Chair of
Software Engineering

Model-based Contracts for C#

Bachelor Thesis

By: Tobias Kiefer
Supervised by: Nadia Polikarpova
Prof. Dr. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

Programming-by-contract has been proven a powerful method for software developers to write bug-free, high-quality software ever since it became popular with the appearance of the Eiffel programming language. This thesis deals with model-based contracts, which represent an extension of the classic Design by Contract approach. Just like the original concept the core component of this mechanism, the *Mathematical Model Library*, first was available for the Eiffel programming language only. This report describes the porting of the library to the .Net platform using the C# language, the application of model-based contract techniques to existing .Net software, as well as automated white box testing experiments of the contract-enhanced code using the Pex testing tool.

Contents

1. Introduction

1.1. Design by Contract

In Eiffel, programmers are able to express parts of the program specification within the actual code, without having to use external tools or languages. To enable this, Eiffel provides special assertions known as *contracts*. The relevant types of contracts in this context are *preconditions*, *postconditions*, and *class invariants*. While pre and postconditions relate to and essentially are part of a single method, a class invariant relates to a whole class. A precondition is a predicate that is checked and has to be true at the beginning of a method. In the same way, a postcondition must evaluate to true just after the execution of the method. A class invariant is an expression constraining the state of a whole type of objects, and can probably best be imagined as a pre and postcondition for all methods of the class.
Combined with run-time checking of the contracts the Design by Contract methodology can be a great help for programmers while developing and testing, especially since they do not need to be familiar with formal specification techniques. On the other hand, traditional contracts seem rather limited in a way that they leave a lot of the programmer's informal understanding of the program specification uncovered[1].

1.2. Model-based Contracts and the MML

Model-based contracts is an approach to extend the classic Design by Contract concept by associating each class with a mathematical model, consisting of so called *model queries* provided by immutable *model classes*. As the model classes are simply translations of mathematical concepts (e.g. sets, functions,..) into the respective programming language, the model queries can be used inside the traditional contract expressions. This allows the programmer to express even more powerful specifications without having to learn any new and complicated formal techniques or languages.

Listing 1.1: A model-based contract in C#

```
public new void AddLast(T item)
{
    // new == old & item
    Contract.Ensures(m_Sequence ==
        Contract.OldValue(m_Sequence).Extended(item));

    base.AddLast(item);
}
```

A typical example for a contract involving a model query is shown in listing 1.1. It is a contract for the `AddLast`[1]function of the DSA `DoublyLinkedList` implementation. The model for this class was chosen to be a sequence, and thus `m_Sequence`, the model query, is a member object of type `MML.Sequence`. In C#/.Net contracts `Contract.Ensures` denotes the static postcondition function which will check if the specified expression evaluates to true. Note that despite appearing at the top of the method body, the postcondition function will always be executed after all other code contained in the method. The contract checks (at the end of the method) if the resulting object is indeed the old object with the `item` appended at the end. For a more detailed introduction into model based contracts and some more background on related work in software specification see [1] and [2].

In the original Eiffel implementation the model classes are encapsulated in a library named the *Mathematical Model Library(MML)*[2]. Since recently Microsoft has developed a contract framework for their .Net platform[3], there exist the opportunity of porting the MML and thereby creating the possibility of applying the model-based contracts approach to a much larger software base. The porting of the MML code from Eiffel to C# is described in the second chapter of this report, while the third chapter covers the development of model-based contracts for DSA[4], a library of algorithms and data structures for the .Net platform. Chapter 4 describes test experiments involving contract-based automatic testing of the DSA code using the Pex[5] testing tool.

[1]Because we are dealing with a cross-language project, and for the sake of readability, throughout this document functions will be referred to mostly by their plain names, i.e., without return types and parameters, as long as this does not cause any ambiguity.

2. The C# Version of the MML

2.1. Initial Situation and the Porting Process

The first step towards a fully-functional, usable port of the MML was to inspect the current state of the code, both on the Eiffel and the C# side. As there had already been an attempt at porting the code to C# during a previous project[6], the question was to decide if the existing code would be worth building upon. Unfortunately was there neither a project report available, nor was there any information about completeness or correctness of the existing C# code. Still, since the amount of ported code looked quite substantial, it seemed worth of being manually inspected for compliance with the corresponding Eiffel implementation.
At the beginning of this process some minor structural differences became obvious; they will be described in the next section which is about the overall structure of the library on a class level. The major workload, however, was to deal with numerous implementation-based issues: bugs had to be fixed and missing functionality had to be implemented. These topics will be dealt with in Section 2.3. Even after these steps and the whole porting process, there were a few remaining inconsistencies left between the original Eiffel implementation and the C# version, mostly caused by differences and constraints regarding the two programming languages. Section 2.4 concludes this chapter by summarizing these minor discrepancies.

In general, though, it has to be noted that porting the library from Eiffel to C# was rather straightforward. To illustrate this, an example is shown in listings 2.1 and 2.2. This code was already part of the initial version of the port, and one can see how the developer could easily use the functional programming inspired features of the C# language, along with a lambda expression, to beautifully convert the agent-based Eiffel code without having to explicitly implement delegates or even manual iteration.

Listing 2.1: The `is_constant` function of class `MML_BAG` in Eiffel

```
is_constant (c: INTEGER): BOOLEAN
        -- Are all values equal to 'c'?
    do
        Result := values.for_all (agent reference_equal (c, ?))
    end
```

Listing 2.2: The `IsConstant` function of class `MML.Bag` in C#

```
[Pure]
public bool IsConstant(int c)
{
    return Array.TrueForAll(values, x => Equals(x, c));
}
```

2.2. The Overall Structure of the Library

The current version of the MML is a rather small piece of code. It consists of six classes, each one representing a different mathematical "object" or concept. Figure 2.1 shows a class diagram of the whole library. The class names should be self-explanatory. For a more detailed documentation about the individual classes and their functionality see Appendix A.
Although the diagram shows only the C# version of the code, the Eiffel version looks nearly the same. The only difference is that the Eiffel MML classes all have a common base class (`MML_MODEL`), which basically only serves the needs of an agent-related workaround and is thus not needed in C#. On the other hand, the C# classes `MML.Sequence` and `MML.Set` implement the `IEnumerator` interface. This enables users of the class to call the `GetEnumerator` function to receive an iterator for the data structure. Even though this functionality was seldom used in practice, this decision made for the initial version for the port has not been altered during this project.

From an object-oriented perspective, another design decision worth mentioning is that only little use of inheritance is made. Of course with the inheritance structure being the same in Eiffel and C#, apart from the above mentioned

exceptions, this observation is also true for the Eiffel version. Although reasoning about the original design goals of the library is beyond the scope of this thesis, this phenomenon most likely dates back to a decision made by the initial MML developer. He states that "The inheritance hierarchy of a class library modeling mathematical concepts does not necessarily correspond to the natural taxonomy of mathematical structures." ([2], p.55)

