Refactoring Refinement Structure of Event-B Machines

Tsutomu Kobayashi[1(✉)], Fuyuki Ishikawa[2], and Shinichi Honiden[1,2]

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

Abstract. Refinement in formal specifications has received significant attention as a method to gradually construct a rigorous model. Although refactoring methods for formal specifications have been proposed, there are no methods for refactoring of refinement structures in formal specifications. In this paper, we describe a method to restructure refinements in specifications of Event-B, a formal specification method with supports for refinement. The core of our method is decomposition of refinements. Namely, when an abstract Event-B machine A, a concrete machine C refining A, and a slicing strategy are provided, our method constructs a consistent intermediate machine B, which refines A and is refined by C. We show effectiveness of our methods through two case studies on representative usages of our method: decomposition of large-scale refinements and extraction of reusable parts of specifications.

Keywords: Event-B · Refinement · Abstraction · Refactoring · Interpolation

1 Introduction

Formal specification methods with refinement mechanisms have been gaining much interest, because they help developers to do rigorous modeling while lessening the burden of modeling and verification. In particular, Event-B [1], which has a flexible refinement mechanism including support for *horizontal refinement*, mitigates the complexity of modeling and verification by distributing it amidst multiple steps, which form a refinement chain. In modeling in Event-B, developers construct specifications with a set of *machines*. After constructing an abstract machine, they introduce more aspects of the target system by constructing a new machine with more details and verifying the consistency between the new machine and the abstract one.

The refinement mechanism of Event-B enables developers to design structures composed of refinements. In other words, developers can decide aspects of the target system that are considered in each refinement step. Thus, the design is impor-

This work is partially supported by JSPS KAKENHI Grant Number 26700005.

J. Fitzgerald et al. (Eds.): FM 2016, LNCS 9995, pp. 444–459, 2016.
DOI: 10.1007/978-3-319-48989-6_27

tant for understandability, ease of verification, maintainability, and reusability. However, refinement structures have not been a target of refactoring.

In this paper, as a foundation to support refinement restructuring, we propose a method for decomposing refinements. In particular, for given consistent (i.e. proved) machines M_{A} and M_{C} such that M_{C} refines M_{A}, our method helps users to construct an *intermediate* machine M_{B} such that M_{C} refines M_{B} and M_{B} refines M_{A}. This enables users to decompose a refinement step into several substeps. The decomposition method can be combined with merging of refinements, which is simpler than decomposition, to restructure refinements.

We show the usefulness of refinement restructuring through two case studies. The first shows how decomposing large-scale refinements can improve the maintainability of existing machines. The second shows how to extract parts of existing machines and reuse them for constructing new machines of another system that is different from the original system at the first glance.

The remainder of this paper is organized as follows. First, we provide background on Event-B in Sect. 2. We then describe our proposal for decomposing (and merging) refinement in Sect. 3. Next, we show two case studies in Sect. 4. In Sects. 5 and 6, we discuss application of our method and related work, respectively. Finally, we conclude this study in Sect. 7.

2 Background

A model in Event-B is composed of contexts and machines. The static properties of the target system are specified in contexts, whereas its dynamic properties are specified in machines as predicates of invariants and events. Machines can refer to specifications in contexts. The main part of a specification of events consists of guards and actions. Guards are predicates of necessary conditions for executing the state transitions of an event. Actions describe the state transitions of an event with *before-after predicates* (BAPs), which are relationships between the before and after states of variables.

For example, a machine `ma` (Fig. 1) has specifications of variables a and b, invariant `typ_a`, and events `initialisation` and `evt_a`. An action is composed of the variables that are changed by the action and a BAP. In BAPs, the after states of variables are expressed using variables with primes, such as a'. In the figure, event `evt_a` increases the values of a and b by 1 and 2, respectively, and it can be executed if $0 \leq a$.

In modeling in Event-B, new aspects and details are gradually introduced to a machine through a refinement mechanism. A machine M_{C} can be defined as a refinement of another machine M_{A}. Here, M_{C} and M_{A} are called a concrete machine and an abstract machine, respectively.

We use the symbols V_{A} and V_{C} to denote M_{A}'s variables and M_{C}'s variables, respectively. The invariants in a concrete machine M_{C} can refer to V_{A} in addition to V_{C}. Those that refer to both variables in V_{A} and those in V_{C} are called *gluing invariants*, because they connect the state spaces of two machines.

variables: a, b

typ_a: $\{a, b\} \subset \mathbb{N}$

Event evt_a
  when
    grd_a1: $0 \leq a$
  then
    act_a1: a :| $a' = a + 1$
    act_a2: b :| $b' = b + 2$
  end

Event initialisation
  begin
    init_a1: a :| $a' = 0$
    init_a2: b :| $b' = 0$
  end

Fig. 1. Abstract machine ma

V_C does not need to be a superset of V_A. If $V_A \not\subseteq V_C$, some of the variables in V_A are *replaced* with some of the variables in V_C. In such a replacement, developers also need to provide gluing invariants that refer to the replaced variables (in V_A) and replacing variables (in V_C), in order to prove consistency between an abstract machine and a concrete machine.

Moreover, events in M_C may refine events in M_A. Concrete events, which refine events in the abstract machine (abstract events), need to have guards that are stronger than the guards of abstract events. Also, the actions of concrete events should simulate the actions of their abstract events.

