SUMMARY Refinement-based formal specification is a promising approach to the increasing complexity of software systems, as demonstrated in the formal method Event-B. It allows stepwise modeling and verifying of complex systems with multiple steps at different abstraction levels. However, making changes is more difficult, as caution is necessary to avoid breaking the consistency between the steps. Judging whether a change is valid or not is a non-trivial task, as the logical dependency relationships between the modeling elements (predicates) are implicit and complex. In this paper, we propose a method for analyzing the impact of the changes of Event-B. By attaching labels to modeling elements (predicates), the method helps engineers understand how a model is structured and what needs to be modified to accomplish a change.

key words: formal method, formal specification, impact analysis, refinement, event-B, modification support

1. Introduction

One key challenge to system dependability is dealing with the increasing complexity in system modeling and verification. One of the most promising approaches is using abstraction effectively. Specifically, refinement mechanisms decompose the complexity in modeling and verification into multiple steps with different abstraction levels.

Event-B is a formal method that uses a refinement mechanism to specify models of complex systems [1]. We can gradually introduce and verify concepts and constraints in a system while moving from abstract, prescriptive representations to concrete, realizable ones. The effectiveness of refining the mitigating complexity has been demonstrated, for example, in industrial experiences with Event-B [2, 3].

However, making changes becomes more difficult when using multiple refinement steps due to the dependency relationships between the model elements (predicates) in different steps. Careless changes could easily break the consistency between the steps. In addition, the relationships vary in each step because of the flexibility of the refinement mechanisms in Event-B (e.g., representation being changed by using concrete terms and/or strengthening constraints).

As the relationships are often implicit, without very costly rich documentation, it is very difficult to judge how desired changes can be made by selecting and reusing the proper parts of a model. This difficulty does not allow models to be reused to develop similar models that can be applied for a different situation.

In our previous research, we developed a heuristic method for clarifying relationships between predicates through refinements [4]. This method helps engineers understand the intention of each refinement step by extracting the core relationships between predicates through refinements by tracing variables used in the predicates. These core relationships present how the concepts of events are succeeded through refinements, so extracting relationships improves the understanding of how the system described by the model works. Understanding the behavior of a system makes it easier to judge whether an existing model can be applied to another case with little modification. However, there are still obstacles to reusing an existing Event-B model for a different situation. One of the obstacles is that some modifications made to a model are not as straightforward as they appear. As the size of the model increases, overlooking invariants becomes more likely, and engineers may try to conduct unpermitted modifications. Our novel method intends to remove this obstacle and encourages engineers to develop Event-B models effectively by reusing existing models.

In this paper, we present a method of change impact analysis for Event-B models with multiple steps of different abstraction levels. Change impact analysis is an analysis that estimates what needs to be modified to accomplish a change [5]. Our method classifies and outputs the characteristic of the logical dependency relationships of modeling elements (predicates) by applying our method to the Event-B model we want to analyze. By labeling classified characteristics on each element and showing whether each element accepts changes or not, understanding what is necessary for the intended changes becomes easier. We also present a strategy for modifying elements according to their characteristics to show how our analysis can support reusing models. The analysis process can be implemented using a satisfiability modulo theories (SMT) solver, and its effectiveness is demonstrated through case studies.

In the remainder of this paper, we first explain the structure of the Event-B model in Sect. 2. We then present the proposed method for analyzing the impact of changes.
to Event-B and the proposed strategy for modification in Sect. 3 with an illustrative example. We present case studies evaluating our proposed method in Sect. 4, and we conclude the paper in Sect. 5.

2. Background

2.1 Event-B

An Event-B model consists of two types of modules: Context and Machine. A Context module defines the sets and constants used in the model. A Machine module defines the system specifications or how the state variables changes through events while satisfying invariants. Each event contains predicates, namely guards and actions. An event can occur when its guards are all fulfilled and variable values are changed as described in its actions. In Event-B, we start by constructing an abstract model. This model is refined into more concrete ones in a stepwise manner. The Rodin platform provides intensive tool support, including editors, proof obligation generators, and theorem provers [6], [7].

The consistency of models is defined by proof obligations, which define predicates whose satisfaction is required. Intuitively speaking, we first need to ensure consistency within a model step; an invariant always holds (i.e., is established by initialization and then maintained by any possible event occurrences). We also need to ensure consistency between model steps; (a proof on) an invariant in an abstract step is maintained in a refinement (concrete step). We focus on the latter type of consistency and two types of proof obligations.

One type of proof obligation is guard strengthening. Let A denote a list of axioms $A_1,A_2,\ldots$. Let I and G denote a list of abstract invariants $I_1,I_2,\ldots$ and abstract guards $G_1,G_2,\ldots$, and let J and H denote a list of concrete invariants $J_1,J_2,\ldots$ and concrete guards $H_1,H_2,\ldots$. Then, the guard strengthening is expressed as follows.

$$A, I, J, H \vdash G_i$$

Guard strengthening shows that concrete guards imply abstract guards. In other words, concrete guards are stronger than or equal to the abstract ones so that the concrete model does not allow unexpected events to occur in an abstract model.

The other type of proof obligation is action simulation. In addition to the symbols defined above, let Q and T denote a list of abstract before-after-predicates $Q_1,Q_2,\ldots$ and concrete before-after-predicates $T_1,T_2,\ldots$ that denote the effects of actions respectively. Then, the action simulation is expressed as follows.

$$A, I, J, H, T \vdash Q_i$$

Action simulation ensures that the effects of concrete actions do not contradict those of abstract actions and thus does not leading to unexpected states in the abstract model.

2.2 Example

We developed an example Event-B model to demonstrate our method. The model presents a unit with a simple smart grid system that can generate and store electricity for the owner’s use or can supply other systems with it. The generator can store electricity in two types of storage batteries, batteryA and batteryB. BatteryA is used mainly for the owner and batteryB can also be used for the owner, or the electricity can be sent to other units. Before using or sending the stored electricity, the system checks for available batteries.