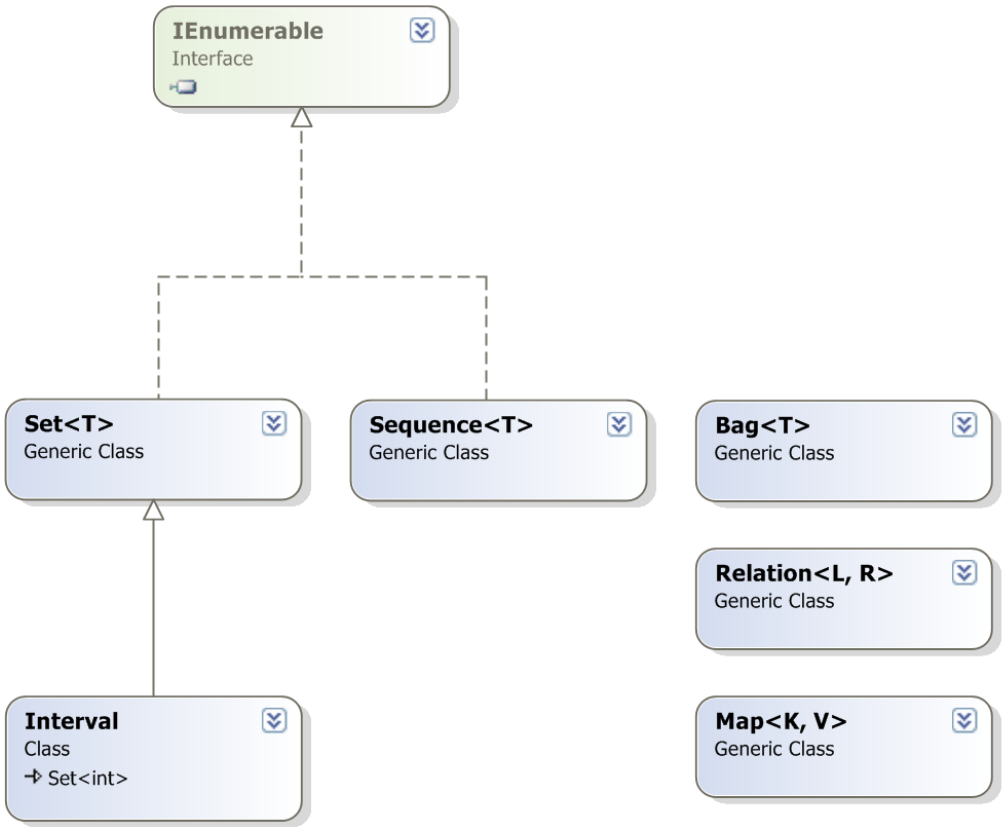


Figure 2.1.: A class diagram of the MML in C#

2.3. Improvements Made

Although a significant part of the Eiffel code has already been ported during the aforementioned initial project, there were still several improvements to be made. First of all there were still some functions in the Eiffel code that didn't have a C#-counterpart yet. For the port to provide the same functionality as the Eiffel version the following functions had to be newly implemented[1]:

- Class `Map`:
  - `ToBag`
- Class `Intervall`:
  - `Intervall(Tuple<int, int>)`[2]
- Class `Sequence`:
  - `ToBag`
  - `Removed`
  - `Restricted`
  - `ExtendedAt`[3]

In addition to the newly implemented functions there were some implementation errors, showing up during the manual revision of the code, that had to be fixed. Luckily, there were only two functions affected:

- `Set.Disjoint`
  In this case the error was restricted to the C# code, with the Eiffel code being correct. The failure was rather simple, as the original code returned the expression `Exists(other.Has)`, which paradoxically evaluates to true iff the sets are not disjoint (i.e., have a common element).

[1] The class and function names listed here correspond to the C# version of the code. The corresponding Eiffel entities have generally very similar names, adhering to the Eiffel style: i.e. `MML_MAP` and `to_bag`. Therefore, for the sake of readability, mentioning both versions of the names will often be omitted throughout this document unless it is considered useful.

[2] The constructor corresponding to the `from_tuple` creation procedure in Eiffel.

[3] Additionally, the implementation of `Sequence.Extended` was changed completely to make use of `ExtendedAt`

- `Relation.Has`
  This issue in the C# function was not caused by a mistake in porting the code, but rather revealed a similar bug in the original Eiffel implementation. It is illustrated in listing 2.3 as an example for the bugfixing aspect of the porting related work during this project.
  In contrast to the first bug described, the problem here is a little bit less obvious to see. The original implementation is commented out in the first two lines of the method body. The `Array.IndexOf` function returns the *first* index containing `x` on the left "side"(of the modeled relation). Then in the next line it is checked if the "right" value corresponding to this index matches the `y` value.
  Unfortunately this means that in a relation containing the pairs $(1, 2)$ and $(1, 3)$, i.e., the `lefts` array containing the tuple $(1, 1)$ and `rights` containing $(2, 3)$, the call to `Has(1, 3)` always returns false.
  In C# the issue was finally fixed using the `ImageOf` function, which returns *all* indices of occurrences of `x`. Moreover, the maintainer of the Eiffel MML code has been informed and the corresponding code was corrected as well.

Listing 2.3: A fixed bug in `MML.Relation.Has`

```
public bool Has(L x, R y)
{
// int i = Array.IndexOf(lefts, x);
// return i >= 0 && Equals(rights[i], y);

    return ImageOf(x).Has(y);
}
```

In addition to these major porting tasks like implementing new methods and correcting errors, a lot of work has also been done in areas that may at first not that obviously be seen as being important to contributing towards a fully functional port of the Eiffel code. Nonetheless, one goal of the porting effort was to reduce the remaining differences between the two MML versions and to improve the overall quality of the port. Therefore the following additional work has been done:

- **Aliases**
  In Eiffel it is easy to declare an arbitrary alias, made out of non-alphanumeric symbols, for a function. The MML interface makes extensive use of this. In C# it was attempted to implement these aliases using overloaded operators. During this project new operators were implemented to make the interface more similar to that of the Eiffel version. The current state of all aliases for both versions can be found in Table 2.1.

- **Contracts**
  Preconditions and class invariants corresponding to their respective equivalents in the Eiffel code have been added to all classes.

- **Comments and code structure**
  During the revision of the code, comments have been added as a natural process of understanding foreign code that is very sparsely documented. In addition to that, the structure of the source code files (e.g. the order member functions appearing in a class declaration) has been changed to match with the structure of the Eiffel code. This provides for easy comparison of corresponding code in both languages and has proven already useful while manually revising the code.

- **XML documentation**
  The code already contained some comments related to the automatic, XML-based generation of documentation using Visual Studio. These comments have been revised and extended. New XML comments have been added for functions that were previously not documented. Now the whole MML interface is covered by the automatically generated documentation, which is available in Appendix A of this report and as HTML and PDF files in the '`Documents/Documentation`' subdirectory on the project repository[7].

2.4. Differences between Eiffel and C# Versions

Even after all the above mentioned efforts to bring the C# code in line with the original implementation, there are still some minor differences remaining. There are several reasons for this: At a first look one immediately notices that while the class and function names are very similar, the C# version does not mimic the Eiffel style and naming conventions. It rather uses the CamelCase convention which is popular among Windows and Java developers.

This decision was made by the original developer of the C# port and has not been modified during this revision.

Apart from the slightly different function names caused by different naming conventions there are very few differences left regarding functions and functionality. There are currently no Eiffel functions that don't have a corresponding equivalent in the C# code. On the other hand, there are even additional functions in the C# code that were not part of the initial Eiffel version. These functions were not implemented during this project but were part of the initial porting attempt. The following C# functions have no Eiffel counterpart:

- Class `Map`:
  - `Has(K x)`
- Class `Sequence`:
  - `Reverse`

In addition to the intentional decisions regarding function implementation and naming described above, there are also constraints caused by the target language playing a role as a reason for inconsistencies between the two versions. Probably the most important difference in this regard is the lack of flexibility in the support of "operator aliases" for functions in C#. As already denoted in the previous section, in Eiffel aliases for functions may be arbitrary strings, as long as they do not contain alphanumeric symbols.

In C# no such flexibility exists. Eiffel aliases can therefore only be translated if they match a valid C# operator. Thus aliases like '`#`' for `Count` will probably never be available in C#. In the case of the '`<=`' alias for the `IsPrefixOf` function the C# compiler requires a matching '`>=`' operator to be defined. As no such operator or functionality exists in the original Eiffel code, neither of the two operators has been implemented in C#.
On the other hand, a lot of aliases were already supported in the initial port, and during this project the number was even increased. Table 2.1 summarizes all aliases of the MML interface, listing the respective names of the corresponding functions for both languages, as well as information about when they appeared in the C# version. Here "initial" means that they had already been ported during the previous project, "current" means that the alias has been added during this project, and "missing" denotes that the alias is not available in the C# port.

Table 2.1.: Function aliases of the MML interface

Alias	Function(Eiffel)	Function(C#)	Version
Bag:			
[ ]	occurrences	Occurrences	initial
|==| [1]	is_model_equal	[2]	initial
#	count	-	missing
&	extended	Extended	current
/	removed	Removed	current
|	restricted	Restricted	initial
+	union	Union	initial
-	difference	Difference	initial
Map:			
[ ]	item	Item	initial
|==| [1]	is_model_equal	[2]	initial
|	restricted	Restricted	initial
+	override	Override	initial
Relation:			
[ ]	has	Has	initial
|==| [1]	is_model_equal	[2]	initial
|	restricted	Restricted	initial
+	union	Union	initial
*	intersection	Intersection	initial
-	difference	Difference	initial
^	sym_difference	SymDifference	initial

Continued on next page

Table 2.1 – Continued from previous page

Alias	Function(Eiffel)	Function(C#)	Version

Sequence:

[ ]	item	Item	initial
#	count	-	missing
|==|[1]	is_model_equal	[2]	initial
<=	is_prefix_of	IsPrefixOf	missing
&	extended	Extended	current
+	concatenation	Concatenation	initial

Set:

[ ]	has	Has	initial
|	filtered	Filtered	current
#	count	-	missing
|==|[1]	is_model_equal	[2]	initial
<=	is_subset_of	IsSubsetOf	current
>=	is_superset_of	IsSupersetOf	current
&	extended	Extended	current
/	removed	Removed	current
+	union	Union	initial
*	intersection	Intersection	initial
-	difference	Difference	initial
^	sym_difference	SymDifference	initial

[1] The corresponding operator in C# is the '==' operator.
[2] In the C# implementation the '==' operator involves not a single function call but an equivalent expression.