For instance, suppose that machine mc (Fig. 2)[1] is defined as a refinement of ma (Fig. 1). In mc, a variable a is inherited from ma, variables c, d, e, and f are newly introduced, and a variable b, which is specified in ma, has disappeared. The gluing invariant gluinv_c1 describes the relationship among b, c, and d. Event evt_c is defined as a concrete event of evt_a of ma.

variables: a, c, d, e, f

typ_c: $\{a, c, d, e, f\} \subset \mathbb{N}$
gluinv_c1: $b = c + d$
inv_c1: $\mathrm{mod2}(a + e) = 0 \Rightarrow a < 1$
inv_c2: $\mathrm{mod2}(e + f) = 1$

Event evt_c
  refines evt_a
  when
    grd_c1: $0 \leq a \wedge 0 \leq c$
    grd_c2: $\mathrm{mod2}(a + f) = 0$
  then
    act_c1: a :| $a' = a + 1$
    act_c3: c :| $c' = c + 1$
    act_c4: d :| $d' = d + 1$
    act_c5: e :| $e' = f + 2$
    act_c6: f :| $f' = f + 3$
  end

Event initialisation
  begin
    init_c1: a :| $a' = 0$
    init_c3: c :| $c' = 0$
    init_c4: d :| $d' = 0$
    init_c5: e :| $e' = 1$
    init_c6: f :| $f' = 2$
  end

Fig. 2. Concrete machine mc

The refinement mechanism enables two styles of refinement, namely, gradual addition of concrete elements (*horizontal refinement*) and transformation of expressions to make them closer to the implementation (*vertical refinement*).

[1] Assume that a function $\mathrm{mod2}(n)$ that returns n modulo 2 is defined in a context.

The consistency of the specified machine is represented in the form of sequents, called *proof obligations* (POs), generated from the specification. POs include sequents of the machine's self-consistency and its consistency with the abstract machine. Developers confirm consistency by discharging all POs. When POs cannot be discharged, developers need to modify the specification.

For instance, one of the primary PO types is *invariant preservation* (written as $evt/inv/\mathrm{INV}$), which means an invariant inv holds after an event evt occurs.

The rule of PO $evt/inv/\mathrm{INV}$ is as follows:[2] $I, J, H(evt), T(evt) \vdash inv'$, where I and J are invariants of an abstract machine and a concrete machine, respectively, $H(evt)$ and $T(evt)$ are respectively the guards and BAPs of event evt in the concrete machine, and inv' is inv with the before-state variables replaced by after-state variables.

For example, the PO `mc`/`evt_c`/`inv_c1`/INV is as shown in Fig. 3.

`grd_c2` BAP of `act_c1` BAP of `act_c5` ... $\vdash$ Modified `inv_c1`	$\mathrm{mod2}(a+f)=0$ $a'=a+1$ $e'=f+2$ ... $\vdash$ $\mathrm{mod2}(a'+e')=0 \Rightarrow a'<1$

Fig. 3. Invariant preservation of `inv_c1` by `evt_c` in `mc` (provable)

3 Approach

3.1 Method Overview

We assume that we have given consistent (proved) machines M_{A} and M_{C} such that M_{C} refines M_{A}. The goal of our decomposition method is to construct an intermediate machine M_{B} such that M_{C} refines M_{B} and M_{B} refines M_{A} by using as much of the original specifications as possible. For this purpose, users give a slicing criterion as a set of variables V_{B0}, which actually may be given by selecting variables in V_{C}. The first step is to use this criterion for syntactic slicing from M_{A} and M_{C} to construct the initial base M_{B0}. The actual criterion for slicing V_{B} is extended from V_{B0} because of consistency constraints. In general, the result of this first step M_{B0} may have POs that are not provable. Thus, the second step adds complementary predicates to M_{B0} to make a consistent intermediate machine M_{B}. By handling replacement of variables through refinement and proof obligations, our decomposition method deals with both horizontal refinement and vertical refinement. Combined with merging of refinements, the decomposition method is extended as a restructuring method (Sect. 3.4).

[2] Actually static predicates (axioms) and predicates of event parameters are also included in POs. We will omit them for the sake of simplicity.

3.2 Step 1 of Decomposing Refinement: Slicing

Finding Additional Variables. If M_{C} refines M_{A}, then M_{C} inherits variables $V_{\text{A}} \cap V_{\text{C}}$ from M_{A}. The remaining variables of M_{A}, that is, $V_{\text{A}} \setminus V_{\text{C}}$, are replaced with some of variables in $V_{\text{C}} \setminus V_{\text{A}}$, as described in Sect. 2.

As shown on the left side of Fig. 4, if $V_{\text{A}} \cap V_{\text{C}} \not\subseteq V_{\text{B}}$, then the variables in $(V_{\text{A}} \cap V_{\text{C}}) \setminus V_{\text{B}}$ $(= V_{\text{A}\overline{\text{B}}\text{C}})$, which is a subset of $V_{\text{A}} \setminus V_{\text{B}}$, is specified in M_{C}. The variables in $V_{\text{A}} \setminus V_{\text{B}}$ are, however, replaced with other variables in M_{B}. Thus, selecting such a V_{B} makes the refinement "M_{C} refines M_{B}" inconsistent. Therefore, V_{B} must satisfy $V_{\text{A}} \cap V_{\text{C}} \subseteq V_{\text{B}}$. In addition, to take advantage of predicates in existing machines to construct M_{B}, V_{B} should satisfy $V_{\text{B}} \subseteq V_{\text{A}} \cup V_{\text{C}}$; otherwise, a user needs to design new variables $(V_{\text{B}} \setminus (V_{\text{A}} \cup V_{\text{C}}))(= V_{\overline{\text{A}}\text{B}\overline{\text{C}}})$ and predicates of them. Thus, V_{B} should be as depicted on the right side of Fig. 4. Hereinafter, we will use the symbols $V_{\text{A}\overline{\text{BC}}}$, $V_{\text{AB}\overline{\text{C}}}$, V_{ABC}, $V_{\overline{\text{A}}\text{BC}}$, and $V_{\overline{\text{AB}}\text{C}}$ to represent $V_{\text{A}} \setminus V_{\text{B}}$, $V_{\text{B}} \setminus V_{\text{C}}$, $V_{\text{A}} \cap V_{\text{C}}$, $V_{\text{B}} \setminus V_{\text{A}}$, and $V_{\text{C}} \setminus V_{\text{B}}$, respectively.

