Event-B Control Flow Modeling based on iUML-B State Machine

Han Peng
College of Computer Science
Northwestern Polytechnical University
Xi'an, China
hansbeng2016@gmail.com

Chenglie Du
College of Computer Science
Northwestern Polytechnical University
Xi'an, China
Ducl@nwpu.edu.cn

Haobin Wang
College of Computer Science
Xi'an Aeronautical University
Xi'an, China
z83054539@163.com

***Abstract*—There are some limitations in expressing the order of actions using Event-B. To solve this problem, the event refinement structure method (ERS) is proposed to facilitate modeling of the system's control flow. However, the event refinement structure cannot be translated to a behavior semantic model such as the communication sequence process (CSP) or labeled transition system (LTS) directly, thus it is not convenient for engineers to verify the control flow. In this paper, we first propose a general method to model the control flow of the Event-B model with various iUML-B state machines. Then we prove by simulation that the event trace of the iUML-B state machine is the same as that of the event refinement structure method. Finally, we use a case study of a lift control system to prove the practicality of our method.**

Keywords—Event-B; control flow modeling; iUML-B state machine; atomicity decomposition; event refinement structure

I. INTRODUCTION

Event-B [1] is a formal method that evolved from B method [2] and action system [3]. It uses set theory and first order logic to model the system, and is applied in different fields including control systems. In an Event-B model, the behavior of the system is expressed in terms of the sequence of events. For small systems that contain only a small number of events, one can observe the behavior of the system by simulation. However, when the number of events in the Event-B model is relatively large, it is difficult to accurately observe and predict the behavior of the system. The root cause of this problem is that Event-B lacks a mechanism for explicitly expressing system control flows.

To solve this problem, researchers have proposed a number of methods to model the Event-B control flow. Some of the outstanding methods include the CSP||B method [4], the flow method [5] and the event refinement structure method (ERS) [6]-[9]. The CSP||B method expresses the control flow of the system using communication sequence process (CSP), which is a formal language that is too difficult for most engineers to learn. The Flow method uses a new set of graphical symbols to express the order between events, which in fact increases the difficulty of learning. Engineers need an intuitive, non-formal or semi-formal symbolic system that can easily model the control flow of the system and translate the control flow model into a formal behavior model. ERS method meets this demand to a large extent; it uses a tree structure based on the Jackson structure diagram (JSD) to express the relationship between the abstract event and the subsequent concrete events as well as the order of concrete events. This tree structure can generate the corresponding Event-B code. The ERS method also presents eight atomicity decomposition patterns to express the relationships between events, which make it very suitable for Event-B modeling.

However, there are some limitations in the ERS method. First of all, it uses an undirected graph to express the order of events, which is easily misinterpreted. Secondly, although the tree structure of ERS is very good in expressing the relationship between abstract events and concrete events (vertical direction), it cannot express the control flow of the system in the same refinement level (horizontal direction) explicitly. Thirdly, the tree structure of ERS cannot be directly converted into a formal semantic model, such as CSP or Labeled Transition System (LTS).The reason for this problem is obvious, JSD diagrams can express the static relationship between entities well, but cannot express the control flow explicitly. In contrast, the state diagram of Unified Modeling Language (UML) is better at expressing the system's control flow.

iUML-B [10], [11] is a graphical "front end" of Event-B, using "UML-like" class diagram and state machine to describe the system's states and actions. The graphical model drawing by iUML-B can generate Event-B code directly on the Rodin [12] platform, and the generated code can be filled into the Event-B machine automatically, the proof obligations can be discharged automatically as well. Fig. 1 shows an example of iUML-B state machine.

In this paper, we propose a method to model control flow of system using various iUML-B state machines, which has the following advantages:

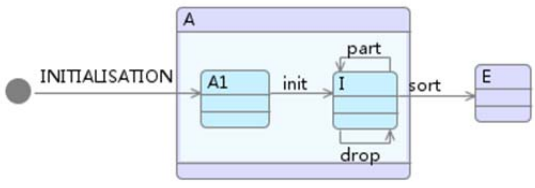

Fig. 1. Example of iUML-B state machine.

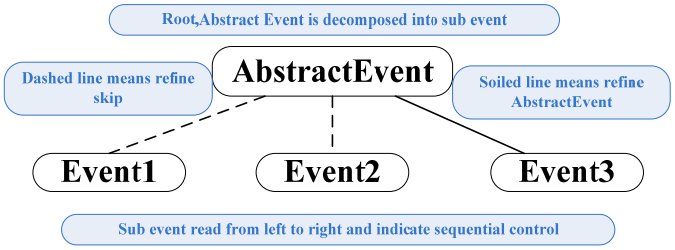

Fig. 2. principles of atomicity decomposition.

