Analysis on Strategies of Superposition Refinement of Event-B Specifications

Tsutomu Kobayashi(✉) and Fuyuki Ishikawa

National Institute of Informatics, Tokyo, Japan
{t-kobayashi,f-ishikawa}@nii.ac.jp

Abstract. The superposition refinement with the Event-B modeling method is useful because it supports construction of models in multiple abstraction levels, and thus mitigates the burden of constructing rigorous models. With such a refinement mechanism, developers can choose which subset of a target system's elements is specified in each abstraction level (refinement strategy). Although differences of refinement strategies for a model affect the complexity of modeling and verification, the effect has not been studied. We propose our automatic refinement refactoring method, which constructs abstract versions of a given Event-B model according to a refinement strategy different from the original one. We applied the refactoring method to construct various refactored versions of large Event-B models and compared them. As a result, we found that the granularity and frequently used variables are important factors for reducing the complexity. We consider the findings important to help Event-B modelers to design and change refinement strategies.

Keywords: Event-B · Refinement · Formal specifications
Design exploration

1 Introduction

Event-B [1] has been attracting strong attention. The primary advantage of Event-B is its flexible refinement mechanism to deal with the complexity of contemporary software. It supports *superposition refinement*, which enables developers to gradually introduce elements of target systems to models.

Although it is important to consider designing of Event-B refinement, existing studies lack explicit discussions on it. Because of the flexibility of superposition refinement, the design space of refinement in Event-B is large. Developers can choose the granularity and the order of introducing elements of target systems into models. Guides for designing Event-B refinement include a textbook showing good refinement design examples [1] and domain-specific guidelines [13]. However, they do not explicitly discuss refinement strategies themselves nor explain why some refinement strategies are better than others.

This work was supported by JST, ACT-I grant number JPMJPR17UA.

J. Sun and M. Sun (Eds.): ICFEM 2018, LNCS 11232, pp. 357–372, 2018.
https://doi.org/10.1007/978-3-030-02450-5_21

In our previous research, we have proposed methods for planning good refinement strategies before constructing models [6] and refactoring refinement strategies of constructed models without breaking consistency [7]. For planning, it is essentially difficult to plan concrete refinement strategies before starting modeling, and thus the support our method provides is limited. Moreover, developers often have difficulties in making design decisions before constructing and end up reconstructing models later. For refactoring, our refactoring method helps developers to construct consistent refactored models through the use of a new refinement strategy. However, developers must face the task of coming up with that strategy. In addition, the method can only be partially automated.

We tackled the problem of analyzing how to design Event-B refinement strategy by solving those problems of our previous work. First, we automated our refactoring method to support easy and flexible refactoring of refinement strategy. Second, we constructed variants of sample models by giving different refinement strategies to our tool and compared various refinement strategies.

The problem we address is novel and important. Methods on verification of refinement have been actively studied in formal methods area. However, as far as we know, design analyses of refinement that take complexity and usability into account have never been studied. From an engineering viewpoint, refinement design is equally important as verification. In fact, there have been many studies on this problem in other areas such as object-oriented design [11].

The contributions of this paper are as follows:

- Automation of our refactoring (generating additional predicates for consistency, automating proof of refactored models, and handling Event-B models)
- Evaluation on automation of refactoring
- Evaluation on effectiveness of refactoring
- Discussion on preferable refinement strategies
- Proposal of a tool-assisted design space exploration of Event-B refinement

The rest of this paper is organized as follows. First, we provide a background on Event-B in Sect. 2. Next, we explain our previous work on refactoring refinement in Event-B and our new proposal on automation of refactoring in Sect. 3. We then describe experiments for comparing various refinement strategies in Sect. 4. In Sects. 5 and 6, we discuss our methods, experiments, threats to validity, and related work. Finally, we summarize this study in Sect. 7.

2 Superposition Refinement in Event-B

2.1 Event-B and Superposition Refinement

Event-B [1] is a formal modeling method with a flexible refinement mechanism, which is designed to mitigate the complexity of contemporary software systems. Specifically, Event-B supports a special style of refinement, which is called *superposition (horizontal) refinement*. For mitigation of complexity in modeling and verification, it enables developers to gradually introduce elements of a target

system to models. In other words, it helps developers to distribute the complexity over several steps. An important point of superposition refinement is that developers can design multiple ways of introduce elements.

Another style of refinement that is popular in classical formal methods is called *data (vertical) refinement*. Event-B also supports this style. This is oriented for deriving executable program codes from specifications. A typical example is conversion from a set-theoretic operation to an operation on an array. In contrast to superposition refinement, the design space of data refinement is limited. In fact, there is a semi-automated tool [9] to do data refinement.

2.2 Modeling in Event-B

In Event-B, a unit of a model (machine) consists of variables, invariants, and events. An event basically consists of guards and actions, which are necessary conditions for triggering the event and state transitions of the event, respectively.

After constructing a model, the development environment of Event-B (Rodin) generates *proof obligations* (POs), which are formulae of consistency of the model. A primary sort of PO is that an occurrence of event e does not violate an invariant i (*invariant preservation*, written as e/i/INV).

If a developer declares that a model is a refinement of another model, other sorts of POs are generated. Such POs include *guard strengthening* (GRD), which requires guards of a concrete event to be stronger than guards of corresponding events in the abstract model, and *action simulation* (SIM), which requires that concrete behavior corresponds to abstract behavior. Guard strengthening ($e_{\mathrm{C}}/g_{\mathrm{A}}$/GRD, where e_{C} is an event in the concrete machine and g_{A} is a guard of the abstract event of e_{C} in the abstract machine) demands that the conjunction of guards of e_{C} is stronger than a guard of the abstract event g_{A}. The (simplified) formula of $e_{\mathrm{C}}/g_{\mathrm{A}}$/GRD is $I_{\mathrm{A}} \wedge I_{\mathrm{C}} \wedge G_{\mathrm{C}} \Rightarrow g_{\mathrm{A}}$, where I_{A} and I_{C} are abstract invariants and concrete invariants, and G_{C} is guards of e_{C}. Developers can be confident with the consistency of the model by discharging all generated POs.

