Computing and Informatics, Vol. 35, 2016, 411–440

A SET OF REFACTORING RULES FOR UML-B SPECIFICATIONS

Mehrnaz Najafi, Hassan Haghighi, Tahereh Zohdi Nasab

Faculty of Computer Science and Engineering
Shahid Beheshti University G.C.
Evin, Tehran, Iran
e-mail: M.Najafi@mail.sbu.ac.ir, {h_haghighi, t.zohdinasab}@sbu.ac.ir

Abstract. UML-B is a graphical formal modelling notation which is based on UML and relies on Event-B and its verification tools. In this paper, we propose annealing and introduce subtyping rules as well-known refactoring rules which can improve and assist the derivation of object-oriented design from an abstract specification written in UML-B. We prove that the proposed annealing rules are behavior preserving. We also demonstrate the applicability and effectiveness of our refactoring rules by applying them on two UML-B specifications.

Keywords: UML-B, refactoring, refinement, object-oriented design, annealing, introduce subtyping

1 INTRODUCTION

UML-B [1] is a graphical formal modelling notation which is based on Event-B [2] for its underlying semantics and is closely integrated with the Event-B verification tools [3]. The graphical modelling environment of UML-B allows development of a formal model through the use of visual objects at the abstraction level. The supporting tools ensure the model is verifiable and thus accurate [4]. On the other hand, UML-B supports object-oriented modelling concepts and has some resemblance with UML [5].

Object-orientation is perceived differently by different authors. The object paradigm, in the sense of object-oriented programming, provides principles like subtyping, inheritance and polymorphism [6]. Class instantiation, class inheritance, polymorphism, and generic class parameters (or templates) as four object-oriented

architectural constructs [7], that are almost universal, underpin the paradigm and provide modularity and reuse capabilities. Object-oriented formal specification languages such as Object-Z, UML-B and VDM++ share these core features with their programming language counterparts. However, the way they are utilized to capture requirements associated with a problem domain is often quite different from the way in which they are used to implement a specific solution to a problem. The result is that an object-oriented specification does not usually directly resemble, in a structural sense, the design of the desired implementation.

To bridge this gap between specification and implementation, specification refactoring rules have been proposed. These rules allow the structure of specification to be incrementally transformed to represent a design [8]. Refactoring is a technique which has long been used by programmers to improve the structure of their code once it got unreadable. More generally, refactoring is an important notion with regard to improving an existing code respecting flexibility, maintainability and reusability [9]. Although Fowler coined the word *refactoring* as a general term for frequently occurring clean-up operations on programs [10], he also proposes to use this term for the process of remodelling object-oriented software to improve an existing design whilst preserving its behavior [10]. Therefore, a recent trend is to apply the concepts of refactoring to higher levels of abstraction. Consequently, model refactoring is emerging as a desirable means to improve design models using behavior preserving transformations. Applying refactoring as early as possible during the software life cycle can improve the quality of design and reduce the complexity and cost in successive development phases [11].

Rules for refactoring have been well documented [12] and formalized [13] at the programming language level; however, we propose the use of more general rules, similar to refactoring rules presented by Fowler [10], to apply at the UML-B specification level in order to introduce a design rather than improve an existing one. More precisely, we are going to extend the process of modifying an UML-B specification to introduce design elements (this process has already been known as refinement in the literature) by the notion of refactoring, which Fowler describes as the application of simple rules for the improvement of the design of existing code [10].

In summary, as two main differences to Fowler's approach,

- our rules apply to abstract specifications and are utilized to moving from an abstract specification to an object-oriented design without the need to consider irrelevant details at the programming language level, and
- the motivation of our rules is primarily to introduce designs rather than improve existing ones [14].

Nevertheless, formal specification refactoring rules have one important similarity with the Fowler's: they must be equivalence transformations in terms of behavioral interpretation [14]. As Fowler puts it, all these refactorings should not change externally visible behavior of a program.

There are two major aspects a process of turning functional specifications into well-structured object-oriented software must encompass. First, at a high level, a purely functional specification must be reorganized and refined to create a reasonable object-oriented design. Second, at a lower level, the object-oriented design must be further refined to object-oriented code [7]. The focus of this paper is on the first aspect. Deriving object-oriented designs from object-oriented formal specifications using a sequence of step-wise structural transformations that are behavior preserving has clear benefits: constructed designs conform to the specification, and the design process is clearly documented as a sequence of tractable and repeatable steps [15]. Besides these benefits, we follow one more contribution: If the program code is refactored, however, the specifications need also to be changed. This can be facilitated by specification refactoring rules which allow such changes to be made systematically along with the changes to the code [16].

A number of approaches have been so far presented to derive designs from UML-B specifications. Said et al. [12] described class and state machine refinement in UML-B. Said [17] completed the previous approach [12] by introducing refinement of context diagrams and machines in package diagrams. Although rules given by Said [12, 17] are appropriate for deriving designs from UML-B specifications, they have not been introduced as refactoring rules. More precisely, rules given by Said [12, 17] do not support the decomposition of one class into classes in one step (called annealing) and the introduction of subtyping hierarchy in UML-B specifications while *annealing* and *introduce subtyping* are well-known refactoring rules in other object-oriented formal specification languages, such as Object-Z (as annealing and introduce inheritance rules) and VDM++ (as annealing and subtyping rules).

In this paper, we propose *annealing* and *introduce subtyping* rules which facilitate introducing structure in the process of moving from specification towards design. We have concentrated on these two particular refactoring rules because annealing adds structure (but no redundancy) to a specification by splitting a class into two classes (aiming for more fine-grained classes); also, introduce subtyping establishes a relationship between classes that have many features in common (with the aim of increasing reusability). Having these two rules in place, when combined with non-structural refactoring steps, our approach becomes a powerful method for incremental introduction of specification-based structural design in UML-B. Moreover, since the rules are behavior-preserving, this process ensures that any software design produced will be correct with respect to the original specification.

The paper is organized as follows: In Section 2, we review UML-B and related work done to introduce refactoring rules for object-oriented specification languages. Section 3 gives our refactoring rules for UML-B specifications. In Section 4, we prove that all four styles of annealing (i.e. *shared-event annealing, shared-attribute annealing, annealing as association and annealing as subtyping*) are behavior preserving. Section 5 is devoted to the evaluation of our method. The main part of this section shows the applicability of the proposed rules by employing them to introduce design elements into two abstract functional specifications written in UML-B. Section 6 concludes the paper and gives some directions for future work.

2 PRELIMINARIES

In this section, we first present a brief description of UML-B. Then, we review some related work performed to introduce refactoring rules for object-oriented specification languages.

2.1 UML-B

Four interlinked diagram types (package, context, class and state machine) are provided in UML-B. As in UML, package diagrams provide a structure of the model, but also cater for the concept of refinement. The diagram shows the *refines* relationships between *machines*, the *extends* relationships between *contexts* and the *Sees* relationships from *machines* to *contexts* [18].

UML-B mirrors the Event-B approach where static data (sets and constants) are modelled in a separate package called a *context*. The context diagram is similar to a class diagram but has only constant data (the static part of the model) represented by ClassTypes, Attributes and Associations. Axioms (given properties about the constants) and Theorems (assertions requiring proof) may be attached to the ClassTypes. ClassTypes define *carrier* sets or constant subsets of other ClassTypes.

The behavioral parts (variables and events) are modelled in a class diagram which is used to describe the *machine*. Classes represent subsets of the ClassTypes introduced in the context. Associations and attributes of the class are similar to those in the context but represent variables instead of constants. An attribute defines a data value of an instance of a class. An association is a special case of an attribute that defines a relationship between two classes [12]. Classes may own events that modify the variables. Event parameters can be added to an event providing local variables to be used in the transition's guards and actions. These parameters can be used to model inputs and outputs. Class events implicitly utilize a parameter to non-deterministically select the affected instance of the class [18].

At last, state machines may be used to model behavior. Transitions represent events with implicit behavior associated with the change of state. The event can only occur when the instance is in the source state and, when it fires, the instance changes to the target state. Hence state machines model class variables similar to attributes. Additional guards and actions can be attached to the transition in the property view [18].