- iUML-B state machine is a directed graph, so it can express the order of events explicitly;
- iUML-B state machine is a variant of the UML state diagram, so it is easily understood by engineers;
- iUML-B state machine can be easily converted to LTS to verify its behavior properties.

We use the iUML-B state machine to construct the *And*, *Sequence*, *Xor*, and *Loop* decomposition patterns of ERS method, and demonstrate that the order of events obtained by this method is the same as that of the ERS method. We illustrate our approach on a lift control system case study.

The remainder of this paper is organized as follows. Section II presents preliminary details for ERS method and the problem in it. In Section III, we present our approach for modeling four decomposition patterns of ERS using iUML-B state machine. We also verify the correctness of this method by simulation. In Section IV, we model the control flow of a lift control system using iUML-B state machine. Section V discusses our approach and compares the results of this work with existing work. Section VI presents related work. In Section VII, we draw some conclusions and provide some perspectives related to our future research.

II. BACKGROUND

A. Overview of ERS Methods

The ERS method, also known as atomicity decomposition method, was first proposed by Butler. It uses the tree structure to express the patterns of event refinement structure, as Fig. 3 shows.

The *AbstractEvent* of the root node is an abstract atomic event. At the bottom of the graph, this abstract event is split into three concrete events *Event1*, *Event2* and *Event3*, where *Event1* and *Event2* refine the *skip*, and *Event3* refines the *AbstractEvent*. The trace of concrete events is:

<Event1, Event2, Event3>.

In addition to the *Sequence* pattern shown in Fig. 3, the ERS method also includes *And*, *Xor* and *Loop* patterns. The meaning and event order of these patterns is:

1) And pattern

It indicates that an abstract event is split into many concrete events that can occur in any interleaved manner. For example, if abstract event is split into two concrete events *AndEvent1* and *AndEvent2*, then the event order can be:

<Event1, AndEvent1, AndEvent2, Event3>

or

<Event1, AndEvent2, AndEvent1, Event3>.

It should be noted that in *And* pattern and the remaining several patterns, *Event1* and *Event3* are not the result of the decomposition of abstract events, but only two auxiliary events.

2) Xor pattern

It indicates that an abstract event is split into many concrete events. Only one of these events can be executed. For example, if an abstract event is split into two concrete events *XorEvent1* and *XorEvent2*, then the event order of concrete events can be:

<Event1, XorEvent1, Event3> or

<Event1, XorEvent2, Event3>

3) Loop pattern

It indicates that an abstract event is refined into one concrete event that can occur many times. For example, if abstract event is refined into one concrete event *LoopEvent*, then the event order of concrete event can be (we use '*' to represent an event that can occur 0 or more times):

<Event1, (LoopEvent), Event3>.*

B. The Problem of ERS Method

The advantage of ERS method lies mainly in the graphical representation of the event decomposition architecture, as shown in Fig. 3. This method facilitates the exploration and evaluation of various event decomposition strategies and refinement strategies for the entire system. At the same time, one can deduce the event order of the system by traversing each leaf node of the tree structure so that they can analyze whether the behavior of the system is consistent with the requirement.

However, ERS method also has the following problems:

- The event trace in the atomicity decomposition diagram is not obvious. For example, in Fig. 3, if one were to analyze the trace of the events in the lowest level, he would need to trace back to the top of the abstract root node, because the bottom layer does not provide this information.
- Alkhammash [7] used ERS method to model the control flow of the system and used UML-B to model the data-oriented requirement, which separated the control flow model and the functional model of the system. The idea of this method is correct because it achieved the separation of concerns. But in the final step, people will have to compose the functional model $Function_M$ and the control flow model $Controller_M$ to get the system model $System_M = (Function_M \parallel Controller_M)$. This step will introduce unexpected changes in the behavior of the system.
- The basic modeling elements of the ERS method are events and connections. This modeling form cannot be directly transformed into a behavioral semantic model,

Fig. 3. Atomicity decomposition architecture.

such as the CSP or LTS model, and is therefore not conducive to the modeling and analysis of behavior properties (e.g., safety and liveliness properties in concurrent situations).

III. EXPRESSING ERS PATTENRS BY IUML-B

In this section we will use the iUML-B state machine to establish four atomicity decomposition patterns of the ERS method, including *Sequence* pattern, *Loop* pattern, *And* pattern, and *Xor* pattern. We first give a general event decomposition method, and then present iUML-B state machine representation for each pattern and analyze its event trace by simulation. The simulation results show that the event trace of the iUML-B state machine is the same as that of the ERS method.

