Construction of Abstract State Graphs for Understanding Event-B Models

Daichi Morita[1](✉), Fuyuki Ishikawa[2], and Shinichi Honiden[1,2]

[1] The University of Tokyo, Tokyo, Japan
[2] National Institute of Informatics, Tokyo, Japan
{d-morita,f-ishikawa,honiden}@nii.ac.jp

Abstract. Event-B is a formal method that supports correctness by construction in system modeling using stepwise refinement. However, it is difficult to understand the rigorous behaviors of models from Event-B specifications, such as the reachable state space or the possible sequences of events. This is because the Event-B model is described in a style that lists events that have concurrently been enabled depending on their guard conditions. This paper proposes a method that helps in understanding the rigorous behaviors of an Event-B model by creating an abstract state graph. The core of our method involves dividing the concrete state space by using the guard conditions of individual events to extract states that are essential to enable possible transitions to be understood. Moreover, we further divided the state space by using the guard conditions of events in the models before refinement to support understanding of changes in behaviors between the models before and after refinement. Our unique approach facilitated finding of invariants that were not specified but held, which were useful for validation.

1 Introduction

Event-B [1] is a formal specification language based on first-order predicate logic and set theory. It adopts a refinement mechanism to allow developers to gradually build a model while ensuring its correctness by using these mathematical methods. Developers in Event-B modeling start from the most abstract machine to build the model and then refine it by building a more concrete machine that introduces new aspects so that it is closer to the comprehensive machine to be obtained. This refinement process is continued until the comprehensive machine is obtained that includes all the target aspects. This refinement mechanism reduces the difficulty in rigorously modeling and verifying a complicated model by enabling focus on individual small steps.

Event-B is a state-based formal method and the behavior of an Event-B model is expressed by states and transitions. Thus, it is important for developers to comprehend the reachable state space or the possible sequences of events in terms of validation. However, this is difficult because infinite sets, such as

This work is partially supported by JSPS KAKENHI Grant Number 17H01727.

© Springer International Publishing AG 2017
K.G. Larsen et al. (Eds.): SETTA 2017, LNCS 10606, pp. 250–265, 2017.
https://doi.org/10.1007/978-3-319-69483-2_15

integers, can be used as types of variables and thus the size of the state space can be infinite or too large to comprehend.

ProB [12] is one of the standard tools to check Event-B models. Although ProB provides several methods of visualizing the state space [9,11], there have been problems with these methods. They have required developers to specify that the range of infinite sets be finite to generate a graph because they constructed the state space by exhaustive simulation. Thus, the state space was restricted and developers could miss unexpected behaviors outside the space. Moreover, there are no methods of graph visualization that takes refinement into consideration, even though it would be useful to know how a concrete machine can refine an abstract one.

This paper proposes two methods of graph visualization of an Event-B model from the specifications without simulation to enable behaviors to be rigorously understood. The first method involves constructing an abstract state graph using predicate abstraction [8], which is useful to enable developers to explore the full state space and not to overlook the differences in behaviors, according to the range of infinite sets. Our key idea was to use guard conditions of events for predicate abstraction, which allowed us to extract essential insights into possible transitions (event occurrences) in each state. The second method was graph visualization that took refinement into account. It is useful for developers to validate behaviors by checking the correspondence between the states and transitions of abstract and concrete machines. Moreover, these graphs are useful for developers to find stronger invariants than those described in the specifications and help them to validate the state space. The unique feature in our approach is that we first constructed apparently reachable states from the specified predicates, such as invariants, and we then examined actual (un)reachability. This approach could expose unexpectedly unreachable states, which represented implicit expectations or faults.

We have organized the rest of this paper as follows. Section 2 provides the necessary background on Event-B and Sect. 3 describes our methods of generating graphs. Section 4 explains how we evaluated our methods by providing various applications. Section 5 relates our study to other studies and Sect. 6 concludes the paper.

2 Preliminaries

2.1 Event-B Models

An Event-B model consists of two modules, which are called machine and context. A context contains a set of constants and a set of axioms. Axioms are predicates that denote the constraints that the constants must satisfy. A machine contains a set of variables, a set of invariants and a set of events. Invariants are predicates that denote the safety properties that developers require variables and constants to satisfy. That they are actually invariants is verified by proving some statements, which are called "proof obligations". An event mainly consists of some guards and actions. Guards are predicates to denote the conditions under

which an event is to be enabled. Actions are called before-after predicates that denote the relationships between the values of variables just before and after an event. The values of the constants cannot be changed by the events. Thus, the states of the model consist of the dynamic values of the variables and the static values of the constants. The transitions between them are triggered by the occurrence of the events.