2.3 Example: Cars on the Bridge

We describe a variant of an Event-B example model "Cars on the Bridge" [1, Chap. 2]. It is about traffic between a mainland and an island, which are connected with a one-way bridge (Fig. 1, right). The requirements include: (R1) The number of cars outside of the mainland should not exceed the capacity (constant *cap*). (R2) When a car is going on the bridge towards the mainland, traffic lights on the mainland should prevent cars on the mainland from departing.

In Event-B, a developer first constructs an abstract model that disregards some elements of the target system. For example, Fig. 2 shows an abstract model of Cars on the Bridge (Fig. 1, left). The variable n_{out} is the number of cars on the island or the bridge. The invariant **inv_A1** represents requirement (R1). The event describes the behavior of a car's departure from the mainland. Various POs including mainland_out_abs/**inv_A1**/INV are generated and proved.

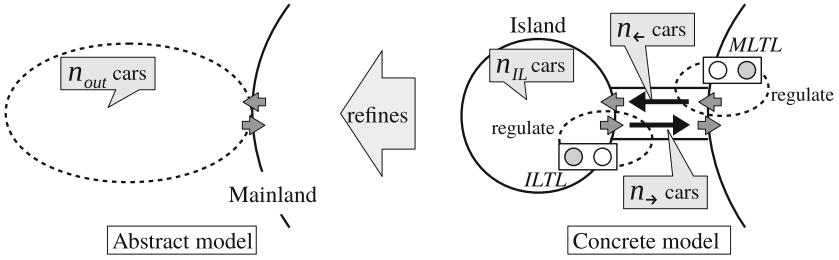

Fig. 1. Cars on the bridge example

variables: n_{out}

invariants:
inv_A1: $n_{\text{out}} \leq cap$
...

Event mainland_out_abs
when
***grd1*:** $n_{\text{out}} < cap$
then
***act1*:** $n_{\text{out}} := n_{\text{out}} + 1$
end

Fig. 2. M_{CarsA}: part of abstract model of example

After constructing an abstract model, a concrete model with more elements is constructed. For instance, Fig. 3 shows a concrete model of the example (Fig. 1, right). The number of cars on the island is n_{IL}, and the number of cars going left and right are $n_{\leftarrow}$ and $n_{\rightarrow}$, respectively. Those variables *replace* the abstract variable n_{out} (**inv_C1**). Variables of traffic lights on the mainland and the island ($MLTL$ and $ILTL$) are also introduced into the model. The invariant **inv_C2** and the guard ***grd2*** satisfies the requirement (R2). In this model, the PO mainland_out_con/***grd1***/GRD is dischargeable because the its formula is:

$$(MLTL = green) \wedge (MLTL = green \Rightarrow n_{\rightarrow} = 0)$$
$$\wedge\ (n_{\leftarrow} + n_{\text{IL}} + 1 < cap) \wedge (n_{\text{out}} = n_{\leftarrow} + n_{\text{IL}} + n_{\rightarrow}) \Rightarrow n_{\text{out}} < cap.$$

variables:
$n_{\leftarrow}, n_{\text{IL}}, n_{\rightarrow}, MLTL, ILTL$

invariants:
inv_C1: $n_{\text{out}} = n_{\leftarrow} + n_{\text{IL}} + n_{\rightarrow}$
inv_C2: $MLTL = green \Rightarrow n_{\rightarrow} = 0$
...

Event mainland_out_con
refines mainland_out_abs
when
***grd2*:** $MLTL = green$
***grd3*:** $n_{\leftarrow} + n_{\text{IL}} + 1 < cap$
then
***act1*:** $n_{\leftarrow} := n_{\leftarrow} + 1$
end

Fig. 3. M_{CarsC}: part of concrete model of example

The refinement in Event-B is done in this way. First, developers blackbox traffic lights and state that "Somehow, the numbers of cars satisfy these invariants and behave like these events." They then construct a concrete model to describe that "It turned out that the cars' invariants and behaviors of the abstract model are due to traffic lights." This flexible refinement mechanism allows developers to freely design the elements introduced in each refinement step. For example, they can also introduce traffic lights before introducing cars.

Henceforth, we will use the term *refinement strategy* (RS) to mean a sequence of introduced variables in each step. We will also use the term *refinement chain* (RC) to mean a sequence of Event-B models $[M_0, M_1, \ldots, M_n]$ such that M_{i+1} refines M_i, where $0 \leq i \leq n-1$.

3 Automated Refinement Refactoring

3.1 Refinement Refactoring

We will here describe our previous work on refinement refactoring [7].

Refinement refactoring aims to improve the value of given Event-B models by changing the refinement strategy of given verified models. The refactored models have a different refinement strategy than that of the given models. By refactoring, the expression of models other than the most concrete model can be changed without changing the most concrete model, because a refinement strategy dominates the expression of models. In other words, refactoring corresponds to obtaining projection of the most concrete model onto a new state space. For example, we can improve the maintainability of a model by decomposing one refinement step into several small steps. In addition, a reusable part of an existing model can be extracted with refactoring by obtaining a projection onto a state space of reusable variables. Thus, refactoring facilitates engineering use of constructed models by obtaining a new projection of the most concrete model.

