Extending OCL to Better Express UML Qualified Associations

Alla Dove, Aditi Barua and Yoonsik Cheon

TR #14-20
March 2014

Keywords: formal specification, constraints, map, qualified association, OCL, UML.

1998 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications—languages; D.2.4 [*Software Engineering*] Software/Program Verification—class invariants, formal methods, programming by contract; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—assertions, invariants, pre- and post-conditions, specification techniques.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

Extending OCL to Better Express UML Qualified Associations

Alla Dove, Aditi Barua and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A
alladove@gmail.com; abarua@miners.utep.edu; ycheon@utep.edu

***Abstract*—A qualified association in the Unified Modeling Language (UML) is an association that allows one to restrict the objects referred in an association using a key called a qualifier. A qualified association can appear in a constraint written in the Object Constraint Language (OCL) to specify a precise UML model. However, the OCL notation fails to provide appropriate support for expressing certain types of constraints written using qualified associations. In this paper we first describe a deficiency of OCL in expressing qualified associations and then propose a small extension to OCL to make it more expressive. The key idea of our extension is to view a qualified association as a map and provides a language construct to manipulate it as a first class entity in OCL. For this, we also extend the OCL standard library to introduce a wide range of map-specific collection operations. Our extension makes OCL not only more expressive but also amenable to a more direct translation to programming languages for various implementation uses of OCL constraints.**

***Keywords*— Formal specification, constraints, map, qualified association, OCL, UML.**

I. Introduction

The Object Constraint Language (OCL) is a formal, textual notation designed specifically for use with UML diagrams such as class diagrams to specify additional business rules or constraints that the diagrams have to satisfy [9]. While it enables software developers to construct more precise UML models by reducing ambiguities occurring in diagram-based models, its notation lacks expressiveness for certain UML concepts such as qualified associations. A qualified association is an association that allows one to restrict the objects referred in an association using a key called a *qualifier* [8]. An optional qualifier at an association end enables the indexing of many associations between classes. It partitions associations into key-to-value mappings, where the key comes from the qualifier and the value is given by the associated class.

In this paper we first point out a deficiency of OCL in its support for qualified associations. A qualified association can be conceptually viewed as a key-to-value map, but OCL doesn't provide a notation for manipulating it as a map; its notation is only for traversing associated classes through qualified associations. In other words, a qualified association is not a first-class entity in OCL, and this limits its expressiveness in writing constraints on or using qualified associations. We then propose a small extension to OCL to improve its expressiveness. Our extension consists of an extension to the notation and a new standard collection library class. We extend the OCL notation for navigating a qualified association to view it as a key-to-value map, and our new collection library class provides a wide range of operations to manipulate this map. Our preliminary evaluations show that our small extension makes OCL more expressible but also the resulting constraints more readable, understandable and directly translatable to various programming languages for implementations [3] [5].

The remainder of this paper is structured as follows. In the next section we explain OCL by applying it to a small example that will be used throughout this paper. In Section III we identify and describe a deficiency of OCL in supporting qualified associations. In Section IV we explain our approach of extending OCL to better support qualified associations. Our extension includes both the notation and the collection library. In Section V we apply our extension to our running example and produce a series of small OCL specifications. In Section VI we conclude this paper with a concluding remark.

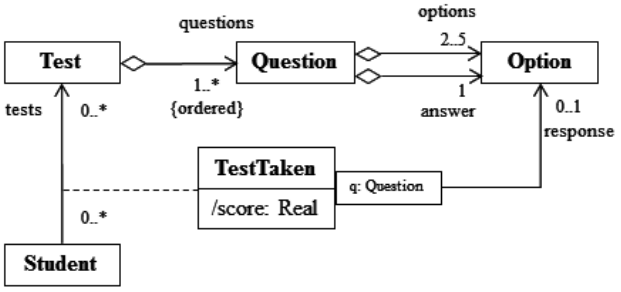


Figure 1: Example UML class diagram

II. The Object Constraint Language

The Object Constraint Language (OCL) is a textual, declarative notation used to specify constraints or rules that apply to UML models [9]. OCL can play an important role in