A. *Atomicity Decomposition Model Based on iUML-B State Machine*

1) *General Method*

The general approach to get atomicity decomposition using the iUML-B state machine is described as follows:

a) An abstract state machine is used to describe the event trace of the abstract model.

b) In the refinement model, a node in the abstract state machine is changed into a super node of the concrete state machine.

c) New states and events are added in the super node as refinement events in the next level.

d) The refinement event is linked to the edge of the new state machine, and ensures that its event trace meets the requirements for decomposition and refinement.

According to the above general method, we first show the initial abstract state machine model. Then we show the iUML-B state machine representation of the four decomposition patterns.

The initial abstract state machine model is shown in Fig. 4.

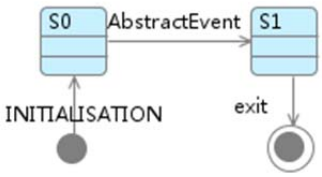

Fig. 4. Abstract state machine.

The corresponding code of event is:

```
AbstractEvent ≙
STATUS
Ordinary
WHEN
        @Isin_S_0  :  S_0 = TRUE
THEN
        @leave_S_0  :  S_0 ≔ FALSE
        @Enter_S_1  :  S_1 ≔ TRUE
END
```

The event trace of abstract model is

<AbstractEvent>.

2) *Sequence pattern*

In order to get the correct event trace, in the Sequence decomposition pattern, we change the *S0* state of the abstract state machine to super node and add three sub-states *s0_1*, *s0_2*, *s0_3* and two events *Event1*, *Event2* in its nested state machine, and then let *Event3* refine the abstract event *AbstractEvent*, as the Fig. 5 shows.

The resulting code of events and invariants are:

```
Event1 ≙
STATUS
Ordinary
WHEN
        @Isin_S0_1  :  S0_1 = TRUE
THEN
        @leave_S0_1  :  S0_1 ≔ FALSE
        @Enter_S0_2  :  S0_2 ≔ TRUE
END
```

```
Event2 ≙
STATUS
Ordinary
WHEN
        @Isin_S0_2  :  S0_2 = TRUE
THEN
        @leave_S0_2  :  S0_2 ≔ FALSE
        @Enter_S0_3  :  S0_3 ≔ TRUE
END
```

```
Event3 ≙
extended
STATUS
Ordinary
REFINES
AbstractEvent
WHEN
@isin_S0  :  S0 = TRUE
@Isin_S0_3  :  S0_3 = TRUE
THEN
@leave_S0  :  S0 ≔ FALSE
@enter_S1  :  S1 ≔ TRUE
@leave_S0_3  :  S0_3 ≔ FALSEEND
```

```
@S0_1_substateof_S0  :  (S0_1 = TRUE) ⇒ (S0 = TRUE)
@S0_2_substateof_S0  :  (S0_2 = TRUE) ⇒ (S0 = TRUE)
@S0_3_substateof_S0  :  (S0_3 = TRUE) ⇒ (S0 = TRUE)
```

The simulation result shows that the event trace of this pattern is:

<Event1, Event2, Event3>.

3) *Loop pattern*

Fig. 5. Sequence decomposition pattern.

In order to get the loop event, in the *Loop* decomposition pattern, we change the *S0* state of the abstract state machine to the super node, and add two sub-states of *s0_1*, *s0_2* and *Event1* and one event *LoopEvent* in its sub state machine, where *LoopEvent* is the reflex edge of state *s0_2*, and then let *Event3* refine the abstract event *AbstractEvent*, as Fig. 6 shows:

The resulting code of events and invariants are:

Event1 ≙
STATUS
Ordinary
WHEN
@Isin_S0_1 : S0_1 = TRUE
THEN
@leave_S0_1 : S0_1 ≔ FALSE
@Enter_S0_2 : S0_2 ≔ TRUE
END

LoopEvent2 ≙
STATUS
Ordinary
WHEN
@Isin_S0_2 : S0_2 = TRUE
THEN
Skip
END

Event3 ≙
extended
STATUS
Ordinary
REFINES
AbstractEvent
WHEN
@isin_S0 : S0 = TRUE
@Isin_S0_3 : S0_3 = TRUE
THEN
@leave_S0 : S0 ≔ FALSE
@enter_S1 : S1 ≔ TRUE
@Leave_S0_3 : S0_3 ≔ FALSE
END

@S0_1_substateof_S0 : (S0_1 = TRUE) ⇒ (S0 = TRUE)
@S0_2_substateof_S0 : (S0_2 = TRUE) ⇒ (S0 = TRUE)

The event trace obtained by simulation is:

$$<Event1, (LoopEvent)^{*}, Event3>.$$

Loop pattern can also be implemented using the pseudo-state node of iUML-B state machine, as shown in Fig. 7.