Our refactoring method receives a concrete model M and a set of variables V as input and manually produces a model $M'(V)$ (*intermediate model*) that is an abstract version of M. The input V is a subset[1] of all variables declared in the given model M and its abstract models. The output model $M'(V)$ should be consistent with M, and thus all POs (such as invariant preservation, guard strengthening, and action simulation) of $M'(V)$ should be dischargeable. Moreover, the set of variables contained in $M'(V)$ should be V. Refactoring is achieved with two operations: refinement merging and refinement decomposition. For a given refinement chain $[M_A, M_B, M_C]$, refinement merging constructs a model M_{B+C}, which refines M_A and is constructed from M_B and M_C. For a given refinement chain $[M_A, M_C]$, refinement decomposition constructs a model M_B such that $[M_A, M_B, M_C]$ is a refinement chain. By merging a refinement chain $[\ldots, M]$ and decomposing it into $[\ldots, M'(V), M]$, we can obtain a model $M'(V)$ that is written with V and consistent with M.

[1] V cannot be an arbitrary subset. See our previous work [7] for conditions of V.

The key challenge of refinement refactoring is guaranteeing consistency in refinement decomposition. A naïve approach towards refinement decomposition is *slicing*, namely constructing $M'(V)$ as a collection of parts of M that can be written with V. For example, suppose that we try to construct a model of the example disregarding the traffic lights (*MLTL* and *ILTL*). In other words, we try to construct a model that describes properties and behavior relevant to the number of cars on the bridge and the island that are controlled by the traffic lights, without describing the behavior of traffic lights. By slicing, we obtain the model $M'_{\mathrm{CarsC}}(\{n_{\leftarrow}, n_{\mathrm{IL}}, n_{\rightarrow}\})$ shown in Fig. 4. Although the model should refine M_{CarsA}, this intermediate model lacks the consistency of mainland_out_int/***grd1***/GRD, because it lacks the invariant **inv_C2** and the guard ***grd2***, which were necessary hypotheses for the consistency in the original model. Thus, slicing often drops predicates that are hypotheses of consistency proofs of the original model.

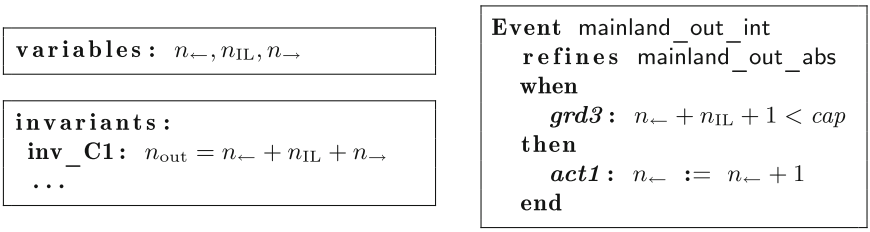

Fig. 4. $M'_{CarsC}(\{n_{\leftarrow}, n_{\mathrm{IL}}, n_{\rightarrow}\})$: a part of intermediate model (obtained by slicing, not consistent) of example.

Our refactoring method addresses this problem by supporting the construction and addition of new predicates, which we call *complementary predicates* (CPs). CPs should be able to be expressed with variables of an intermediate model and should function as a missing hypothesis of a proof of an intermediate model. CPs can be found by analyzing a proof of the original model because they correspond to lemmas in the original proof (Sect. 2.3). For instance, in the proof of mainland_out_con/***grd1***/GRD in the original example model M_{CarsC}, there is a lemma $n_{\rightarrow} = 0$, which can be derived from hypotheses **inv_C2** and ***grd2***. This lemma can be expressed with variables $\{n_{\leftarrow}, n_{\mathrm{IL}}, n_{\rightarrow}\}$ and we can discharge the PO in the intermediate model (mainland_out_int/***grd1***/GRD) by adding this lemma to the model as a new guard of event mainland_out_int. Thus, our method achieves a consistent refinement decomposition by slicing and analyzing the original proof.

3.2 Automation with Heuristics

The method in our previous work, which include manual analysis on many proofs, is demanding and difficult. Therefore, we propose an automation of refinement

refactoring by constructing the following three functionalities. **Complementary Predicates Generator.** Obtaining CPs is the most difficult and time-consuming part of manual refinement refactoring. Our method uses heuristics and Craig's interpolation (with Z3 [5]) to automate this process. **Proof Finder.** Automatic prover of Rodin cannot discharge all POs of refactored models. Our method finds parts of the original proof that correspond to the proof of refactored models and reuses them. **Merger and Slicer.** We have also developed rule-based automation of merging and slicing.

Those functionalities are implemented as a plug-in of Event-B's development environment[2]. This automation enables us to analyze the effects of refinement strategies on complexity of models and verification (Sect. 4).

Manually finding CPs is significantly difficult and demanding. To manually find a CP for a PO, a developer must find a corresponding PO in the original model, analyze the proof of it, and find hypotheses that are essential to discharge the PO and written in variables of the intermediate model. Repeating this process to find CPs for all POs of large-scale models is demanding. In addition, developers need to repeat a difficult task of deeply understanding the proofs of original models. Therefore, we made this process systematic and automatic.