3. Model Based Contracts for the DSA Library

3.1. Introduction

With the C# version of the MML being complete and fully functional, the next project step was to demonstrate its usability in practice, making use of the vast amount of C# and .Net software available. Because of the nature of the mathematical models represented by its classes, the MML is particularly well suited to model collection or container classes. A library providing such classes is the *Data Structures and Algorithms(DSA)* library. Since its code is open-source, it seemed to be a good candidate for being extended with contracts. This chapter describes the integration of model-based contracts into the existing DSA implementation using the MML in conjunction with the .Net *Code Contracts* framework.

Figure 3.1 shows a class diagram of the `Dsa.DataStructures` namespace. Small helper classes such as nodes for list classes or anything else not directly affected by the contract implementation have been omitted. The classes with names ending with "Contracts" have been added during this project as a wrapper for their parent classes. A typical method of such a wrapper class contains only the model-based contracts and a call to the respective base function of the parent class. In this way the contracts could mostly be separated from the original code. The initial intention behind this design idea was to provide more flexibility for the case that multiple older versions of the DSA implementation had to be tested, for example if the recent version hadn't contained any detectable bugs. But as Chapter 4 will show this has not even been necessary.

Nonetheless there have been some difficulties showing up during the process of the integration of the contracts, these will be described in Section 3.2. Section 3.3 gives an overview of contracts that are incomplete, i.e., listing all methods where the contracts are not able to fully capture the effects of the method in regard to the chosen model. To conclude this chapter the last section will cover the overhead that was involved with implementing the contracts, measured in working hours and lines of code.

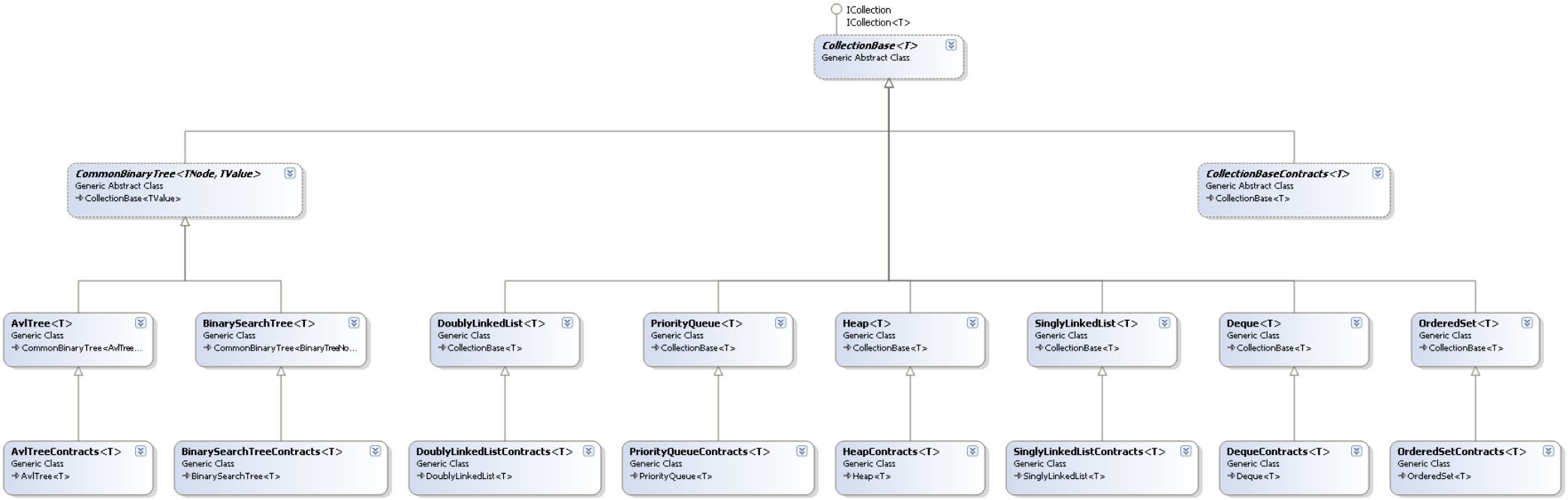


Figure 3.1.: A class diagram showing the MBC-enhanced DSA library

3.2. Contract Development

Whereas the most useful kind of application of model-based contracts and specifications in general would probably be the case where these techniques are involved during the development process right from the beginning, it was clear that the task of applying model-based contracts to software that has not been developed with such concepts in mind would sooner or later bring up some issues.

One of the main problems was to find the right level of abstraction for the model of each class. Since it didn't make sense to completely provide alternate implementations for almost everything, including helper functions, it was clear that the models would have to be build on top of some existing "low level" functions of the DSA implementation. Thus for building the model objects out of the data contained in the target class mostly the functions `GetEnumerator` and `ToArray` were used which are present in all relevant classes of the `DSA.DataStructures` namespace. As a consequence, though, as these functions are involved in creating the model object, they can't have contracts based on these models in their implementation.

In general, during the contract development the main task was to find suitable models for the DSA classes. A suitable model has to be not too implementation-specific but it does also not have to ignore important effects of the target code by being too abstract. As a consequence of finding a compromise between these two extremes leaving some methods with incomplete specifications is often unavoidable. Incomplete contracts are covered in more detail during the next section, while Table 3.1 gives a short summary of all target classes along with their models.[1]

As already mentioned there was the idea of separating the contract from the implementation, and this caused the contracts to be placed in an inconsistent way throughout the DSA project. As can be seen in Figure 3.1 the most common approach chosen for the contracts was to place them in classes derived from the original DSA classes. This was of course only possible for non-abstract classes at the leafs of the inheritance tree. To integrate contracts into abstract classes there were two possibilities:

[1] Note that in this table and throughout the remaining document for readability reasons the original DSA class names are used, although in a strict sense the contract-enhancements are only present in the derived classes (suffixed with "Contracts"). In a similar manner, generic parameters will often be omitted.

Table 3.1.: Contract-enhanced DSA classes and their models

Class	Model
AVLTree	public Map<Sequence<Direction>, TValue> m_Map
BinarySearchTree	public Map<Sequence<Direction>, TValue> m_Map
CollectionBase	public MML.Bag<T> m_Bag
CommonBinaryTree	public Map<Sequence<Direction>, TValue> m_Map
Deque	public MML.Sequence<T> m_Sequence
DoublyLinkedList	public MML.Sequence<T> m_Sequence
Heap	public MML.Sequence<T> m_Sequence public MML.Bag<T> m_Bag
OrderedSet	public MML.Sequence<T> m_Sequence
PriorityQueue	public MML.Sequence<T> m_Sequence
SinglyLinkedList	public MML.Sequence<T> m_Sequence

- Providing a `Contract Class` using the `ContractClassFor` Attribute:

  This was done in case of the `CollectionBase` class, which contained mostly abstract functions. The class `CollectionBaseContracts` is therefore the `Contract Class` containing contracts for the abstract functions declared in `CollectionBase`.

- Writing the contracts directly into the implementation code:

  Since the first approach only works for abstract functions, the contracts for the `CommonBinaryTree` class, containing a lot of non-abstract functions while being an interior node in the inheritance tree, had to be written directly into the class implementation.

Another problem was the inconsistent implementation of the `GetEnumerator` function in various DSA classes. For example, the enumerator returned by the `AVLTree` and `BinarySearchTree` objects requires a call to the `Reset` method before it can be properly used, whereas the enumerator returned by the `DoublyLinkedList` class for example does not need such a call, and even throws an exception when the `Reset` function is called. This required that the construction of the `m_Bag` model query had to be specialized in classes derived from `CollectionBase`, although the query itself belongs to the most abstract level in the inheritance tree.