Note that the *Loop* pattern shown in Fig. 6 indicates that *LoopEvent* can occur "0 times or more times", while the *Loop* pattern shown in Fig. 7 ensures that *LoopEvent* occurs at least once. Thus, the event trace of the state machine shown in Fig. 7 is ('+'an event that can occur one or more times):

$$<Event1, (LoopEvent)^{+}, Event3>.$$

4) And pattern

In the *And* pattern, we first change the *S0* state of the abstract state machine to super node and add two sub-states *s01*, *s02* and one event *Event1* to its sub state machine. Then state *S02* is split into two orthogonal state machines into which sub-states and events are added. Finally, let *Event3* refine the abstract event *AbstractEvent*, as Fig. 8 shows.

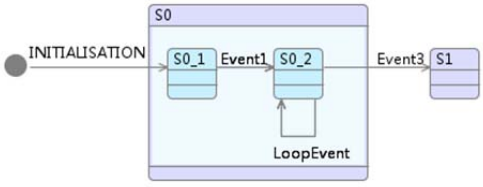

Fig. 6. Loop decomposition pattern 1.

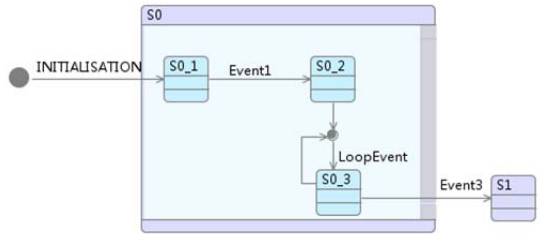

Fig. 7. Loop decomposition pattern 2.

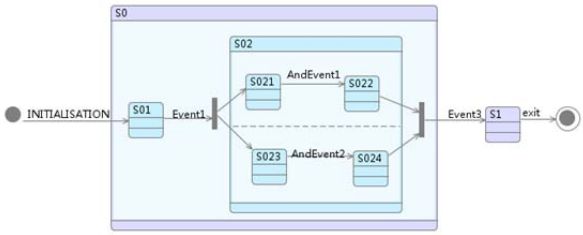

Fig. 8. And decomposition pattern.

The resulting code of event and invariant are:

Event1 ≙
STATUS
Ordinary
WHEN
@Isin_S01 : S01 = TRUE
THEN
@leave_S01 : S01 ≔ FALSE
@enter_S021 : S021 ≔ TRUE
@enter_S02 : S02 ≔ TRUE
@Enter_S023 : S023 ≔ TRUE
END

Event3 ≙
STATUS
Ordinary
REFINES
AbstractEvent
WHEN
@isin_S024 : S024 = TRUE
@Isin_S022 : S022 = TRUE
THEN
@leave_S024 : S024 ≔ FALSE
@leave_S02 : S02 ≔ FALSE
@leave_S0 : S0 ≔ FALSE
@leave_S022 : S022 ≔ FALSE
@Enter_S1 : S1 ≔ TRUE
END

@S02_substateof_S0 : (S02 = TRUE) ⇒ (S0 = TRUE)
@S021_substateof_S02 : (S021 = TRUE) ⇒ (S02 = TRUE)
@S022_substateof_S02 : (S022 = TRUE) ⇒ (S02 = TRUE)
@S023_substateof_S02 : (S023 = TRUE) ⇒ (S02 = TRUE)
@S024_substateof_S02 : (S024 = TRUE) ⇒ (S02 = TRUE)
@S01_substateof_S0 : (S01 = TRUE) ⇒ (S0 = TRUE)

The event traces obtained by simulation are:

<Event1, AndEvent1, AndEvent2, Event3> or

<Event1, AndEvent2, AndEvent1, Event3>.

There is no other possible event trace in this state machine.

5) Xor pattern

In the *Xor* decomposition pattern, we change the *S0* state of the abstract state machine to super node and add three sub states *s0_1*, *s0_2*, *s0_3* and three events *Event1*, *XorEvent1* and *XorEvent2* into the sub state machine. Let *XorEvent1* and *XorEvent2* have the same source state but distinct target states, and finally let *Event3* refine the abstract event *AbstractEvent*, as Fig. 9 shows.

The resulting code of events and invariants are:

```
Event1 ≙
STATUS
Ordinary
WHEN
        @Isin_S0_1  :  S0_1 = TRUE
THEN
        @leave_S0_1  :  S0_1 ≔ FALSE
        @Enter_S0_2  :  S0_2 ≔ TRUE
END
```

```
XorEvent1 ≙
STATUS
Ordinary
WHEN
        @Isin_S0_2  :  (S0_2 = TRUE)
THEN
        @leave_S0_2  :  S0_2 ≔ FALSE
        @enter_S0_3  :  S0_3 ≔ TRUE
END
```

```
XorEvent2 ≙
STATUS
Ordinary
WHEN
        @Isin_S0_2  :  (S0_2 = TRUE)
THEN
        @leave_S0_2  :  S0_2 ≔ FALSE
        @Enter_S0_4  :  S0_4 ≔ TRUE
END
```

The event traces obtained by simulation are:

<Event1, XorEvent1, Event3> or

<Event1, XorEvent2, Event3>

There will be no other possible event traces in this state machine.