Predicates sufficient for proofs in the original model can be systematically obtained as interpolants of the formulae of POs. In other words, a formula X that satisfies *hypotheses* $\Rightarrow X \Rightarrow$ *consequence* is enough to derive *consequence*. Thus, if such interpolants can be added to the intermediate model, the model becomes consistent. However, such interpolants cannot always be added to the model because X may use variables that are not in variables of the intermediate model. This is because the set of identifiers of X (an interpolant) is a subset of identifiers used in both of *hypotheses* and *consequence*. For example, mainland_out_con/***grd1***/GRD in M_{CarsC} is as follows:

$$(MLTL = green \land n_{\leftarrow} + n_{\mathrm{IL}} + 1 < cap \land (MLTL = green \Rightarrow n_{\rightarrow} = 0) \\ \land\ n_{\mathrm{out}} = n_{\leftarrow} + n_{\mathrm{IL}} + n_{\rightarrow}) \Rightarrow n_{\mathrm{out}} < cap. \tag{1}$$

The identifiers common in the hypotheses part and the consequence part are $\{n_{\mathrm{out}}, cap\}$, but n_{out} cannot be used in the intermediate model. Thus, an interpolant of the formula of a PO of the original model cannot always be a CP.

Our method provides heuristics to convert a formula of PO into an equivalent formula such that an interpolant of the converted formula becomes a CP. The heuristic for GRD converts the original formula $I_{\mathrm{A}} \land I_{\mathrm{C}} \land G_{\mathrm{C}} \Rightarrow g_{\mathrm{A}}$ into:

$$I_{\mathrm{AB}} \land \tilde{I}_{\mathrm{C}} \land \tilde{G}_{\mathrm{C}} \Rightarrow g_{\mathrm{A}} \lor \neg \tilde{I}_{\mathrm{A}} \lor \neg I_{\mathrm{BC}} \lor \neg G_{\mathrm{BC}}, \tag{2}$$

where $\tilde{I}_{\mathrm{A}}$, $\tilde{I}_{\mathrm{C}}$, and $\tilde{G}_{\mathrm{C}}$ are abstract invariants, concrete invariants, and concrete guards that contain dropped variables, and I_{AB}, I_{BC}, G_{BC} are those that do not contain dropped variables (i.e., they are obtained by slicing). We also defined

[2] https://github.com/trarse-nii/SliceAndMerge.

heuristics for SIM and INV. For example, the heuristic converts (1) into:

$$(MLTL = green \wedge n_{\leftarrow} + n_{\mathrm{IL}} + 1 < cap \wedge (MLTL = green \Rightarrow n_{\rightarrow} = 0)) \\ \Rightarrow (n_{\mathrm{out}} < cap \vee \neg(n_{\mathrm{out}} = n_{\leftarrow} + n_{\mathrm{IL}} + n_{\rightarrow})). \quad (3)$$

The heuristics are designed so that (a) interpolants of converted formulae are always written with identifiers of the intermediate model (i.e., the interpolants can be added to the intermediate model) and (b) adding the interpolants to the intermediate model makes the model consistent. Let V_{A}, V_{B}, and V_{C} be variables of the abstract model, the intermediate model, and the concrete model, respectively. By definitions, variables of $(V_{\mathrm{A}} \backslash V_{\mathrm{B}})$ do not occur in the hypotheses part of (2) and variables of $(V_{\mathrm{C}} \backslash V_{\mathrm{B}})$ do not occur in the consequence part. Thus, the set of variables common in the hypotheses part and the consequence part is guaranteed to be a subset of V_{B}. Therefore, the interpolant can be added to the intermediate model. For instance, predicate $n_{\leftarrow} + n_{\mathrm{IL}} + n_{\rightarrow} + 1 < cap$ can be obtained as an interpolant of (3). In addition, by (2), $X \Rightarrow g_{\mathrm{A}} \vee \neg \tilde{I}_{\mathrm{A}} \vee \neg I_{\mathrm{BC}} \vee \neg G_{\mathrm{BC}}$. By strengthening the hypotheses part of this formula,

$$I_{\mathrm{A}} \wedge I_{\mathrm{B}} \wedge G_{\mathrm{B}} \wedge X \Rightarrow g_{\mathrm{A}} \vee \neg \tilde{I}_{\mathrm{A}} \vee \neg I_{\mathrm{BC}} \vee \neg G_{\mathrm{BC}},$$

where I_{B} and G_{B} are invariants and guards of the intermediate model. Since $I_{\mathrm{A}} \Rightarrow \tilde{I}_{\mathrm{A}}$, $I_{\mathrm{B}} \Rightarrow I_{\mathrm{BC}}$, and $G_{\mathrm{B}} \Rightarrow G_{\mathrm{BC}}$,

$$I_{\mathrm{A}} \wedge I_{\mathrm{B}} \wedge G_{\mathrm{B}} \wedge X \Rightarrow g_{\mathrm{A}}.$$

Thus, by adding X to the corresponding event as a guard, the GRD of the intermediate model becomes dischargeable. Therefore, our tool makes intermediate models consistent by adding interpolants of formulae converted from POs.

The proof finder aims to reuse proofs on the original model. Predicates of a model (invariants, guards, actions, etc.) related to a PO dominate the contents of the generated PO. However, the POs of a refactored model are generated from a mixture of multiple steps of original models because refactoring decomposes after merging of multiple models. Hence, it is not straightforward to find which proof on the original model should be reused to discharge POs of a refactored model. To address this problem, we added traceability information, which shows the predicates of original models used to generate a PO of a refactored model to the refactored model. The proof finder uses this traceability information to find corresponding proof in the original model and follows the same proof tree for proving a PO in the refactored model. Although the proof finder does not work for arbitrary proofs, we did not find any problems in our experiments (Sect. 4). Thus, we automated reusing the proof of an original model for verifying a refactored model by adding traceability.