model-based software development because UML diagrams generally lack sufficient precision to enable the transformation of a UML model to complete code [5]. A UML diagram alone often cannot express rich semantics of and all relevant information about an application. As an example, consider the class diagram depicted in Figure 1 that models an on-line test-taking application. The application allows students to take tests consisting of multiple choice questions. Each question has two to five options, one of which is a correct answer. However, the class diagram does not express the fact that the answer to a question should be one of its options. OCL allows one to precisely specify this kind of additional constraints on UML modeling elements. It is based on set theory and predicate logic and supplements UML diagrams by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. The above constraint, for example, can be written in OCL as a class invariant as follows.

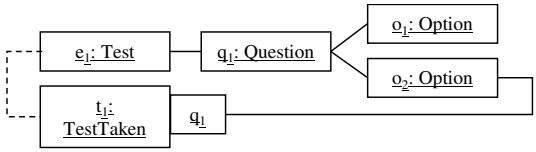
context Question
inv: options→includes(answer)

It states that for each question its options should include its answer. The collection operation *includes* tests if an element appears in a collection; as shown, an OCL collection operation is invoked using an arrow (→) notation. OCL supports a wide range of collection operations to write sophisticated queries and constraints by navigating associations and manipulating associated objects. For example, we can specify an operation, say *calcAverage,* which calculates the average of student's test scores by using such collection operations as *size*, *collect*, and *sum* as follows.

context Student::calcAverage(): Real
pre: testTaken→size() > 0
post: result = testTaken→collect(score)→sum()
/ testTaken→size()

In OCL an association class (e.g., TestTaken) that is part of an association relationship between two other classes can be referred to by using its name.

In UML, an association may have an optional *qualifier* at an association end to enable the indexing of many associations between classes [8]. It allows partitioning associations into key-to-value mappings, where the key comes from the qualifier and the value is given by the associated class. A qualified association is the UML equivalent of a programming concept variously known as associative arrays, maps, and dictionaries. In Figure 1, for example, the association between TestTaken and Option is a qualified association with a qualifier *q* of type Question. Here the keys are objects of type Question and the values are objects of type Option. It models student responses to test questions by stating that for each question *q* there may be zero or one option, the answer provided by a student. The object diagram shown below depicts one possible object configuration that conforms to the class diagram. A test e_1 has only one question q_1 consisting of two options o_1 and o_2, and a test taken by a student, t_1, says that the student's response to the question q_1 is the option o_2.



One can refer to a qualified association when writing OCL constraints. For example, we can define the value of the derived attribute *score* of the TestTaken class as follows.

context TestTaken::score: Real
derive: tests.questions
→select(ql self.response[q] = q.answer)→size())
/ tests.questions→size()

The expression *self.response*[*q*] denotes an Option object associated with the TestTaken object *self* through the qualifier *q*; if a qualifier is omitted, e.g. *self.response*, it denotes all the associated objects regardless of their qualifiers. The *select* operation selects all the test questions that are correctly answered. In the following section we will show more OCL constraints written using qualified associations to describe a shortcoming of the OCL support for qualified associations.

III. The Problem

As shown in the previous section, OCL provides a special notation to navigate through a qualified association. One needs to specify the value for a qualifier to the navigation in square brackets, e.g., *response*[*q*], to obtain the associated objects. If the value for a qualifier is left out from the navigation, e.g., *response*, it denotes all the associated objects regardless of the qualifier value. The way to navigate through a qualified association is consistent with that of an unqualified association in that both produce a collection of associated objects that could be manipulated using various collection operations.