Figure 1 shows a part of a model that includes several events related through refinements, and Fig. 2 lists the invariants for each model. Figure 3 presents the relationships between symbols defined earlier using this model. The Event-B model is composed of three different abstraction models: the “abstract model,” the “first refinement model,” and the “second refinement model.” The abstract model MR0 is the first model to be developed, which started from 5 events: Initialization, UseBattery, MakeBattery, Unable2Use, and Able2Use (to discriminate between events in a different model, hereinafter we refer to them as model_event: for example, MR0_UseBattery). The abstract model MR0 does not discriminate between the two batteries, and an event deals with the operations of using electricity and sending electricity that are the same from the viewpoint of consuming power. The first refinement model, MR1, focuses on the specification of sending electricity to other units, and actions for implementing this specification are added to the events. Therefore, the number of events increases to 9 through the first refinement. After the second refinement, all specifications of the Smart Grid system are specified in MR2, which has 13 events. Through refinements, refined events properly derive events in the abstract model. For example, the refined events MR2_HasA and MR2_HasAll properly derive the event MR1_HasBattery, and grd1 in MR1_HasBattery can be derived from gr1 in MR2_HasA and invariants battery $\in N$, batteryA $\in N$, batteryB $\in N$, and battery = batteryA + batteryB fulfills guard strengthening.

$$batterynN,$$

batteryA $\in N$,

batteryB $\in N$,

battery = batteryA + batteryB,

batteryA $>$ 0 \wedge

batteryB $=$ 0 \wedge

powerChecked = FALSE \wedge

battery $>$ 0 \wedge

powerChecked = FALSE \wedge

We focus on the difficulty in modifying an existing Event-B model for reuse and evolution. The difficulty in
the modifying mainly lies on the implicitness and the complexity of the relationships between predicates and invariants. For example, it is difficult to judge which predicate is modifiable without comprehending the invariants of the model. Modifying act2 in MR2_HasAll to available := A preserves the consistency of the model, but modifying act2 in MR2_HasA (which looks similar to MR2_HasAll) to available := all breaks the consistency since it will violate inv19 after the event. It seems easy to judge whether the predicates are modifiable in this case. However, in actual use, the number of invariants is likely to grow much larger than a minimum-sized model that is like more than 100 invariants, and this makes analyzing all of relationships between predicates and invariants difficult and burdensome. Another problem related to this difficulty is that the set of variables can change through refinements. To preserve consistency of the model, some invariants called gluing invariants refer to variables of the abstract and the concrete model to explain the relationships between them. Therefore, comprehending how the predicates are refined and what modification can be made is possible by following these relationships between predicates. As the size of a model increases, the number of invariants also increases (including gluing invariants), and following the relationships between predicates becomes difficult. From these reasons, it is important to have assistance in figuring out where and how we can modify the model.

2.3 Related Work

Change impact analysis is an essential task in software engineering. It identifies the potential consequences of a change or estimates what needs to be modified to accomplish a change [8]. Besides simple cases of just traversing traceability links, dependability is the key for the analysis to understand what happens to an element when we change another element. Although investigation on analysis of the source code has been active (e.g., [9]), few studies have focused on targeting formal specifications. Wen et al. presented a method of identifying the impact of requirement changes on design changes by using a formal representation of components and their states [10]. Slicing technique is originally developed for debugging programs by decomposing them and extracting statements that use variables under observation [11]–[13]. There are some approaches for state-based models using this technique, for example, model...
comprehension, model checking, and reuse [14]. However, the main purpose of slicing is for extracting statements, not mainly for comprehending the range of change impact. To our knowledge, change impact analysis of logic-based formal models, which analyzes logical dependency relationships (i.e., dependency of predicates that supports the satisfaction of a given predicate), has not been investigated.

The most significant point in Event-B is designing how a specification is gradually made concrete through refinements. There have been many studies supporting this design. Traichaiyaporn and Aoki [15] adapted concepts from the goal model and the goal refinement patterns of a goal-based knowledge acquisition in automated specifications (KAOS) method [16] as a way closer to how humans refine requirements in a stepwise manner, making refinements easy to understand. Hoang et al. provided design patterns for Event-B [17]. Yeganefard et al. provided a guideline of this refinement design for controller systems [18]. Ma-
toussi et al. used a goal model, and Said et al. used UML, as the high-level representation of refinement design [19], [20]. These methods are focused on converting the design patterns of refinements written in other languages into Event-B, and they do not expect the changes of refinements. Our focus in this paper is a reuse that was not predicted at design time.

There is a lot of several pieces of research on improving the readability and comprehension of Event-B models that may help engineers modify them. Kobayashi et al. [21] developed a method of refactoring refinement structures of an Event-B model. This method applies refactoring on an Event-B model by automatically constructing an intermediate abstraction leveled model. With this refactoring, large complicated refinements are sliced into small simple ones, making it easy to understand the refinement structures of specifications. However, this method focuses on refactoring the structure of models in order to make the part of models easier to extract for reusing and does not target reusing the whole models.

Fathabadi et al. [22] assessed model decomposition techniques that decompose a complex and large Event-B model through refinement. Atomicity decomposition [23] can provide insights into appropriate event decomposition and can support engineers in developing an Event-B model without getting into the technical details in their experiments. These methods develop Event-B model by developing parts of model beforehand and assembling them, not in a monolithic order. Their methods could be applied for supporting reusing models. However, these methods do not support assembling parts that were out of expectation of being assembled beforehand. Our method focuses on this problem and analyzes the impact of modification associated with the change of the strength of predicates in Event-B models. In this paper, we support comprehending what modification is needed to assemble the modified part with the other parts of the model.

Model checkers can also be used to analyze the formal specifications of formal models. ProB [24] is a model checker for the Event-B model that uses simulated models to help engineers comprehend specifications. ProB can also be used to generate test cases and can help engineers confirm whether a specification described in natural language is properly implemented. AnimB [25] is another model checker that helps us understand Event-B models by showing animated models with a graphical interface. However, these tools deal with specific traces and do not help us understand the relationships between predicates in different steps. Our method helps us understand the relationships between predicates and refined ones and decide how to modify the model without breaking the consistency.

3. Method for Analyzing and Classifying Predicate Refinement

3.1 Overview

Our idea is to facilitate comprehension of the degree of difficulty in modifying predicates by showing the results of analyzing and classifying logical dependency relationships of modeling elements (i.e., predicates of guards and actions) by labeling it directly on each predicate. This information makes it possible to judge how the predicate should be modified with the label since some modification steps can be skipped according to the labels. Support for this judgment plays an important role in change impact analysis for Event-B models. Comprehension of how much impact changes have is useful for modifying models because it is possible to estimate the modification’s difficulty.