For example, let us take Abrial's model of "controlling cars on a bridge" (from [1, Chap. 2]) into account. There is a mainland, an island, a bridge between them, and traffic lights that control cars going to and coming from them in the model. There was only the mainland and island in the initial model. It consisted of a context $Ctx0$ and a machine $Mac0$ shown in Fig. 1. The constant d defined in $Ctx0$ denotes the maximum number of cars allowed to be on the island. The variable n defined in $Mac0$ represents the number of cars on the island. The invariant $inv0_2$ means that constant d is actually the maximum number of cars on the island. The states of the model consist of two values of variable n and constant d, such as $(n, d) = (0, 1)$. The state space is infinite because d can be an arbitrary natural number that is more than zero. The event *init* is the initialization event, ML_out is the event corresponding to the transition of a car from the mainland to the island, and ML_in is its inverse event. The guard $grd1$ of the event ML_out is $n < d$ and the action $act1$ is the before-after predicate $n' = n + 1$. The value of the variable just after an event has occurred makes its before-after predicate true. A primed variable, such as n' appearing in $act1$ in a before-after predicate, denotes the value of the variable just after an event has occurred. Thus, the before-after predicate $act1$ means that the value of variable n just after the event is equal to the value of variable n just before it has occurred plus one.

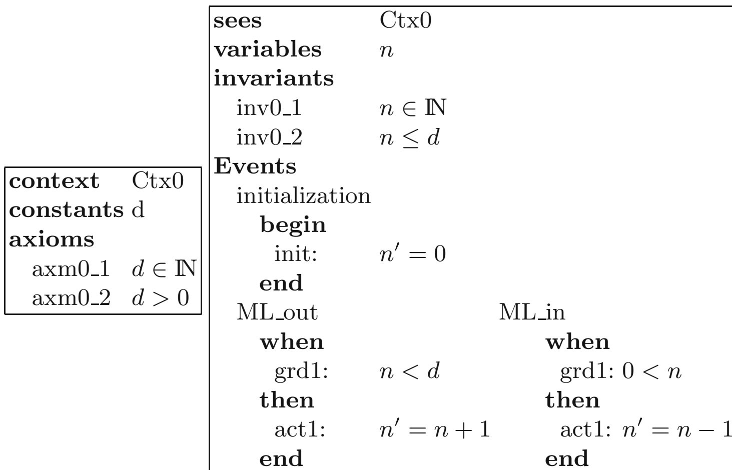

Fig. 1. Event-B specifications of $Mac0$

2.2 Event-B Refinement

The refinement mechanism in Event-B is a way of gradually building a model. An abstract machine is refined by a new machine and new features and details are introduced into the abstract machine. A new machine is called a concrete machine. A sequence of machines linked by a refinement relationship is called a “refinement chain”. An increasingly more complicated but accurate model is built through stepwise refinements.

For example, $Mac1$ (Fig. 2) refines $Mac0$. A one-way bridge is introduced into the abstract machine. The variable a is the number of cars on the bridge going to the island, b is the number on the island and c is the number on the bridge coming to the mainland. The variable n defined in $Mac0$ is replaced by these three variables and the invariant $inv1_2$ denotes the relationship between n and a, b, c. The states consist of four values of the variables a, b, c and the constant d. The invariant $inv1_3$ denotes that the bridge is one-way. The two events ML_out and ML_in in $Mac0$ are refined as they are events on these

refines $Mac0$
sees Ctx0
variables a, b, c
invariants
  inv1_1 $a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge c \in \mathbb{N}$
  inv1_2 $a + b + c = n$
  inv1_3 $a = 0 \vee c = 0$
Events
  initialization
    begin
      init: $a' = 0 \wedge b' = 0 \wedge c' = 0$
    end
  ML_out
    refines ML_out
    when
      grd1: $a + b < d$
      grd2: $c = 0$
    then
      act1: $a' = a + 1$
    end
  ML_in
    refines ML_in
    when
      grd1: $0 < c$
    then
      act1: $c' = c - 1$
    end
  IL_in
    when
      grd1: $0 < a$
    then
      act1: $a' = a - 1$
      act2: $b' = b + 1$
    end
  IL_out
    when
      grd1: $0 < b$
      grd2: $a = 0$
    then
      act1: $b' = b - 1$
      act2: $c' = c + 1$
    end

Fig. 2. Event-B specifications of $Mac1$

three variables. ML_out in $Mac1$ corresponds to the transition of a car from the mainland to the bridge and ML_in is its inverse event. These refined events must simulate the original ones. The before-after predicate of ML_out is $a' = a+1$ but it implicitly contains $b' = b$ and $c' = c$, which means that the values of missing variables in the predicate are equal to the values just before the event. Thus, its precise before-after predicate is $a' = a+1 \wedge b' = b \wedge c' = c$, and it simulates that of the original because the invariant $a + b + c = n$ holds. Guards of the refined event must not contradict those of the original. The guard of the event ML_out in $Mac1$ is $a + b < d$ and does not contradict the guard $n < d$ of the abstract one because the invariant $a + b + c = n$ holds.