4 Experiments on Models Constructed with Refactoring

4.1 Evaluation Criteria

We considered that good strategies would effectively mitigate development complexity in Event-B because the primary goal of the Event-B refinement mecha-

nism was to mitigate such complexity. We focused on two kinds of complexities: the complexity of model itself and the complexity of verification.

Local Model Complexity. We checked *the numbers of variables, invariants, and events of each step* to evaluate the complexity of the model of each step. If a step is small, it tends to be easy to understand the step because developers can focus on a small number of elements. Therefore, we considered that the numbers should be *well-distributed over multiple steps* if developers follow a good strategy.

Proof Complexity. We checked *the number of all generated POs and the number of POs that failed to be discharged by automatic provers (manually discharged POs)* to evaluate the proof complexity. The number of manually discharged POs are checked to evaluate actual burden of proving because Rodin has automatic provers, which discharge most of the relatively simple proofs. In addition, we also considered the *local model complexity* informative to evaluate this complexity. If there is a non-dischargeable PO in a model, making modifications to a part of the model affects multiple POs. This is because POs and the contents of Event-B models are interrelated. For instance, if an invariant i needs to be modified, not only preservation of i by all events, but also GRD and SIM of related events should be checked again. Therefore, distributing model contents and POs limits the range of modification propagation, and thus reduces complexity. The number of invariants and events also affects the number of generated POs. Thus, we considered that effective strategies *distribute number of variables, invariants, events, and POs well.*

4.2 Comparison Settings and Hypotheses

As the materials, we used models [2] of a train system [1, Chap. 17] and models of an autonomous satellite flight formation system [12]. Both were constructed by modelers experienced in Event-B.

There are two important characteristics of RSs: *granularity* and *order*. For example, an RS $[\{a, b, c\}]$, which introduces three variables in one step and another RS $[\{a\}, \{b\}, \{c\}]$, which introduces them one-by-one are different in granularity. An RS $[\{a\}, \{b\}, \{c\}]$ and another RS $[\{c\}, \{b\}, \{a\}]$ are different in order of variable introduction. In an experiment we conducted, we compared strategies that differed in granularity and order to check the evaluation criteria.

To examine differences of granularity, we made the following comparisons:

Original vs. Merged. Comparison with a model constructed by merging the original models. For example, when an RC $[M_1, M_2, M_3]$ followed an RS $[\{n_{\mathrm{IL}}\}, \{n_{\leftarrow}, n_{\rightarrow}\}, \{MLTL\}]$, we constructed a model M_{1+2+3} that followed an RS $[\{n_{\mathrm{IL}}, n_{\leftarrow}, n_{\rightarrow}, MLTL\}]$ and compared $[M_1, M_2, M_3]$ and $[M_{1+2+3}]$.

Hypothesis 1: The number of POs of the merged model is less than that of the original models. This is because there is no need of checking consistencies between several steps (e.g., GRD and SIM) in the merged model. This means that decomposition adds several POs but mitigates the local model complexity.

Original vs. Decomposed. Comparing a step of original models and models constructed by decomposing the step of original models. Complementary predicates are generated and added to the model through refinement decomposition, and the complexity is affected by CPs. Therefore, to eliminate the effect of CPs, we compared decomposed models and a model constructed by re-merging the decomposed model. For instance, when the RC $[M_1, M_2, M_3]$ was given, we decomposed M_2 to construct another RC $[M_1, M_{21}, M_{22}, M_3]$ that follows an RS $[\{n_{\mathrm{IL}}\}, \{n_{\leftarrow}\}, \{n_{\rightarrow}\}, \{MLTL\}]$. We then constructed a model M_{2*} by merging M_{21} and M_{22}, and then compared (M_{21}, M_{22}) and M_{2*}. Since there are multiple ways of decompositions, we compared several of them.

Hypothesis 2: Although the sum of the number of invariants may increase due to the introduction of typing invariants[3], invariants are distributed over several steps. This also means decomposition mitigates the local model complexity. *Hypothesis 3:* The number of CPs (new guards and actions) affects the number of POs because GRD and SIM should be checked for them. However, the new GRDs and SIMs are relatively simple because CPs correspond to lemmas of original proof and thus proofs for CPs are simpler than the original proofs. This means that decomposition adds several POs, which are easily discharged by automatic provers. *Hypothesis 4:* The number of POs of the re-merged model is almost the same as that of the original model. This is because CPs are introduced as guards and actions, which do not affect the number of INVs. This means that the comparison of original models and re-merged models is fair.

To examine differences of order, we made the following comparisons:

Swapping Two Steps. Comparison with models constructed by swapping two continuous steps in the original strategy. By the same reason as Original vs. Decomposed, we used models constructed by swapping twice instead of the original models. For example, when the RC $[M_1, M_2, M_3]$ was given, we constructed another RC $[M_{S121}, M_{S122}, M_3]$ that follows an RS $[\{n_{\leftarrow}, n_{\rightarrow}\}, \{n_{\mathrm{IL}}\}, \{MLTL\}]$. We then re-swapped them to construct another RC $[M_{SS121}, M_{SS122}, M_3]$ that follows an RS $[\{n_{\mathrm{IL}}\}, \{n_{\leftarrow}, n_{\rightarrow}\}, \{MLTL\}]$, and then compared $[M_{S121}, M_{S122}, M_3]$ and $[M_{SS121}, M_{SS122}, M_3]$. We calculated standard deviation of the number of POs to compare distributions of them.