2.2 Related Work

Said et al. [12] proposed rules for refinement of classes and state machines. In case of refinement of classes, a refined class may drop some of attributes of its abstract class, and a refined class may introduce new attributes (or variables). In case of refinement of state machines, the structure of a refined state machine is an elaboration of the structure of its abstraction machine either by replacing each

transition by one or more transitions or by elaborating an abstract state by a nested state machine. In addition, there are two methods for moving a class event in a refinement:

1. move to a refined class as a transition of a state machine and
2. move to a new class in a refinement either as a class event or a transition in a state machine.

Rules given by Said [17] support refinement of machines, classes, state machines and context diagrams. Refinement rules for machine include composition, decomposition with shared event approach, and machine refinement via refining its class diagram. Rules for class diagram include machine variable, event and invariant refinements (according to Event-B refinement rules for these notions), class and state machine refinements, adding new classes and dropping abstract classes. Class refinement rules include adding new attributes and associations to a refined class, dropping abstract attributes and associations, refinement of class events and invariants and also state machine refinement.

Rules for refinement of state machines [17] include state elaboration (an abstract state may be elaborated by nested state machines), transition elaboration (a transition may be replaced by one or more transitions), flattening state machine (removing nested state machines from the structure of a machine) and state grouping (adding new structure or state to a state machine and nesting some of its states in the new structure). Refinement rules for context diagrams [17] include adding new attributes and associations to extended ClassTypes and also adding new ClassTypes to the refinement.

There is some work which introduces refactoring rules in other specification languages. Lano and Goldsack [19, 20, 21] propose annealing (for decomposing classes) in VDM++. Moreover, they present formal definitions of refinement, subtyping, and subclassing. However, the problem of invariant distribution in the decomposition process is the focus of their work, and a complete approach to specification-based object-oriented design has not been developed.

McComb and Smith [8] present a minimal set of refactoring rules, namely *introduce generic parameters*, *introduce inheritance* and *introduce polymorphism* and show that the combination of these three rules with compositional class refinement [22] and annealing [7, 15] yields a complete method for design in Object-Z. The annealing rule allows for the decomposition of one class into two, effectively partitioning its data and functionality. They propose the annealing rule by considering one class, called framing class, holding a reference to an object of the other class, called component class. It is the responsibility of the framing class to invoke operations upon and manage the state of the component class [7].

Introduce inheritance rule offers the means to build an inheritance hierarchy from existing classes [8]. This rule is most effectively applied to link together classes that contain common features in order to maximize the potential for reuse, but the classes need not share any features at all [14]. Finally, Lu and Zhu [23] propose

a set of more fine-grained refactoring rules than what was presented by McComb and Smith [7, 8, 14, 22].

3 REFACTORING RULES FOR UML-B

In this section, our refactoring rules for UML-B specifications are introduced. At first, *annealing* is proposed in four styles, and then, the *introduce subtyping* rule is presented to reuse data constructs and operations in classes.

3.1 Annealing

Annealing rule allows decomposition of one class into two or more classes by partitioning its data and functionality [7]. It is similar in intention to the Fowler's *Extract Class* refactoring [10], which performs the same function at the programming language level. With Fowler's *extract class* refactoring, when we have one class doing a function that should be done by two, we should create a new class and move the relevant fields and methods from the old class into the new one [10].

Investigating the related work, we have found that there are two ways to propose annealing rule: by using either multiple inheritance or association. In the *annealing as association* style, one of the new classes must hold a reference to the other one (or other ones) while in the *annealing as multiple inheritance* style, one of the new classes must inherit the other one (or other ones).

Also, there are two ways to decompose a machine in UML-B similar to what we have in Event-B [24]: shared variable decomposition (called A-style decomposition) and shared event decomposition (called B-style decomposition). In A-style decomposition, a machine is decomposed into arbitrary number of sub-machines based on a shared variable in that machine. Those variables accessed by events of distinct sub-machines are called shared variables. In B-style decomposition, a machine is decomposed into arbitrary number of sub-machines based on a shared event in that machine. Those events shared in two or more sub-machines are called shared events. In order to decompose machines in UML-B, one should consider those state machines which do not exist in machines in Event-B. Thus, state machine refinement techniques, i.e., state grouping and flattening, are defined in UML-B in order to define machine decomposition [17].

Since a class in UML-B has events and attributes similar to those in a machine, in the following subsections, we present annealing in four styles, called *annealing as association*, *annealing as subtyping*, *shared-event annealing* and *shared-attribute annealing*.

3.1.1 Annealing as Association

As we said earlier, in *annealing as association*, one of the new classes must hold a reference to the other one (or other ones); hence, we must use the notion of object in the class diagram. Using this style, a class will be decomposed into two or more

new classes. Figure 1 shows annealing of class A to two new classes B_1 and B_2. In this figure, class A is decomposed into classes B_1 and B_2 such that B_1 holds a reference to (is associated with) B_2, and hence, B_1 has an attribute which is an object of class B_2. Also, attribute *attribute* of class A is considered as an attribute of class B_2. Events *event1* and *event2* are partitioned over classes B_1 and B_2.

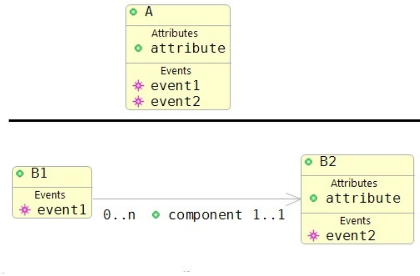

Figure 1. Annealing as association of class A to classes B_1 and B_2

The following sequence must be followed for annealing class C to classes $C_1, \ldots, C_n$ with association style; suppose that class C_n holds references to classes $C_1, \ldots, C_{n-1}$:

1. A new machine (refinement machine) which refines the existing machine (abstract machine), whose class diagram contains class C, is introduced in the package diagram.
2. Classes $C_1, \ldots, C_n$ are introduced in the class diagram of the refinement machine defined in the previous step.
3. Attributes in the form of objects of classes $C_1, \ldots, C_{n-1}$ are considered in class C_n so that they are equivalent to associations from C_n to $C_1, \ldots, C_{n-1}$. Also, surjective, injective, total and functional values of these associations must be determined according to semantics of $C_1, \ldots, C_n$.
4. The events of class C are first partitioned over $C_1, \ldots, C_{n-1}$ in the class diagram based on the semantics which are considered for $C_1, \ldots, C_{n-1}$. Although all of the events of C are considered in class C_n, it may be needed to change the specification of these events to *promotion* in order to make them equivalent with their counterparts in C. For instance, if event e in C is considered in C_1 based on event partitioning and attribute a whose type is C_1 is considered in C_n according to step 3, event e should be also considered in C and its body is in the form of promotion (i.e., a.e).
5. The attributes of class C are partitioned over $C_1, \ldots, C_n$ in the class diagram based on event partitioning and also semantics which are considered for $C_1, \ldots, C_n$. It should be noted that it is possible to remove some of the attributes of C because attribute removing from a class is a valid refinement in UML-B.

6. The invariants and state machines of class C are distributed according to events and attributes partitioning in the class diagram. State machine grouping and flattening techniques must be used here in order to decompose state machines among $C_1, \ldots, C_n$. For attributes a1, …, am, an invariant based on P(a1, …, am) is copied to Ci (i: 1..n) if and only if Ci contains all a1, …, am.
7. The associations of class C are distributed over $C_1, \ldots, C_n$ based on the semantics of C, $C_1, \ldots, C_n$.
8. Predicates that say each attribute or association of class C is considered in one of classes $C_1, \ldots, C_n$ are regarded as gluing invariants in the refinement machine.
9. A new context is introduced in the package diagram that *extends* the context seen by the abstract machine. The refinement machine *sees* this new context. The ClassType of C is considered as ClassType of class C_n. Also, a new ClassType is considered for each class Ci (i: $1..n-1$) in the context diagram of this new context.