New events can be introduced into abstract models. IL_in and IL_out in $Mac1$ are new events. IL_in corresponds to the transition of a car from the bridge to the island and IL_out is its inverse event. They do not need to modify any abstract variables so that the abstract model is not contradicted, i.e., they need to refine null event *skip* in which guard is *true* and action has no meaning.

3 Method

3.1 Construction of Abstract State Graph (CASG)

The goal discussed in this subsection was to construct a state graph from the Event-B specifications, which is useful for understanding the rigorous behavior of an Event-B model. The state space that may be infinite is abstracted by predicate abstraction. We called the graph an "abstract state graph" as in Graf and Saïdi [8] and called our method of constructing the abstract state graph CASG.

Let us assume that we have a machine M. We use symbols Inv_M to denote the conjunction of all invariants and axioms that appear in the refinement chain that precedes M, and BA_{evt} for the conjunction of all before-after predicates of the event evt and the event that refines evt in the refinement chain. For example, $Inv_{Mac1} = (d > 0 \wedge n \leq d \wedge a + b + c = n \wedge (a = 0 \vee c = 0))$ and $BA_{ML_out} = (n' = n+1 \wedge a' = a+1 \wedge b' = b \wedge c' = c)$. We also use symbols G_{evt} to denote the conjunction of all the guards of event evt and Evt_M to denote the set of the events of model M.

An abstract state graph of the Event-B machine consists of (S, I, L, δ), where S is a set of abstract states, I is a set of initial abstract states, L is a labeling function of the set S and δ is a transition function with guard conditions. An abstract state that constitutes S is defined by a predicate and a set of states that satisfy the predicate. For example, $0 \leq n \leq 2$ represents the set of states $\{(n, d) \mid n, d \in \mathbb{N}, 0 \leq n \leq 2\}$. After this we will use a predicate to denote an abstract state, i.e., we will refer to abstract states and predicates as exchangeable words. We will use "concrete states" to refer to states of the model to distinguish them from abstract states.

First, let us define set S of abstract states. We use $sat(p)$ to denote that the predicate p is satisfiable. Set S is constructed as:

$$S = \left\{ s \mid E \subseteq Evt_M,\ s = Inv_M \wedge \bigwedge_{evt \in E} G_{evt} \wedge \bigwedge_{evt \in Evt_M \setminus E} \neg G_{evt}\ \ , sat(s) \right\}.$$

This definition constructs abstract states by making equivalence classes of concrete states that satisfy invariants in terms of enabled events in each state. The possibly infinite state space is reduced into finite state space. However, note that the state space is approximated by the invariants and it may include some unreachable concrete states. This is discussed in Subsect. 3.5. This approximation is reasonable since invariants are properties that developers require the model to satisfy and they are verified by discharging proof obligations. Although the idea that states that have the same enabled events are regarded as being the same is similar to the method described in Leuschel and Turner [13], our method does not require developers to specify the range of infinite sets. It also provides predicates that explain the conditions of individual states.

The set I of initial abstract states is the set of abstract states that the before-after predicate of the initialization event satisfies.

The labeling function L for each $s \in S$, to specify the events enabled in an abstract state, is defined as:

$$L(s) = \{evt \in Evt_M \mid sat(G_{evt} \wedge s)\}.$$

The transition function δ is then constructed. We use $after(s)$ for each predicate s to denote a predicate where all variable symbols are replaced with primed variable symbols, which means that they are values just after events have occurred. Note that $after$ only replaces variable symbols, and not constant symbols. The δ for each $s \in S$ and $evt \in L(s)$ is defined as:

$$\delta(s, evt) = \{(s', g) \mid s' \in S, g = s \wedge BA_{evt} \wedge after(s'), sat(g)\}.$$

The predicate $BA_{evt} \wedge after(s')$ is like the weakest precondition if the state will be s' just after evt has occurred. Thus, g is a guard condition of the transition. Note that even if there is an edge, the corresponding transition cannot always occur in M because of our approximation.

Let us take the model $Mac0$ (Fig. 1) as an example. The graph constructed by using CASG is outlined in Fig. 3. The two lines in each ellipse, such as $\{ML_out\}$ and $n = 0 \& d > 0$ in the top ellipse, denote the enabled events and abstract states. Type invariants have been omitted. The two lines beside the arrow denote the name of the event and the guard condition.