Our idea of labeling the type of logical dependency relationships on each predicate is based on the idea that this method has to be able to support the smallest modifications, such as only modifying one predicate. Since models can be reused for similar situations through this method, it must be able to make minor changes. Therefore, we analyzed the relationships between other predicates and invariants in order for each predicate to comprehend the situation of each predicate within the model. With labels, we can modify models from the predicate level, which allows us to make the most of existing models in developing new models.

Our method labels both guards and actions, but the labels labeled on guards and on actions are different. We define the former guard labels and the latter action labels. We explain the aim of each label.

We start with our discussion on the guard labels. Through refinements, guards need to be stronger or equal to each other (the proof obligation of guard strengthening). Modifying guards to make them stronger is not that difficult since it always fulfills proof obligations. Therefore, we want to know how to modify the guard into one that cannot be derived from a previous guard. From here, we only focus on this modification. During this modification, it is not certain whether such a guard can be replaced directly without modifying another part of its event or invariants, depending on how its last refinement was done.

Two main things need to be considered when modifying guards. One is that all invariants are fulfilled even after the event and the other is that guard strengthening is fulfilled. The reason why we need to consider the first point is that it is not verified in cases that are not in the previous guard. Other cases had already been verified when the previous model was developed, but for those cases, we need to verify them by checking all invariants and confirming that no cases violate the invariants. For the second point, the modified guard must be confirmed as stronger or equal to the one before refinement. Since the variables used can change with gluing invariants through refinements, we need to understand how the refinement was conducted in order to comprehend how we can modify guards. When the previous guard is equal to the one before refinement, no other guard that cannot derive the previous guard can replace it since the guard can only replace the previous guard by being stronger or equal to it. Therefore, comprehending whether the guard is modifiable is important in order to judge where to modify.

When refining actions, refined actions may be kept
equal to each other or pruned (less non-deterministic). The actions need to satisfy the proof obligation of an action simulation, thus keeping the constraints on the effect defined in the abstract actions. However, variables used in the model can change through refinements. In some cases, some variables in action can be updated with several values; in other cases, some variables in action can only be updated with the one in the refined action. Distinguishing these cases is important because the former case may be able to modify the action directly while the latter case at least needs to modify other actions, guards, or invariants to modify the action.

Next, we provide analysis and classification for each of both the guards and the actions. The results of analysis are attached as labels on guards and actions in concrete models so that engineers can use the information to judge which modifications are easily acceptable (or not).

3.2 Classifying Guard Labels

Before classifying with our method, we dealt (syntactically) with multiple guards as one single guard as the conjunction of all of the guards. This is because we focused on the states under which an event can occur, which are defined by the conjunction. Thus, we attached a label to a set of guards in each event, not for each guard.

Figure 4 shows how guards are classified. In order to classify guards, our method first analyzes a guard by checking if the guard has become stronger than the abstract one; in other words, we check if the guard is equal to the abstract one (Step (a) in Fig. 4). This can be checked with the following logical expression.

\[ A, I, J, G, T \vdash H \]

When this logical expression is true, the abstract guard implies the concrete guard. Since this guard fulfills strengthening, the guard is equal to the abstract guard. On the other hand, when this logical expression is false, the guard is stronger than the abstract guard. Our method utilizes the Z3 SMT solver to find a state that is a counterexample of this logical expression. If the solver finds a counterexample of this logical expression, it is considered an example of the state that is pruned in the refinement, meaning that the \( H \) concrete guard is stronger than the \( A \) abstract guard.

When the concrete guard is stronger than the abstract one, our method checks whether there are any ways to modify the guard without also changing actions or invariants (Step (b) in Fig. 4). This can be checked with the following logical expression.

\[ A, I, J, G, T \vdash H \]

When the solver finds a counterexample, the guard is labeled with \( Extracted \). This label shows that the guard was extracted from the abstract guard by pruning other valid cases through the refinement. The guard could be modified directly without modifying actions or invariants since the event does not violate any invariants and at least fulfills proof obligations when occurring under the situation in the counterexample found in this step. On the other hand, if the solver could not find a counterexample, the guard is labeled with \( Limited \). This label shows that the guard was limited to its range with refined actions or invariants. To modify the guard, actions or invariants are also modified.

In Step (a) in Fig. 4, when the guard is equal to the abstract guard, the method checks if the guard is symbolically different from the abstract one (Step (c) in Fig. 4). This can be checked by checking the logical expression using the symbols we defined in Sect. 2.1 below.

\[ A, I, J, G \vdash H \]

When this logical expression is true, the abstract guard implies the concrete guard. Since this guard fulfills strengthening, the guard is equal to the abstract guard. On the other hand, when this logical expression is false, the guard is symbolically different from the abstract guard. Our method utilizes the Z3 SMT solver to find a state that is a counterexample of this logical expression. If the solver finds a counterexample of this logical expression, it is considered an example of the state that is pruned in the refinement, meaning that the \( H \) concrete guard is stronger than the \( A \) abstract guard.

When the concrete guard is stronger than the abstract one, our method checks whether there are any ways to modify the guard without also changing actions or invariants (Step (b) in Fig. 4). This can be checked with the following logical expression.

\[ A, I, J, G, T \vdash H \]

When the solver finds a counterexample, the guard is labeled with \( Extracted \). This label shows that the guard was extracted from the abstract guard by pruning other valid cases through the refinement. The guard could be modified directly without modifying actions or invariants since the event does not violate any invariants and at least fulfills proof obligations when occurring under the situation in the counterexample found in this step. On the other hand, if the solver could not find a counterexample, the guard is labeled with \( Limited \). This label shows that the guard was limited to its range with refined actions or invariants. To modify the guard, actions or invariants are also modified.

In Step (a) in Fig. 4, when the guard is equal to the abstract guard, the method checks if the guard is symbolically different from the abstract one (Step (c) in Fig. 4). If the guard is symbolically different, it is labeled with \( Replaced \) since variables used in the guard are replaced, while the meaning of the guard does not change with gluing invariants. If the guard is symbolically the same, it is labeled with \( Unchanged \), because it does not change.

3.3 Classifying Action Labels

Like guard labels, Fig. 5 shows how actions are classified. In order to classify actions, our method first analyzes an action by checking if the action can be derived from abstract guards
and actions (Step (d) in Fig. 5). This is checked by the following logical expression. Let \( k \) be an integer for which \( T_k \) is the concrete action under analysis.

\[
A, I, J, G, Q \vdash T_k
\]

When this logical expression is true, the action is implied by the conjunction of an abstract guard, actions, and invariants, meaning that this action is necessary in this refinement. If it is false and the SMT solver finds a counterexample, we move on to Step (e) in Fig. 5.