3.3. Incomplete Contracts

According to [1] a contract is incomplete if it does not fully capture the effect of a method in regard to the chosen model an the return value of a function. If, like in the DSA case, contracts are applied to a piece of software that was initially developed without such concepts in mind, a certain amount of contracts will most likely have to remain incomplete.
Yet, even incomplete contracts can still be very useful. Comparing the figures in this section with the results presented in Chapter 4 it can be seen that faults were detected automatically even in classes where a lot of contracts are incomplete. Below is a list of all methods with incomplete contracts[2]:

- In class `DoublyLinkedList`:
  - `public new DoublyLinkedListNode<T> Head`
  - `public new DoublyLinkedListNode<T> Tail`
- In class `SinglyLinkedList`:
  - `public new SinglyLinkedListNode<T> Head`
  - `public new SinglyLinkedListNode<T> Tail`
- In class `Heap`:
  - `public Heap(IEnumerable<T> collection)`
  - `public Heap(IEnumerable<T> collection, Strategy strategy)`
  - `public override void Add(T item)`
  - `public override bool Remove(T item)`
- In class `PriorityQueue`:
  - `public PriorityQueue(IEnumerable<T> collection)`
  - `public PriorityQueue(IEnumerable<T> collection, Strategy strategy)`
  - `public override void Add(T item)`
  - `public T Dequeue()`

- In class `CommonBinaryTree`:
  - `public TNode Root`
  - `public TNode FindNode(TValue value)`
  - `public TNode FindParent(TValue value)`
- In class `AvlTree`:
  - `public AvlTree(IEnumerable<T> collection)`
  - `public override void Add(T item)`
  - `public override bool Remove(T item)`
  - `public new int Height(AvlTreeNode<T> node)`
- In class `BinarySearchTree`:
  - `public BinarySearchTree(IEnumerable<T> collection)`
  - `public override void Add(T item)`
  - `public override bool Remove(T item)`

The incomplete contracts in the list and tree classes are basically all related to the fact that these classes publicly expose their node-based implementation: the `Head` and `Tail` nodes in case of the lists as well as the `Root` node for the trees. The models, a `Sequence` of values for the lists and a `Map` of paths to values for the trees, do not capture the underlying node structure, leading to incomplete contracts for some methods. Still, these models have proven to be a good compromise, as can be seen by looking at the test results in Chapter 4.

In case of the heap class the model is a `Sequence` corresponding to the array-based implementation of the heap. An invariant checks if the heap condition is always fulfilled. In addition to that, the `Bag` part of the model checks if the expected items are contained in the heap. However, there can exist two

[2]Unless not noted otherwise, during this section a method listed as having incomplete contracts in a base class (see Figure 3.1) implies that the corresponding method in a derived class also has incomplete contracts. This applies analogously to methods being listed as having no contracts.

or more heap representations of the exact same content (in terms of a `Bag`), both fulfilling the heap condition, where the order of the items (in terms of a `Sequence`) is slightly different. This results in incomplete contracts with respect to the `Sequence` model and it may be indicating that the model of the class should be more abstract. However since the semantics of a heap were represented quite well with the aforementioned model, it was kept. The same facts apply to the `PriorityQueue` class, which has the same models, and basically only wraps `Heap`.

In addition to methods with incomplete contracts there were some methods in the relevant DSA classes left completely without contracts. These functions can generally be divided into two categories:

- Non-relevant methods:

  Some functions were not considered relevant for being enhanced with contracts. The reason for this was mostly that these functions were not really a part of the "data structure related" interface (e.g. methods inherited from C# interfaces and base classes, related to concurrency, etc.). This applies to the following methods which are all part of `CollectionBase`:

  - `public bool IsSynchronized`
  - `bool ICollection<T>.IsReadOnly`
  - `object ICollection.SyncRoot`
  - `public void CopyTo(T[] array, int arrayIndex)`

- Methods involved in model queries:

  As already mentioned in Section 3.2 there exist functions that could not be extended with contracts because they were used for constructing the model object in a model query. If these methods contained contracts involving the same type of model query, a cyclic dependency would occur. The following methods are therefore left without contracts. Again, they are all defined (first) in the `CollectionBase` class:

  - `public int Count`
  - `public abstract T[] ToArray()`
  - `public abstract IEnumerator<T> GetEnumerator()`
  - `IEnumerator IEnumerable.GetEnumerator()`

3.4. Specification Overhead

A very important aspect of reasoning about the practical usefulness of the model-based contracts specification technique is to look at the amount of work the developer has to provide to implement the contracts. In regard to overhead measured in working-time the total time needed for extending the DSA library with model-based contracts was about 50 person-hours. Furthermore there were about additional 8 person-hours needed to learn about the MBC concepts and to familiarize with their practical application. Regarding overhead measured in amount of code, detailed figures are given in Table 3.2.

In this table all classes of the `DSA.DataStructures` namespace that have been enhanced with contracts during this project are listed by name in the first column. The second column specifies the total number of lines of code (LOC) for a given class, whereas the third column describes the part of this of code (in LOC) that is used for contracts. The rest of the columns are rather self-explanatory and describe in detail which pieces of code were added to the DSA library during the contract development phase.
It is important to note, though, that for the overhead values in column two only the LOC containing the actual contracts have been measured, not the additional overhead caused by the decision to create designated classes for the contracts. Because as this decision was purely based on design ideas it had no influence on the functionality of the contracts. As can be seen, the overhead is rather low compared to the overall amount of code in most classes.
It may also be especially useful to compare these values with the figures given in Chapter 4 to get an impression on how much specification overhead was needed to successfully detect faults in certain classes.

Table 3.2.: Specification overhead for the DSA library

Class	LOC overall	LOC contracts	Routines added	Preconditions added	Postconditions added	Invariants added
`AVLTree`	391	46	1	0	17	1
`BinarySearchTree`	213	8	0	0	6	0
`CommonBinaryTree`	536	117	11	2	19	5
`Deque`	231	30	1	0	15	2
`DoublyLinkedList`	458	50	2	2	24	2
`Heap`	390	19	1	0	8	3
`OrderedSet`	158	22	2	0	5	4
`PriorityQueue`	216	30	1	0	11	3
`SinglyLinkedList`	492	53	2	2	26	2
`CollectionBase`	244	68	6	0	12	1
`Total`	3329	443	27	6	143	23

4. Automatic Testing of the MBC-Enhanced DSA Library

4.1. The Pex Testing Tool

The previous chapters describe the long way of manual development work up to the point where the result is a C# library enhanced with over 150 handwritten model-based contracts. Most of this work, though, was done just in order to lay the foundation for the final project step: contract-based automated testing experiments using the Pex testing tool.
Pex is a software by Microsoft Research that provides automatic whitebox testing of .Net software. In contrast to e.g. black box testing approaches, Pex constantly analyzes the execution of the code-under-test to find new test inputs and to cover branches that are yet unexplored. This functionality has been reported to be especially useful in conjunction with contracts, because in this case the contracts will act as "guides" for the code explorations, and of course also as testing oracles[8]. These features make Pex the ideal tool for the purposes of this project, i.e., for providing evidence for the usefulness of model-based contracts in conjunction with automatic testing on the .Net platform. The following section describes how a Pex testing environment was set up as part of the MBC solution in Visual Studio, and how the actual testing experiments were being conducted. Section 4.4 finally summarizes all relevant results extracted from the testing experiments.

4.2. The Testing Project

Pex is a tool that can be used comfortably from within the Visual Studio IDE, and it can basically be used in two different modes: The first mode works without the creation of a dedicated test project, and Pex explorations can be triggered right from the context menu. The second one is the *unit testing mode*, in which Pex offers a lot of more flexibility. During this project the latter mode was used, and so the `DSATest` project was created as a part of the MBC solution by letting Pex generate test classes containing parametrized unit tests

out of the contract-enhanced classes of the `DSA.DataStructures` namespace. In such a test class, a test method for every public function is generated by Pex. These test methods take objects of the "type under test"[1]as parameters, and basically only wrap the respective method calls of these objects.
The most important parts of a typical test class generated by Pex can be seen in Listing 4.1. The `PexClass` attribute denotes this class as a test class, specifying the type of object to test (`AvlTreeContracts`), plus a few limiting parameters for the Pex exploration. The higher these limiting values, the longer Pex will explore a single method of this class before giving up and moving on to the next method. As can be seen in line 20, these values can also be refined for individual methods as parameters of the `PexMethod` attribute. This attribute denotes the function as a test method, and has to be present (with or without parameters) for any method that shall be explored by Pex.
However, to really make use of its potential, Pex needs to be given a little more information. The `PexGenericArguments` attribute in line 6 had to be added to tell Pex which data type it shall use for the generic type parameter of the classes, which in this case means that Pex will only instantiate and test trees of type `AvlTreeContracts<int>`. This attribute was necessary in all test classes to get Pex to work correctly, meaning that all test experiments were limited to collections of `int` values.
Furthermore Pex initially had problems instantiating an appropriate implementation of the `IComparer<int>` interface, so a helper class named `IntComp`, which wraps the default comparer for `int` values, hat to be implemented in order to make Pex work correctly. In a similar way, Pex often had problems to instantiate the collection classes or corresponding node classes themselves. In these cases *factory classes*, taking care of the construction of the "object under test" out of objects of more basic or intrinsic types, had to be added to the project. There exist factory classes for all tested classes, and most classes have been tested with and without their utilization.