A graph constructed by using CASG is an abstraction of the actual state graph of an Event-B model, in which state space may be infinite. Solving satisfiability problems enables us to explore the full state space when checking the existence of transitions. An important aspect of this abstraction is that transitions that actually occur in the model are in the graph and transitions that are not in the graph do not occur in the model.

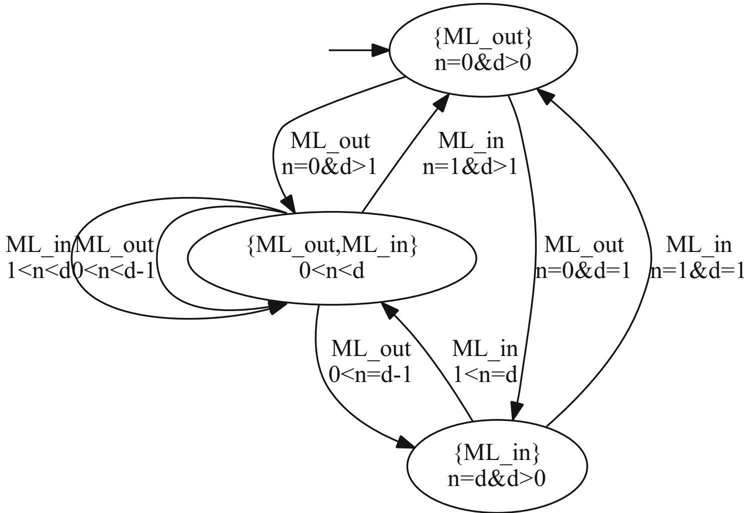

Fig. 3. Abstract state graph of $Mac0$

3.2 Construction of Refinement Abstract State Graph (CRASG)

This subsection explains how we constructed a graph that took refinement into consideration. We assumed that we had machines M_A and M_C, such that M_C refined M_A. Each state in M_A was refined by some states in M_C through the refinement. We wanted to reflect such a relation between the graphs of M_A and M_C. In other words, each abstract state of the graph of M_C constructed with the method described in this subsection corresponds to one abstract state of the graph of M_A constructed with CASG. We called the method of constructing the abstract state graph that took refinement into account CRASG.

Let us define a binary relation R_V to clarify the relation between the abstract state graphs for M_A and M_C. The R_V is a binary relation between the abstract states of the graphs. Let S_A be the abstract states set of the graphs for M_A that is constructed by using CASG, and let S_C be the abstract states set of the graph for M_C that is constructed by using CRASG. The R_V is defined as:

$$R_V = \{(s, s') \in S_C \times S_A \mid sat(s \wedge s')\}.$$

Here, $(s, s') \in R_V$ means that there is a concrete state in s that corresponds to a state in s'. Our main objective in this subsection is to explain how we constructed the graph of M_C, such that R_V is a function, which means each abstract state in S_C corresponds to one abstract state in S_A.

Let us now construct the abstract states set S. Let $Evt = Evt_{M_C} \cup Evt_{M_A}$. The S is constructed as:

$$S = \left\{ s \mid E \subseteq Evt,\ s = Inv_{M_C} \wedge \bigwedge_{evt \in E} G_{evt} \wedge \bigwedge_{evt \in Evt \setminus E} \neg G_{evt}, sat(s) \right\}.$$

The construction of S splits the state space of M_C by the equivalence relation, where the enabled events of M_C are the same and those of M_A are the same if the space is projected onto the space of M_A. Therefore, the abstract states in the graph of M_A that are constructed by using CASG are divided even more into S, and R_V becomes a function. Note that unlike CASG, there can be states where the same events of M_C are enabled. Then, the remainder of the construction of the graph is similar to that with CASG.

For example, the graph of $Mac1$ that is constructed by using CRASG is given in Fig. 4. The squares denote the abstract states of the $Mac0$ graph. The dashed arrows mean that they correspond to transitions in the $Mac0$ graph. The predicates in the graph have been omitted to the extent that they can be understood. The guard labels have been completely omitted.