In Step (e) in Fig. 5, our method checks if the action can be derived from previous guards and actions. The logical expression checked here is:

\[
A, I, J, Q, H, T_{ik} \vdash T_k.
\]

If a counterexample is also found at this point, the label of the action is Chosen. The counterexample shows the other value that the variable can have in this event without changing any other part of the model. Therefore, we can say that this action has several choices for updating its variable in the refinement and its value was chosen. Since this action can be modified without changing any other actions or its guard, there is a chance of modifying it directly. However, if no counterexample is found, the action is labeled Restricted. An action labeled Restricted has been restricted to the value it has because of the stronger guard, refined actions, or new invariants through refinement. But this action can be modified if the guard or other actions are modified along with it since it has a counterexample in Step (e). Therefore, actions labeled Restricted can be modified without modifying invariants.

In Step (d) in Fig. 5, if there is no counterexample, our method then checks if the action is symbolically different from all abstract actions (Step (f) in Fig. 5). If the action is symbolically different from every abstract action, it is labeled Specified since the value of the variable updated in the action is specified only to the one in the action through refinement because of the restriction made by the invariants or the relationship described by the gluing invariants. This action cannot be modified unless the invariants are changed. If the action is symbolically the same, it is labeled Untouched, as the action remains the same through refinement.

### 3.4 Modification Strategies

We have seen the classification of guard labels: Extracted, Limited, Replaced, and Unchanged. And we have seen the classification of action labels: Chosen, Restricted, Specified, and Untouched. From here, we discuss how the labels are used to modify models effectively.

First, we discuss modifying a guard. The strategy for modifying a guard effectively can be specified comprehensively below.

**Extracted**

As shown in Sect. 3.2, the Extracted label means that there are some cases in which an event is pruned during the refinement. In order to include these cases so that the event can occur, we can modify directly without modifying actions or invariants. Since there is a chance that our intended modification is included in the pruned cases, our intended modification should be tried directly first. If it fails, we can apply the strategy of label Limited below.

**Limited**

The label Limited tells us that we need to modify actions or invariants as well as the guard since the previous guard is derived from them and cannot be weakened any further unless they are changed. In this step, we first determined which invariants were related to the guard by finding common variables from the changes we made in the guard. Then, we checked to see if we can avoid violating invariants by modifying actions. If it was possible, we found other invariants that have common variables and adjusted actions until no more adjustments were needed. If it was not possible to
avoid violating invariants by modifying on actions, we moved on to the modifying strategy of label Replaced.

**Replaced**

*Replaced* indicates that the guard seems different but in actuality, it is impossible to modify the guard (except for making it stronger) by only modifying the predicates in the event. If we really need to modify this guard, we might be able to modify it by modifying gluing invariants related to the guard, which is highly inadvisable since this modification affects the entire model and makes it difficult to conserve consistency.

**Unchanged**

The *Unchanged* label is similar to the *Replaced* label, but the guard is both semantically and symbolically the same as its abstract guard. The only modification allowed here is making the guard stronger or changing variables by adding new ones to the model and preparing some gluing invariants to depict the relationships between the abstract guard and the new guard. However, we should keep in mind that there is still a possibility that we can develop an event with the guard we want by starting modification from a more abstract event since it has a weaker guard.

The guard that is considered the easiest to modify is labeled *Extracted*, followed in order by the ones labeled *Limited*, *Replaced*, and *Unchanged*. While it may be possible to modify the guard directly with *Extracted*, guards with other labels require us to modify a broader part of the model. Guard labels allow us to set our focus on a proper part of the model to modify each guard effectively. Therefore, guard labels are well classified for helping us modify guards.

Now, we discuss modifying an action. Like with guard labels, we detail the strategy for modifying an action effectively (below).

**Chosen**

We defined the *Chosen* label in Sect. 3.3. The variable in the action labeled *Chosen* has other values that it can have without modifying others part of the model. Therefore, the intended modification should be tried directly at first, and if it does not work, the strategy described in *Restricted* label below should be tried next.

**Restricted**

The *Restricted* label shows that the variable in the action can only take one value because the other part of the event is being refined. In order to modify this action, it might be possible to modify it by also adjusting other parts of the models. To modify it, we checked the invariants that were related to the action by finding common variables in the action. Then, we checked if we could avoid violating the invariants we found by adjusting the guard and other actions. Repeatedly finding invariants that have common variables with the guard or action when they are adjusted until the event does not violate any invariants. If we find out that we cannot modify the action by only modifying other actions and the guard, we apply the strategy of the *Specified* label.

**Specified**

The *Specified* label shows that the action cannot be modified unless the invariants change. As we mentioned about the modifying strategy of the *Replaced* guard label, modifying invariants could affect other events and make it difficult to conserve the consistency of the model, but it is not necessarily impossible. However, there is another way to modify the actions with *Specified* label that does not modify invariants of the model, explained in the strategy of the *Untouched* label.

**Untouched**

Actions labeled *Untouched* do not change through refinement, and they cannot be modified unless new variables are added with their gluing invariants to the model because of the action simulations. By adding new variables with gluing invariants that refer to the new variables and the variables used in the action, we can modify actions into more concrete ones, which do not violate the proof obligation of action simulations. Another choice for a developing new event that has the action we want is to give up modifying this event and start modifying from a more abstract event that has a different action label, similar to the strategy for the *Unchanged* guard label.

The *Chosen* action is expected to be the easiest to modify — followed in order by *Restricted*, *Specified*, and *Untouched* — since the area of the model we need to comprehend and modify in the beginning of modification enlarges in this order. While we may only finish modifying the action when the action is labeled *Chosen*, we have to comprehend how the event contributes to the model when the action is labeled *Restricted*, and we even have to start considering changing the whole model by modifying invariants when the action is labeled *Specified* and *Unchanged*. Since the action labels we defined present where to focus and prevent us from trying to modify in impossible ways, it is considered that the action label is well classified for supporting the modification of actions as well.

In above, we confirmed that the classification we made for guards and actions helps engineers modify the model by clarifying the modifications the predicates need. In the next section, we demonstrate some examples of modifying models using the model we developed in Sect. 2.2 to confirm some characteristics of refinements presented by labels.