Hypothesis 5: Swapping may change the distribution of number of invariants and POs. This is because some variables are frequently used in invariants and others are rarely used. For instance, let us assume that we are going to construct models with variables $\{a, b\}$ and invariants $\{f(a), g(a, b)\}$. If we construct models by following $[\{a\}, \{b\}]$, we can distribute the invariants because $f(a)$ is introduced in the first step and $g(a, b)$ is introduced in the second step. In contrast, if we construct models by following $[\{b\}, \{a\}]$, both f and g are introduced in the second step because both depend on a. The sum of number of invariants may slightly increase due to the introduction of typing invariants. This means that orders of RSs are important.

[3] Rodin requires variables' typing information. Although typing information is usually inferred from normal invariants, slicing may remove such invariants. In this case, invariants of typing information (e.g., $MLTL \in COLOR$) must be newly provided.

Table 1. Merging and decomposing of results obtained for Train example.

Models	ΔV	I	ΣI	E	CP	PO	ΣPO	MPO	ΣMPO	Auto%
Tr	4, 3, 1, 1	8, 9, 3, 4	24	6,8,8,8	-	35, 63, 16, 13	127	7, 18, 6, 5	36	72%
TrM	9	24	24	8	-	110	110	32	32	71%
Tr1	4	8	8	6	-	35	35	7	7	80%
Tr1DA	1, 1, 1, 1	1, 2, 4, 5	12	3, 4, 4, 6	0, 0, 1, 0	0, 0, 13, 25	38	0, 0, 2, 5	7	82%
Tr1DMA	4	12	12	6	1	35	35	7	7	80%
Tr1DB	1, 1, 1, 1	1, 4, 3, 4	12	4, 5, 5, 6	1, 7, 1, 0	2, 26, 7, 17	52	0, 6, 1, 0	7	87%
Tr1DMB	4	12	12	6	9	36	36	8	8	78%

Reversing Multiple Steps. Comparison with models constructed by reversing steps in the original strategy. Again, we used models constructed by reversing twice instead of the original models. For instance, when the RC $[M_1, M_2, M_3]$ was given, we constructed another RC $[M_{R1}, M_{R2}, M_{R3}]$ that follows an RS $[\{MLTL\}, \{n_{\leftarrow}, n_{\rightarrow}\}, \{n_{\mathrm{IL}}\}]$. We then did reversing again to construct another RC $[M_{RR1}, M_{RR2}, M_{RR3}]$ that follows an RS $[\{n_{\mathrm{IL}}\}, \{n_{\leftarrow}, n_{\rightarrow}\}, \{MLTL\}]$, and then compared $[M_{R1}, M_{R2}, M_{R3}]$ and $[M_{RR1}, M_{RR2}, M_{RR3}]$. The hypothesis is the same as that of swapping.

4.3 Results

Due to space limitations, we omitted the results on Flight Formation Systems models, which tend to be similar to those for the Train example. The Train model and additional information are available on the Web[4].

With our automated refactoring tool, we succeeded in constructing all models and discharging all POs. This means our tool generated correct CPs. Combined with the SMT solvers plug-in of Rodin, our proof finder discharged all POs. Therefore, we conclude that our tool is appropriate for this experiment.

Table 1 shows the experiment results we obtained on granularity. Each row lists the numbers of a set of models (i.e., an RS). For example, in the original Train example (the first row "Tr"), there are four steps that introduce 4, 3, 1, and 1 variables (ΔV). Each step has 8, 9, 3, and 4 invariants (I), 24 invariants in total (ΣI). E shows events in each step, CP shows the numbers of CPs introduced in each step, PO shows the number of all POs, MPO shows manually discharged POs, and Auto% shows the rate of automatically discharged POs.

Row 2 (TrM) in Table 1 shows the results obtained for the merged model. Columns ΔV, I, and E show that TrM introduces things introduced through four steps in the original machines in one-shot. ΣPO shows that the number of POs decreased from 127 to 110 through merging. This result supports Hypothesis 1.

Rows 3–7 (Tr1*) in Table 1 show the results obtained on decomposition of the first step of the original model (the numbers are those of the first step of Tr). The first step of Tr introduces four variables: $resrt, resbl, rsrtbl$, and OCC. Row 4 (Tr1DA) shows the results obtained for a decomposed strategy that introduces $resrt, resbl, rsrtbl$, and OCC one-by-one. Row 6 (Tr1DB) shows the results

[4] http://tkoba.jp/publications/icfem2018/.

Table 2. Results obtained in swapping and reversing the Train example.

Models	ΔV	I	ΣI	E	CP	PO	σ_{PO}	MPO	σ_{MPO}	Auto%
Tr	4, 3, 1, 1	8, 9, 3, 4	24	6, 8, 8, 8	-	35, 63, 16, 13	19.9	7, 18, 6, 5	5.2	72%
TrS12	3, 4, 1, 1	4, 15, 3, 4	26	6, 8, 8, 8	0, 0, 0, 0	8, 85, 33, 13	30.5	1, 23, 9, 5	8.3	73%
TrSS12	4, 3, 1, 1	8, 13, 3, 4	28	6, 8, 8, 8	0, 0, 0, 0	36, 53, 33, 13	14.2	7, 16, 9, 5	4.1	71%
TrS23	4, 1, 3, 1	8, 2, 10, 4	24	6, 7, 8, 8	0, 5, 0, 0	35, 25, 61, 15	17.1	7, 6, 19, 6	5.5	72%
TrSS23	4, 3, 1, 1	8, 9, 3, 4	24	6, 8, 8, 8	0, 9, 0, 0	35, 65, 16, 23	18.7	7, 16, 5, 10	4.2	73%
TrS34	4, 3, 1, 1	8, 9, 1, 6	24	6, 8, 8, 8	0, 0, 4, 0	28, 45, 5, 19	20.0	7, 18, 2, 10	5.8	72%
TrSS34	4, 3, 1, 1	8, 9, 2, 6	25	6, 8, 8, 8	0, 0, 4, 0	35, 63, 7, 25	20.3	7, 18, 2, 9	5.8	72%
TrR	1, 1, 3, 4	1, 3, 6, 17	27	2, 2, 7, 8	0, 0, 1, 0	2, 7, 13, 94	37.7	0, 2, 2, 28	11.6	72%
TrRR	4, 3, 1, 1	8, 11, 5, 3	27	7, 8, 8, 8	0, 0, 1, 0	36, 49, 19, 6	16.3	9, 15, 5, 2	4.9	72%