Listing 4.1: A part of the `AvlTreeTTest` class

```
1 [PexClass(typeof(AvlTreeContracts<>),
2           MaxBranches = 80000,
3           Timeout = 400,
4           MaxRunsWithoutNewTests = 200)]
5
6 [PexGenericArguments(typeof(int))]
7 [PexUseType(typeof(IntComp))]
```

[1]The type which the test class was generated from

```
[PexUseType(typeof(AvlTreeContracts<int>))]
[PexAllowedException(typeof(NullReferenceException))]

[PexAllowedExceptionFromTypeUnderTest(
                              typeof(InvalidOperationException))]
[PexAllowedExceptionFromTypeUnderTest(
                              typeof(ArgumentException))]

[TestClass]
public partial class AvlTreeTTest
{
    /// <summary>Test stub for Add(!0)</summary>
    [PexMethod(MaxBranches = 160000,
               Timeout = 600,
               MaxRunsWithoutNewTests = 600)]
    public void Add<T>([PexAssumeUnderTest]AvlTreeContracts<T>
        target, T item)
        where T : IComparable<T>
    {
        target.Add(item);
    }
    ...
}
```

4.3. The Testing Procedure

As already denoted in the previous section, the general approach during the testing phase was to run the Pex explorations, observing errors, warnings, and timeout messages, and then refine the Pex attributes, parameters, and limits for the next test iteration. Therefore the values used for the `MaxBranches`, `Timeout`, and `MaxRunsWithoutNewTests` were determined experimentally for each class: by increasing limiting values until either errors were found, or the testing time for a single method was considered infeasible long (e.g. several hours). The main limiting factor in this regard was the `MaxBranches` value, which has been doubled several times for all classes during the testing experiments. It should also be noted that in all cases the test runs were executed multiple times per class even when the parameters remained constant. This

Table 4.1.: Contract-enhanced DSA classes and their models

Class	Max Branches	Testing Time	Faults	Design Issues
`AVLTree`	80000	0:23:05	0	1
`BinarySearchTree`	80000	0:21:03	1	0
`CommonBinaryTree`	120000	1:22:30	0	0
`Deque`	120000	2:25:07	0	0
`DoublyLinkedList`	120000	2:51:05	2	1
`Heap`	1600000	1:01:10	1	0
`OrderedSet`	240000	0:09:44	0	0
`PriorityQueue`	120000	1:05:05	0	0
`SinglyLinkedList`	120000	2:28:22	2	1
`Total`		12:07:11	6	3

was due to the fact that Pex explorations are nondeterministic and that therefore there was no guarantee for finding a (detectable) fault in a single run. Also because of these reasons it is that the figures in column two of Table 4.1 are not representative for the *complete* amount of time that was invested into testing the class, they only represent the accumulated length of the two *final* test runs.

4.4. Testing Results

The overall numbers of detected issues can be seen in columns 4 and 5 of Table 4.1. Issues denoted as *faults* represent clearly undesired program behaviour, which mostly can be fixed by simple corrections in the implementation, while problems classified as *design issues* describe more subtle kinds of issues where the "suspicious" program behaviour was most likely intended by the developers, and is not that easy to change.

Although the classes `OrderedSet`, `Deque`, `CommonBinaryTree`, and `PriorityQueue` have been tested at least as extensively as the rest of the relevant code, there could no problems be detected. In all other tested classes, though, there were issues showing up during the experiments, and throughout the following subsections they will be described in detail and in a uniform way: First the problems will be presented by the Description and Remarks sections, followed by the code of the contract that detected the issue. Additionally, a

path — relative to the 'Documents/TestReports' subdirectory of the project's SVN repository[7] — to the detailed auto-generated HTML test report of the relevant test run will be given.

4.4.1. AvlTree

- **Design Issue** in `Add`:

  Description:

  `AvlTree` allows adding duplicate items as keys in the tree. `BinarySearchTree` disallows this. Still both share a lot of code in the common base class `CommonBinaryTree`. No special code for handling duplicate nodes is available. The comments on `Remove` or `FindParent` functions suggest that every item only appears once. Furthermore there is no comment at all about this fundamentally differing behaviour of the two binary search tree implementations.

  Remarks:

  - This might not be a bug in the strict sense, but it is surely not desirable. Of course this is highly speculative, but such an issue might have been caused by two people implementing the two tree classes independently.

  Failing Contract:

```
// Tree allows no duplicate values
Contract.Invariant(m_Map.ToBag().Domain.ForAll(
                          i => m_Map.ToBag()[i] == 1));
```

  (CommonBinaryTree.cs, line 262)

  Test Report:

  `AvlTree-120801.213241.4124.pex/`

4.4.2. BinarySearchTree

- **Fault** in `Add`:

Description:

The `Count` member gets increased even when an item is already in the tree. `Add` calls `InsertNode`, which only adds an item to the tree if it is not already present, i.e., `BinarySearchTree` does not allow duplicate elements. Still `Add` increases the `Count` in any case. In the end, this leads to the `Count` value not matching the actual amount of nodes in the tree anymore.

Remarks:

- This could easily be fixed by increasing `Count` only if the item was actually added. Therefore this can be considered a fault.

Failing Contract:

```
Contract.Invariant(this.GetType()
                       == typeof(AvlTreeContracts<int>)
                       || Count == m_Bag.Count);
```

(CollectionBaseContracts.cs, line 35)

Test Report:

`BinarySearchTree-Add-120801.182723.5624.pex/`

4.4.3. Heap

- **Fault** in `Remove`:

Description:

Heaps of generic type `<int>` suffer from the misinterpretation of zero-initialized empty array fields as keys of value 0. The Heap is implemented as an array, and initially an array of size 4 is constructed. If it is full, its size will be doubled by the `Add` function. So in any case there is at least one empty field at the end. In C# memory is initialized to 0 by default, and for `Heap<int>` objects the `Remove` function regards these empty fields as keys of value 0. In the end this causes the call to `Remove(0)` to always remove the last element in the heap (regardless of its value), and also to violate the heap condition.

Remarks: -

Failing Contract:

```
// Object state
Contract.Ensures(Contract.Result<bool>() ?
                  Contract.OldValue(m_Bag).Removed(item)
                                                  == m_Bag
                 : Contract.OldValue(m_Bag) == m_Bag);
```

(CollectionBaseContracts.cs, line 169)

Test Report:

`Heap-120802.153211.5196.pex/`

4.4.4. DoublyLinkedList

- **Fault** in `AddAfter`, `AddBefore`:

  Description:

  The `AddAfter` and `AddBefore` functions take a node and an item as parameters. When an arbitrary node (not in the tree) is passed, the item will not be added to the tree. Still the `Count` member is increased at the end of the method.

  Remarks:

  - This could easily be fixed by increasing `Count` only if the item was actually added. Therefore this can be considered a fault.
  - Similar issues apply to the `SinglyLinkedList` class.

  Failing Contract:

```
Contract.Invariant(this.GetType()
                       == typeof(AvlTreeContracts<int>)
                       || Count == m_Bag.Count);
```

(CollectionBaseContracts.cs, line 35)

Test Report:

`DoublyLinkedList-120730.022938.3784.pex/`

- **Design Issue** in `Add`:

  Description:

  The `Next` and `Previous` fields in the `DoublyLinkedListNode` class are publicly writable. Therefore the user can manipulate the list structure at any time without using the `DoublyLinkedList` interface. Because of this the list object "looses track" of the node count.

  Remarks:

  - The generated test only works with the factory method guessed by Pex, not with the manually written one. Because the auto generated method sets the `Head`/`Tail` references which would not be allowed using a normal constructor and public functions.
  - Similar issues apply to the `SinglyLinkedList` class.

  Failing Contract:

```
Contract.Invariant(this.GetType()
                       == typeof(AvlTreeContracts<int>)
                       || Count == m_Bag.Count);
```

  (CollectionBaseContracts.cs, line 35)

  Test Report:

  `DoublyLinkedList-120802.102635.2948.pex/`

4.4.5. SinglyLinkedList

- **Fault** in `AddAfter`, `AddBefore`:

  Description:

  The `AddAfter` and `AddBefore` functions take a node and an item as parameters. When an arbitrary node (not in the tree) is passed, the item will not be added to the tree. Still the `Count` member is increased at the end of the method.

  Remarks:

  - This could easily be fixed by increasing `Count` only if the item was actually added. Therefore this can be considered a fault.