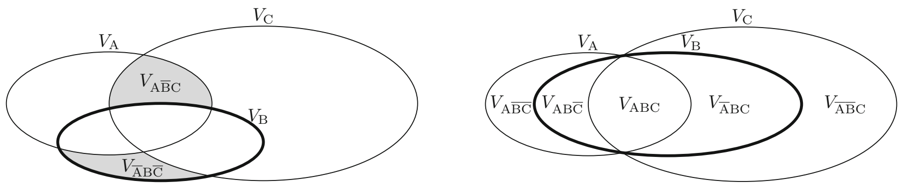

Fig. 4. Variables of M_{B} : invalid case (left) and valid case (right)

This method assumes that $V_{\text{B0}} = V_{\text{ABC}} \cup V_{\overline{\text{A}}\text{BC}}$, which is a subset of V_{C}, is given as an input. This is because it is easy for a user to select the criterion from V_{C}, without considering which variables in V_{A} must be replaced. To construct M_{B}, the remaining variables $V_{\text{B}} \setminus V_{\text{B0}}(= V_{\text{AB}\overline{\text{C}}})$ need to be identified. The remainder of this section describes a heuristic for automatically finding $V_{\text{AB}\overline{\text{C}}}$.

To replace abstract variables with concrete ones in a refining machine, a user needs to provide gluing invariants about the relationships between the two sets of variables. In the case of constructing M_{B} as a machine that refines M_{A}, the set of newly introduced variables $V_{\overline{\text{A}}\text{BC}}$ is a subset of $V_{\text{C}} \setminus V_{\text{A}}$. Therefore, some of the gluing invariants in M_{C} may not be specified in M_{B}. M_{B}'s gluing invariants can describe the relationship between $V_{\text{A}\overline{\text{BC}}}$ and $V_{\overline{\text{A}}\text{BC}}$, but cannot describe the relationship between $V_{\text{AB}\overline{\text{C}}}$ and $V_{\overline{\text{A}}\text{BC}}$. Hence, $V_{\text{AB}\overline{\text{C}}}$ can be obtained as
$V_{\text{AB}\overline{\text{C}}} = \{v \in V_{\text{A}} \setminus V_{\text{C}} | \exists i \in \text{ginv}(M_{\text{C}}).v \in (\text{var}(i) \cap V_{\text{A}}) \wedge (\text{var}(i) \cap V_{\text{C}}) \subseteq V_{\overline{\text{A}}\text{BC}}\}$
, where $\text{ginv}(M)$ represents the gluing invariants in a machine M and $\text{var}(p)$ represents the variables that occur in predicate p.

For example, let us assume that a and e are selected to be specified in `mb` $(V_{\text{B0}} = V_{\text{B}} \cap V_{\text{C}} = \{a, e\})$. In `mc`, a gluing invariant `gluinv1:` $b = c + d$ describes replacement of b (of `ma`) with c and d (of `mc`). By contrast, in `mb`, `gluinv1` cannot describe replacement of b, since neither c nor d is selected to be specified in `mb`. Therefore, $V_{\text{AB}\overline{\text{C}}} = \{b\}$; namely `mb` should specify b in addition to a and e.

Finding Certain Specifications through Slicing. For a predicate p and a set of variables V, we say p is *expressible* by V if and only if $\mathrm{var}(p) \subseteq V$.

Predicates in M_{A} and M_{C} that are expressible by V_{B} are necessary (but not always sufficient, as described later) in M_{B}, because they certainly express the properties of V_{B}, which should be consistent with M_{A} and M_{C}. Therefore, in this step, invariants, guards, and BAPs in M_{A} and M_{C} that are expressible by V_{B} are specified in M_{B}.[3] For example, `mb0` (Fig. 5) is constructed by collecting predicates that are expressible by $V_{\mathrm{B}} = \{a, b, e\}$ from `ma` and `mc`.

```
variables: a, b, e
```

```
typ_b: {a, b, e} ⊂ ℕ
inv_c1: mod2(a + e) = 0 ⇒ a < 1
```

```
Event evt_b
  refines evt_a
  begin
    act_c1: a :| a′ = a + 1
    act_a2: b :| b′ = b + 2
  end
```

```
Event initialisation
  begin
    init_c1: a :| a′ = 0
    init_a2: b :| b′ = 0
    init_c5: e :| e′ = 1
  end
```

Fig. 5. Sliced machine `mb0`

Note that an action `mc/evt_c/act_c5`, which assigns a value to $e (\in V_{\mathrm{B}})$, is not specified in `mb0`, because $f (\in V_{\overline{\mathrm{ABC}}})$ occurs in its BAP.

We implemented this step as a plugin tool of Event-B's IDE, named SliceAndMerge[4]. Users of the tool can select V_{B0} with checkboxes and obtain a sliced machine M_{B0}. The tool also supports analysis of dependencies between invariants and variables, and merging of refinements (described in Sect. 3.4).

3.3 Step 2 of Decomposing Refinement: Complementing

Possible Lack of Consistency in M_{B0}. Although all POs of M_{A} and M_{C} are discharged, M_{B0}, which is constructed from fragments of the machines, is not ensured to be consistent. For example, some invariants in M_{B0} (from M_{A} and M_{C}) may not be preserved by events in M_{B0}, because the specification of M_{B0}'s events may be only part of the specification of M_{A} and M_{C}'s events.