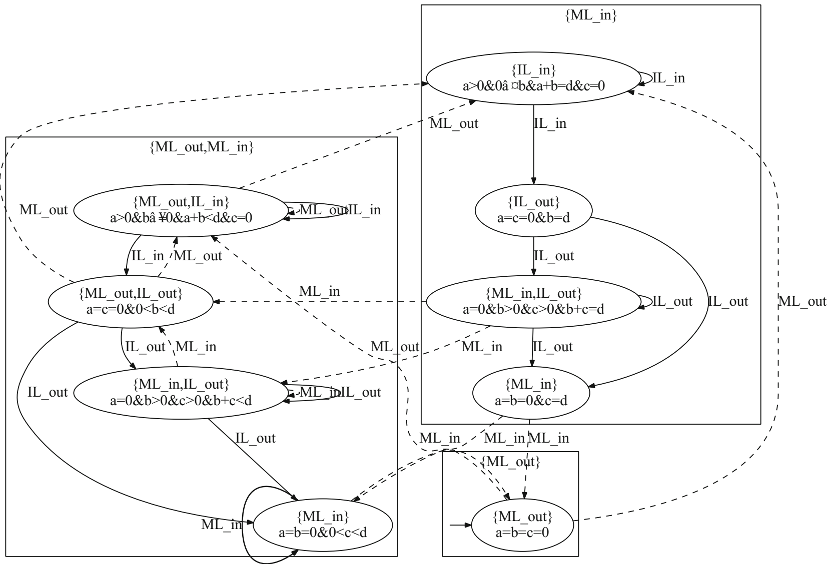

Fig. 4. Abstract state graph of $Mac1$ by taking refinement into consideration

The graph of M_C that is constructed by using CRASG is the refinement of the graph of M_A that is constructed with CASG. The abstract states of the graph of the concrete model can be grouped by which abstract state of the graph for M_A they satisfy because the binary relation R_V is a function. Moreover, the transitions of the graph for M_C can be grouped by which transition of the graph for M_A they simulate. These facts indicate that the graph provides a visualization of how M_C simulates M_A in terms of states and transitions.

3.3 Implementation

These two methods need to solve numerous satisfiability problems. Event-B is based on first-order predicate logic, and thus predicates used in Event-B models

are first-order predicates within a practical range. A satisfiability modulo theories (SMT) solver [3] is one of the tools used to solve them automatically. Although the range of problems that it can solve is limited, it can solve many of the predicates in practical Event-B models. We used Z3 [5] as an SMT solver for our implementation. We succeeded in automatically creating a graph for $Mac0$, $Mac1$ and $Mac2$ (as a result of the second refinement of "controlling cars on a bridge") by using CASG and the graph for $Mac1$ and $Mac2$ by using CRASG.

3.4 Checking for Existence of Transitions

A transition corresponding to an edge in the graph of the model constructed with our methods could not always actually occur in the model because of our approximation. However, developers could check whether or not each transition (s, evt, s') could actually occur by using linear temporal logic (LTL [7]) model checking. The properties to check for occurrences could be directly represented by using $\text{LTL}^{[e]}$ [15] that introduced the operator $[evt]$, which meant the next executed event was evt. The condition that a transition (s, evt, s') could occur was formulated by $\text{LTL}^{[e]}$ in the form $\neg G \neg (s \wedge [evt] \wedge X(s'))$.

The LTL model checker could be used in ProB [14], which also supports $\text{LTL}^{[e]}$. Note that it adopts lower approximation and model checking is done under some values of constants. Thus, developers need to appropriately set the range.

3.5 Checking Validity and Strengthening Invariants

This subsection introduces another unique use of the graphs that were constructed with our methods due to our construction. It promotes the enhancement of invariants to strictly represent their expectations on acceptable and unacceptable states.

Sufficient invariants should ideally be given to strictly distinguish expectations on reachable and unreachable state spaces. However, this does not always hold, even at the level of the model appearing in [1], which is one of the most well-known references on Event-B. One reason for this is that developers completely understand the reachable concrete state space and decide not to add invariants to the model because it is redundant and these would not matter. In other words, reachable spaces are indirectly constrained by other means, such as guard conditions in events. This implicit approach may cause problems when other developers try to understand and revise the model in this case. Another reason is that they do not understand the reachable state space and have missed the invariants held in the model. There probably will be unexpected behaviors in the model in this case. Therefore, it is worth suggesting invariants that hold to confirm validity.

Our approach approximates the concrete state space with the invariants of a model. Thus, our graph can provide the difference between the actual reachable state space and the invariants by examining reachability. Other visualization methods [9,11] could not provide this because their construction was based

on the simulation of the model and did not take invariants into consideration. Developers then checked each of the suggested invariants; they could be invariants that should have been added or they represented unexpectedly attained unreachable states caused by faults in the model (e.g., overly strong guard conditions). There are three methods that suggest invariants that can be achieved by using our graph.

First, let us assume there is an unreachable abstract state s from the initial abstract state. Then, its negation $\neg s$ is an invariant. This is because if there are no transitions into the abstract state in the model, then all the events preserve the negation.

Second, let us assume that there is a transition $s \xrightarrow{evt,g} s'$ in the graph that actually does not occur in the model (recall Subsect. 3.4). There are some unreachable concrete states in the source abstract state s of the transition in that case. We can find such a state by using an SMT-solver and finding an assignment of the predicate $s \wedge g \wedge BA_{evt} \wedge s'$. Developers can then find some invariants by investigating why this state was unreachable.