obtained for another decomposed strategy that introduces $OCC, rsrtbl, resbl$, and $resrt$ one-by-one (i.e. in the reversed order of Tr1DA). Rows 5 and 7 (Tr1DMA and Tr1DMB) show the results obtained for a strategy constructed by merging the four steps of Tr1DA and Tr1DB, respectively.

By comparing Tr1 and others, we see no difference in the total numbers of variables and events, but an increasing number of invariants. From ΣPO, we see no significant difference from Tr1 except for Tr1DB. This is because 9 CPs (as guards) and related POs were generated by following the RS of Tr1DB ($[\{OCC\}, \{rsrtbl\}, \{resbl\}, \{resrt\}]$). As column MPO shows, these new POs were simple enough for automatic provers to discharge. We can see that we succeeded in distributing variables, invariants, and POs over several steps by decomposition. Therefore, we consider that the results support Hypotheses 2–4.

Table 2 shows the experiment results on the order in which variables are introduced. In this table, standard deviations of PO and MPO (σ_{PO} and σ_{MPO}) are shown. Row 1 (Tr) shows the results obtained with the original strategy. Rows 2–3, 4–5, 6–7 (TrSn(n + 1), TrSSn(n + 1)) show the results obtained by swapping steps 1–2, 2–3, and 3–4, respectively. Rows 8–9 (TrR, TrRR) show the results obtained by reversing. ΔV of the swapped or reversed models' strategy are simply swapped or reversed numbers of ΔV of Tr.

We see no significant differences in ΣI. It is interesting that there are steps with a large number of invariants in TrS12 (second step) and TrR (fourth step). Both of them introduce four variables introduced in the first step of the original strategy ($\{resrt, resbl, rsrtbl, OCC\}$). This is because those variables are frequently used in the invariants. It is also found that the steps with a large number of invariants (the second step of TrS12 and the fourth step of TrR) have a large number of POs (PO and MPO), and result in high standard deviations of those strategies. Thus, we consider that the result supports Hypothesis 5.

Summary. (1) Strategies with more steps have more POs in total but distribute POs well, especially if strategies are carefully chosen taking dependence into consideration. This is preferable as discussed in Sect. 4.1. (2) Due to the dependence of invariants on variables, the order in which variables are introduced affects variance of invariants and POs. In general, important variables that are written in many invariants should be introduced in early steps.

5 Discussion

5.1 Effects of Refactoring to POs

Refactoring adds several POs to the original models and also removes several POs from the original models. Generating CPs by refinement decomposition results in the generation of new POs related to the CPs. However, as CPs can be seen as lemmas, the generation of CPs helps automatic provers to discharge difficult POs. In addition, as we saw in Sect. 4.3, refinement merging removes several GRDs and SIMs.

Although changing the order of a strategy (such as swapping and reversing) involves slicing and generation of CPs, several POs are removed through it. In fact, TrR (reversed) has 116 POs but Tr (original) has 127 POs. We found that several POs about consistency between two models (such as GRD and SIM) are removed through the changing order. This is because concrete and strong guards (which were originally introduced in later steps) are introduced in early steps after changing order, and thus there is no need to strengthen them in later steps.

5.2 Dependence of Invariants on Variables

As we saw in experiments on the order of refinement strategies, dependence of invariants on variables is important for detailed analyses on complexity mitigation with refinement. The dependence is obviously problem specific. Thus, changing the order of strategy will not have much effect if the dependence is not strong.

The dependence also strongly affects whether an invariant is dropped in slicing (i.e., whether CP is generated). To analyze this, not only variables in an invariant but also the structure of the invariant is important. For example, an invariant **inv1:** $f \in a \rightarrow b$, which means that "f is a total function from a to b", limits the value of f. Although variables a and b appear in **inv1**, the values of them are not limited by **inv1**. Therefore, **inv1** can be a hypothesis in a proof related to f but it cannot be a hypothesis in a proof related to a or b. Because CP is generated by a lack of hypothesis in a proof, we need to consider whether the value of a variable is limited by an invariant.

Our future work will include detailed analyses on refactoring while taking dependence into consideration.

5.3 Use of Automated Refactoring in Development

We also consider that search for a good refinement strategy is important in development process. Automated refinement refactoring can be used to search for a desirable strategy to improve flexibility against change and actually refactor models. However, CPs generated by the current method are sometimes redundant or non-human-friendly. Therefore, we are planning to improve our CP generator so that it will not only generate correctly but also be easy to understand. Possible approaches include applying metrics of formula understandability and a method to generate simple interpolants [3].

5.4 Threats to Validity

Internal Validity Threats. Our analyses rely on artificial data constructed with our method. Thus the method, in particular the CPs it generates, may have affected the obtained results. However, we carefully designed the experiment to eliminate the effect of CPs (such as double-reversing). We also discussed how refactoring would affect POs (Sect. 5.1). Therefore, although user studies for further analyses are included in our future direction, we conclude that the analyses given in this paper are valid.