For instance, the PO shown in Fig. 3 (`mc/evt_c/inv_c1`/INV), which has a succedent specified with the after-states of a and e, is provable because predicates including `grd_c2` and `act_c5` are in the antecedent.

Although the preservation of the same invariant `inv_c1` by the event `evt_b` in `mb0` (`mb0/evt_b/inv_c1`/INV) should also hold, this is not provable because predicates `grd_c2` and `act_c5`, which are essential for proving that `inv_c1` is preserved, are not included in the antecedent (Fig. 6), as they are predicates about variable f $(\in V_{\overline{\mathrm{ABC}}})$.

[3] BAPs that are expressible by $V_{\mathrm{B}} \cup V'_{\mathrm{B}}$ are also specified, where V'_{B} represents the set of after-state variables of V_{B}.

[4] Available at http://tkoba.jp/software/slice_and_merge/.

BAP of `act_c1`	$a' = a + 1$
...	...
$\vdash$	$\vdash$
Modified `inv_c1`	$\mathrm{mod2}(a' + e') = 0 \Rightarrow a' < 1$

Fig. 6. Invariant preservation of `inv_c1` by `evt_b` in `mb0` (unprovable)

Complementary Predicates for Consistency. Since M_{A} and M_{C} are consistent, they have predicates that are essential for the consistency.

When such predicates are expressible by V_{B}, the consistency of M_{B} can be guaranteed by including them in M_{B}. Obviously in simple cases, this can be realized by slicing (Sect. 3.2).

However, as described above, sometimes M_{B0} is inconsistent because M_{B0}, which is obtained by a syntactic predicate-level slicing, sometimes lacks some of these predicates. In such cases, predicates that are essential for discharging POs need to be added to M_{B0}, so that the resulting M_{B} is consistent. Moreover, such predicates need to be expressible by V_{B}. We call such additional predicates *complementary predicates* (CPs). Predicates that are essential for the consistency of original machines can be found from the specifications or the proof of consistency of M_{A} and M_{C}, and they can be "translated" into V_{B} as CPs. Some CPs may work as gluing invariants. We discuss how often CPs are required and how hard finding them is in Sect. 5. The rest of this section describes ways to do this.

Finding CPs Using Rule-based Analysis. In some cases, part of a predicate is expressible by V_{B} but the remainder of it is not; thus, the predicate cannot be obtained through predicate-level slicing. Simple heuristics can be used to find parts of such predicates that are expressible by V_{B}. For instance, a predicate $0 \le a$ can be found by extracting a part that is expressible by V_{B} from `mc/evt_c/grd_c1` ($0 \le a \wedge 0 \le c$). A possible implementation of this is to convert predicates into CNF and extract clauses that are expressible by V_{B}.

Finding CPs from Existing Proofs. The essence of the consistency of M_{A} and M_{C} can be found by examining the proof of consistency and making an inference.

For example, the proof of `mc/evt_c/inv_c1`/INV can be summarized in terms of goals (succedents) as shown on the left side of Fig. 7. The initial goal GLc0: $\mathrm{mod2}(a' + e') = 0 \Rightarrow a' < 1$ can be derived because GLc1: $\mathrm{mod2}(a' + e') \neq 0$ can be derived from hypotheses including a guard `grd_c2`.

A proof with the same root goal is possible using the vocabulary of `mb` if the goal GLb1: $\mathrm{mod2}(a' + e') \neq 0$ can be derived from an event-local predicate (guard or BAP) p that is expressible by V_{B}. GLb1 can be transformed into GLb3: $\mathrm{mod2}(a + e') = 0$ by `act_c1`. We need to find p such that GLb3 can be derived from p, because there is no predicate about e' in `mb0`. A solution is to view GLb3 itself as p and add an action such as `act_NEW` : e `:|` $\mathrm{mod2}(a + e') = 0$ to `mb0` (the right side of Fig. 7).

GLc0: $mod2(a' + e')=0 \Rightarrow a' < 1$ (modified inv_c1)
- GLc1: $mod2(a' + e') \neq 0$
  - GLc2: $mod2((a + 1) + (f + 2)) \neq 0$ (by act_c{1,5})
    - GLc3: $mod2(a + f + 1) \neq 0$
      - GLc4: $mod2(a + f)=0$
        - GLc5: $\top$ (by grd_c2)

GLb0: $mod2(a' + e')=0 \Rightarrow a' < 1$ (modified inv_c1)
- GLb1: $mod2(a' + e') \neq 0$
  - GLb2: $mod2((a + 1) + e') \neq 0$ (by act_c1)
    - GLb3: $mod2(a + e')=0$
      - GLb4: $\top$ (by act_NEW)

Fig. 7. Proof of mc/evt_c/inv_c1/INV (left) and mb/evt_b/inv_c1/INV (right)

Finding CPs as Craig Interpolant. The essence of consistency of M_{A} and M_{C} as expressed by V_{B} can often be found as a Craig interpolant of the proof of consistency.

Let ϕ_{C}: $\mathbf{Ant_C} \vdash \mathbf{Suc_C}$ be a sequent of the proof of consistency in M_{C}. A sequent ϕ'_{C}: $\mathbf{Ant'_C} \vdash \mathbf{Suc'_{BC}}$ such that $\mathcal{V}(\mathbf{Suc'_{BC}}) \subseteq V_{\mathrm{B}}$ can be inferred by using inference rules for sequent calculus such as negation rules, where $\mathcal{V}(\mathbf{X})$ denotes the set of variables in $\mathbf{X}$.