### 3.5 Example of Using Labels

We demonstrate the process of modifying models using the labels we defined and labeled on the model with our method.
Figures 6 and 7 focus on guards and actions of a part of a Smart Grid model that were already analyzed by our method and had labels attached. Table 1 shows the result of labeling with the events in Figs. 6 and 7 in detail. Here, we describe how the events were modified using these labels.

3.5.1 Cases with Guard Labels

First, we can see the label Extracted on the guard of MR1_Unable2Use. This label shows that this guard is stronger than that in M0_Unable2Use and has some cases pruned through refinement. We can confirm this characteristic of this label with an example of the case that is pruned through refinement below.

\[
\text{battery} = 0, \ function\text{Checked} = \text{FALSE}, \\
\text{power\text{Checked}} = \text{FALSE}
\]

Therefore, we can modify the MR1_Unable2Use event by modifying grd1 into the following predicate:

\[
\text{battery} = 0 \land \ function\text{Checked} = \text{FALSE} \\
\land \ power\text{Checked} = \text{FALSE}
\]

However, we need to understand that not all modifications can be made without modifying other parts of the event since the label only shows what modification is impossible with its predicate. For example, we cannot modify the MR1_Unable2Use event by replacing the grd1 into the following predicate:

\[
\text{battery} = 0 \land \ function\text{Checked} = \text{TRUE} \\
\land \ power\text{Checked} = \text{TRUE}.
\]

This modification cannot be made since it does not fulfill the proof obligation, guard strengthening. We should keep in mind that the label only helps us decide a strategy for modifying predicates, but it does not guarantee that the modification can be made with the strategy.

Here, we focus on the guard in MR2_Unable2Use. It is labeled Replaced and its characteristics show that the strength of this guard is equal to the one before refinement, even though several places on the guard have changed. To confirm this, we needed to examine invariants to understand why they are equal. With inv1, inv8, inv9, inv10, inv11, inv14, inv17, inv18, and inv19, the guard of MR1_Unable2Use is stronger than that of MR2_Unable2Use. Let Axioms denote a list of axioms of this model.

\[
\text{Axioms}, \ inv1, \ inv8, \ inv9, \ inv10, \ inv11, \ inv14, \ inv17, \ inv18, \ inv19, \ \text{grd1} \text{ in } MR1\text{\_Unable2Use} \\
\rightarrow \\
\text{grd1} \text{ in } MR2\text{\_Unable2Use}
\]

From the guard strengthening, we can see that MR2_Unable2Use is stronger than MR1_Unable2Use. Therefore, guards are clearly equal here and the guard in MR2_Unable2Use cannot be modified without making the guard stronger or also modifying invariants through the refinement. As for modifying the invariants, one example is
Fig. 7  Labeled actions in Smart Grid model

Table 1  Analysis result of example model

| MR0: abstract model | MR1: first refinement | MR2: second refinement | Label |
|---------------------|-----------------------|------------------------|-------|
| **Event** Unable2Use | **Event** Unable2Use | **Event** Unable2Use | **Label** |
| **grp1** | **grp1** | **grp1** | **Label** |
| battery = 0 ∧ functionChecked = FALSE | battery = 0 ∧ functionChecked = FALSE ∧ powerChecked = TRUE | batteryA > 0 ∧ batteryB > 0 ∧ powerChecked = FALSE | Extracted |
| **ac1** use := unable | **ac1** use := unable | **ac1** use := unable | Replaced |
| **ac2** send := unable | **ac2** send := unable | **ac2** send := unable | Untouched |
| **ac3** functionChecked := TRUE | **ac3** functionChecked := TRUE | **ac3** functionChecked := TRUE | Untouched |
| **Event** HasBattery | **Event** HasBattery | **Event** HasA | **Label** |
| **grp1** | **grp1** | **grp1** | **Label** |
| battery > 0 ∧ powerChecked = FALSE | batteryA > 0 ∧ batteryB > 0 ∧ powerChecked = FALSE | available := A | Extracted |
| **ac1** powerChecked := TRUE | **ac1** powerChecked := TRUE | **ac2** available := A | Untouched |
| **ac2** available := all | **ac2** available := all | **ac2** available := all | Restricted |
| **Event** HasAll | **Event** HasA | **Event** HasA | **Label** |
| **grp1** | **grp1** | **grp1** | **Label** |
| batteryA > 0 ∧ batteryB > 0 ∧ powerChecked = FALSE | batteryA > 0 ∧ batteryB > 0 ∧ powerChecked = FALSE | available := A | Limited |
| **ac1** powerChecked := TRUE | **ac1** powerChecked := TRUE | **ac1** powerChecked := TRUE | Untouched |
| **ac2** available := all | **ac2** available := all | **ac2** available := all | Chosen |
modifying the invariant \texttt{inv18} into the following predicate:

\begin{align*}
\text{inv18}': \\
power\text{Checked} = \text{TRUE} \land \text{available} \in (A,\text{all}) \\
\Rightarrow \text{batteryA} \geq 0,
\end{align*}

when we want to modify the guard into the one below.

\begin{align*}
\text{grd1}' : \\
\text{function}\text{Checked} = \text{FALSE} \land \\
power\text{Checked} = \text{TRUE} \land \\
\text{available} \in [A]
\end{align*}

However, since we changed an invariant of the model, we have to check if other events preserve proof obligations and modify them to adapt to the new invariant.

The guard in MR2.HasAll is labeled \textit{Limited}. This characteristic shows that this guard is stronger than the one before refinement and can be confirmed with the following logistic expression.

\begin{align*}
\text{Axioms}, \text{inv11, grd1 in MR1 HasBattery,} \\
\neg (\text{grd1 in MR2 HasAll}) \\
\models \bot
\end{align*}

Therefore, we can see that the guard in MR2.HasAll is stronger than that in MR1.HasBattery. However, we can see that we cannot make any direct modification with the guard (except for making it stronger) since this guard can be derived from \textit{inv18, inv19, act1} in MR2.HasAll, and \textit{act2} in MR2.HasAll.

\begin{align*}
\text{Axioms, inv18, inv19, act1 in MR2 HasAll,} \\
\text{act2 in MR2 HasAll} \\
\models \\
\text{grd1 in MR2 HasAll}
\end{align*}

To modify this guard, we should seek invariants that have a variable in common with the guard from the strategy. We found that there are \textit{inv8, inv9, inv10, inv11, inv12, inv13, inv17, inv18, inv19, and inv20}. Here, imagine that we want to modify the guard as below.

\begin{align*}
\text{batteryA} > 0 \land \text{batteryB} > 0 \land \\
\text{powerCheck} = \text{FALSE}
\end{align*}

Focusing on the \textit{inv18}, we can expect that by modifying the \textit{act2} in MR2.HasAll to \textit{available := A}, the modification on guard does not violate any invariants. Therefore, we confirmed we can modify the event with label \textit{Limited} by modifying other parts of the model. However, the results of this modification are exactly the same as the MR2.HasA event. There are some cases in which the results of adjusting guards and actions are the same as other events. Therefore, we should keep in mind that this could happen during the modification.