However, interpreting a qualified association as an unqualified association by specifying a qualifier value limits the expressiveness of the OCL language, as a qualified association cannot be viewed or manipulated as a set of key-value pairs, e.g., one cannot write constraints on the keys (i.e., qualifier values). Let's consider our running example of the on-line test-taking application shown in Figure 1 in Section II. One important domain constraint for this application is that student's responses to test questions should be the options of the questions, which may be written as a class invariant as follows.

context TestTaken
inv: tests.questions→forAll(q|
not self.response[q].oclIsUndefined()
implies q.options→includes(self.response[q])

The *forAll* iterator operation is similar to a universal quantifier in logic and asserts that a predicate holds for each element of a collection. The invariant states that for each question of the test taken if a response is provided by a student, it should be one of the options of the question. The invariant constraint restricts the responses a student can provide to the options of the questions by putting a constraint on the value side of the key-value pairs of a qualified association. However, it doesn't impose any constraints on the key side of the qualified association. As shown in Figure 2, for example, it doesn't prevent a student from answering a question that is not included in the test; q_2 is not in the test and thus the qualified association between t_1 and q_2 trivially satisfies the invariant. The set of test questions answered by a student can be larger than the set of questions of the test taken.

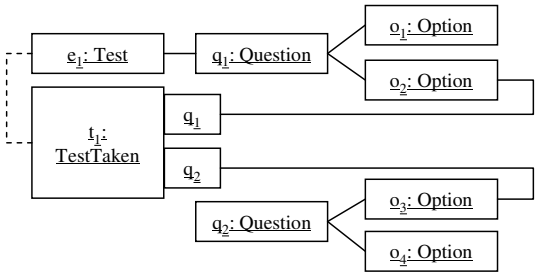


Figure 2: Sample object configuration

One possible fix would be to introduce another constraint to disallow student-provided responses to the questions not included in the test.

context TestTaken
inv: Question.allInstances()→forAll(q|
not tests.questions→includes(q)
implies self.response[q].oclIsUndefined())

The invariant is written using the *allInstances* operation that denotes the set of all instances of a type, and states that if a question q is not a question of the test taken, *response*[q] is undefined. Although this invariant constrains the values of the key-value pairs of a qualified association, its real purpose or intention is to indirectly constrain the keys or qualifier values of the association. This indirect, convoluted way of writing assertions not only confuses the readers, especially, about the purposes of the assertions but also the resulting constraints are long and complex, making them less readable, understandable and amenable to formal manipulations and treatments. In general, more direct and concise assertions are preferred. In this particular case, a better solution would be to write a direct statement requiring the qualifier values be a subset of the test questions, e.g., dom $f \subseteq$ tests.questions, where f denotes the key-value mapping of the qualified association and dom denotes its keys. In the following section we explain our approach for writing such a direct and concise constraint on a qualified association.

IV. Our Approach

As explained in the previous sections, a qualifier is a property of a binary association and is an optional part of an association end [8]. A qualifier holds a set of association attributes which model the keys that are used to index a subset of relationship instances. Conceptually, a qualified association thus can be viewed as a map from qualifier values (keys) to associated objects (values). When navigating through a qualified association, however, this map view is lost because it produces only the associated objects. This is the reason that OCL lacks expressiveness in writing constraints on or using qualified associations. The key idea of our approach is to make a qualified association a first class entity in OCL by exposing it as a map and allowing one to query and manipulate the map directly. For this we make a small extension to the OCL notation to denote the qualified association map and introduce a new collection library class to model a map.

A. An Extension to OCL Notation

In OCL, to navigate through a qualified association one can append to the navigation an optional qualified value enclosed in a pair of square brackets, e.g., *response*[q], to obtain the associated objects; a navigation with no qualified value (e.g., *response*) denotes all the associated objects regardless of their qualifier values. We propose a small extension to this OCL notation to denote the map itself modeling a qualified association, that we call a *qualified association map*. Our proposed notation is to use an empty pair of square brackets, e.g., *response*[] (see Figure 3).