- Similar issues apply to the `DoublyLinkedList` class.

Failing Contract:

```
Contract.Invariant(this.GetType()
                       == typeof(AvlTreeContracts<int>)
                       || Count == m_Bag.Count);
```

(CollectionBaseContracts.cs, line 35)

Test Report:

`SinglyLinkedList-120730.023012.2492.pex/`

- **Design Issue** in `AddLast`:

Description:

The `Next` field in the `SinglyLinkedListNode` class is publicly writable. Therefore the user can manipulate the list structure at any time without using the `SinglyLinkedList` interface. Because of this the list object "looses track" of the node count.

Remarks:

- The generated test only works with the factory method guessed by Pex, not with the manually written one. Because the auto generated method sets the `Head`/`Tail` references which would not be allowed using a normal constructor and public functions.
- Similar issues apply to the `DoublyLinkedList` class.

Failing Contract:

```
Contract.Invariant(this.GetType()
                       == typeof(AvlTreeContracts<int>)
                       || Count == m_Bag.Count);
```

(CollectionBaseContracts.cs, line 35)

Test Report:

`SinglyLinkedList-120801.235505.5376.pex/`

5. Conclusion and Future Work

5.1. Conclusion

During this project it has been shown that the ability of model-based contracts to detect serious faults and design flaws is not limited to code written in the rather academic Eiffel language, but also applies to a more common software development scenario: using C# and .Net.

Furthermore, combining the description of the specification deployment process in Chapter 3, and in this regard especially the overhead measured in working-time and lines of code, with the testing results presented in Chapter 4, it has been demonstrated, although in a rather small scale, that model-based contracts can be a powerful and very efficient specification and verification mechanism — especially when combined with automatic white box testing.

As a positive side effect, the *Mathematical Model Library* has been completely ported to .Net, and has also been extensively used and practically tested during the contract development and testing phases. In the end, this process did even slightly improve the Eiffel version of the library by having revealed a bug that was present in both the C# port and the original implementation.

5.2. Future Work

With the *Mathematical Model Library* now being fully usable, considering the the success of the experiments, the large amount of existing .Net software, and the availability of the sophisticated Pex testing tool, there exists a lot of potential for further experiments on the .Net platform to demonstrate the capabilities and maybe also to explore the limitations of the model-based contract approach.

As a follow-up to this project there will similar experiments be conducted targeting the .Net collection classes, but there could also lie some potential in trying to apply model-based-contracts to software other than collection-like datastructures. For example, with the QuickGraph[9] library being available for C# one could possibly try to model graph data structures using the MML

library, and as a more long-term perspective it could even be considered to determine the usefulness of the MBC-approach for the specification and verification of application software.

A. API Documentation for the C# MML Port

The following documentation of the .Net version of the MML was automatically generated using XML comments in the C# code. It is also available as a hyperlinked PDF file and as an HTML version in the '`Documents/Documentation/`' subdirectory of the SVN project repository[7].

MML - C# Version

Generated by Doxygen 1.8.2

Thu Nov 1 2012

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 MML.Bag< T > Class Template Reference

Public Member Functions

- Bag ()
- Bag (T x)
- Bag (T x, int n)
- bool Has (T x)
- bool IsEmpty ()
- bool IsConstant (int c)
- int Occurrences (T x)
- override bool Equals (object obj)
- override int **GetHashCode** ()
- Bag< T > Extended (T x)
- Bag< T > ExtendedMultiple (T x, int n)
- Bag< T > Removed (T x)
- Bag< T > RemovedMultiple (T x, int n)
- Bag< T > RemovedAll (T x)
- Bag< T > Restricted (Set< T > subdomain)
- Bag< T > Union (Bag< T > other)
- Bag< T > Difference (Bag< T > other)

Static Public Member Functions

- static bool operator== (Bag< T > b1, Bag< T > b2)
- static bool operator!= (Bag< T > b1, Bag< T > b2)
- static Bag< T > operator& (Bag< T > b1, T t)
- static Bag< T > operator/ (Bag< T > b1, T t)
- static Bag< T > operator| (Bag< T > b, Set< T > s)
- static Bag< T > **operator+** (Bag< T > b1, Bag< T > b2)
- static Bag< T > **operator-** (Bag< T > b1, Bag< T > b2)

Properties

- Set< T > Domain `[get]`
- int this[T x] `[get]`
- int Count `[get]`

3.1.1 Detailed Description

Finite bag

Template Parameters

T	Type of bag elements

3.1.2 Constructor & Destructor Documentation

3.1.2.1 MML.Bag< T >.Bag ( )

Create an empty bag

3.1.2.2 MML.Bag< T >.Bag (T *x*)

Create a singleton bag

Parameters

x	The only element in the bag

3.1.2.3 MML.Bag< T >.Bag (T *x*, int *n*)

Create a bag with multiple occurrences of the same element

Parameters

x	The element of the bag
n	Number of occurrences

3.1.3 Member Function Documentation

3.1.3.1 Bag<T> MML.Bag< T >.Difference (Bag< T > *other*)

This bag with all occurrences of values from *other* removed

3.1.3.2 override bool MML.Bag< T >.Equals (object *obj*)

Is *obj* a bag with the same elements?

3.1.3.3 Bag<T> MML.Bag< T >.Extended (T *x*)

Current bag extended with one occurrence of *x*

3.1.3.4 Bag<T> MML.Bag< T >.ExtendedMultiple (T *x*, int *n*)

Current bag extended with *n* occurrences of *x*

3.1.3.5 bool MML.Bag< T >.Has (T *x*)

Is *x* contained?

3.1.3.6 bool MML.Bag< T >.IsConstant (int *c*)

Are all values equal to *c* ?

3.1.3.7 bool MML.Bag< T >.IsEmpty ( )

Is bag empty?

Returns

3.1.3.8 int MML.Bag< T >.Occurrences (T *x*)

How many times *x* appears

3.1.3.9 static bool MML.Bag< T >.operator!= (Bag< T > *b1*, Bag< T > *b2*) [static]

Are the two objects not equal?

3.1.3.10 static Bag<T> MML.Bag< T >.operator& (Bag< T > *b1*, T *t*) [static]

Operator equivalent for extended operation

See Also

Extended(T)

>

3.1.3.11 static Bag<T> MML.Bag< T >.operator/ (Bag< T > *b1*, T *t*) [static]

Operator equivalent for removed operation

See Also

Removed(T)

>

3.1.3.12 static bool MML.Bag< T >.operator== (Bag< T > *b1*, Bag< T > *b2*) [static]

Are the two objects equal? (Operator is equivalent to the |=| alias in Eiffel)

3.1.3.13 static Bag<T> MML.Bag< T >.operator| (Bag< T > *b*, Set< T > *s*) [static]

Operator equivalent for restricted operation

See Also

Restricted(Set<T>)

>

3.1.3.14 Bag<T> MML.Bag< T >.Removed (T *x*)

Current bag with one occurrence of *x* removed if contained

3.1.3.15 Bag<T> MML.Bag< T >.RemovedAll (T *x*)

Current bag with all occurrences of *x* removed, if contained

3.1.3.16 Bag<T> MML.Bag< T >.RemovedMultiple (T *x*, int *n*)

Current bag with at most *n* occurrence of *x* removed if contained

3.1.3.17 Bag<T> MML.Bag< T >.Restricted (Set< T > *subdomain*)

Bag that consists of all elements of this bag that are in *subdomain* .