### 3.5.2 Cases with Action Labels

Now, we focus on the labels for actions and confirm their characteristics.

We focused on an action in MR2.HasAll first. The action \textit{act2} in MR2.HasAll was labeled \textit{Chosen}. This label informed us that the variable in this action could have a different value without changing the guard or other actions. In this case, we could change \textit{act2} to \textit{available := A} without violating proof obligations.

In contrast with the case of MR2.HasAll, we could see that \textit{act2} in MR2.HasA was labeled \textit{Restricted}, which means that this other part of the event had to be modified together when the action was modified. We used the following logical expression to confirm that this action could not be modified without modifying something else.

\begin{align*}
\text{Axioms, inv18, inv19, inv20, grd1 in MR2 HasA,} \\
\text{act1 in MR2 HasA, \neg (available := A)} \\
\models \bot
\end{align*}

For an example of modifying the action along with other actions or guards, we needed to check invariants with common variables that were \textit{inv14, inv15, inv16, inv17, inv18, inv19, and inv20}. Imagine that we wanted to modify the action into \textit{available := B}. By focusing on \textit{inv19}, we could expect to modify the action without violating any invariants by modifying the \textit{grd1} in MR2.HasA to the following predicate.

\begin{align*}
\text{batteryA} > 0 \land \text{batteryB} > 0 \land \\
\text{powerCheck} = \text{FALSE}
\end{align*}

Therefore, we could see that there is a way to modify the action with the label \textit{Restricted} that only modifies within the event.

Finally, we checked the characteristics of the refinements of the \textit{Specified} label from the MR1.Unable2Use event. The \textit{Specified} label shows that we needed to modify invariants to modify it. This is easily apparent from the \textit{inv7} and \textit{act1} in MR1.Unable2Use. We could not modify \textit{act1} since it is necessary because of the action simulation. Therefore, we needed to modify or remove the \textit{inv7} to modify the action, though this could break many proof obligations of other events.

We did not check the guard label \textit{Unchanged} or the action label \textit{Untouched} since the characteristics of the refinement were simple and it was apparent to be difficult to modify their predicates.

We saw that the labels showed the characteristics of refinements we explained at the classification method and could modify the predicates using the strategy we proposed. Therefore it is probable that our method can help engineers in comprehend the relationships between predicates through refinements and help them reuse models by modifying their predicates according to the labels labeled on the predicates.
4. Experiments

4.1 Settings

We applied our method to a model for controlling cars on a bridge from Abrial [1] as a case study of model modification using the labels of our method. This Bridge model was refined 3 times, and its events increased from 3 to 17. This model of a narrow bridge that linked a mainland to a small island and used sensors and traffic lights to control the number of cars on the island and the cars getting onto the bridge. In this experiment, we defined MR0 as a Machine model operated with zero refinement, in other words, the most abstract Machine model, and the number attached to its name increased as the model was refined. For example, Machine model was defined as MR1 in the first refinement model and as MR2 in the second refinement model (we refer to the name of the model as MR0, MR1, ... for simplicity hereafter). We applied our method to this model in order to label each predicate and modify some of them according to their label type.

4.2 Case Studies

In these case studies, we focused on evaluating whether our labels could help us understand how their predicates should be modified and help develop new events that are not trivial; for example, an event that already exists. Figure 8 presents how events were modified or judged as difficult to modify in each case study we made. Figure 9 lists the invariants for the Bridge model.

4.2.1 Modification Using Guard Labels

Table 2 shows the distribution of the guard labels labeled on the Bridge model. In this model, we could find all kinds of labels, except for Replaced. However, we only attempted to modify guards labeled Extracted and Limited, since guards with other labels can only be made stronger, which was applicable to any guard.

Case 1: Extracted

In this case, we developed a new event by modifying the guard of ML_out_1 event in model MR2. The ML_out_1 event represented a car crossing the bridge to the island. This event can occur when the traffic light on the mainland is green and the total number of cars on the island and the bridge including the one getting on it is smaller than the maximum number of cars allowed to be on them. In this model, variable ml_tl presented the traffic light on the mainland and il_tl presented the traffic light on the island. Variable a presented the number of cars heading to the island on the bridge. Variable b presented the number of cars on the island. Variable c presented the number of cars heading to the mainland on the bridge. In addition, constant d presented the maximum number of cars allowed to be on the island and the bridge.

We attempted to modify this event for when there is a need to let vehicles cross the bridge from the mainland to the island even if the traffic light is red (for example, letting an ambulance cross the bridge in an emergency). In order to make such an event, we needed to modify \( ml_{tl} = \text{green} \) into \( ml_{tl} = \text{red} \). Since this guard is labeled Extracted, we could see that there were some guards that could replace this guard without strengthening it. Therefore, we attempted to modify the guard directly, and we adjusted it to fulfill each proof obligation to successfully develop the event by modifying the guard as below.

\[
ml_{tl} = \text{red} \land a + b + 1 < d \land il_{tl} = \text{red} \land c = 0
\]

As a result, we successfully developed a new event by only modifying a guard labeled Extracted.

Case 2: Limited

In this case, we attempted to develop a new event by modifying the guard of the ML_out_1 event in model MR3. This event was a refined version of the one we modified in Case 1. In this model, the event ML_out_1 presents the event of a car crossing the bridge to the island like the one in Case 1; however, while ML_out_1 in model MR2 controls the system with only traffic lights, ML_out_1 in model MR3 controls the system using sensors. In this model, variable ml_out_10 presented the signal of cars getting onto the bridge from the mainland and different from the ones in MR2. We deal with variables \( a, b, \) and \( c \) as the number of cars that the system was aware of, separated from the actual number of cars, which were presented by variables \( A, B, \) and \( C \).