3.1.2 Annealing as Subtyping

We consider *annealing as subtyping* in order to decompose a class in the class diagram into two or more classes where one of them is the super type of the rest. The following sequence must be followed for annealing class C to classes $C_1, \ldots, C_n$ with subtyping style; suppose that class C_n is the subtype of classes $C_1, \ldots, C_{n-1}$:

1. A new machine (refinement machine) which refines the existing machine (abstract machine), the class diagram of which contains class C, is introduced in the package diagram.
2. Classes $C_1, \ldots, C_n$ are introduced in the class diagram of the refinement machine defined in the previous step.
3. SuperType connections from C_n to classes $C_1, \ldots, C_{n-1}$ are considered in the class diagram.
4. Events of class C are first partitioned over $C_1, \ldots, C_n$ in the class diagram based on the semantics of $C_1, \ldots, C_n$. Event decomposition may be needed; however, it should be a valid refinement.
5. Attributes of class C are partitioned over $C_1, \ldots, C_n$ in the class diagram based on event partitioning and the semantics of $C_1, \ldots, C_n$.
6. Invariants and state machines of class C are distributed according to events and attributes partitioning in the class diagram. Invariant distribution of annealing as subtyping is the same as that of annealing as association.
7. The associations of class C are distributed over $C_1, \ldots, C_n$ based on the semantics of C, $C_1, \ldots, C_n$.
8. Gluing invariants are devised in the same way as those of annealing as association.

9. A new context is added into the package diagram in the same way as that of annealing as association.

3.1.3 Shared-Attribute Annealing

For better understanding of *annealing with shared-attribute style*, we first give a sample of this style of annealing. In Figure 2, class A has events e1 to e4 and attributes a1 to a3. The solid lines show which attributes are used by an event. Class A is decomposed into two new classes B_1 and B_2 based on shared attribute a2. Attributes like a2 that are accessed by events of distinct sub-classes (i.e. B_1 and B_2) are called shared attributes. Attributes a2_1 and a2_2 have the same properties as attribute a2. In other words, we rename attribute a2 to a2_1 and a2_2 because we cannot have attributes with the same name in different classes in UML-B tool [19]. Events e2 and e3 must be distributed over B_1 and B_2. Thus, we consider part of e2 which relates to a2 as event e2_1 of B_2 and also, part of e3 which relates to a2 as event e3_1 of B_1. The following sequence corresponds to *shared-attribute annealing* of class C to classes $C_1, \ldots, C_n$:

1. A new machine (refinement machine) which refines the existing machine (abstract machine), the class diagram of which contains class C, is introduced in the package diagram.
2. Classes $C_1, \ldots, C_n$ are introduced in the class diagram of the refinement machine defined in the previous step.
3. Events of class C are partitioned over $C_1, \ldots, C_n$.
4. Attributes of C are distributed according to event partitioning. A shared attribute sa is renamed to sa_i in each class Ci (i: 1..n) which should have sa.
5. Invariants of C are distributed over $C_1, \ldots, C_n$. Invariant distribution of shared attribute annealing is the same as that of annealing as association. It should be noted that shared attribute renaming must be performed.
6. State machines of C are distributed according to event partitioning and also by applying state grouping and flattening techniques.
7. Parts of events are built in $C_1, \ldots, C_n$. If e1 is an event of Ci (i: 1..n) that modifies a shared attribute sa_i, then an event is built from e1 in each sub-class Cj (j: 1..n) where sa_j is accessed (sa_i and sa_j have been obtained from a same attribute sa).
8. Predicates that say each attribute or association of class C, except the shared attribute, is considered in one of classes $C_1, \ldots, C_n$ are regarded as gluing invariants in the refinement machine. For shared attribute sa, predicates that say sa_i in each class Ci (i: 1..n) are identical to sa in class C are also considered as gluing invariants in the refinement machine.
9. A new context is introduced in the package diagram that *extends* context seen by the abstract machine. The context diagram of the new context contains a new

ClassType for each class Ci (i: 1..n). Also, the refinement machine *sees* the new context.

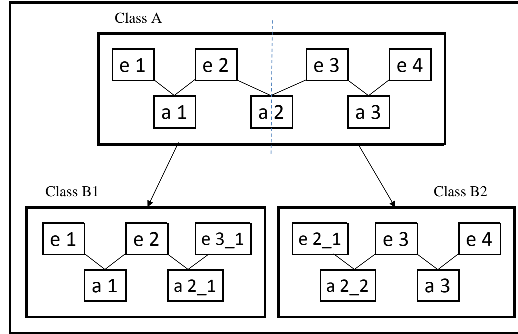

Figure 2. Shared attribute annealing of class A to classes B_1 and B_2

3.1.4 Shared-Event Annealing

For better understanding of *annealing with shared-event style*, we first give a sample of this style of annealing: Figure 3 shows annealing of class A based on shared event e2. In order to perform this type of annealing, event e2 must be decomposed into e2_1 and e2_2. Event e2_1 is part of event e2 that relates to a1, and event e2_2 is part of event e2 which relates to a2. The following sequence must be regarded for *shared-event annealing* of class C to classes $C_1, \ldots, C_n$:

1. A new machine (refinement machine) which refines the existing machine (abstract machine), the class diagram of which contains class C, is introduced in the package diagram.
2. Classes $C_1, \ldots, C_n$ are introduced in the class diagram of the refinement machine defined in the previous step.
3. Attributes of C are partitioned over $C_1, \ldots, C_n$.
4. Events of C are distributed over $C_1, \ldots, C_n$ according to attribute partitioning. Also, the shared event must be split in order to be regarded in classes which should have this event.
5. Invariants and state machines of C are distributed similar to what was done in annealing as association.
6. Predicates that say each attribute or association of class C is considered in one of classes $C_1, \ldots, C_n$ are regarded as gluing invariants in the refinement machine.

7. A new context is added into the package diagram in the same way as that of shared-attribute annealing.

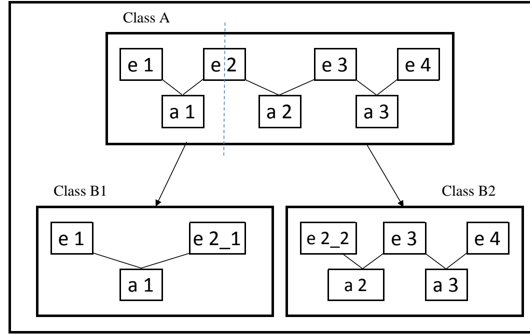

Figure 3. Shared event annealing of class A to classes B_1 and B_2

3.2 Introduce Subtyping

We propose *introduce subtyping* to reuse data constructs and operations in classes. This rule offers a means to build a subtyping hierarchy from existing classes. It is similar to the Fowler's *Extract Hierarchy* refactoring rule, which creates a hierarchy of classes in which each subclass represents a special case [10]. *Introduce subtyping* rule creates a subtyping relationship between any two classes in the specification, as long as the addition of the relationship does not result in a circular dependency. This rule not only adds subtyping relationship, but also hides (using existential quantifier) every feature of the supertype in order to preserve the meaning of the specification.

In order to introduce subtyping relationship between two existing classes C_1 and C_2 when C_2 is semantically a subtype of C_1 (by semantically, we mean that the subtyping relationship does not exist in the diagram; however, the specifier can infer this relationship between C_1 and C_2 by referring to their meanings, and thus, he/she can create subtyping relationship between the classes using the *introduce subtyping* rule), we must first define a new machine (refinement machine) which refines the existing machine (abstract machine) the class diagram of which contains classes C_1 and C_2. Classes C_1 and C_2 are introduced in the class diagram of the refinement machine as refined classes which have the same attributes, associations, events and invariants as those existing in the class diagram of the abstract machine. Next, we draw a supertype connection from C_2 to C_1 and then hide all inherited features of C_1 in C_2. Also, the refinement machine *sees* the context seen by the abstract machine.

To reuse features of the supertype, the designer must make further changes to the subtype to unhide, and perhaps rename, the features inherited from the supertype.

4 PROOF SKETCH

The following theorem demonstrates that the annealing rule is behavior preserving.

Theorem 1 (Behavior Preserving)**.** For a given UML-B class diagram *umlbs*, let *umlbd* be a UML-B class diagram obtained after applying annealing on *umlbs*. Then, we have U2B (*umlbs*) *REF* U2B (*umlbd*), where U2B (s) denotes the Event-B specification generated from UML-B specification s using U2B [25]. In addition, *REF* is used as the refinement notation.

To prove this theorem, we use the notion of refinement in Event-B [26] because proof obligations for UML-B specifications are done using Event-B prover in Rodin. Also, automatic conversion of UML-B specifications to Event-B specifications is available considering U2B. Therefore, we first convert UML-B class diagrams before/after applying annealing to Event-B specifications using U2B translation rules. After checking consistency of machines [27] in the Event-B specifications, we use Event-B refinement rules to prove that the Event-B specification, which is the translation of the class diagram after applying the refactoring rule, is a valid refinement of the Event-B specification, which is the translation of the class diagram before applying the refactoring rule.