3.1.3.18 Bag<T> MML.Bag< T >.Union (Bag< T > *other*)

Bag that contains all elements from this bag and *other*

3.1.4 Property Documentation

3.1.4.1 int MML.Bag< T >.Count [get]

Total number of elements

3.1.4.2 Set<T> MML.Bag< T >.Domain [get]

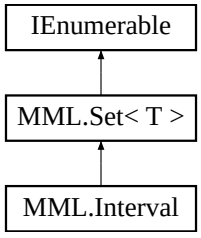
Set of values that occur at least once

3.1.4.3 int MML.Bag< T >.this[T x] [get]

Indexer alias for Occurrences

3.2 MML.Interval Class Reference

Inheritance diagram for MML.Interval:



Public Member Functions

- Interval (int l, int u)
- Interval (Tuple< int, int > tuple)
- int Lower ()
- int Upper ()

Additional Inherited Members

3.2.1 Detailed Description

Closed integer intervals

3.2.2 Constructor & Destructor Documentation

3.2.2.1 MML.Interval.Interval (int *l*, int *u*)

Create interval [*l* , *u*]

3.2.2.2 MML.Interval.Interval (Tuple< int, int > *tuple*)

Create interval from tuple[*tuple*]

3.2.3 Member Function Documentation

3.2.3.1 int MML.Interval.Lower ( )

Lower bound

3.2.3.2 int MML.Interval.Upper ( )

Upper bound

3.3 MML.Map< K, V > Class Template Reference

Public Member Functions

- Map ()
- Map (K key, V value)
- bool Has (V x)
- bool Has (K x)
- bool IsEmpty ()
- bool IsConstant (V c)
- V Item (K k)
- Set< V > Image (Set< K > subdomain)
- Sequence< V > SequenceImage (Sequence< K > s)
- Bag< V > ToBag ()
- override bool Equals (object obj)
- override int **GetHashCode** ()
- Map< K, V > Updated (K k, V v)
- Map< K, V > Removed (K k)

- Map< K, V > Restricted (Set< K > subdomain)
- Map< K, V > Override (Map< K, V > other)
- Relation< V, K > Inverse ()

Static Public Member Functions

- static bool operator== (Map< K, V > m1, Map< K, V > m2)
- static bool operator!= (Map< K, V > m1, Map< K, V > m2)
- static Map< K, V > operator| (Map< K, V > m, Set< K > s)
- static Map< K, V > operator+ (Map< K, V > m1, Map< K, V > m2)

Properties

- V this[K k] `[get]`
- Set< K > Domain `[get]`
- Set< V > Range `[get]`
- int Count `[get]`

3.3.1 Detailed Description

Finite map

Template Parameters

K	Type of key elements
V	Type of value elements

3.3.2 Constructor & Destructor Documentation

3.3.2.1 MML.Map< K, V >.Map ( )

Create an empty map

3.3.2.2 MML.Map< K, V >.Map (K *key*, V *value*)

Create a singleton map

Parameters

key	The only key in the map
value	The only value in the map

3.3.3 Member Function Documentation

3.3.3.1 override bool MML.Map< K, V >.Equals (object *obj*)

Is *obj* a map with the same key-value pairs?

3.3.3.2 bool MML.Map< K, V >.Has (V *x*)

Does map contain value *x*

3.3.3.3 bool MML.Map< K, V >.Has (K *x*)

Does map contain key *x*

3.3.3.4 Set<V> MML.Map< K, V >.Image (Set< K > *subdomain*)

Set of values corresponding to keys in *subdomain*

3.3.3.5 Relation<V, K> MML.Map< K, V >.Inverse ( )

Relation consisting of inverted pairs from this map

3.3.3.6 bool MML.Map< K, V >.IsConstant (V *c*)

Are all values equal to *c* ?

3.3.3.7 bool MML.Map< K, V >.IsEmpty ( )

Is map empty?

3.3.3.8 V MML.Map< K, V >.Item (K *k*)

Value associated with *k*

3.3.3.9 static bool MML.Map< K, V >.operator!= (Map< K, V > *m1,* Map< K, V > *m2*) [static]

Are the two objects not equal?

3.3.3.10 static Map<K, V> MML.Map< K, V >.operator+ (Map< K, V > *m1,* Map< K, V > *m2*) [static]

Operator equivalent for override operation

See Also

Override(Map<K, V>)

>

3.3.3.11 static bool MML.Map< K, V >.operator== (Map< K, V > *m1,* Map< K, V > *m2*) [static]

Are the two objects equal? (Operator is equivalent to the |=| alias in Eiffel)

3.3.3.12 static Map<K, V> MML.Map< K, V >.operator| (Map< K, V > *m,* Set< K > *s*) [static]

Operator equivalent for restricted operation

See Also

Restricted(Set<K>)

>

3.3.3.13 Map<K, V> MML.Map< K, V >.Override (Map< K, V > *other*)

Map that is equal to *other* on its domain and to this map on its domain minus the domain of *other*

3.3.3.14 Map<K, V> MML.Map< K, V >.Removed (K *k*)

Current map without key *k* and the corresponding value

3.3.3.15 Map<K, V> MML.Map< K, V >.Restricted (Set< K > *subdomain*)

Map that consists of all key-value pairs of this map whose key is in *subdomain*

3.3.3.16 Sequence<V> MML.Map< K, V >.SequenceImage (Sequence< K > *s*)

Sequence of images of *s* elements under this map

3.3.3.17 Bag<V> MML.Map< K, V >.ToBag ( )

Bag of map values

3.3.3.18 Map<K, V> MML.Map< K, V >.Updated (K *k*, V *v*)

Current map with *v* associated with *k* . If *k* already exists, the value is replaced, otherwise added.

3.3.4 Property Documentation

3.3.4.1 int MML.Map< K, V >.Count [get]

Map cardinality

3.3.4.2 Set<K> MML.Map< K, V >.Domain [get]

Set of keys

3.3.4.3 Set<V> MML.Map< K, V >.Range [get]

Set of values

3.3.4.4 V MML.Map< K, V >.this[K k] [get]

Indexer alias for Item

3.4 MML.Relation< L, R > Class Template Reference

Public Member Functions

- Relation ()
- Relation (L left, R right)
- bool Has (L x, R y)

- bool IsEmpty ()
- Set< R > ImageOf (L x)
- Set< R > Image (Set< L > subdomain)
- override bool Equals (object obj)
- override int **GetHashCode** ()
- Relation< L, R > Extended (L x, R y)
- Relation< L, R > Removed (L x, R y)
- Relation< L, R > Restricted (Set< L > subdomain)
- Relation< R, L > Inverse ()
- Relation< L, R > Union (Relation< L, R > other)
- Relation< L, R > Intersection (Relation< L, R > other)
- Relation< L, R > Difference (Relation< L, R > other)
- Relation< L, R > SymDifference (Relation< L, R > other)

Static Public Member Functions

- static bool operator== (Relation< L, R > r1, Relation< L, R > r2)
- static bool operator!= (Relation< L, R > r1, Relation< L, R > r2)
- static Relation< L, R > operator| (Relation< L, R > r, Set< L > s)
- static Relation< L, R > operator+ (Relation< L, R > r1, Relation< L, R > r2)
- static Relation< L, R > operator* (Relation< L, R > r1, Relation< L, R > r2)
- static Relation< L, R > operator- (Relation< L, R > r1, Relation< L, R > r2)
- static Relation< L, R > operator^ (Relation< L, R > r1, Relation< L, R > r2)

Properties

- bool this[L x, R y] `[get]`
- Set< L > Domain `[get]`
- Set< R > Range `[get]`
- int Count `[get]`

3.4.1 Detailed Description

Finite relation

Template Parameters

L	Domain type of the relation
R	Range type of the relation

3.4.2 Constructor & Destructor Documentation

3.4.2.1 MML.Relation< L, R >.Relation ( )

Create an empty relation

3.4.2.2 MML.Relation< L, R >.Relation (L *left*, R *right*)

Create a singleton relation

Parameters

left	Left component of the only element
right	Right component of the only element

3.4.3 Member Function Documentation

3.4.3.1 Relation<L, R> MML.Relation< L, R >.Difference (Relation< L, R > *other*)

Set of values contained in this relation but not in *other*

3.4.3.2 override bool MML.Relation< L, R >.Equals (object *obj*)

Is *obj* a relation with the same pairs?

3.4.3.3 Relation<L, R> MML.Relation< L, R >.Extended (L *x*, R *y*)

Current relation extended with pair (*x* , *y*) if absent

3.4.3.4 bool MML.Relation< L, R >.Has (L *x*, R *y*)

Is *x* related to *y* ?

3.4.3.5 Set<R> MML.Relation< L, R >.Image (Set< L > *subdomain*)

Set of values related to any value in *subdomain*

3.4.3.6 Set<R> MML.Relation< L, R >.ImageOf (L *x*)

Set of values related to *x*

3.4.3.7 Relation<L, R> MML.Relation< L, R >.Intersection (Relation< L, R > *other*)

Relation that consists of pairs contained in both this relation and *other*

Parameters

other	

Returns

3.4.3.8 Relation<R, L> MML.Relation< L, R >.Inverse ( )

Relation that consists of inverted pairs from this relation

3.4.3.9 bool MML.Relation< L, R >.IsEmpty ( )

Is map empty?

Returns

3.4.3.10 static bool MML.Relation< L, R >.operator!= (Relation< L, R > *r1,* Relation< L, R > *r2*) [static]

Are the two objects not equal?