We attempted to modify this event when the sensors malfunction and had to increment the number of cars on the bridge. However, since this event’s guard is labeled Limited, we can understand that the replaced guard has to be a stronger version from guard strengthening in order avoid breaking the consistency of the model if we directly modify the guard. Since the event we intended to develop required a guard including \( ml_{out_1} = \text{FALSE} \), we checked and found out that this could not become a stronger guard from the logical expression below.

\[
\text{inv}_{30}, (\text{guard in MR2}_ML_{out_1}) = \text{TRUE}
\]

\[\vdash \perp\]

Therefore, modifying the guard directly without breaking the consistency of the model was impossible. If we really desire to modify this guard, actions and invariants must be modified along with the guard to preserve the consistency, which is likely to affect many
other events and make it difficult to preserve the consistency of the model.

4.2.2 Modification Using Action Labels

Table 3 shows the distribution of the action label on the Bridge model. We were able to find all kinds of action labels. Here, we attempted modification on actions labeled Chosen, Restricted, and Specified. We did not attempt modification on actions labeled Untouched since the method is the same as the one operated on actions labeled Specified.

Case 3: Chosen

In this case, we developed a new event by modifying one action of event ML_out_1 in model MR2. The situation where it could occur was already explained in Case 1. This event increased the number of cars crossing the bridge heading for the island and changed the value of variable ml_pass to TRUE. Variable ml_pass describes whether there was at least one car heading to the island and il_pass describes whether there was at least one car heading to the mainland. When these are TRUE, they describe at least one car being on the bridge; when these are FALSE, no car was on the bridge.

We attempted to modify this event for when there is a need to deal with a vehicle that gets onto the bridge and does not leave, instead parking where it got onto bridge. To fulfill this goal, we had to modify the second action ml_pass := TRUE to ml_pass := FALSE in
our plan since a car heading to the vehicle base left the road to the island soon after getting onto the bridge. We had to consider that no car would pass over the bridge while the number of cars on the bridge increased. Since the second action was the labeled Chosen, we could see that FALSE was one other value that could be replaced with TRUE. Therefore, we attempted to modify the action directly, and we confirmed that all proof obligations were fulfilled without modifying any other part of the model.

As a result, we succeeded in developing a new event by only modifying an action that has a label Chosen.

**Case 4: Restricted**

In this case, we developed a new event by modifying one action of the ML out 1 event in model MR3. The situation which it could occur was already explained in Case 2. This event increases the number of cars the system is aware of and the system changes the state of ml pass to TRUE and resets the sensor for detecting cars getting onto the bridge from the mainland.

We attempted to modify this event for cases when there is a need for dealing with cars that cross the bridge in a row without stopping. To fulfill this goal, we modified the third action ml out 10 = FALSE to ml out 10 = TRUE because a car got onto the bridge soon after the car in front of it. However, since the third action was the labeled Refined, we could understand that we needed to modify the third action and at least some other actions or its guard. Therefore, for our first modification step, we checked invariants that were related to the third action. Since the third action had the ml out 10 variable, inv36, inv37, inv38, and inv39 had to be considered from our strategy. By analyzing the behavior of the event, variable A was also
updated through the event in order to fulfill invariants after the event. We added $A := A + 1$ to the action and checked invariants that were related to this new action again and found inv48 and inv49. To fulfill these invariants, we modified the guard by making it stronger with $A + B + 1 < d \land C = 0$. After confirming that the new event we developed fulfilled all proof obligations, we succeeded in developing a new event with the action $ml_{out\_10} = TRUE$.

Summarizing the above, we succeeded in developing a new event by only modifying predicates within the event that was indicated by the Restricted label.

**Case 5: Specified**

In this case, we attempted to develop a new event by modifying one action of the ML_out_2 event in the MR3 model. The situation in which it could occur resembled the ML_out_1 event, but the sum of cars on the bridge and the island differed from the one with ML_out_1. After the event, the sum of cars on the bridge reached the maximum number $d$; therefore, the event changed the traffic light on the mainland to red so that no more cars could get onto the bridge.

We attempted to modify this event for the same case we considered in Case 4. To fulfill this goal, we had to modify the fourth action $ml_{out\_10} = FALSE$ to $ml_{out\_10} = TRUE$ with the same reason in Case 4. However, Case 4 differs in that the fourth action is labeled Specified. Therefore, we had to modify invariants to modify the fourth action, which required us to be extremely careful with the delicate relationships between events and invariants. To confirm that modifying invariants is necessary, we assumed that it was possible to develop the event with $ml_{out\_10} = TRUE$ out of ML_out_2 without modifying invariants and breaking the consistency of the model. Because of the inv30 invariant, the second action $ml_{pass} = red$ had to be modified into $ml_{pass} = green$. However, this violated the proof obligation of action simulation, since $ml_{pass} = green$ contradicted with the second action in ML_out_2 in MR2. Therefore, we could not modify the fourth action $ml_{out\_10} = FALSE$ of the ML_out_2 event.

Regarding the above, modifying the fourth action without modifying invariants was impossible. If we really desire to modify this guard, we have to modify invariants along with the action to preserve the consistency, which is difficult and burdensome.

### 4.2.3 Evaluation Summery

These cases made it possible for us to comprehend what is necessary to modify predicates from their labels, and we modified the predicates unless the modifications did not involve invariants that made it difficult to preserve the consistency of the model. Moreover, the results of the successful modifications were all non-trivial. In cases like Case 3, label Chosen supported modifications and greatly reduced the burden of modifications by making analyzing relationships between invariants and predicates unnecessary to preserve consistency of the model. Therefore, our change impact analysis is efficient at supporting the modification of models from the view of understanding the process of modification and confirming the consistency.

### 5. Conclusions

We proposed analyzing the impact of the changes of refinements made to formal specifications and described a method for analyzing the changes to help engineers change the formal specifications of a formal model. In Sect. 3, we discussed the labeling method for analyzing and classifying logical dependency relationships of predicates and developed a modification strategy for each label. In Sect. 4, we conducted a case study and confirmed that our method can support modifying models.