Third, let us assume that there is abstract state s that contains a concrete state unreachable from other abstract states. The condition is formulated as:

$$after(s) \Rightarrow \bigvee_{(s',evt):(s,g)\in\delta(s',evt)} s' \wedge BA_{evt}.$$

The predicate $s' \wedge BA_{evt}$ means the possible reachable states from s' just after event evt has occurred. Thus, the formula means that abstract state s is actually included in the possible reachable states from other states. There are unreachable concrete states in s if the condition does not hold, and some invariants can be added.

4 Evaluation

4.1 Setting

We evaluated our methods by using three applications of the graphs. Subsection 4.2 explains how we investigated the graph constructed by using CASG. This is useful for understanding the overall behaviors of the model and the validation of the state space by using predicates. Subsection 4.3 describes how we investigated the graph constructed with CRASG. This would help developers understand the details on changes in behaviors caused by refinement. Subsection 3.5 explains how many we found the stronger invariants than those described in the specifications. We used the $Mac2$ model (from [1, Chap. 2]) that refines $Mac1$ to evaluate our methods in addition to $Mac0$ and $Mac1$.

4.2 Abstract State Space Exploration

Our main objective was to help developers understand the behaviors of an Event-B model. Our methods provide state graphs that represent behaviors. Abstraction of the state space reduces the complexity of the original state graph and

makes it easy for developers to comprehend behaviors. In addition, abstract states help them validate the model since these states are represented by predicates on the variables and constants in the model and the predicates can promote developers' understanding of the states.

Here, we will provide an example of exploring the state space by using our graph. Our graph provides the conditions for the variables and constants that represent the set of all concrete states that enable the same events. The example of $Mac0$ (Fig. 1) indicates that, if ML_out and ML_in are concurrently enabled, then developers can see that $0 < n < d$ holds in the state on the left in Fig. 3. The predicate is easy for them to compare with their intentions because they describe various predicates in Event-B modeling and they can perceive the situation with the model from the predicates. If they write the guard of ML_out incorrectly as $n < d - 1$, the predicate of the state with label $\{ML_out, ML_in\}$ will be $0 < n < d - 1$. They can then find that the guard is incorrect because their intention is for the number of cars allowed on the island to be d, but this is not achieved. It is also important for d to be symbolic. Due to this, they can validate that this condition holds no matter what the value of constant d is.

The graph of an Event-B model constructed by using CASG helps developers validate the state space by providing predicates on the variables and constants. However, this method does not completely solve the problem of complexity in the graph if the model is very complicated. One possible solution is for developers to focus on the change caused by refinement, which will be discussed in next section. This cannot completely solve the problem, but it is effective for limited ranges of investigation.

It also helps developers to comprehend overall sequences of the executed events by searching paths in the graph, even though not all sequences of events that correspond to the paths can be executed. The guard conditions of the transitions are useful when validating sequences of events because they can express conditions where the sequences can be executed.

4.3 Refinement Abstract State Graph Exploration

A graph constructed by using CRASG helps developers understand changes in behaviors caused by refinement. Changes are not trivial from the specifications because refinement is across the invariants, guards, and actions. The graph is very useful for understanding some aspects of the effect of refinement.

An aspect is how a concrete model simulates its abstract model. In other words, the correspondences between abstract states and transitions in the model before refinement to those in the model after refinement are drawn in the graphs. Let us take model $Mac1$ (Fig. 2) as an example. There is a square in Fig. 4 that is labeled $\{ML_in\}$, which represents the abstract state of the graph for $Mac0$ that satisfies the predicate $n = d \wedge d > 0$. The predicate $n = d$ means that the number of cars on the island has reached the limit. Thus, the states and transitions in the square describe how the cars are moving between the island and mainland along the bridge to solve traffic jams in the model $Mac1$. There are also two transitions labeled ML_out between the squares labeled $\{ML_out, ML_in\}$, and $\{ML_in\}$

in Fig. 4. Such transitions correspond to the transition in the graph in Fig. 3 that are labeled ML_out between the abstract states labeled $\{ML_out, ML_in\}$ and $\{ML_in\}$, which means that the number of cars on the island has just reached the limit.

Another aspect is how a concrete model does not simulate its abstract model. The abstract model can be refined so that some transitions of the abstract model cannot occur in the concrete model by strengthening guards regardless of whether they have been intended or not. This graph helps developers at such times to discover what transition has occurred and what has not occurred in the concrete model. If developers write the guard of ML_out of $Mac1$ incorrectly as $a + b < d - 1$, the transitions that correspond to the transition labeled ML_out between abstract states labeled $\{ML_out, ML_in\}$ and $\{ML_in\}$ will actually disappear from the graph. Developers can then find the degree of degradation and check whether it is intended.