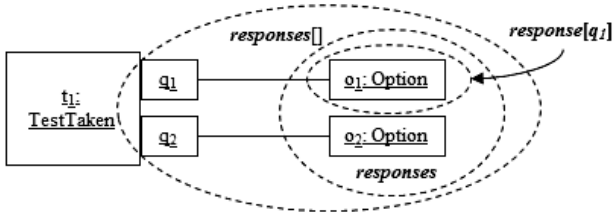


Figure 3: Notation for referring to qualified associations

Once a qualified association map is obtained by using the proposed notation, it can be queried and manipulated by using a wide range of collection operations (refer to Section IV.B for new operations). For example, we can easily express the previously-mentioned key constraint, *dom f* ⊆ *tests.questions* as: *tests.questions*→*includesAll*(*answers*[]→*keys*()), where

keys is a new collection operation introduced to obtain the domain of a qualified association map (see Section IV.B).

The proposed new notation does not contradict or overwrite the standard OCL notation. It is merely a small addition, which can be applied to denote a qualified association map and to express constraints on a qualified association when used in conjunction with new collection operations (see the following subsection). In summary, while the standard OCL notation allows one only to navigate through a qualified association, i.e., by denoting the values of key-value pairs of a qualified association map, our extension allows one to query and manipulate it as a map itself, i.e., a set of key-value pairs.

B. An Extension to OCL Standard Library

Alongside the new notation we also propose to extend the OCL standard library to introduce a new collection type to model a qualified association map. The OCL standard defines one abstract collection type named Collection and four concrete collection types named Set, OrderedSet, Bag, and Sequence [6]. The Collection type is a common supertype of all the concrete collection types and defines operations common to all collection types. Additionally, each concrete collection type has a number of specific collection operations not shared among all concrete collection types.

We propose to introduce a new collection type named Map. This collection type provides a model for a qualified association map. A map is a set of key-value pairs and cannot have duplicate keys, meaning that each key can map to at most one value. A key-value pair is represented as a tuple of type Tuple(key: K, value: V) (see Appendix A.2). Both Set and Map are unique, unordered collections of elements. The only difference between them is that Map consists of tuples. For that reason, we also propose to make the Map a direct subtype of Set. This will ensure for Map to have all the operations of Collection and Set without duplicating them.

Another strong argument for introducing this new collection type is that when translating a UML qualified association to an implementation, an associative array or map is frequently used as a concrete representation to implement the required functionality [1].

The usual collection, iteration and Set-specific operations such as *size*, *includes*, *includesAll*, *isEmpty*, *select*, *collect*, *forAll, union*, *intersection*, *including*, and *asBag* are defined for Map, as they are inherited from Collection and Set. However, elements are assumed to be of type Tuple(key: K, value: V). In addition to these common collection, iteration and Set-specific operations, Map defines many new map-specific collection operations, and Table 1 shows a list of representative operations; refer to Appendix A.2 for a complete list of map-specific collection operations and their specifications. In the following subsection we present a series of small examples to illustrate the use of these new collection operations.

Table 1: Map-specific collection operations

Operation	Description
including(k,v)	Map with (k, v) pair added
excludingKey(k)	Map with key k removed
includesKey(k)	Is k mapped to a value?
includesValue(v)	Is v mapped by a key?
keys()	Domain of self
values()	Range of self
apply(k)	Application of self to k, i.e., self[k]
override(k,v)	Map with (k,v) pair added or replaced
restrictDomain(d)	Domain restriction by keys d
restrictRange(r)	Range restriction by values r
compose(m)	Relational composition of self and m

V. APPLICATION

A. Examples

Using our new notation and collection operations we can easily express the constraint discussed in Section III that motivated our work. Essentially we need to limit the set of keys of the qualified association map to the test questions as shown below.

context TestTaken
inv: tests.questions→includesAll(response[]→keys())
inv: response[]→forAll(p|p.key.options→includes(p.value))

The first invariant expresses the motivating constraint by stating that the set of test questions is a superset of the keys of the qualified association map. That is, only questions included in the test can have responses. The second invariant asserts that the student-provided responses should be the options of the test questions. The *forAll* collection operation is used to constrain each key-value pair of the qualified association map. It would be instructive to compare these new constraints with the ones presented in Section III, which is copied below.

context TestTaken
inv: Question.allInstances()→forAll(q|
  not tests.questions→includes(q)
  implies self.response[q].oclIsUndefined())
inv: tests.questions→forAll(q|
  not self.response[q].oclIsUndefined()
  implies q.options→includes(self.response[q])

The new constraints are not only more concise but also easier to read and understand, as they capture and express the core of the constraints directly, e.g., a relationship between two sets, the test questions and the keys of the qualified association map. We expect such direct constraints to be more amenable to formal and informal treatments of constraints,

e.g., formal verification and transformation to code.