B. Summary

We implemented four atomicity decomposition patterns of ERS method using the iUML-B state machine. The simulation results show that the event traces obtained by the iUML-B state machine are exactly the same as that of the ERS method.

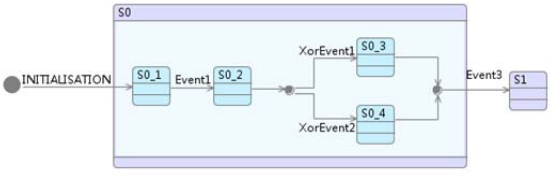

Fig. 9. Xor decomposition pattern 2.

IV. Case Study

In this section, we use the general method proposed in Section III to model the lift control system of Alkhammash [7] with the iUML-B state machine. The complete requirements for the lift system can be found in Alkhammash's article. In this paper, we only care about the events added during the system refinement and the constraints imposed by the atomicity decomposition on event order.

A. The Original Decomposition Process of the Lift System

1) Control Flow Requirements of Lift System

The control flow requirements for the lift system are shown in Table 1, which is cited from Alkhammash's article [7]. Alkhammash called them flow requirements, and described these requirements using ERS method, as shown in Fig. 2 of Section II.

2) Event Execution Trace Specified by ERS Method

The event refinement structure shown in Fig. 3 specifies the following requirements of system event orders:

- In the top-level abstraction model, the *AbstractLiftStop* event and the *AbstractLiftMove* event occur alternatively, that is to say, two or more *AbstractLiftStop* events cannot occur continuously. For the *AbstractLiftMove* event, this constraint is the same.
- In the first layer refinement model, the LiftStop1 event refined the *AbstractLiftStop* event of the abstract model. The event *LiftStop1* can be followed by the *OpenLiftDoor* event or the *NotOpenLiftDoor* event. The *Xor* in Fig. 3 indicates that these two events cannot occur at same time. The leaf nodes order of the tree structure indicates that the *LiftStop1* event must occur before the *OpenLiftDoor* event or the *NotOpenLiftDoor* event (requirement LIFT8). Similarly, the *LiftMove* event refined the *AbstractLiftMove* event of the abstract model, and it must occur after the *CloseLiftDoor* event (requirement LIFT7).
- The third layer of refinement introduces a *RequestFloor* event that can occur more than one time. *RequestFloor* is put before *LiftStop2*, which indicates that only after the passengers have made a choice, the elevator will stop at the required floor (requirement LIFT9).

B. Control Flow Modeling Based on iUML-B State Machine

We use the iUML-B state machine to model the control flow of lift control system. The original model of the system is shown in Fig.10.

1) Abstract model.

The top-level abstract model is shown in Fig. 10. Since there is no interaction between the elevator and the door, it is only necessary to describe the relationship between *LiftMove* event and the *LiftStop* event. This relationship has been modeled in the liftStatemachine0.

2) First refinement

Three events are introduced in the first refinement to express the behavior of the door, namely, *OpenLiftDoor*, *CloseLiftDoor* and *NotOpenLiftDoor*, as shown in Fig. 11.

TABLE I. DESCRIPTION OF FLOW REQUIREMENTS

Flow Requirements	**Example Description**
Sequencing requirements	LIFT7-The floor door closes before the lift is allowed to move
Selection requirements	LIFT8-If a lift is stopped then the floor door for that lift may be open. In this requirement the lift door can be either opened or left closed when the lift is stopped.
Repetition requirements	LIFT9-There might be more than one external floor request in a particular floor, the lift will respond to them (stop) only once

We use another state machine, *FlowStateMachine1*, to constrain the events trace of the system, as shown in Fig. 12. We use the pseudo-state to express the *Xor* relationship between *OpenLiftDoor* and *NotOpenLiftDoor*, and put them into a super state. Then we make the *LiftStop* event to be an ingoing event of super-state, which requires the *LiftStop* event to be executed just before *OpenLiftDoor* or *NotOpenLiftDoor*. Similarly, we make the *LiftMove* event to be the outgoing edge of the super-state, which specifies that the *LiftMove* can be executed only after these two events.