As a limitation of our work, we do not consider state machines in our proofs since we have only concentrated on class diagrams and context diagrams in this paper. In all of the class diagrams, we use the following conventions:

1. C_CT denotes the ClassType of class C. Similarly, Ci_CT denotes the ClassType of class Ci.
2. C_Attributes, C_Events and C_Invariants denote the attributes, events and invariants of class C, respectively. Similarly, Ci_Attributes, Ci_Events and Ci_Invariants denote the attributes, events and invariants of class Ci, respectively.
3. Ci_Associations denotes the associations of class Ci.
4. **type** (Attribute a) returns the type of attribute a, and **GetAssEnd** (Attribute a) returns the name of the class at the end of association a.

For simplicity, we assume that the class diagram before applying this rule is in a machine (called M) that has only one class named C, and also it has only one ClassType named C_CT; considering other classes in the class diagram does not affect our proof. Figure 4 shows the mentioned class diagram. The next subsection presents the translation of UML-B class diagrams before/after applying annealing rules to Event-B specifications using U2B translation rules.

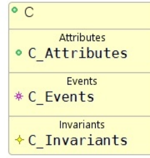

Figure 4. The class diagram considered before applying annealing

4.1 Translation to Event-B

Figure 5 shows Machine M obtained from the translation of the class diagram in Figure 4 using U2B translation rules.

```
MACHINE M
SEES M_implicitContext
VARIABLES
    C
    C_Attributes
INVARIANTS
    C ∈ ℙ(C_CT) ∧ (∀ a ⦂ Attribute | a ∈ C_Attributes ∘ a ∈ C→type(a))
    C_invariants
EVENTS
    INITIALISATION
    C_Events
END
```

Figure 5. Machine M

Now, suppose that we want to anneal class C represented above to classes C_1, ..., C_n using annealing as association. Also, suppose that C_n holds references to C_1, ..., C_{n-1}. If *CD* is the class diagram obtained after applying this style of annealing, Machine M_{AA} in Figure 6 is resulted from the translation of *CD* using U2B translation rules.

Similarly, suppose that we want to anneal class C to classes C_1, ..., C_n (C_n is a subtype of classes C_1, ..., C_{n-1}) using *annealing as subtyping*. Machine M_{AS} in Figure 7 is obtained from the translation of the class diagram after *annealing as subtyping* using U2B translation rules. Finally, Machines M_{SAA} and M_{SEA} in Figures 8 and 9 are resulted from the translation of the class diagram after *shared-attribute annealing* and *shared-event annealing* using U2B translation rules.

MACHINE M_{AA}

SEES M_{AA}_implicitContext

VARIABLES

∀i ⦂ 1 · ·n ∘ Ci

∀i ⦂ 1 · ·n ∘ Ci_Attributes

INVARIANTS

∀i ⦂ 1 · ·n ∘ Ci ∈ ℙ(Ci_CT)

∀i ⦂ 1 · ·n , a ⦂ Attribute | a ∈ Ci_Attributes ∧ a ∉ Ci_Associations ∘ a ∈ Ci → type(a)

(∀i ⦂ 1 · ·n ∘ Ci_invariants) ∧ (∀i ⦂ 1 · ·n−1 , a ⦂ Attribute | a ∈ Cn ↔ Ci)

∀a ⦂ Attribute | a ∈ C_Attributes ∘ (∃i ⦂ 1 · ·n ∘ a ∈ Ci_Attributes)

EVENTS

INITIALISATION

∀i ⦂ 1 · ·n ∘ Ci_Events

END

Figure 6. Machine M_{AA}

4.2 Proof in Event-B

In this subsection, we prove that the Event-B specification corresponding to the class diagram after applying the refactoring rule is a valid refinement of the Event-B specification corresponding to the class diagram before applying the refactoring rule.

MACHINE M_{AS}

SEES M_{AS}_implicitContext

VARIABLES

∀i ⦂ 1 · ·n ∘ Ci

∀i ⦂ 1 · ·n ∘ Ci_Attributes

INVARIANTS

∀i ⦂ 1 · ·n ∘ Ci ∈ ℙ(Ci_CT)

∀i ⦂ 1 · ·n , a ⦂ Attribute | a ∈ Ci_Attributes ∘ a ∈ Ci → type(a)

(∀i ⦂ 1 · ·n ∘ Ci_invariants) ∧ (∀i ⦂ 1 · ·n−1 , a ⦂ Attribute | a ∈ ℙ(Ci))

∀a ⦂ Attribute | a ∈ C_Attributes ∘ (∃i ⦂ 1 · ·n ∘ a ∈ Ci_Attributes)

EVENTS

INITIALISATION

∀i ⦂ 1 · ·n ∘ Ci_Events

END

Figure 7. Machine M_{AS}

MACHINE M_{SAA}

SEES M_{SAA}_implicitContext

VARIABLES

∀i ⸬ 1‥n ∘ Ci

∀i ⸬ 1‥n ∘ Ci_Attributes

INVARIANTS

∀i ⸬ 1‥n ∘ Ci ∈ ℙ(Ci_CT)

∀i ⸬ 1‥n , a ⸬ Attribute | a ∈ Ci_Attributes ∘ a ∈ Ci → type(a)

∀i ⸬ 1‥n ∘ Ci_invariants

∀ a, sa ⸬ Attribute | sa ∈ C_Attributes, a ∈ C_Attributes – {sa} ∘ (∃i ⸬ 1‥n ∘ a ∈ Ci_Attributes) ∧ (∃i ⸬ 1‥n ∘ sa_i ⸬ Attribute, sa_i ∈ Ci_Attributes | sa_i = sa)

EVENTS

INITIALISATION

∀i ⸬ 1‥n ∘ Ci_Events

END

Figure 8. Machine M_{SAA}

4.2.1 Proof for Annealing as Association

Lemmas 1 to 3 below demonstrate that M_{AA} is a valid refinement of M.

Lemma 1 (Attribute Refinement)**.** Attributes in M_{AA} are valid refinements of attributes in M.

MACHINE M_{SEA}

SEES M_{SEA}_implicitContext

VARIABLES

∀i ⸬ 1‥n ∘ Ci

∀i ⸬ 1‥n ∘ Ci_Attributes

INVARIANTS

∀i ⸬ 1‥n ∘ Ci ∈ ℙ(Ci_CT)

∀i ⸬ 1‥n , a ⸬ Attribute | a ∈ Ci_Attributes ∘ a ∈ Ci → type(a)

∀i ⸬ 1‥n ∘ Ci_invariants

∀a ⸬ Attribute | a ∈ C_Attributes ∘ (∃i ⸬ 1‥n ∘ a ∈ Ci_Attributes)

EVENTS

INITIALISATION

∀i ⸬ 1‥n ∘ Ci_Events

END

Figure 9. Machine M_{SEA}

Proof. M_{AA} dropped abstract variable C of machine M, kept C_Attributes in C_1, ..., C_n, and introduced new variables. All of the mentioned changes are valid refinements in Event-B; it should be noted that new variables (Ci_Attributes except that associations from C_n to Ci (i: $1..n-1$)) refine abstract variables (C_Attributes). □

Lemma 2 (Event Refinement)**.** Events in M_{AA} are valid refinements of events in M.

Proof. As we said in the steps of annealing as association, C_n has all events of C but in different specifications (i.e. promotion); therefore we can conclude that these events refine corresponding events in machine M. On the other hand, all of the events of C are distributed in C_1, ..., C_{n-1}; hence, these events refine their corresponding events in machine M. So, action simulation proof obligations of Event-B [27] are proved. Suppose that event e exists in class C. So, it exists in machine M, class C_n and also one of the classes C_1, ..., C_{n-1}. Using U2B translation rules to translate the class diagram obtained after applying *annealing as association* results in one event e in machine M_{AA} which is equivalent to the conjunction of e in class C_n and the instance of the same event existing in one of classes C_1, ..., C_{n-1}. Due to the fact that the mentioned two events refine each other, we conclude that event e in machine M_{AA} refines event e in machine M, too. Thus, if correct initializations of variables are performed in INITIALIZATION, we can conclude from all of the above statements that these refinements are valid in Event-B. □

Lemma 3 (Invariant Preservation)**.** Invariants are preserved and properly devised to relate M_{AA} and M.