An interpolant of ϕ'_{C} $\mathcal{I}$ can be obtained. According to the Craig's interpolation theorem, $\mathcal{I} \vdash \mathbf{Suc'_{BC}}$ is provable. Moreover, $\mathcal{V}(\mathcal{I}) \subseteq \mathcal{V}(\mathbf{Ant'_C}) \cap \mathcal{V}(\mathbf{Suc'_{BC}}) \subseteq V_{\mathrm{B}}$. Thus, $\mathcal{I}'$ corresponds to an embodiment of the essence of ϕ'_{C} in V_{B}.

Let ϕ_{B0}: $\mathbf{Ant_{B0}} \vdash \mathbf{Suc_{B0}}$ be a (unprovable) sequent of consistency in M_{B0}. If another sequent ϕ'_{B0}: $\mathbf{Ant'_{B0}} \vdash \mathbf{Suc'_{BC}}$ can be inferred from ϕ_{B0} by using inference rules, then ϕ_{B0} becomes provable by adding a predicate $\mathcal{I}$ to M_{B0}, because $(\mathcal{I} \wedge \mathbf{Ant'_{B0}}) \vdash \mathbf{Suc'_{BC}}$.

For example, the sequent shown in Fig. 8 can be inferred from the sequent of mc/evt_c/inv_c1/INV (Fig. 3). Variables that occur in the succedent of the sequent are $\{a, a', e'\} \subset V_{\mathrm{B}} \cup V'_{\mathrm{B}}$. The predicate $\mathrm{mod2}(a+e') = 0$ is an interpolant of the sequent, and it is expressible by V_{B}.

By adding the predicate to mb0 as an action act_NEW : e :| $\mathrm{mod2}(a+e') = 0$, the sequent mb/evt_b/inv_c1/INV becomes provable, as shown in Fig. 9 (because (BAP of act_c1) $\wedge$ (BAP of act_NEW) $\Rightarrow$ Modified inv_c1).

Note that if an action is obtained from the sequent of mc/evt_c/inv_c1/INV (Fig. 3) (i.e. action is obtained without applying inference rules), it becomes act_NEW_nondet : a, e :| $\mathrm{mod2}(a' + e') \neq 0$, which is more non-deterministic than necessary (act_NEW).

grd_c2 BAP of act_c5 ... $\vdash$ Modified inv_c1 $\neg$ (BAP of act_c1)	$\mathrm{mod2}(a + f) = 0$ $e' = f + 2$ ... $\vdash$ $\mathrm{mod2}(a' + e') = 0 \Rightarrow a' < 1$ $\neg(a' = a + 1)$

Fig. 8. A sequent inferred from mc/evt_c/inv_c1/INV

BAP of `act_c1`
BAP of `act_NEW`
...
$\vdash$
Modified `inv_c1`

$a' = a + 1$
$\mathrm{mod2}(a + e') = 0$
...
$\vdash$
$\mathrm{mod2}(a' + e') = 0 \Rightarrow a' < 1$

Fig. 9. Invariant preservation of `inv1` by `evt_b` in `mb0` with interpolant (provable)

3.4 Restructuring Refinement

We call a sequence of machines $[M_n, M_{n+1}, \ldots M_{m-1}, M_m]$ *refinement chain (RC)* if M_{i+1} refines M_i for every natural number i such that $n \leq i < m$.

In addition to decomposing, we can *merge* refinements as follows: When there is a RC $[M_0, M_1, M_2]$, merging M_1 and M_2 construct a new machine M_{12} such that M_{12} refines M_0. M_{12}'s variables, invariants, and events are composed of the unions of the variables, invariants, and events of M_1 and M_2.

Refinements can be restructured by merging and decomposing refinements. Suppose that a RC $[M_n, \cdots, M_m]$ is given. First, machines $(M_i)_{i=n+1}^{m}$ are merged as M'_m, which directly refines M_n. Then, a RC $[M_n, M'_m]$ is decomposed by constructing new machines $(\tilde{M}_i)_{i=k+1}^{l}$ that reflect the user's preference of aspects in terms of V_{B0}. As a result, the refinement is restructured into a RC $[M_n = \tilde{M}_k, \tilde{M}_{k+1}, \ldots, \tilde{M}_{l-1}, \tilde{M}_l = M'_m]$. As a result of restructuring refinements, the understandability of a specification increases because the meaning of each refinement step can be changed as the user likes. In Sect. 4.2, we describe an application of restructuring method to extract parts of an existing model for reuse.

4 Case Studies

4.1 Case Study 1: Decomposing Large Refinement Steps

This case study tried to determine whether we can improve maintainability of existing machines by decomposing refinements. One of the authors of this paper decomposed refinements in a large-scale Event-B model with several intermediate machines by following our method and verified their consistency. The target model was a specification about an autonomous satellite flight formation system [12], and it was constructed by a computer scientist who had over four years of experience in modeling in Event-B. The target system was a controller for two spacecraft (leader and follower), which run autonomously while maintaining two-layered communication, namely a higher-layer mode communication and a lower-layer phase communication.

The model has a RC of five steps $[\mathtt{m0}, \mathtt{m1}, \ldots, \mathtt{m5}]$. The second refinement $([\mathtt{m1}, \mathtt{m2}])$ and the third refinement $([\mathtt{m2}, \mathtt{m3}])$ were selected to be decomposed, because they were larger than the other steps. The row of `m2` in Table 1a and the row of `m3` in Table 1b show statistics of `m2` and `m3`, respectively. The N_{V}

and N_{I}[5] in Table 1 respectively list the numbers of variables and invariants of the models. In m2, seven variables and 46 invariants were introduced to specify mode transitions and communications in the spacecraft. In m3, two variables have disappeared, ten variables were introduced (N_{V} is "$-2+10$"), and 72 invariants were introduced to specify the phase transitions in modes of spacecraft.