Further research on how the labels can be used to develop formal models is an important way to promote using formal methods. Since our classification has distinct information on refinement relationships, we might be able to help develop formal models by generating them automatically according to labels in the future or provide templates for modifying models according to their labels.

### References

[1] J.R. Abrial, Modeling in Event-B: System and Software Engineering, 1st ed., Cambridge University Press, New York, NY, USA, 2010.

[2] A. Romanovsky and M. Thomas, Industrial Deployment of System Engineering Methods, Springer Publishing Company, Incorporated, 2013.

[3] Copyright Deploy-Project, “Deploy: Industrial deployment of system engineering methods providing high dependability and productivity,” http://www.deploy-project.eu/index.html, accessed Sept. 15, 2018.

[4] S. Saruwatari, F. Ishikawa, T. Kobayashi, and S. Honiden, “Extracting traceability between predicates in event-b refinement,” Asia-Pacific Software Engineering Conference (APSEC), 2017 24th, pp.61–70, IEEE, 2017.

[5] S.A. Bohner and R.S. Arnold, Software Change Impact Analysis, IEEE Computer Society Press, includes bibliographical references, pp.361–374, Los Alamitos, CA, USA, 1996.

[6] Copyright Dependable Systems and Software Engineering Research Group, “Event-B,” http://www.event-b.org/index.html, accessed Sept. 15, 2018.

[7] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin, “An open extensible tool environment for event-b,” International Conference on Formal Engineering Methods, vol.4260, pp.588–605, Springer, 2006.

[8] R.S. Arnold, Software Change Impact Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[9] M. Acharya and B. Robinson, “Practical change impact analysis based on static program slicing for industrial software systems,” Proc. 33rd International Conference on Software Engineering, pp.746–755, ACM, 2011.

[10] L. Wen and R.G. Dromey, “From requirements change to design change: A formal path,” Proc. Second International Conference
Shinnosuke Saruwatari is a student enrolled in a master’s degree program at the Graduate School of Information Science and Technology, The University of Tokyo. He received his Bachelor of Science degree from The University of Tokyo in 2017. His research interests include Software Engineering, Formal Methods, and Model Checking.

Fuyuki Ishikawa is an associate professor at the National Institute of Informatics (NII), Japan. He is also a visiting associate professor at The University of Electro-Communications. He received a Ph.D. in Information Science and Technology from The University of Tokyo, Japan in 2007. His research interests include Software Engineering, Formal Methods, Testing, Service-Oriented Computing, Cyber-Physical Systems, and Machine Learning Systems Engineering.

Tsutomu Kobayashi is a researcher at the National Institute of Informatics (NII), Japan. He received his Ph.D. degree in Information Science and Technology from the University of Tokyo, Japan, in 2017. His research interests include formal methods, software engineering, and requirements engineering.

Shinichi Honiden received a Ph.D. degree in Electrical Engineering from Waseda University, Tokyo, Japan, in 1986. From 1978 to 2000, he worked at Toshiba Corporation. Since April 2000, he has been a professor at the National Institute of Informatics (NII), Japan, and he has served as a deputy director general at NII since 2012. He also worked as a professor in the Graduate School of Information Science and Technology at the University of Tokyo from 2001 to 2018. He is now a professor at Waseda University and a director at the Center for Global Research in Advanced Software Science and Engineering (GRACE Center). He has been a visiting professor at University College London (UCL) in the UK. His research interests include Agent Technology, Pervasive Computing, and Software Engineering.

Shinnosuke Saruwatari is a student enrolled in a master’s degree program at the Graduate School of Information Science and Technology, The University of Tokyo. He received his Bachelor of Science degree from The University of Tokyo in 2017. His research interests include Software Engineering, Formal Methods, and Model Checking.

on Software Engineering and Formal Methods, 2004. SEFM 2004, pp.104–113, IEEE, 2004.

[11] M. Weiser, “Program slicing,” Proc. 5th International Conference on Software Engineering, pp.439–449, IEEE Press, 1981.

[12] S. Schoenig and M. Ducassé, “A backward slicing algorithm for prolog,” International Static Analysis Symposium, vol.1145, pp.317–331, Springer, 1996.

[13] L.I. Millett and T. Teitelbaum, “Slicing promela and its applications to model checking, simulation, and protocol understanding,” Proc. 4th International SPIN Workshop, pp.75–83, Citeseer, 1998.

[14] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt, “State-based model slicing: A survey,” ACM Computing Surveys (CSUR), vol.45, no.4, p.53, 2013.

[15] K. Traichaiyaporn and T. Aoki, “Refinement tree and its patterns: a graphical approach for event-b modeling,” International Workshop on Formal Techniques for Safety-Critical Systems, pp.246–261, Springer, 2013.

[16] A. Lapouchcian, “Goal-oriented requirements engineering: An overview of the current research,” Depth Report, University of Toronto, vol.32, 2005.

[17] T.S. Hoang, A. Fürst, and J.-R. Abrial, “Event-b patterns and their tool support,” Softw. Syst. Model., vol.12, no.2, pp.229–244, May 2013.

[18] S. Yeganefard, M. Butler, and A. Rezazadeh, “Evaluation of a guideline by formal modelling of cruise control system in event-b,” 2010.

[19] A. Matoussi, F. Gervais, and R. Laleau, “A goal-based approach to guide the design of an abstract event-b specification,” 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, pp.139–148, IEEE, 2011.

[20] M.Y. Said, M. Butler, and C. Snook, “Language and tool support for class and state machine refinement in uml-b,” International Symposium on Formal Methods, vol.5850, pp.579–595, Springer, 2009.

[21] T. Kobayashi, F. Ishikawa, and S. Honiden, “Refactoring refinement structure of event-b machines,” International Symposium on Formal Methods, vol.9995, pp.444–459, Springer, 2016.

[22] A.S. Fathabadi, A. Rezazadeh, and M. Butler, “Applying atomicty and model decomposition to a space craft system in event-b,” NASA Formal Methods Symposium, vol.6617, pp.328–342, Springer, 2011.

[23] M. Butler, “Decomposition structures for event-b,” International Conference on Integrated Formal Methods, vol.5423, pp.20–38, Springer, 2009.

[24] M. Leuschel and M. Butler, “Prob: A model checker for b,” International Symposium of Formal Methods, vol.2805, pp.855–874, Springer, 2003.

[25] C. Métayer, “AnimB.” http://www.animb.org/index.xml, accessed Sept. 15. 2018.