Proof. Events in machine M_{AA} preserve invariants of C_1, ..., C_n which are invariants of C itself. Also, since new variables (Ci_Attributes except that associations from C_n to Ci (i: $1..n-1$)) are refinements of abstract variables (C_Attributes), and events in M hold invariants related to abstract variables, we conclude that the events in M_{AA} hold invariants related to attributes of classes as well. Furthermore, new invariants are introduced due to the new ClassTypes and associations: considering the steps of annealing as association, attributes of each association (i.e. surjective, injective, total and functional) are determined based on C, C_1, ..., C_n semantics. We assume that correct initialization of these variables is performed, and also correct ClassTypes are considered for classes. Thus, the events hold these new invariants. Finally, we provided gluing invariants to relate M_{AA} and M in the steps of annealing as association. More precisely, predicates that say each attribute of class C is considered in one of classes C_1, ..., C_n are regarded as gluing invariants; see last line of invariants of M_{AA}. □

4.2.2 Proof for Annealing as Subtyping

Lemmas 4 to 6 below demonstrate that M_{AS} is a valid refinement of M.

Lemma 4 (Attribute Refinement)**.** Attributes in M_{AS} are valid refinements of attributes in *M*.

Proof. M_{AS} dropped abstract variable C of machine *M*, kept C_Attributes in C_1, ..., C_n and introduced new variables. These changes are valid refinements in Event-B; it should be noted that new variables (C*i*_Attributes) refine abstract variables (C_Attributes). □

Lemma 5 (Event Refinement)**.** Events of M_{AS} are valid refinements of those in *M*.

Proof. Events of M_{AS} are the same as events of *M* considering the steps of annealing as subtyping. So, action simulation proof obligations of Event-B [16] are proved. □

Lemma 6 (Invariant Preservation)**.** Invariants are preserved and properly devised to relate M_{AS} and *M*.

Proof. Events in M_{AS} preserve invariants of C_1, ..., C_n which are invariants of C itself considering that events of M_{AS} are the same as events of *M*. Furthermore, some new invariants are introduced in M_{AS}. Assuming that correct initializations of variables are performed in INITIALISATION, and also correct ClassTypes are considered for classes, we can conclude that the new invariants are preserved in events of M_{AS}. Finally, we provided gluing invariants to relate M_{AS} and *M* in the steps of annealing as sub-typing. More precisely, predicates that say each attribute of class C is considered in one of classes C_1, ..., C_n are regarded as gluing invariants; see last line of invariants of M_{AS}. □

4.2.3 Proof for Shared-Attribute Annealing

Lemmas 7 to 9 below demonstrate that M_{SAA} (as the machine after applying the shared-attribute annealing rule) is a valid refinement of *M* (as the original machine).

Lemma 7 (Attribute Refinement)**.** Attributes in M_{SAA} are valid refinements of attributes in *M*.

Proof. M_{SAA} dropped abstract variable C of machine *M* and kept C_Attributes in C_1, ..., C_n, in a way that the shared attribute of the original class is copied and renamed in the decomposed classes. Other attributes of *M* are distributed in the decomposed classes without any changes. All of the mentioned changes are valid refinements in Event-B; it should be noted that new variables (C*i*_Attributes) refine abstract variables (C_Attributes). □

Lemma 8 (Event Refinement)**.** Events of M_{SAA} are valid refinements of those in *M*.

Proof. Events of M_{SAA} are partitioned over C_1, ..., C_n. For the event that modifies the shared attribute, the parallel composition of events obtained from decomposition of the corresponding original event refines the original event. According to the decomposition rule proposed by Abrial in [28], this is a valid refinement in Event-B. □

Lemma 9 (Invariant Preservation)**.** Invariants are preserved and properly devised to relate M_{SAA} and M.

Proof. Since we distribute invariants in the same manner as in the annealing as association rule, the argument for invariant preservation is straightforward (we have already proved that annealing as association is behavior preserving). Events in M_{SAA} preserve invariants of $C_1, \ldots, C_n$ which are invariants of C itself. Also, since new variables (Ci_Attributes) are refinements of abstract variables (C_Attributes), and events in M hold invariants related to the abstract variables, we conclude that the events in M_{SAA} hold invariants related to attributes of classes as well. Furthermore, new invariants are introduced due to the new ClassTypes for each class Ci (i: 1..n): considering the steps of shared-attribute annealing, variables of each class are distributed based on event partitioning. We assume that correct initialization of these variables is performed, and also, correct ClassTypes are considered for classes. Thus, the events hold these new invariants. Finally, we provided gluing invariants to relate M_{SAA} and M in the steps of shared-attribute annealing. More precisely, predicates that say each attribute or association of class C is considered in one of classes $C_1, \ldots, C_n$, and also for the shared attribute sa, predicates that say sa_i in each class Ci (i: 1..n) are identical to sa in class C are regarded as gluing invariants in the refinement machine. □

4.2.4 Proof for Shared-Event Annealing

Lemmas 10 to 12 below demonstrate that M_{SEA} (as the machine after applying the shared-event annealing rule) is a valid refinement of M (as the original machine).

Lemma 10 (Attribute Refinement)**.** Attributes in M_{SEA} are valid refinements of attributes in M.

Proof. M_{SEA} dropped abstract variable C of machine M, kept C_Attributes in $C_1, \ldots, C_n$, and introduced some new variables. These changes are valid refinements in Event-B; it should be noted that new variables (Ci_Attributes) refine abstract variables (C_Attributes). In other words, attributes do not change; they are just distributed in classes $C_1, \ldots, C_n$. □

Lemma 11 (Event Refinement)**.** Events of M_{SEA} are valid refinements of those in M.

Proof. All of the events of C except the shared one are partitioned in $C_1, \ldots, C_n$; hence, these events refine their corresponding events in machine M. The shared event is decomposed into two events, and its partition depends on the partition of the variables. When the decomposition occurs, parameters are shared between the decomposed events. The guard of a decomposed event inherits the guard on the composed event according to the variable partition [29]. According to the shared-event decomposition rule proposed by Silva and Butler [29], and since we decompose an event based on attributes used in that event, we have a valid refinement in Event-B. □

Lemma 12 (Invariant Preservation)**.** Invariants are preserved and properly devised to relate M_{SEA} and M.

Proof. Since we distribute invariants in the same manner as in the annealing as association rule, the argument for invariant preservation is straightforward. Considering that events of M_{SEA} are the same as events of M, except the shared event which is decomposed, we can conclude that events in M_{SEA} preserve invariants of $C_1, \ldots, C_n$ which are invariants of C itself. Furthermore, for new invariants introduced in M_{SEA}, assuming that correct initializations of variables are performed in INITIALISATION, and also, correct ClassTypes are considered for classes, it can be asserted that the new invariants are preserved in events of M_{SEA}. Finally, we provided gluing invariants to relate M_{SEA} and M in the steps of shared-event annealing. □

5 EVALUATION

In the first two subsections, we apply our refactoring rules and rules given by Said [12, 17] to introduce design elements into two abstract functional specifications written in UML-B. We have used two different specifications in order to show the applicability of all rules proposed in the paper. In the third subsection, we compare our work with the most related work.

5.1 Case Study 1: Mass Transit Railway System

We evaluate our refactoring rules through applying them on an adapted study of the *Mass Transit Railway System* [30]: we do not consider context diagrams and state machines in this case study. The mass transit railway network consists of a set of stations. To travel on the network, a passenger inserts a ticket into a station entrance barrier and, provided that the ticket is valid for entry, access to the network is given. The ticket is returned to the passenger after passing through the barrier. When the destination station is reached, the passenger inserts the same ticket into the station exit barrier, and at this time, provided that the ticket is valid for the trip just completed, exit is permitted. Again, after passing through the barrier, the ticket is returned to the passenger. Also, passengers can purchase several types of tickets. Figure 10 shows the abstract functional specification of this case study written in UML-B.