In Section II we defined the value of the derived attribute *score* of the TestTaken class. We can simplify its formulation and rewrite it by using our extended notation as follows.

context TestTaken::score: Real
derive: response[]→select(p|p.key.answer=p.value)→size())
/ tests.questions→size()

The *select* iteration operation defined in OCL returns a collection with all the elements of the receiver that meets a specified condition. In this example it returns the set of correct responses—i.e., question-and-option pairs; as specified in the condition of the *select* operation, a response *p* is correct if its value (option) is equal to the answer of its key (question).

Below we show several operations of the TestTaken class that can be easily and naturally specified in our extended notation along with new map-specific collection operations.

context TestTaken::isAnswered(q: Question): Boolean
post: result = response[]→includesKey(q)

context TestTaken::unanswered(): Set(Question)
post: result = tests.questions – response[].keys()

context TestTaken::incorrectlyAnswered(): Set(Question)
post: result = response[]→select(p|p.key.answer <> p.value)

context TestTaken::responses(): Map(Question, Option)
post: result = response[]

context TestTaken::addResponse(q: Question, a: Option)
pre: tests.questions→includes(q)
pre: q.options→includes(a)
post: response[] = response[]@pre→including(q, a)

The *includesKey* operation used in the specification of the *isAnswered* operation tests whether a given key is defined by a map. The *keys* operation in the second specification returns the set of all keys defined by a map. The specifications of the last two operations are interesting, as they clearly show the benefit of querying and manipulating a qualified association as a map itself. For example, we can easily specify the behavior of such mutation operations as *addResponse* that changes the values of a qualified association. The new value of the qualified association *response* is its old value with the given new response added; the *including* operation appearing in the post-condition adds a new key-value pair to a map.

As the last example we specify below the *takeTest* operation of the Student class that models a test taking by a student. The operation takes a test to be taken by the student and the student responses to the test, a map of question-option pairs. The operation adds the given test to the set of tests taken by the student and links a new TestTaken object to the newly created Student-Test association.

context Student::takeTest(t: Test, a: Map(Question,Option))
pre: **not** tests→includes(t)
pre: t.questions→includesAll(a→keys())
pre: a→forAll(p| p.key.options→includes(p.value))
post: tests = tests@pre→including(t)
post: testTaken→exist(tt| tt.oclIsNew() **and**
tt.student = self **and** tt.test = t **and** tt.response[] = a)
post: testTaken→includesAll(testTaken@pre)
post: testTaken→size() = testTaken@pre→size() + 1

The first precondition is to ensure that a given test is not already taken by the student, and the other preconditions are for establishing the class invariant of the TestTaken class.

- The set of questions in the map of the student's responses should be a subset of all questions of the test to be taken.
- For each question in the map, the student's response is one of the options of the question.

The first postcondition states that the given test is added to the set of tests taken by the student. The second is to assert that a new TestTaken object is created and linked to the new Student-Test association introduced by the first postcondition. The other two postconditions are to assert that all the previous Student-Test associations are still there and no new one is added. The formulation of the postconditions is a bit convoluted, as the creation of a new TestTaken object has to be asserted indirectly; OCL does provides a direct and concise way for modeling object creation [4].

B. Preliminary Evaluation

A formal evaluation of our proposed extension is pending but a series of example constraints presented in the previously subsection convince us that our extension makes the OCL notation more expressive, readable, and understandable. For example, there is no straightforward way to write the following constraint using the standard OCL notation.

context TestTaken::isAnswered(q: Question): Boolean
post: result = response[]→includesKey(q)

It is because the standard doesn't provide any notation for referring to the qualified association relationship itself, its keys when it is viewed as a map. Our finding is that in general constraints written using our extended notation are more direct and explicit especially when the constraints are on the qualifiers of qualified association relationships. They are also tend to be more concise and less convoluted.