FlowStateMachine1 generates the following control flow code:

```
MoveLift ≙
extended
STATUS
Ordinary
REFINES
MoveLift
WHEN
        @isin_stopped  :  lift = stopped
        @isin_LiftStoped  :  LiftStoped = TRUE
        @Isin_Liftdoorclosed  :  Liftdoorclosed = TRUE
THEN
        @leave_LiftStoped  :  LiftStoped ≔ FALSE
        @enter_moving  :  lift ≔ moving
        @enter_LiftMoving  :  LiftMoving ≔ TRUE
        @Leave_Liftdoorclosed  :  Liftdoorclosed ≔ FALSE
END

StopLift ≙
extended
STATUS
Ordinary
REFINES
StopLift
WHEN
        @isin_moving  :  lift = moving
        @Isin_LiftMoving  :  LiftMoving = TRUE
THEN
        @leave_LiftMoving  :  LiftMoving ≔ FALSE
        @enter_stopped  :  lift ≔ stopped
        @enter_LiftStoped  :  LiftStoped ≔ TRUE
        @Enter_LiftStoped_1  :  LiftStoped_1 ≔ TRUE
END

OpenLiftDoor ≙
STATUS
Ordinary
WHEN
        @isin_LiftStoped_1  :  (LiftStoped_1 = TRUE)
        @Isin_closed  :  door = closed
THEN
        @leave_LiftStoped_1  :  LiftStoped_1 ≔ FALSE
        @enter_Liftdooropen  :  Liftdooropen ≔ TRUE
        @Enter_open  :  door ≔ open
END

CloseLiftDoor ≙
STATUS
Ordinary
WHEN
        @isin_Liftdooropen  :  Liftdooropen = TRUE
        @Isin_open  :  door = open
THEN
        @leave_Liftdooropen  :  Liftdooropen ≔ FALSE
        @enter_Liftdoorclosed  :  Liftdoorclosed ≔ TRUE
        @Enter_closed  :  door ≔ closed
END

NotOpenLiftDoor ≙
STATUS
Ordinary
WHEN
        @isin_LiftStoped_1  :  (LiftStoped_1 = TRUE)
        @Isin_closed  :  door = closed
THEN
        @leave_LiftStoped_1  :  LiftStoped_1 ≔ FALSE
        @Enter_Liftdoorclosed  :  Liftdoorclosed ≔ TRUE
END

@Distinct_states_in_LiftStoped_statemachine1  :  (LiftStoped = TRUE)
⇒ partition{ {TRUE}, {LiftStoped_1} ∩ {TRUE}, {Liftdooropen} ∩ {TRUE}, {Liftdoorclosed} ∩ {TRUE}}
@LiftStoped_1_substateof_LiftStoped  :  (LiftStoped_1 = TRUE) ⇒ (LiftStoped = TRUE)
@Liftdooropen_substateof_LiftStoped  :  (Liftdooropen = TRUE) ⇒ (LiftStoped = TRUE)
@Liftdoorclosed_substateof_LiftStoped  :  (Liftdoorclosed = TRUE) ⇒ (LiftStoped = TRUE)
```

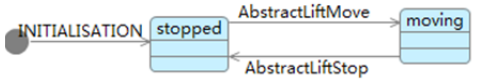

Fig. 10. liftStateMachine0

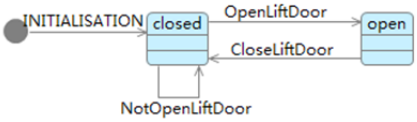

Fig. 11. DoorStateMachine1.

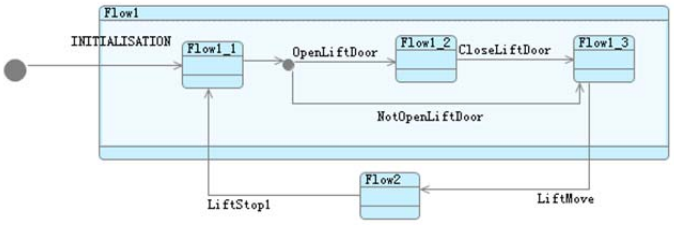

Fig. 12. FlowStateMachine 1.

It should be noted that, *OpenLiftDoor*, *CloseLiftDoor* and *NotOpenLiftDoor* are events belonging to the door object. The relationship between these three events is also expressed in Fig. 11. However, the relationship between events that belong to the lift object and that belong to the door object must specified by *FlowStateMachine1*, otherwise we can only write some Event-B code manually to achieve the same effect. In fact, we use the Flow state machine to replace the event refinement structure graph in the ERS method.

3) The Second Refinement

A loop event named *RequestFloor*, is introduced in this refinement level. It means that the passenger has pressed a lift button, which is a floor number. We note that this event should not be attributed to the lift object, nor should it belong to the door object, rather the environment (people) issued the event. So, we think the *RequestFloor* event is a global event, it should not be added to the door or elevator state machine, but should be manually added in the Event-B model.