The mass transit railway specified in Figure 10 consists of a set of stations and a set of tickets. The event *startTrip* models the effect of inserting a ticket (t) into the entrance barrier of a station (s) at the system level. Also, the operation *finishTrip* models the effect of inserting a ticket (t) into the exit barrier of a station (s). Finally, the operation *reIssueTicket* enables a ticket to be reissued.

Class *Ticket* has two attributes: *value* denotes the current value of the ticket, and *entryPoint* is a set that is either empty or contains the identity of one station which the ticket is inserted to its barrier. Also, this class has three events: *enterStation* models what happens when a ticket is inserted into a station entrance barrier

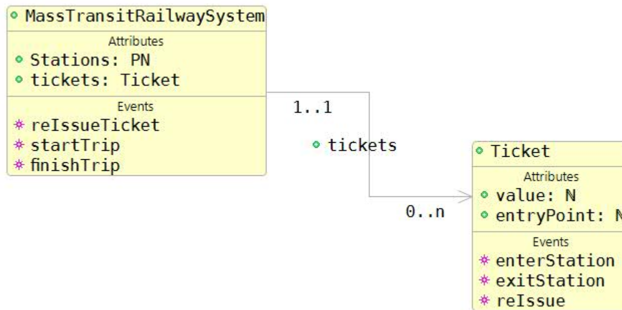

Figure 10. Mass transit railway system specification

and given access to the network, *reissue* enables a ticket to be reissued according to the ticket type (single-trip ticket, multi-trip ticket and season ticket which are determined based on *value*). Event *exitStation* models what happens when a ticket is inserted into a station exit barrier.

This specification is adequate to describe the core functionality of the system, but does not represent an appropriate object-oriented design. A good object-oriented design, and even, a better structured specification would identify stations as separate objects in the system and encapsulate their functionality. Also, one could encapsulate common features of ticket types via a specific class, and thus, reduce the clear redundancy. In the following subsections, we achieve this goal via the step-wise application of behavior preserving rules introduced in this section.

5.1.1 The First Two Steps: Annealing as Association, Dropping Attribute (stations) and Adding New Attribute (stationID)

We apply annealing as association to the class diagram in Figure 10. This rule decomposes class *MassTransitRailwaySystem* into classes *MassTransitRailwaySystem* and *Station*; see Figure 11. Class *Station* models the stations of the mass transit railway system, and class *MassTransitRailwaySystem* models entering and exiting of passengers from stations. Also, we drop attribute *stations: P N* and add new attribute *stationID: N* to class *Station*. The new attribute models the station number.

Now, we show changes needed to be applied to event *startTrip* after applying the mentioned rules. Specification of *startTrip* defined in Figure 11 is as follows:

startTrip
status ordinary
any t, s
where t **in** tickets
then t.enterStation (s)
end

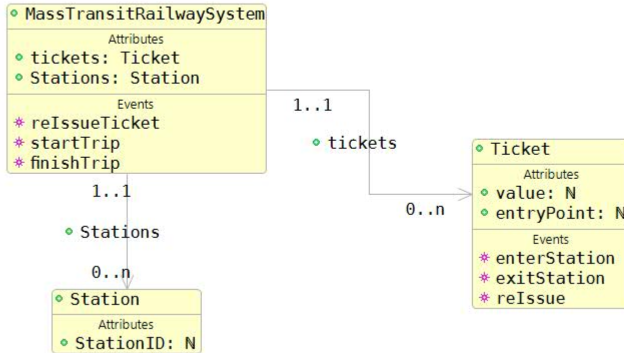

Figure 11. The first two steps of the mass transit railway system design process

Specification of event *startTrip* after applying the mentioned rules becomes as follows (see item 2 in the sequence of *annealing as association* in Subsection 3.1.1):

startTrip
status ordinary
any t, s
where t **in** tickets
then t.enterStation (s.stationID)
end

5.1.2 The Third Step: Annealing as Subtyping

Figure 12 shows the class diagram after applying *annealing as subtyping* to the class diagram in Figure 11. This rule decomposes class *Ticket* into classes *BaseTicket* and *SingleTripTicket*. Class *BaseTicket* captures the features common to tickets, and class *SingleTripTicket* permits only a single trip and only on the day the ticket is purchased. We decompose event *reIssue* into events *reIssueST* and *reIssue* and also event *exitStation* into events *exitStationST* and *exitStation*. In fact, events *reIssueST* and *exitStationST* model reissuing the ticket and exiting the station for this type of ticket, respectively; these decompositions are valid refinements.

5.1.3 The Final Two Steps: Annealing as Subtyping

We apply *annealing as subtyping* to the class diagram in Figure 12 in order to decompose class *BaseTicket* into classes *BaseTicket* and *MultiTripTicket*. Class *BaseTicket* still captures the features common to tickets, and class *MultiTripTicket* models validity for any number of trips provided that the current value of the ticket remains greater than zero. Also, we decompose event *reIssue* into events *reIssueMT* and *reIssue*. Similarly, event *exitStation* is decomposed into events *exitStationMT*

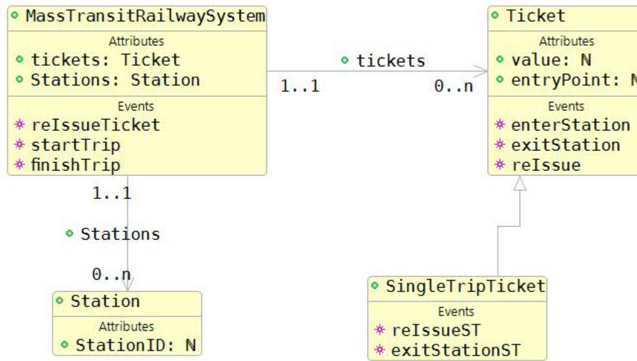

Figure 12. The third step of the mass transit railway system design process

and *exitStation*. Events *reIssueMT* and *exitStationMT* model reissuing the ticket and exiting the station for this type of ticket, respectively.

Next, we apply *annealing as subtyping* to the resulting class diagram again. This rule decomposes class *BaseTicket* into classes *BaseTicket* and *SeasonTicket*. Class *BaseTicket* captures the features common to tickets, and class *SeasonTicket* models validity for either a week, a month or a year; see Figure 13.

The resulting specification represents an improvement, in terms of design, over the original specification shown in Figure 10. This is evidenced by the reduction of redundancy through encapsulating the concept of a station inside a separate class. In addition, the design decision to introduce *BaseTicket* to capture the features common to the tickets is a step toward subtyping concept.

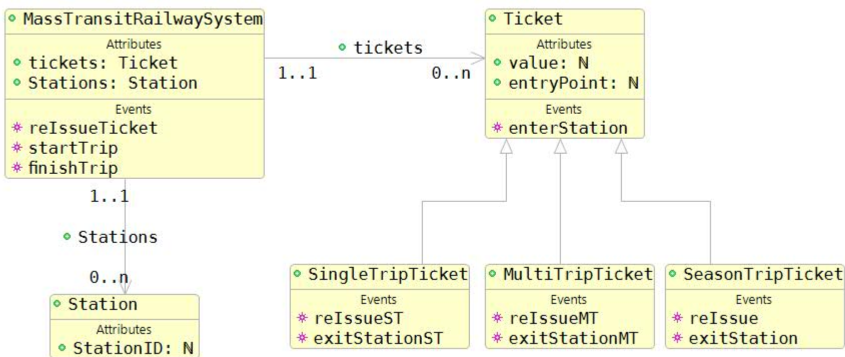

Figure 13. The final two steps of the mass transit railway system design process

5.2 Case Study 2: Daycare Center

Figure 14 shows the abstract functional specification of the daycare center case study modelled in UML-B as a class diagram. Classes *DaycareContext* and *Examiner* model examining kids in a daycare center. Examiners (Doctors and Nurses) and kids are modelled using natural numbers. A set of kids is modelled using the power set notation. For each examining activity, three major sub activities should be performed as follows:

1. Giving the kids exam (ExamineKid)
2. Giving the parent's billing information (GenerateBill)
3. Sending the report to the kid's parents (CreateReport).