Table 1. Results of case study 1

(a) Decomposition of second refinement

	N_{V}	N_{I}	N_{CP}	N_{UCP}	N_{PO}	N_{MPO}
m2	+7	46	–	–	454	53
m2_1	+4	12	21	5	112	12
m2_2	+1	9	10	6	87	0
m2_3	+1	8	12	6	80	5
m2_4	+1	17	0	0	218	33
Sum of m2_*	+7	46	43	17	497	50

(b) Decomposition of third refinement

	N_{V}	N_{I}	N_{CP}	N_{UCP}	N_{PO}	N_{MPO}
m3	-2+10	72	–	–	1127	175
m3_1	-1+3	7	17	4	112	6
m3_2	-1+3	17	17	8	261	30
m3_3	+2	14	3	2	202	30
m3_4	+2	34	0	0	584	81
Sum of m3_*	-2+10	72	37	14	1159	147

We selected slicing criteria V_{B0} to obtain the sliced machines. After that, we found CPs with the approach described in Sect. 3.3. Both of the refinements decomposed with four intermediate machines (m2_1, ..., m2_4, m3_1, ..., and m3_4). Thus, the machines form a RC [m1, m2_1, ..., m2_4, m3_1, ..., m3_4]. The most concrete intermediate machines m2_4 and m3_4 were semantically the same[6] as the corresponding original machines m2 and m3. We selected slicing criteria so that the slicing would distribute aspects in the original machines into small and meaningful sets of concepts. For example, the properties and behavior regarding communication failures, the follower's incoming buffer for mode messages, the leader's outgoing buffer, and the acknowledgement message were specified and verified in m2_1, m2_2, m2_3, and m2_4, respectively.

The results of decomposition are as shown in Table 1. The number of introduced invariants was reduced significantly through the decomposition, and the intermediate machines were more comprehensible than the originals. The replacement of the variables in m3 was also split into two steps. In both m3_1 and m3_2, one variable has disappeared (N_{V} of both machines is "$-1+3$").

We needed to add CPs to the intermediate machines except the most concrete ones. The N_{CP} in the Table 1 list the numbers of added CPs. Similar events in M_{B0}, such as the events of entering phase 1, phase 2, and phase 3, often had the same kind of inconsistency and thus required the same kind of CPs. The numbers of unique CPs (N_{UCP}) show the actual burden of finding CPs.

[5] For the sake of simplicity we did not count invariants for typing.

[6] There were differences in the actual specifications, because several invariants were moved in order to abstract the intermediate machines and the refinement structures of the events were changed.

The N_{PO} and N_{MPO} in Table 1 respectively list the numbers of all POs and the numbers of POs that were manually discharged, including those of POs related to CPs. Most of POs are usually discharged by automatic provers of the IDE for Event-B. Thus the number of manually discharged POs (N_{MPO} in Table 1) corresponds to the actual amount of effort for verification. The results show that our method decreased the labor of verification. For example, rows of `m3` and "sum of `m3_*`" in Table 1b show that the number of manually discharged POs decreased from 175 to 147 through decomposition, despite that the number of all POs increased from 1127 to 1159. This appears to be because direct inclusion of CPs added lemmas to the set of hypotheses. Our future work includes a detailed analysis of this effect.

4.2 Case Study 2: Extracting Reusable Parts of Machines

This case study[7] tried to determine whether we can extract reusable parts of existing machines by using restructuring (Sect. 3.4).

We used a model of a "location access controller" (from [1, Chap. 16]) as the original model $\mathbf{M_O}$ with a RC $[M_{\mathrm{O}1}, \ldots, M_{\mathrm{O}5}]$ (Fig. 10). The model is about a controller of doors between locations according to persons' permission to enter.

Step 1 ($M_{\mathrm{O}1}$): <u>Persons</u> somehow *move* between <u>locations</u> according to the *authorization of persons to locations.*
Step 2 ($M_{\mathrm{O}2}$): *Physical connections* between <u>locations</u> are introduced. Persons *move* between physically connected locations.
Step 3 ($M_{\mathrm{O}3}$): *Doors* with <u>red</u>/*green* lights are introduced. Doors somehow <u>authenticate</u> persons.
Step 4 ($M_{\mathrm{O}4}$): <u>ID cards</u> are introduced. *Doors* read cards and <u>communicate</u> with a controller by <u>messages</u> to <u>authenticate</u>.
Step 5 ($M_{\mathrm{O}5}$): Physical movements of *doors*, *persons*, and <u>lights</u> are considered. <u>Communication</u> is a reaction to a physical event.

Fig. 10. Aspects introduced in each step of original model $\mathbf{M_O}$(Aspects that should be extracted from $\mathbf{M_O}$ to construct $\mathbf{M_N}$ are <u>underlined</u> and those that should be omitted from $\mathbf{M_O}$ are *slanted*.)

We constructed a new model $\mathbf{M_N}$ by reusing parts of $\mathbf{M_O}$. Aspects shown in Fig. 11 are specified in $\mathbf{M_N}$.

First, we constructed a machine M_{mrg} by merging all the machines of $\mathbf{M_O}$. Next, by slicing M_{mrg}, we extracted aspects that were common to $\mathbf{M_O}$ and $\mathbf{M_N}$. Thus, we extracted specifications related to authentication using communication between card readers and a controller (from $M_{\mathrm{O}4}$ and $M_{\mathrm{O}5}$), persons (from $M_{\mathrm{O}1}$), locations (from $M_{\mathrm{O}1}$), and red lights (from $M_{\mathrm{O}3}$ and $M_{\mathrm{O}5}$). In other words, we omitted aspects that would not be included in $\mathbf{M_N}$; i.e., we omitted authorization of persons to locations (from $M_{\mathrm{O}1}$), physical connection of locations (from

[7] Models of this case study are at http://tkoba.jp/publications/fm2016/

– Persons are in locations but do not move to other locations.
– Locations have monitors and consoles with card readers.
– Authenticated persons log in to the server by inserting their ID card in a reader.
– A red light indicates an authentication failure.
– The controller tries to find an unoccupied monitor in the room.
– Consoles communicate with a controller by sending messages.