The graph constructed by using CRASG provides correspondences in the state graph for concrete and abstract models. It is useful for developers to fully understand refinement by comparing it with the models. It also helps developers explore the model by focusing on the change if the model is complicated.

4.4 Checking Validity and Strengthening Invariants

We applied the method described in Subsect. 3.5 to $Mac2$ and found missing invariants that were needed to express precise reachable states. The details are described in Appendix A. As a result, we discovered seven invariants and discharged their proof obligations on the Rodin platform [2], which is equipped with theorem provers. Moreover, we checked that the state space expressed by the invariants was the actual reachable state space of the $Mac2$ model.

However, methods such as these three present several problems. One problem is that the methods are not automatic except for the first one. The methods provide a hint but require some suggestions by developers. Even though they are required to do so, it is difficult for other graph visualization methods to find stronger invariants. Moreover, such suggestions also help them validate the model and the discovered invariants support the building of a more accurate model.

5 Related Work

Our method is classified in terms of abstraction as the predicate abstraction described in Graf and Saidi [8]. Given a set of predicates, it splits the state space of variables appearing in a program or model based on the Boolean value of the predicates. It can solve a state space explosion problem in model checking by providing appropriate predicates. We used it for graph visualization of an Event-B model. We chose the set of guards for the event and the invariants of the model as input for predicate abstraction. This very effectively expressed behaviors of the model because the state space could be approximated by the

invariants and the guards determined the sequences of the executed events. The unique feature of our approach is that we first construct apparently reachable states from specified predicates, such as invariants, and we then examine actual (un)reachability.

Graph visualization is one of the primary methods of enabling the behaviors of a finite state machine to be understood. As described in Dulac et al. [6], the readability of formal specifications is a factor that has not widely been used in industry and visualization often helps people understand specifications. Our methods were aimed at visualizing the behaviors of a possibly infinite state machine of Event-B by reducing it into a finite state machine using predicate abstraction.

There are several other methods of visualizing the state space of an Event-B model. ProB [12] can generate a state graph of the model. The number of states and transitions in the graph are sizably large because it tries to generate the original state space. Thus, it is hard to understand behaviors. Moreover, ProB requires the range of infinite sets to be specified. Thus, the generated state space may be restricted and developers may miss unexpected behaviors. However, the state space of our graph is approximated by invariants so that all the behaviors of the model can be expressed in the graph.

Other methods of visualizing the state space are described in Leuschel and Turner [13]. There are two methods called the deterministic finite automaton (DFA)-abstraction algorithm and the signature merge method. These methods have aimed at reducing the complexity of the graph generated by ProB. The method of DFA abstraction is based on the classical minimization algorithm for DFA. This method produces a graph in which the sequences of transitions are equivalent to those in the original state space, but it cannot effectively reduce the state space and its graph still makes it difficult for developers to understand behaviors. The signature merge method is similar to ours in terms of the way abstraction focuses on enabled events. However, there are no predicates to represent the states and developers thus find our graph is more understandable.

Another similar approach described in Ladenberger and Leuschel [11] is creating projection diagrams. A projection diagram is an abstraction of the original graph that is obtained by using some projection function. The method can reduce complexity more effectively than the two approaches explained above and our methods by focusing on certain variables or some expressions in the model. However, it may lose too much information to enable the overall behavior to be understood, unlike that in ours.

In contrast to visualizing the state space of the model, unified modeling language-B (UML-B) [16] is a method of building an Event-B model by drawing a diagram. It is similar to UML and easy to use by developers who are familiar with it. It mitigates the burden in Event-B modeling but is not suitable for building a complex model.

Another method of understanding the behavior of models is model checking [4]. In particular, Hoang et al. [10] stated that proof obligations in Event-B ensure safety properties; on the other hand, LTL model checking ensured

temporal liveness properties. However, it is difficult to check overall behavior unlike that in our methods, such as the reachable state space or the sequences of executed events.

6 Conclusion

We proposed two methods of constructing the graph of an Event-B model from the specifications. Our methods are useful for graphically understanding the rigorous behavior of the model. They also allow developers to investigate reachable or unreachable states and transitions that cannot be searched by other graph visualizations [11,13]. The second method is useful for checking the correspondence between the graphs of the abstract and concrete machines and understanding changes in behaviors caused by refinement. Additionally, our methods enable developers to enhance invariants to strictly represent their expectations. We concluded that our methods could help developers to understand the behaviors of the model and validate it from various viewpoints. One possible direction in future work is to develop a more effective way of visualization for large systems.

A Appendix

This appendix explains how we investigated the advanced and unique use described in Subsects. 3.5 and 4.4 to discover invariants that were stronger than the invariants described in the specifications by using a graph constructed with CASG. We used the $Mac2$ model and the specifications are in Abrial [1, Chap. 2].

