled transition systems using scenario-based specifications," Imperial College London, Department of Computing, 2009.
[8] N. D'Ippolito, "Synthesis of event-based controllers for software engineering," Citeseer, 2013.
[9] R. Gorrieri, Labeled Transition Systems: Springer International Publishing, pp. 15-34, 2017.
[10] C. Snook and M. Butler, "UML-B:Formal modeling and design aided by UML," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 15, pp. 92-122, 2006.
[11] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," International journal on software tools for technology transfer, vol. 12, pp. 447-466, 2010.
[12] D. H. Vu, A. H. Truong, Y. Chiba, and T. Aoki, "Automated testing reactive systems from Event-B model," in 2017 4th NAFOSTED Conference on Information and Computer Science, 2017, pp. 207-212.
[13] J. R. Abrial, Modeling in Event-B: System and Software Engineering: Cambridge University Press, 2010.
[14] S. Hallerstede, "Structured event-b models and proofs," in International Conference on Abstract State Machines, Alloy, B and Z, 2010, pp. 273-286.
[15] H. Peng, C. Du, L. Rao, and Z. Liu, "LTS semantics model of Event-B synchronization control flow design patterns," Journal of Information Processing Systems, vol. 15, pp. 570-592, 2019.
[16] C. Baier and J.-P. Katoen, *Principles of model checking*: MIT press, 2008.

[17] T. S. Hoang and J. R. Abrial, "Event-B Decomposition for Parallel Programs," in *International Conference on Abstract State Machines, Alloy, B and Z*, 2011, pp. 319-333.

[18] A. Oliveira Gomes, "Formal specification of the arinc 653 architecture using circus," University of York, 2012.

[19] Y. Zhao, Z. Yang, D. Sanán, and Y. Liu, "Event-based formalization of safety-critical operating system standards: An experience report on ARINC 653 using Event-B," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 281-292.

[20] H. Peng, C. Du, L. Rao, and Z. Liu, "Modelling the Embedded Control System Using iUML-B Pattern State Machine," *Journal of Control Science and Engineering*, vol. 2018, 2018.

[21] H. Peng, C. Du, L. Rao, and F. Chen, "A LTS approach to control in event-B," *Scientific Programming*, vol. 2018, 2018.

[22] K. Kraibi, R. B. Ayed, S. Collart-Dutillieu, P. Bon, and D. Petit, "Analysis and Formal Modeling of Systems Behavior Using UML/Event-B," *Journal of Communications*, vol. 14, pp. 980-986, 2019.

[23] D. Dghaym, "Extending the ERS approach for workflow modelling in Event-B," University of Southampton, 2017.

**LEI RAO** is a doctoral student in the School of Advanced Science and Engineering at Hiroshima University. He received the M.D. in Software Engineering from Northwestern Polytechnical University, China in 2020. His current research interests include Software Modeling and Verification, Formal Engineering Methods, Formal Methods, and Fault Prevention for Software Construction.

**SHAOYING LIU** is a Professor of Software Engineering at Hiroshima University, Japan. He received the Ph.D in Computer Science from the University of Manchester, U.K in 1992. His research interests include Formal Methods and Formal Engineering Methods for Software Development, Specification Verification and Validation, Specification-based Program Inspection, Automatic Specification-based Testing, Testing-Based Formal Verification, and Intelligent Software Engineering Environments. He has published a book entitled "Formal Engineering for Industrial Software Development" with Springer-Verlag, twelve edited conference proceedings, and over 200 academic papers in refereed journals and international conferences. He is an Associate Editor for IEEE Transactions on Reliability and Innovations in Systems and Software Engineering, respectively. He is IEEE Fellow, BCS Fellow and member of JSSST, IPSJ, and IEICE.

**HAN PENG** is an Associate Professor at Xi'an Aeronautical University. He received the Ph.D. in Computer Science and Technology from Northwestern Polytechnical University, China in 2019. His research interests include Formal Methods, Formal Verification of Operating Systems, and Airborne System Modeling and Verification. He has published 6 papers in international conferences and refereed journals.