Fig. 11. Aspects of new model $\mathbf{M_N}$(Aspects that should be extracted from $\mathbf{M_O}$ to construct $\mathbf{M_N}$ are underlined and those that should be omitted from $\mathbf{M_O}$ are *slanted.*)

M_{O2}), doors (from M_{O3}), and green lights (from M_{O3} and M_{O5}), in addition to movement of persons (from M_{O1}), which is the primary aspect of $\mathbf{M_O}$.

As a result, we succeeded in automatically extracting the reusable parts from M_{mrg}. In other words, we did not need to add CPs to make the reusable parts consistent. After that, we successfully augmented the reusable parts with specifications that were unique to $\mathbf{M_N}$. We also succeeded in discharging all POs.

Note that not only omitted aspects in $\mathbf{M_O}$ but also extracted aspects were scattered over several refinement steps in the original specification. Therefore, simply copying a single step such as M_{O3} and modifying it is not an effective way of reusing such aspects. In contrast, we succeeded in extracting aspects in a cross-refinement manner by slicing after merging refinement steps.

5 Discussion

5.1 Discussion on Methods

Deriving CPs. All POs originate from specifications. Hypotheses essential to discharge POs are also inferred from specifications. We call predicates that raise a PO ϕ *raisers* of ϕ and predicates that provide hypotheses for discharging ϕ *hypothesis providers* of ϕ.

Suppose that ϕ is a PO in a concrete machine. If the raisers of ϕ are expressible by V_{B}, the hypotheses required to discharge ϕ should also be expressible by V_{B}. However, hypotheses providers are not always specified with vocabulary of V_{B}. Sometimes, a PO ϕ that is expressible by V_{B} is discharged with hypotheses including a hypothesis h that is expressible by V_{B}, and h is implied by hypotheses providers P that are expressible by V_{C} but not expressible by V_{B}. In other words, h is not directly specified in the machine but rather implicitly specified by P in this case. In such cases, ϕ is raised but cannot be discharged in the intermediate machine since the intermediate machine lacks some of hypotheses providers for ϕ. Thus, users need to add CPs that are expressible by V_{B} and able to imply hypothesis h.

However, developers tend to directly specify hypotheses in practice, because hypotheses raisers for POs are usually important properties of a target system; thus, directly specifying hypotheses to discharge the POs is usually a meaningful way of describing the system. Therefore, users do not need to add CPs frequently.

For instance, we did not need to add CPs in the second case study (Sect. 4.2), because all of the hypotheses providers were specified in V_{B} for all of the POs that were expressible by V_{B}.

Specifying a hypothesis provider in the form $h \wedge \mathit{predicate}$ to imply hypothesis h is another common case. Although users need to add CPs, they can be found with simple rules. In other cases, CPs can be found by reviewing the proofs for the original machines, as described in Sect. 3.3. This task is easy for users who are familiar with Event-B.

Therefore, we conclude that finding CPs is neither frequently required nor difficult. As a primary part of our future work, however, we are planning to construct systematic and complete methods for deriving CPs so that developers can easily derive consistent intermediate machines. We will investigate relationships between CPs and Craig interpolation of the completed proof further.

Selecting Slicing Criteria. Users of our decomposition method can select a slicing criterion, namely variables that are specified in the intermediate machine. Users may consider aspects of the intermediate machine and select some of the variables of the concrete machine, or they may consider properties that should be verified in the intermediate machine and select some of the invariants of the concrete machine. In the latter case, the slicing criterion is a set of variables that are required to specify selected invariants. Users can select an arbitrary V_{B0} so long as $V_{\mathrm{A}} \cap V_{\mathrm{C}} \subseteq V_{\mathrm{B}} \subseteq V_{\mathrm{A}} \cup V_{\mathrm{C}}$.

Adding New Concepts of Abstraction. A user can add new concepts of abstraction to the machines, by decomposing refinement after adding new specifications for abstraction to the concrete machine.

One way is adding new variables. For example in Fig. 2, by creating an intermediate machine that have $\{g\}$ as V_{B0} after adding a variable g and an invariant $g = a + e$ and other predicates, a user can construct an intermediate machine for specification of variables b and g instead of variables b, a and e.

The other way is adding new events. Assume that a concrete machine has several events E that have common guards and actions. By selecting variables that occur in common predicates in E as V_{B0}, a user can construct an intermediate machine with an abstract event, which is refined by all events of E.

These appear to be useful for restructuring refinement of existing models.

5.2 Discussion on Applications

Improvement of Maintainability by Decomposition. In our first case study (Sect. 4.1), we decomposed large refinements into smaller ones. The primary benefit of reducing size of specifications is the support of maintaining machines. According to a study conducted in industries [11], activities for formal specifications' maintenance include impact analysis, refactoring identification, and validation. Our decomposition method makes such activities easier because it shrinks the size of the state space and the number of predicates and reveals implicit properties of concrete machines as CPs. In particular, reducing size of specifications can significantly reduce the cost of verification [9] in maintenance. Thus, our decomposition method improves maintainability of each single refinement step. Our future work includes evaluation of trade-off between this and the maintainability of the whole model.

Large Refinement Steps. Large refinement steps such as ones used in our first case study (Sect. 4.1) are common. Developers design refinements on the basis of properties that should be verified or subjects that should be considered in each step. Usually, such properties or subjects are about multiple aspects of the target system. Therefore, including many aspects in one refinement step may seem natural for developers when they construct machines and are in fact common, despite that smaller refinements are easier to comprehend. Thus, we believe our decomposition method is effective for most existing Event-B machines.