We applied the first method and discovered three unreachable states from the graph. One of them is represented by

$$a = 0 \land 0 < b < d \land c = 0 \land ml_tl = il_tl = red \land ml_pass = il_pass = true.$$

We then tried to add the predicate

$$\neg(a = 0 \land 0 < b < d \land c = 0 \land ml_tl = il_tl = red \land ml_pass = il_pass = true)$$

as an invariant to the $Mac2$ model on the Rodin platform [2]. As proof obligation is automatically discharged by them, the predicate is actually an invariant of the model. This invariant is equivalent to:

$$(a = c = 0 \land ml_tl = il_tl = red \land ml_pass = il_pass = true) \Rightarrow (b = 0 \lor b = d),$$

which means that if all the traffic lights are red, the flags are true and there are no cars on the bridge, then the number of cars on the island is zero or has reached its capacity. Developers can check if the situation is valid in the model.

We investigated the number of transitions in the $Mac2$ graph constructed by using CASG that could occur in the second method. We specified the range of the constant d from one to 10 because it seemed to be sufficient from our investigation of the model. We checked all 58 edges in the graph and discovered

16 edges that did not actually occur. One of them was the transition labeled IL_in from the abstract state represented by:

$$a > 0 \wedge b \geq 0 \wedge a+b < d-1 \wedge c = 0 \wedge ml_tl = green \wedge il_tl = red \wedge il_pass = true$$

to another represented by:

$$\begin{aligned} a = c = 0 \wedge b < d - 1 \wedge ml_tl = green \wedge il_tl = red \wedge il_pass = true \\ \wedge (b = 0 \vee (b > 0 \wedge ml_pass = false)). \end{aligned} \tag{1}$$

A concrete state where the transition can occur is:

$$(a, b, c, d, ml_tl, il_tl, ml_pass, il_pass) = (1, 1, 0, 4, green, red, false, true).$$

However, it is actually unreachable because the condition $ml_pass = false$ requires the event ML_tl_green to occur and ML_out_1 and ML_out_2 must not subsequently occur. There was some suggestion that the model always satisfies $a > 0 \Rightarrow ml_pass = true$ because $a > 0$ means ML_out_1 or ML_out_2 has occurred at least once just after ML_tl_green has taken place. Then, we added it as an invariant to the Rodin platform, but its proof obligation was not automatically discharged. Due to an analysis of the failure of the proof, which is often used in Event-B, we added $(ml_tl = red \wedge a + b \neq d) \Rightarrow a = 0$ as an invariant and all proof obligations were automatically discharged.

Finally, let us take $Mac2$ as an example of the third method. The abstract state represented by the predicate (1) does not satisfy the condition. All the transitions into it are labeled ML_tl_green. Since ML_tl_green makes ml_pass false, all concrete states where ml_pass is true in the abstract state are unreachable. There was some suggestion that the predicate $a = b = 0 \wedge ml_tl = green \Rightarrow ml_pass = false$ was an invariant. Therefore, we added it and proof obligation was discharged.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(6), 447–466 (2010)
3. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, Chap. 26, pp. 825–885. IOS Press (2009)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
5. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
6. Dulac, N., Viguier, T., Leveson, N.G., Storey, M.D.: On the use of visualization in formal requirements specification. In: Proceedings of the IEEE Joint International Conference on Requirements Engineering, pp. 71–80 (2002)

7. Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1980), pp. 163–173. ACM, New York (1980)
8. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). doi:10.1007/3-540-63166-6_10
9. Ham, F.V., van de Wetering, H., Van Wijk, J.J.: Visualization of state transition graphs. In: Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS 2001), p. 59. IEEE Computer Society, Washington, DC (2001)
10. Hoang, T.S., Schneider, S., Treharne, H., Williams, D.M.: Foundations for using linear temporal logic in Event-B refinement. Form. Aspects Comput. **28**(6), 909–935 (2016)
11. Ladenberger, L., Leuschel, M.: Mastering the visualization of larger state spaces with projection diagrams. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 153–169. Springer, Cham (2015). doi:10.1007/978-3-319-25423-4_10
12. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
13. Leuschel, M., Turner, E.: Visualising larger state spaces in ProB. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 6–23. Springer, Heidelberg (2005). doi:10.1007/11415787_2
14. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1985), pp. 97–107. ACM, New York (1985)
15. Plagge, D., Leuschel, M.: Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. Int. J. Softw. Tools Technol. Transf. **12**(1), 9–21 (2010)
16. Snook, C., Butler, M.: UML-B and Event-B: an integration of languages and tools. In: Proceedings of the IASTED International Conference on Software Engineering (SE 2008), pp. 336–341. ACTA Press, Anaheim (2008)