A nice side benefit of our extension is the translation of constraints to code for various uses of design constraints

during implementation, including transformation of models into source code in model-driven development [5], runtime constraint checks [3], and even verification and validation of UML/OCL models [7]. In most cases, it is straightforward to translate constraints to implementations; it can be done systematically and thus fully automated. This is because developers frequently use such data structures as associative arrays and maps to reify qualified associations in their implementations [1] and most modern programming languages provide map data structures, e.g., Map in Java and C++, associative arrays in PHP, and Dictionary in C#. In Java, for example, the qualified association *response*[] can be easily translated to a field of type Map<Question, Option>.

VI. Conclusion

In this paper we first pointed out a limitation of OCL in supporting qualified associations and then proposed a small extension to both the OCL notation and the standard collection library. Our extension allows one to query and manipulate qualified associations as key-values pairs that we call *qualified association maps*. While a more rigorous evaluation is still needed, a series of small example specifications written using our extension shows that the notion of qualified association maps not only improves the expressiveness of OCL but also produces constraints that are in general more direct, concise, readable, and amenable to various types of formal and informal manipulations, e.g., translation to code. From this we conclude that our extension meet the needs of both software specifiers and programmers. Two most important contributions of our work is that (1) we showed the need of OCL to provide a better, more expressive way of wring constraints on or using qualified associations and (2) we addressed this need by proposing a small extension to the OCL notation and its collection library.

References

[1] D. H. Akehurst, W. G. J. Howells, and K. D. McDonald-Maier, Implementing Associations: UML 2.0 to Java 5. Journal of Software and Systems Modeling, 6 (1):3-35, 2006.

[2] C. Avila, et al., Runtime Constraint Checking Approaches for OCL, A Critical Comparison, *International Conference on Software Engineering and Knowledge Engineering*, July 1-3, 2010, pp. 293-398.

[3] Y. Cheon, et al., Checking Design Constraints at Run-time Using OCL and AspectJ, *International Journal of Software Engineering,* 2(3):5-28, December 2009.

[4] A. Hamie, et al., Reflections on the Object Constraint Language, *The Unified Modeling Language 98: Beyond the Notation, LNCS*, vol. 1618, pp. 162-172, Springer, 1998.

[5] K. Lano, *Model-Driven Software Development with UML and Java*. Course Technology, 2009.

[6] Object Management Group, *Object Constraint Language*, version 2.3.1. Jan. 2012. Available from http://www.omg.org/spec/OCL/.

[7] M. Richters and M. Gogolla, Validating UML Models and OCL Constraints, *The Unified Modeling Language 200, LNCS*, vol. 1939, pp. 2650277, Springer, 2000.

[8] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language *Reference Manual*, 2nd ed. Addison-Wesley, 2004

[9] J. Warmer and A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, 2nd ed. Addison-Wesley, 2003.

Acknowledgements

The work of Dove is supported in part by the National Science Foundation (NSF) Graduate Research Fellowship, and the work of Barua and Cheon is supported in part by NSF grant DUE-0837567. Any opinion, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

A. Appendix

A.1 *An Extension to OCL Notation*

We extend the OCL notation for navigating a qualified association. In OCL, the production rule representing a navigation call expression [6] (Section 9.4.37) is defined in terms of the production rule for navigating to an association class [6] (Section 9.4.38).

AssociationClassCallExpCS ::= *OclExpressionCS* '.'
  simpleNameCS ('[' *argumentsCS*? ']')? *isMarkedPreCS*?
AssociationClassCallExpCS ::=
  simpleNameCS ('[' *argumentsCS*? ']')? *isMarkedPreCS*?

The production rule *argumentsCS* represents a sequence of arguments [6] (Section 9.4.40), and our extension is to make it optional (denoted by using a meta-symbol "?"). This allows us to introduce a new expression like *response*[] to denote an qualified association map. The optional production rule *isMarkerPreCS* represents the marking @pre in an OCL expression [6] (Section 9.4.39).