Effectiveness of Systematic Extraction of Reusable Parts. In our second case study (Sect. 4.2), we automatically extracted reusable parts of an existing model. Manually extracting such parts is not impossible, namely developers can extract such parts by examining several machines of the original model and copy-and-pasting. However, the number of predicates that should be examined is large. In addition, such predicates are usually scattered over several machines. Therefore, manual examination is tedious and error-prone. Our method makes this process more systematic (and sometimes automatic).

Feasibility of Automatic Extraction of Reusable Parts. In our second case study (Sect. 4.2), we extracted aspects of "authentication using communication between card readers and a controller" as reusable parts of the original machines. In the original machines, these aspects were introduced through several refinement steps and it seemed that they were dependent on other parts. However, they were actually independent of other parts, and we succeeded in extracting them in an automatic manner. We often see this kind of independence of parts embedded in machines. Our method is an automatic extraction of such parts. Although users sometimes need to add CPs, most of the predicates can be found with rules as we described in this section.

6 Related Work

Decomposition of Event-B machines (in "shared variable" style [2] and "shared event" style [3]) is one of primary mechanisms to deal with complexity of modeling in Event-B. The aim of these methods is decomposing a large single machine into several components. Conversely, our goal is decomposing and merging **refinement structure** of multiple machines.

There have been many studies on refactoring software models for the purpose of organizing and understanding of them. Refactoring rules for UML/OCL [5,8], ASM [14], Alloy [7], and Object-Z [10,11] have been proposed. Most of these rules are similar to popular refactoring rules such as move and modification, as well as rules for parameterization of expressions and introduction of inheritance and polymorphism. The goal of our work is similar to theirs, but we take a different approach based on refinement, namely by manipulating refinement structures according to criteria of the vocabulary of a machine.

From the point of view of refactoring of verifications, study by Whiteside [13] has a similar goal to ours. Their study manipulates proofs in proof assistants by providing a proof script framework that handles proof trees in a hierarchical way. One of their primary contributions is refactoring of proof scripts, including manipulating expressions of proof scripts, changing styles of proof, and generalizing tactics. Our approach, namely refinement refactoring considering the vocabulary of a module, is different from theirs.

A number of significant studies on formal methods have used Craig interpolation of logic formulas, which we found to be important for finding CPs. One of the primary applications of interpolation is counterexample-guided abstraction refinement [4] in model checking, which constructs a series of interpolants from the spurious behaviors of an abstract model and uses them to refine the model. One study [6] used interpolation to automatically construct a behavior model of a system from its goal model. The approach described therein updates a behavior model by using interpolants of counterexamples and goals. We believe we can use Craig interpolation in a similar way to systematically find CPs in future.

7 Conclusion and Future Work

We proposed a method to restructure the refinements of Event-B machines according to refactoring criterion in terms of the vocabulary of a new machine. Our method finds necessary variables and predicates from the original machines and helps to find complementary predicates to make the new machine consistent. It helps users to construct an abstraction of an existing machine that focuses on certain aspects of the original machine. By using our method, we split up refinements in large-scale Event-B machines and succeeded in constructing small and consistent machines. Moreover, our methods automatically extracted reusable parts of an existing model. We conclude that our method can help users to do refactoring of refinements in Event-B. Our primary future work will be a trial to enhance our method for finding complementary predicates to guarantee that generated intermediate machine is consistent with original machines.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)
2. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. Fundamenta Informaticae **77**(1–2), 1–28 (2007)
3. Butler, M.: Decomposition structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00255-7_2

4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). doi:10.1007/10722167_15
5. Correa, A., Werner, C., Barros, M.: An empirical study of the impact of OCL smells and refactorings on the understandability of OCL specifications. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 76–90. Springer, Heidelberg (2007). doi:10.1007/978-3-540-75209-7_6
6. Degiovanni, R., Alrajeh, D., Aguirre, N., Uchitel, S.: Automated goal operationalisation based on interpolation and SAT solving. In: Proceedings of the 36th International Conference on Software Engineering, pp. 129–139. ACM, New York (2014)
7. Gheyi, R., Borba, P.: Refactoring alloy specifications. Electron. Notes Theoret. Comput. Sci. **95**, 227–243 (2004)
8. Marković, S., Baar, T.: Refactoring OCL annotated UML class diagrams. In: Briand, L., Williams, C. (eds.) MODELS 2005. LNCS, vol. 3713, pp. 280–294. Springer, Heidelberg (2005). doi:10.1007/11557432_21
9. Matichuk, D., Murray, T., Andronick, J., Jeffery, R., Klein, G., Staples, M.: Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification. In: Proceedings of the 37th International Conference on Software Engineering. pp. 722–732. ACM, New York (2015)
10. McComb, T., Smith, G.: A minimal set of refactoring rules for object-Z. In: Barthe, G., Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 170–184. Springer, Heidelberg (2008). doi:10.1007/978-3-540-68863-1_11
11. Stepney, S., Polack, F., Toyn, I.: Refactoring in maintenance and development of Z specifications and proofs. ENTCS **70**(3), 50–69 (2002)
12. Tarasyuk, A., Pereverzeva, I., Troubitsyna, E., Latvala, T.: The formal derivation of mode logic for autonomous satellite flight formation. In: Koornneef, F., Gulijk, C. (eds.) SAFECOMP 2015. LNCS, vol. 9337, pp. 29–43. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24255-2_4
13. Whiteside, I.J.: Refactoring Proofs. Ph.D. thesis, The University of Edinburgh (2013)
14. Yaghoubi Shahir, H., Farahbod, R., Glässer, U.: Refactoring abstract state machine models. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 345–348. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30885-7_28