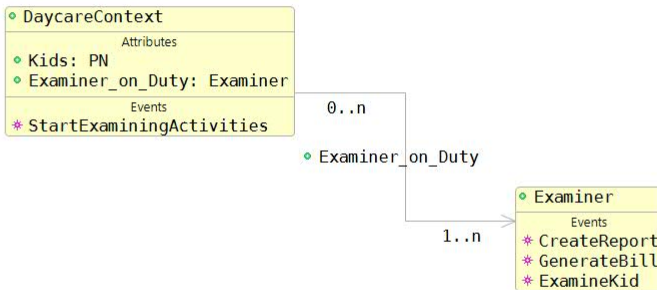

Figure 14. Daycare center specification

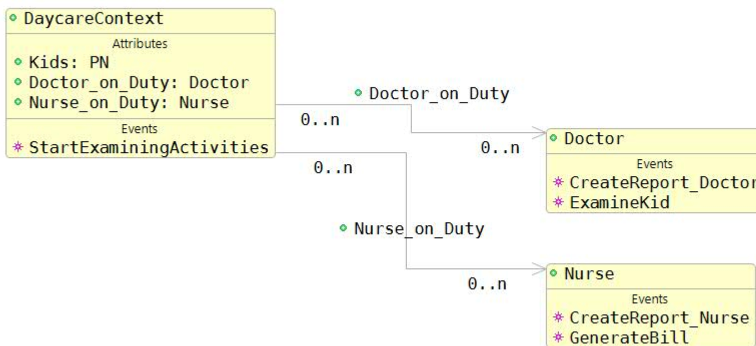

Figure 15. Steps 1–2 of the daycare center system design process

Now we use shared-event annealing, annealing as association, add attribute, annealing as subtyping and introduce subtyping in order to introduce design elements into this class diagram.

5.2.1 Steps 1-2: Shared-Event Annealing and Adding New Attribute (Nurse_on_Duty)

We apply shared-event annealing on *CreateReport* in the class diagram in Figure 14 (Doctor writes part of the report, and Nurse writes the rest of it). This rule decomposes class Examiner into classes Doctor and Nurse; see Figure 15. Part of the event CreateReport related to writing the report by Doctor is considered as event CreateReport_Doctor in class Doctor, and its part related to writing the report by Nurse is considered as event CreateReport_Nurse in class Nurse. Also, we add new attribute Nurse_on_Duty to class DaycareContext.

5.2.2 Steps 3-6: Dropping Attribute (Kids), Annealing as Association and Adding New Attributes

Figure 16 shows the class diagram after dropping attribute (Kids) and annealing as association. The second rule decomposes DaycareContext into classes DaycareContext and Kid. Class Kid models kids. Also, we introduce new attributes KName and KAge into class Kid; these new attributes capture name and age of kid.

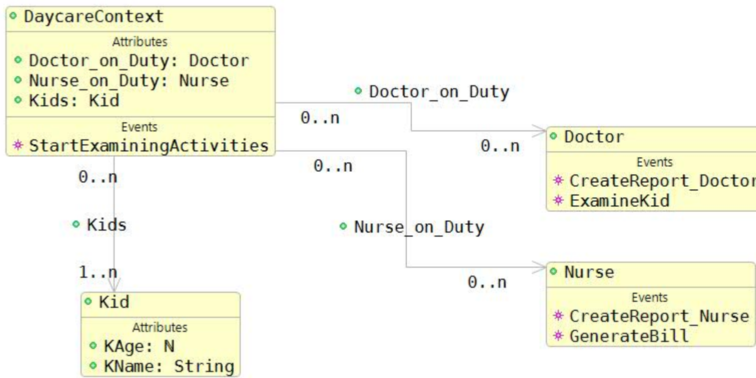

Figure 16. Steps 3–6 of the daycare center system design process

5.2.3 Steps 7-9: Adding New Attributes (exam_fee, dKid and nKid)

We add new attributes exam_fee and dKid to class Doctor. We also add new attribute nKid to class Nurse (Figure 17).

5.2.4 Steps 10-12: Annealing as Subtyping and Adding New Attributes (EName and SSN)

Figure 18 shows the class diagram after applying annealing as subtyping and adding new attributes. Annealing as subtyping decomposes class Doctor into classes Doctor

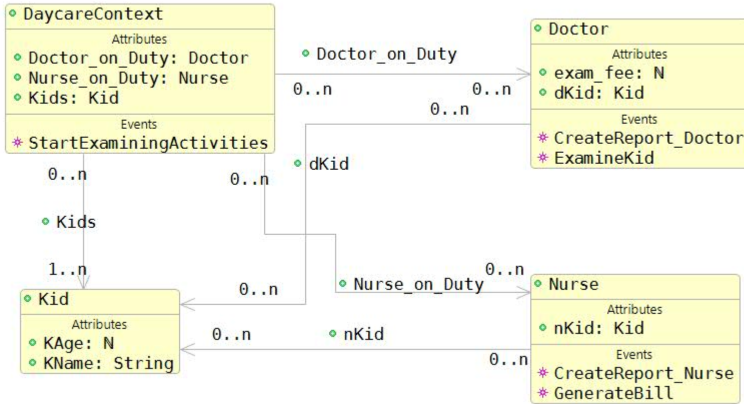

Figure 17. Steps 7–9 of the daycare center system design process

and Employee. Employee models employees in the context (Here, we only focus on doctors and nurses working in a daycare center). We also add new attributes SSN and EName to class Employee that capture social security number and name of employee.

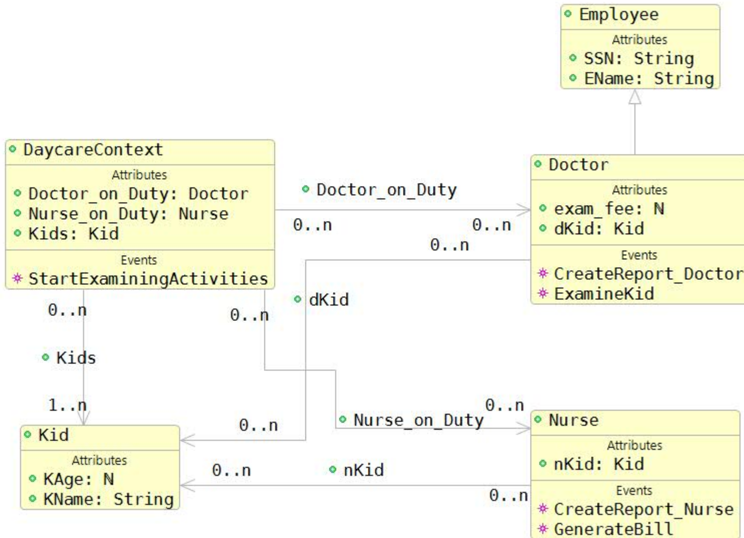

Figure 18. Steps 10–12 of the daycare center system design process

5.2.5 The Final Step: Introduce Subtyping

We apply Introduce Subtyping on the class diagram in Figure 18 to create a subtyping relationship between Employee and Nurse; see Figure 19.

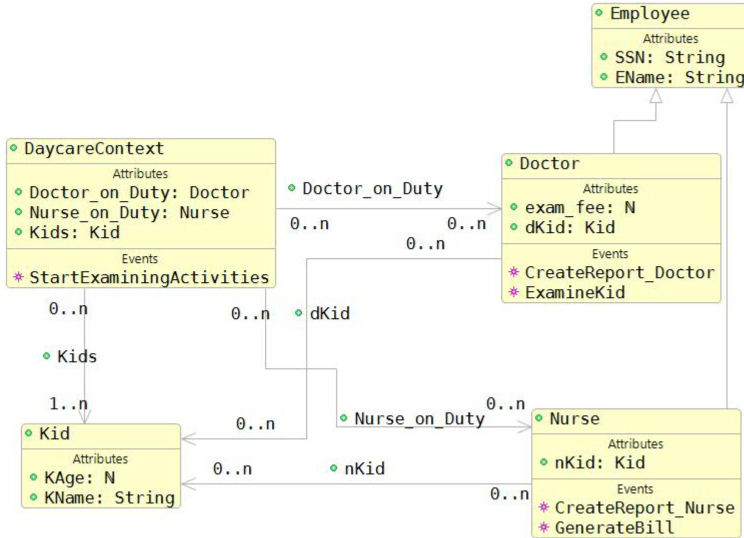

Figure 19. The final step of the daycare center system design process

In this case study, we used some of the rules presented in Section 3. Introduce subtyping is most effectively applied to link together classes that contain common features in order to maximize the potential for reuse; for this purpose, the Employee class is created as a superclass for Nurse and Doctor. In addition, by applying annealing as association and adding new attributes, the kid class is introduced for the purpose of achieving single responsibility that is one of the basic principles of object-oriented design. Through these transformations, we see how the classes reflect their intended purpose, object-oriented design principles are met more, and the final design becomes closer to the implementation.