As we know, passengers should choose at least one floor and the lift will stop at the corresponding floor. That is, the *RequestFloor* event should occur at least once before the *LiftStop* event occurs. ERS method cannot specify this requirement directly. But the iUML-B state machine can express it, as shown in Fig. 13.

The source state of the event *LiftStop* is extended into a super-state and *RequestFloor* event is inserted into it. This does not mean that *RequestFloor* event belongs to the elevator object. In fact, the control flow state machine is more like a composed state machine, where the state is a composition of states.

C. Summary

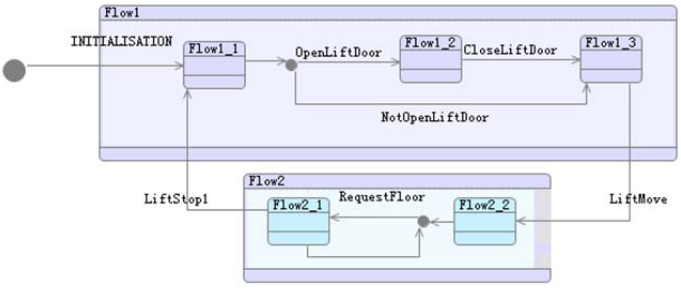

Fig. 13. FlowStateMachine2 - refined control flow model.

In this section we use the iUML-B state machine to construct a simple lift control system. We use iUML-B state machine, instead of ERS method, to model and refine the control flow. At each refinement level, we use an object state

machine to specify the action of an object (lift, door) itself, while using control flow state machine to constrain the overall event order of the lift control system. We obtained event traces that satisfy the requirements of lift system by combining these two types of state machines. The results of this method are the same as that of ERS method.

V. Discussion

Modeling the system control flow is an unavoidable step for all formal methods, but Event-B does not have behavioral semantics. Researchers have proposed many approaches to solve this problem. We use iUML-B state machine to model control flow of Event-B, and compare it with other methods in this field, as shown in Table 2.

We can see that, compared with the ERS method, the iUML-B state machine cannot express the relationship (that is, event structure) between the different refinement levels. However, iUML-B state machine has some advantages in the expression of event trace. Moreover, the two types of iUML-B state machines (object state machine and flow state machine) can be directly converted into Event-B code and embedded into the Event-B model. However, in the ERS method, this step requires the support of other plugins. Finally, ERS graph cannot be translated into LTS directly, while for iUML-B state machine, this translation is easy.

Compared with the flow method, which express the event order using an event based style, the iUML-B state machine uses the state based style to express the control flow. This makes the control flow easier for engineers to understand.

CSP has strict process algebra semantics, so the CSP||B method is more stringent than the iUML-B state machine. Therefore, we can consider the conversion between the CSP and iUML-B state machine, so that our approach can have more rigorous formal behavior semantics.

VI. Realted Work

Our method is inspired by Hallerstede's paper [13], [14]. He presents a method of defining a structured Event-B model, using assertions as nodes, and labels events on the transition edge. Hallerstede also pointed out that, during the refinement of the Event-B model, edge refinement (event refinement) and node refinement (state refinement) are similar, and proved the equivalence between edge refinement diagram and node refinement diagram.

iUML-B has been applied to various fields as a "UML-like" semi-formal modeling language. Fathabadi [15] uses iUML-B state machine to establish the thread scheduling model of the many-core system. Hoang [11] used iUML-B to model and verify the behavior of hemodialysis machine. Snook [10] used iUML-B state-machines to model the protocol execution involving the entities' interactions of Virtual local area network. Said and Butler [16] extended the UML-B meta-model to support the refinement of the UML-B state machine, and define a series of rules to verify the correctness of the refinement. However, all of the above work has not explicitly put forward to the concept of flow state machine and control flow of Event-B.

CSP is a formal system based on process algebra, which explicitly supports the control flow modeling. The CSP||B method [4], [17], [18] is an integrated formal method that combines Event-B with CSP to explicitly model control flows in Event-B. Schneider has proved that, as long as the CSP control flow model is deadlock-free, and the Event-B functional model is non-divergent, it can be concluded that the model obtained by the composition of these two models is deadlock-free. IUML-B state machines are not as powerful as CSP in expressing formal behavioral semantics. But as a semi-formal modeling language, it is easier than CSP to learn.

Alexei [5] proposed a method of applying a control flow constraint to an Event-B model without having to modify it, named flow language. The flow language uses ena, dis, and fis to express the order of the events, and uses the flow plug-in to provide graphical symbols to model the event orders. The flow language also uses OR, AND, and XOR to express the "non-exclusion or", "concurrency" and "exclusion or" relationships between events. However, the modeling elements of the flow method are the events and the relationship between events, rather than state and conversion of the state transition system. This makes flow language difficult to convert to a formal semantic model.