A.2 *A New Collection Type, Map*

The Map type is a template type with two type parameters, K for keys and V for values. A concrete map type is obtained by substituting actual types for K and V, e.g., Map(Person, Address). A map is a set of tuples of type Tuple(key: K, value: V), mapping keys to values. A map cannot have duplicate keys meaning that each key can map to at most one value.

The Map(K,V) is a subtype of Set(Tuple(key: K, value: V)) and inherits all the collection operations defined by Collection and Set-specific operations, including =, <>, size, includes, excludes, count, includesAll, excludesAll, isEmpty, notEmpty, union, intersection, including, excluding, asSequence and asBag. However, such operations as max, min, and sum are not defined for Map. The equality operation (=) and including operations are redefined as follow.

= (m : Map(K,V)) : Boolean
post: result = (self→forAll(t| m→includes(t)) **and**
  m→forAll(t| self→includes(t)))

including(t: Tuple(key: K, value: V): Map(K, V)
The map containing all elements of self and e.
pre: self→excludesKey(e.key).
post: result→forAll(e | self→includes(e) **or** (e = t))
post: self→forAll(e | result→includes(e))
post: result→includes(t)

Map defines the following new operations.

including(k: K, v: V): Map(K, V)
The map containing all elements of self and a k-v pair.
pre: self→excludesKey(k).
post: result→forAll(e | self→includes(e) **or**
(e.key = k **and** e.value = v))
post: self→forAll(e | result→includes(e))
post: result→includes(Tuple(key = k, value = v))

excludingKey(k: Key): Map(K, V)
The map containing all element of self without those with k as the key.
post: result→forAll(e | self→includes(e) **and** (e.key <> k))
post: self→forAll(e | result→includes(e) = (e.key <> k))
post: result→excludesKey(k)

includesKey(k: K): Boolean
True if k is mapped to a value by self, false otherwise.
post: result = self→exist(key = k)

excludesKey(k: K): Boolean
True if k is not mapped to a value by self, false otherwise.
post: result = self→forAll(key <> k)

includesValue(v: V): Boolean
True if a key is mapped to v by self, false otherwise.
post: result = self→exist(value = v)

excludeValue(v: V): Boolean
True if no key is mapped to v by self, false otherwise.
post: result = self→forAll(value <> v)

keys(): Set(K)
The domain of self.
post: result = self→collect(key)→asSet()

values(): Collection(V)
The range of self.
post: result = self→collect(value)

values(keys: Set(K)): Collection(V)
The relational image of a set of keys.
post: result =
self→select(keys→includes(key))→collect(value)

apply(k: K): V
The application of self to k.
pre: self→includesKey(k)
post: result = (self→any(key = k)).value

override(map: Map(K,V)): Map(K,V)
Relational overriding.
post: result = map→iterate(t; acc: Map(K,V) = self|
acc→excluding(t.key)→including(t.key, t.value))

restrictDomain(dom: Set(K)): Map(K,V)
Domain restriction.
post: result = self→select(dom→includes(key))

antirestrictDomain(dom: Set(K)): Map(K,V)
Domain anti-restriction.
post: result = self→select(dom→excludes(key))

restrictRange(ran: Set(V)): Map(K,V)
Range restriction.
post: result = self→select(ran→includes(value))

restrictRange(ran: Set(V)): Map(K,V)
Range anti-restriction.
post: result = self→select(ran→excludes(value))

compose(map: Map(V,V2)): Map(K,V2)
Forward relational composition.
post: result = self→iterate(t; acc: Map(K,V) = Map()|
if map→includesKey(t.value)
then acc→including(t.key, map→apply(t.value))
else acc)

composeBackward(map: Map(V,V2)): Map(K,V2)
Backward relational composition.
post: result = map→iterate(t; acc: Map(K,V) = Map()|
if self→includesKey(t.value)
then acc→including(t.key, self→apply(t.value))
else acc)