5.3 Comparison with Related Work

To compare our rules, i.e., *annealing* and *introduce subtyping*, with their counterparts in VDM++ and Object-Z, we first consider the following criteria:

- CA: Covered styles of annealing
- P: Has it been proved that the rules are behavior preserving?

Table 1 shows the comparison for *annealing* (*ANL* abbreviates for annealing). As demonstrated in the previous subsections, all 4 forms of the annealing rule proposed in this paper are applicable for UML-B specifications.

Criterion	UML-B	Object-Z	VDM++
CA	ANL as association ANL as subtyping shared-event ANL shared-attribute ANL	ANS as association	ANL as association ANL as multiple inheritance
P	Yes	Yes	No

Table 1. Comparison of annealing

In the previous subsections, we demonstrated the applicability of the proposed rule for *introduce subtyping* in UML-B. However, this rule does not exist in VDM++ [19]. Lano and Goldsack [19] only present formal definition of subtyping, and they do not propose any introduce subtyping rule. In contrast, an *introduce subtyping* rule which is similar to our rule has been added to Object-Z [14]. McComb and Smith [14] proved this rule is behavior preserving in Object-Z, but we did not show the behavior preserving proof of this rule in the current work.

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced two refactoring rules, called *annealing* and *introduce subtyping*, in order to introduce design elements into an abstract functional specification written in UML-B. We then proved that all proposed styles of annealing are behavior preserving. Next, as two case studies, we introduced some steps of the design process into abstract functional specifications of the mass transit railway system and the daycare center system in UML-B.

As our future work, we are going to

1. provide refactoring rules for other diagrams in UML-B,
2. prove that introduce subtyping is behavior preserving,
3. extend our proofs to contain state machines.

REFERENCES

[1] Snook, C.—Butler, M.: UML-B and Event-B: An Integration of Languages and Tools. Proceedings of the IASTED International Conference on Software Engineering (SE 2008), 2008, pp. 336–341.

[2] Behm, P.—Benoit, P.—Faivre, A.—Meynadier, J. M.: Météor: A Successful Application of B in a Large Project. Lecture Notes in Computer Science, Vol. 1708, 1999, pp. 369–387.

[3] Abrial, J. R.—Hallerstede, S.—Mehta, F.—Métayer, C.—Voisin, L.: Specification of Basic Tools and Platform. RODIN Deliverable D10, 2005.

[4] Razali, R.—Snook, C.: Comprehensibility of UML-Based Formal Model: A Series of Controlled Experiments. Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering, 2007.

[5] Booch, G.—Jacobson, I.—Rumbaugh, J.: The Unified Modeling Language – A Reference Manual. Addison Wesley, 1998.

[6] Coplien, J. O.: Multi-Paradigm Design for C+. Addison-Wesley, 1999.

[7] McComb, T.: Refactoring Object-Z Specifications. Fundamental Approaches to Software Engineering, Springer-Verlag, Lecture Notes in Computer Science, Vol. 2984, 2004, pp. 69–83.

[8] McComb, T.—Smith, G.: A Minimal Set of Refactoring Rules for Object-Z. Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '08), 2008, pp. 170–184.

[9] Philipps, J.—Rumpe, B.: Refactoring of Programs and Specifications. In: Kilov, H., Baclawski, K. (Eds.): Practical Foundations of Business System Specifications. Springer, 2003, pp. 281–297.

[10] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[11] Zhang, J.—Lin, Y.—Grey, J.: Generic and Domain Specific Model Refactoring Using a Model Transformation Engine. In: Beydeda, S., Book, M., Gruhn, V. (Eds.): Model-Driven Software Development. Research and Practice in Software Engineering, Springer, Vol. II, 2005, pp. 199–218.

[12] Said, M. Y.—Butler, M.—Snook, C.: Language and Tool Support for Class and State Machine Refinement in UML-B. Proceedings of 16th International Symposium on Formal Methods (FM 2009), 2009, pp. 579–595.

[13] Comelio, M.—Cavalcanti, A.—Sampaio, A.: Refactoring by Transformation. Proceedings of REFINE 2002, Electronic Notes in Theoretical Computer Science, Elsevier, Vol. 70, 2002, No. 3, pp. 311–330.

[14] McComb, T.—Smith, G.: Refactoring Object-Oriented Specifications: A Process for Deriving Designs. Technical Report SSE-2006-01, University of Queensland, 2006.

[15] McComb, T.—Smith, G.: Architectural Design in Object-Z. 2004 Australian Software Engineering Conference (ASWEC), 2004, pp. 77–86.

[16] Smith, G.—Helke, S.: Refactoring Object-Oriented Specifications with Inheritance-Based Polymorphism. Proceedings of Fifth International Symposium on Theoretical Aspects of Software Engineering (TASE), 2011.

[17] Said, M. Y.: Methodology of Refinement and Decomposition in UML-B. Ph.D. thesis. University of Southampton, 2010.

[18] Snook, C.—Butler, M.: UML-B: A Plug-in for the Event-B Tool Set. Abstract State Machines, B and Z, Springer, Lecture Notes in Computer Science, Vol. 5238, 2008, p. 344.

[19] Lano, K.—Goldsack, S.J.: Refinement, Subtyping and Subclassing in VDM++. Proceedings for the Second Imperial College Department of Computing Workshop on Theory and Formal Methods, 1994, pp. 341–363.

[20] Goldsack, S.—Lano, K.: Annealing and Data Decomposition in VDM++. ACM SIGPLAN, Vol. 13, 1996, No. 4, pp. 32–38.

[21] Lano, K.—Goldsack, S.: Refinement of Distributed Object Systems. Proceedings of Workshop on Formal Methods for Open Object-based Distributed Systems, 1996.

[22] McComb, T.—Smith, G.: Compositional Class Refinement in Object-Z. 14th International Symposium on Formal Methods (FM 2006), 2006, pp. 205–220.

[23] Liu, H.—Zhu, B.: Refactoring Formal Specifications in Object-Z. 2008 International Conference on Computer Science and Software Engineering, 2008, Vol. 2, pp. 342–345.

[24] Pascal, C.—Silva, R.: Event-B Model Decomposition. DEPLOY Plenary Technical Workshop, 2009.

[25] Snook, C.—Butler, M.: U2B – A Tool for Translating UML-B Models into B. UML-B Specification for Proven Embedded Systems Design, Springer, 2004.

[26] Abrial, J. R.—Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundamenta Informaticae, Vol. 77, 2007, No. 1-2, pp. 1–28.

[27] Rodin User's Handbook v. 2.5. Available on: `handbook.event-b.org/current/html/`.

[28] Abrial, J. R.: Event Model Decomposition. Technical Report 626, ETH Zurich, 2009.

[29] Silva, R.—Butler, M.: Shared Event Composition/Decomposition in Event-B. In: Aichernig, B. K., de Boer, F. S., Bonsangue, M. M. (Eds.): Proceedings of the 9th International Conference on Formal Methods for Components and Objects (FMCO 2010). Lecture Notes in Computer Science, Vol. 6957, 2011, pp. 122–141.

[30] Duke, R.—Rose, G.: Formal Object-Oriented Specification Using Object-Z. Macmillan, UK, 2000.

Mehrnaz Najafi is an M.Sc. student at the Faculty of Computer Science and Engineering, Shahid Beheshti University, Iran, from where she also received her B.Sc. and M.Sc. degrees in software engineering. Her research interest includes formal program development, and she has more than 6 papers in this area.

Hassan Haghighi is Assistant Professor at the Faculty of Computer Science and Engineering, Shahid Beheshti University, Iran. He received his Ph.D. degree in software engineering from Sharif University of Technology, Iran, in 2009. His main research interest includes using formal methods in the software development life cycle, and he has more than 50 papers in this area.

Tahereh Zohdi Nasab is an M.Sc. student at the Faculty of Computer Science and Engineering, Shahid Beheshti University, Iran. She received her B.Sc. degree in computer engineering software from University of Tehran, Iran, in 2009. Her research interest is formal program development.