ERS method [6]-[9] presents a method that integrates structural refinement and control flow refinements for building an Event-B model, as described in Section II. In order to get the system model, one has to combine ERS model (ERS_M) and UML-B model ($UML\text{-}B_M$). That is to say, $System_M = UML\text{-}B_M \,||\, ERS_M$, where the ERS model is a control flow model, while the iUML-B model is a functional Event-B model. If we use the iUML-B state machine instead of ERS to construct the system's control flow model, the above-mentioned combination process can be completed directly by code generation.

TABLE II. Comparison of Major Control Flow Modeling Methods

Ability / Method	Formal behavior semantics	Express ability	Convertible to LTS
ERS	No	Event structure	No
Flow Method	No	Event order	No
CSP||B	Yes	Event order	Yes
iUML-B	No	Event order	Yes

VII. Conclusion

In this paper, we have proposed a method that facilitates the establishment of an Event-B control flow model through a semi-formal iUML-B state machine, which explicitly expresses the control flow of the Event-B model using state transitions. This control flow model can be directly converted to Event-B's state variables and embedded into its functional model. The JSD-style control flow model in ERS method is replaced by the state machine, which makes it easier to observe and analyze the control flow model.

In the future, we intend to translate iUML-B state machine to a behavioral semantic model such as CSP or LTS, which

allows us to verify the behavior properties of the system as early as possible.

ACKNOWLEDGMENT

The authors are very grateful to the chief editor and reviewers for their comments and suggestions, which are helpful in improving the paper.

REFERENCES

[1] J.-R. Abrial, Modeling in Event-B: System and Software Engineering: Cambridge University Press, 2010.

[2] J.-R. Abrial, The B-book: assigning programs to meanings: Cambridge University Press, 1996.

[3] R. J. R. Back and F. Kurki-Suonio, "Distributed cooperation with action systems," Acm Transactions on Programming Languages & Systems, vol. 10, pp. 513-554, 1988.

[4] S. Schneider, H. Treharne, and H. Wehrheim, "A CSP approach to control in event-B," presented at the Proceedings of the 8th international conference on Integrated formal methods, Nancy, France, 2010.

[5] A. Iliasov, "Use Case Scenarios as Verification Conditions: Event-B/Flow Approach," in Software Engineering for Resilient Systems - Third International Workshop, SERENE 2011, Geneva, Switzerland, September 29-30, 2011. Proceedings, 2011, pp. 9-23.

[6] A. S. Fathabadi, M. Butler, and A. Rezazadeh, "Language and tool support for event refinement structures in Event-B," Formal Aspects of Computing, vol. 27, pp. 499-523, 2015.

[7] E. Alkhammash, M. Butler, A. S. Fathabadi, and C. Cîrstea, "Building traceable Event-B models from requirements," Science of Computer Programming, vol. 111, pp. 318-338, 2015.

[8] A. S. Fathabadi, A. Rezazadeh, and M. Butler, "Applying Atomicity and Model Decomposition to a Space Craft System in Event-B," in International Conference on NASA Formal Methods, 2011, pp. 328-342.

[9] A. Salehi Fathabadi and M. Butler, "Applying Event-B Atomicity Decomposition to a Multi Media Protocol," in Formal Methods for Components and Objects: 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers, F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds., ed Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 89-104.9.

[10] C. Snook, T. S. Hoang, and M. Butler, "Analysing Security Protocols Using Refinement in iUML-B," in NASA Formal Methods Symposium, 2017, pp. 84-98.

[11] T. S. Hoang, C. Snook, L. Ladenberger, and M. Butler, "Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-Simulation," in International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, 2016, pp. 360-375.

[12] 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.

[13] S. Hallerstede and C. Snook, "Refining Nodes and Edges of State Machines," in Formal Methods and Software Engineering: 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings, S. Qin and Z. Qiu, Eds., ed Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 569-584.

[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] A. Salehi Fathabadi, C. Snook, and M. Butler, "Applying an Integrated Modelling Process to Run-time Management of Many-Core Systems," in Integrated Formal Methods: 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings, E. Albert and E. Sekerinski, Eds., ed Cham: Springer International Publishing, 2014, pp. 120-135.

[16] M. Y. Said, M. Butler, and C. Snook, "A method of refinement in UML-B," Software & Systems Modeling, vol. 14, pp. 1557-1580, 2015.

[17] S. Schneider, H. Treharne, and H. Wehrheim, "The behavioural semantics of Event-B refinement," Formal Aspects of Computing, vol. 26, pp. 251-280, 2014.

[18] S. Schneider, H. Treharne, and H. Wehrheim, "Bounded Retransmission in Event-B||CSP: a Case Study," Electronic Notes in Theoretical Computer Science, vol. 280, pp. 69–80, 2011.