Additionally, from the experiment results obtained on changing the order of the refinement strategy, we concluded that introducing important variables in early steps is effective in reducing complexity. Although this claim seems natural, the complexity increased by refactoring in every case we examined. In other words, we didn't see any mitigation of complexity with refactoring. This is because the original models were constructed by experienced modelers in an ideal order (i.e., important variables were introduced in early steps). Therefore, we are planning to use models in which variables were introduced in a bad order to confirm the method's validity.

External Validity Threats. In terms of generalization, there may be a concern about the variations and practicality of the materials we used in the experiments. For variations, we believe our findings about granularity and order are general enough and not domain-specific. In terms of practicality, in fact, the models were constructed by experienced modelers. Although we believe the models are appropriate for examining our general findings, analyses on models constructed by inexperienced modelers would be an interesting subject for future work.

Construct Validity Threats. For the sake of simplicity, we used local model complexity and proof complexity as evaluation criteria. However, we were aware that strategies that have atomic steps (e.g., those that introduce only one variable in one step) are not optimal. Too much decomposition of refinement often causes models that lack conceptual integrity and have many meaningless POs. Thus, although our findings show that decomposing refinement steps tend to be effective, we will also consider costs of long refinement chains in our future work.

6 Related Work

There have been studies to connect Event-B models with other modeling methods and requirement analysis methods, such as UML [10] and KAOS [8]. Because such modeling methods are widely used, there have been analyses that studied the design of such models, such as decomposition into components and refinement in KAOS. In particular, in the area of object-oriented design, such studies [11] have been very active. However, by connecting Event-B models and other modeling methods and analyze models in other notations, the expressiveness of

model is limited to that of other modeling notations. More importantly, analysis methods do not consider proof obligations, which need to be considered for formal refinement. Therefore, Event-B's flexible and rigorous formalism cannot be handled with such methods.

There have been case studies of Event-B modeling by experts [1,4]. The study in [4] is particularly interesting because multiple researchers have constructed different Event-B models for the same subject problem. Their models are sophisticated and good learning materials for other developers. However, they do not explain why the strategies they used are better than other possible strategies. There are also guidelines of Event-B modeling for subjects of a particular domain [13]. Although their guides are detailed, they are domain-specific and not applicable to other areas. It will be interesting to analyze more subjects of their models with our method because they are good examples of experts' models.

Our previous work [6] proposed evaluation criteria of refinement strategy based on the number of variables, and a planning method to distribute introduction of variables as much as possible. However, it is not applicable in realistic situation because the planner requires a list of invariants and variables before constructing models. In addition, the planning method is conceptual, and cannot handle details of model and POs, which are necessary in empirical analyses.

Our approach in this paper establishes a method to construct a consistent models and analyze them by automating our refactoring method. By using the automatic refactoring, we succeeded in comparing and discussing refinement strategies, considering predicates and POs by using actual Event-B models. As a result, general and domain-independent findings about refinement strategies were obtained. Our method also enables developers to search the design space of Event-B models constructed in development.

7 Conclusion and Future Work

Our goal was exploration and analysis on the design space of Event-B's flexible superposition refinement, which have never studied in the formal methods area. To this end, we provided an automatic method to construct a consistent refactored model from given models according to given refinement strategies. We defined heuristics and applied Craig's interpolation to generate predicates to resolve inconsistencies occurring through the changing of a refinement strategy. As this enabled us to flexibly change the refinement strategy of a given model, we conducted an experiment in which we compared models constructed by following various refinement strategies from the viewpoint of complexity. As a result, we found that doing fine-grained refinement and introducing frequently used variables to the model earlier are effective to reduce complexity of modeling and verification of each step. In addition, we discussed the effects that refactoring would have on complexity and dependence between predicates and variables. We conclude that our method and experiments will benefit Event-B modelers designing refinement strategies.

Our future work will primarily proceed in two directions. The first will be to analyze the relationship between dependence and refinement strategies to make our design space exploration more sophisticated. The second will be to conduct user studies and compare the result with that of our experiment to check the validity.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R.: Train system. http://deploy-eprints.ecs.soton.ac.uk/124/
3. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 313–329. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_22
4. Boniol, F., Wiels, V.: The landing gear system case study. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 1–18. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_1
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). https://doi.org/10.1007/978-3-540-78800-3_24
6. Kobayashi, T., Ishikawa, F., Honiden, S.: Understanding and planning Event-B refinement through primitive rationales. In: Ait Ameur, Y., Schewe, K.D. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. LNCS, vol. 8477, pp. 277–283. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_24
7. Kobayashi, T., Ishikawa, F., Honiden, S.: Refactoring refinement structure of Event-B machines. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 444–459. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_27
8. Matoussi, A., Gervais, F., Laleau, R.: A goal-based approach to guide the design of an abstract Event-B specification. In: 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 139–148. IEEE (2011)
9. Requet, A.: BART: a tool for automatic refinement. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 345–345. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_33
10. Said, M.Y., Butler, M., Snook, C.: Language and tool support for class and state machine refinement in UML-B. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 579–595. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_37
11. Subramanyam, R., Krishnan, M.S.: Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Trans. Softw. Eng. **29**(4), 297–310 (2003)
12. Tarasyuk, A., Pereverzeva, I., Troubitsyna, E., Latvala, T.: The formal derivation of mode logic for autonomous satellite flight formation. In: Koornneef, F., van Gulijk, C. (eds.) SAFECOMP 2015. LNCS, vol. 9337, pp. 29–43. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24255-2_4
13. Yeganefard, S., Butler, M., Rezazadeh, A.: Evaluation of a guideline by formal modelling of cruise control system in Event-B. In: Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), pp. 182–191. NASA, April 2010