3.4.3.11 static Relation<L, R> MML.Relation< L, R >.operator* (Relation< L, R > *r1,* Relation< L, R > *r2*) [static]

Operator equivalent for intersection operation

See Also

Intersection(Relation<L, R>)

>

3.4.3.12 static Relation<L, R> MML.Relation< L, R >.operator+ (Relation< L, R > *r1,* Relation< L, R > *r2*) [static]

Operator equivalent for union operation

See Also

Union(Relation<L, R>)

>

3.4.3.13 static Relation<L, R> MML.Relation< L, R >.operator- (Relation< L, R > *r1,* Relation< L, R > *r2*) [static]

Operator equivalent for difference operation

See Also

Difference(Relation<L, R>)

>

3.4.3.14 static bool MML.Relation< L, R >.operator== (Relation< L, R > *r1,* Relation< L, R > *r2*) [static]

Are the two objects equal? (Operator is equivalent to the |=| alias in Eiffel)

3.4.3.15 static Relation<L, R> MML.Relation< L, R >.operator^ (Relation< L, R > *r1,* Relation< L, R > *r2*) [static]

Operator equivalent for symmetric difference operation

See Also

SymDifference(Relation<L, R>)

>

3.4.3.16 static Relation<L, R> MML.Relation< L, R >.operator| (Relation< L, R > *r,* Set< L > *s*) [static]

Operator equivalent for restricted operation

See Also

Restricted(Set<L>)

>

3.4.3.17 Relation<L, R> MML.Relation< L, R >.Removed (L *x,* R *y*)

Current relation with pair (*x* , *y*) removed if present

3.4.3.18 Relation<L, R> MML.Relation< L, R >.Restricted (Set< L > *subdomain*)

Relation that consists of all pairs in this relation whose left component is in *subdomain* .

3.4.3.19 Relation<L, R> MML.Relation< L, R >.SymDifference (Relation< L, R > *other*)

Relation that consists of pairs contained in either this relation or *other* , but not in both

3.4.3.20 Relation<L, R> MML.Relation< L, R >.Union (Relation< L, R > *other*)

Relation that consists of pairs contained in either this relation or *other*

3.4.4 Property Documentation

3.4.4.1 int MML.Relation< L, R >.Count [get]

Cardinality

3.4.4.2 Set<L> MML.Relation< L, R >.Domain [get]

The set of left components

3.4.4.3 Set<R> MML.Relation< L, R >.Range [get]

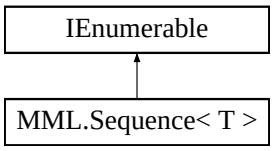
The set of right components

3.4.4.4 bool MML.Relation< L, R >.this[L x, R y] [get]

Indexer alias for Has

3.5 MML.Sequence< T > Class Template Reference

Inheritance diagram for MML.Sequence< T >:



Public Member Functions

- System.Collections.IEnumerator **GetEnumerator** ()
- Sequence ()
- Sequence (T x)
- bool Has (T x)
- bool IsEmpty ()
- bool IsConstant (T c)
- T Item (int i)
- Bag< T > ToBag ()
- int Occurrences (T x)
- override bool Equals (object obj)
- override int **GetHashCode** ()
- bool IsPrefixOf (Sequence< T > other)
- T First ()
- T Last ()
- Sequence< T > ButFirst ()
- Sequence< T > ButLast ()
- Sequence< T > Front (int upper)
- Sequence< T > Tail (int lower)
- Sequence< T > Interval (int lower, int upper)
- Sequence< T > RemovedAt (int i)
- Sequence< T > Restricted (Set< int > subdomain)
- Sequence< T > Removed (Set< int > subdomain)
- Sequence< T > Extended (T x)
- Sequence< T > ExtendedAt (int i, T x)
- Sequence< T > Prepended (T x)
- Sequence< T > Concatenation (Sequence< T > other)
- Sequence< T > ReplacedAt (int i, T x)
- Relation< T, int > Inverse ()
- Sequence< T > Reverse ()

Static Public Member Functions

- static bool operator== (Sequence< T > s1, Sequence< T > s2)
- static bool operator!= (Sequence< T > s1, Sequence< T > s2)
- static Sequence< T > operator& (Sequence< T > s1, T t)
- static Sequence< T > operator+ (Sequence< T > s1, Sequence< T > s2)

Properties

- T this[int i] `[get]`
- Interval Domain `[get]`
- Set< T > Range `[get]`
- int Count `[get]`

3.5.1 Detailed Description

Finite sequence. Indexing starts from 0.

Template Parameters

T	Type of sequence elements

3.5.2 Constructor & Destructor Documentation

3.5.2.1 MML.Sequence< T >.Sequence ( )

Create an empty sequence

3.5.2.2 MML.Sequence< T >.Sequence (T *x*)

Create a singleton sequence

Parameters

x	The only element in the sequence

3.5.3 Member Function Documentation

3.5.3.1 Sequence<T> MML.Sequence< T >.ButFirst ( )

This sequence without the first element

3.5.3.2 Sequence<T> MML.Sequence< T >.ButLast ( )

This sequence without the last element

3.5.3.3 Sequence<T> MML.Sequence< T >.Concatenation (Sequence< T > *other*)

The concatenation of the this sequence and *other*

3.5.3.4 override bool MML.Sequence< T >.Equals (object *obj*)

Is *obj* a set with the same elements in the same order?

3.5.3.5 Sequence<T> MML.Sequence< T >.Extended (T *x*)

Current sequence extended with *x* at the end

3.5.3.6 T MML.Sequence< T >.First ( )

First element

3.5.3.7 Sequence<T> MML.Sequence< T >.Front (int *upper*)

Prefix up to *upper*

3.5.3.8 bool MML.Sequence< T >.Has (T *x*)

Is *x* contained?

3.5.3.9 Sequence<T> MML.Sequence< T >.Interval (int *lower,* int *upper*)

Subsequence from *lower* to *upper* .

3.5.3.10 Relation<T, int> MML.Sequence< T >.Inverse ( )

Relation of values in this sequence to their indexes

3.5.3.11 bool MML.Sequence< T >.IsConstant (T *c*)

Are all values equal to *c* ?

3.5.3.12 bool MML.Sequence< T >.IsEmpty ( )

Is the sequence empty?

3.5.3.13 bool MML.Sequence< T >.IsPrefixOf (Sequence< T > *other*)

Is this sequence prefix of *other* ?

3.5.3.14 T MML.Sequence< T >.Item (int *i*)

Value at position *i*

3.5.3.15 T MML.Sequence< T >.Last ( )

Last element

3.5.3.16 int MML.Sequence< T >.Occurrences (T *x*)

How many times does *x* occur

3.5.3.17 static bool MML.Sequence< T >.operator!= (Sequence< T > *s1,* Sequence< T > *s2*) [static]

Are the two objects not equal?

3.5.3.18 static Sequence<T> MML.Sequence< T >.operator& (Sequence< T > *s1,* T *t*) [static]

Operator equivalent for extended operation

See Also

Extended(T)

>

3.5.3.19 static Sequence<T> MML.Sequence< T >.operator+ (Sequence< T > *s1,* Sequence< T > *s2*) [static]

Operator equivalent for concatenation operation

See Also

Concatenation (Sequence<T>)

>

3.5.3.20 static bool MML.Sequence< T >.operator== (Sequence< T > *s1,* Sequence< T > *s2*) [static]

Are the two objects equal? (Operator is equivalent to the |=| alias in Eiffel)

3.5.3.21 Sequence<T> MML.Sequence< T >.Prepended (T *x*)

Current sequence prepended with *x* at the beginning

3.5.3.22 Sequence<T> MML.Sequence< T >.Removed (Set< int > *subdomain*)

Current sequence with all elements with indexes from *subdomain* removed.

3.5.3.23 Sequence<T> MML.Sequence< T >.RemovedAt (int *i*)

This sequence with element at position *i* removed

3.5.3.24 Sequence<T> MML.Sequence< T >.ReplacedAt (int *i,* T *x*)

This sequence with *x* at position *i*

3.5.3.25 Sequence<T> MML.Sequence< T >.Restricted (Set< int > *subdomain*)

Current sequence with all elements with indexes outside of *subdomain* removed.

3.5.3.26 Sequence<T> MML.Sequence< T >.Reverse ( )

Reverses the order of the elements in a Sequence.

3.5.3.27 Sequence<T> MML.Sequence< T >.Tail (int *lower*)

Suffix from *lower*

3.5.3.28 Bag<T> MML.Sequence< T >.ToBag ( )

Bag of sequence values

Returns

3.5.4 Property Documentation

3.5.4.1 int MML.Sequence< T >.Count [get]

Number of elements

3.5.4.2 Interval MML.Sequence< T >.Domain [get]

Set of indexes

3.5.4.3 Set<T> MML.Sequence< T >.Range [get]

Set of values

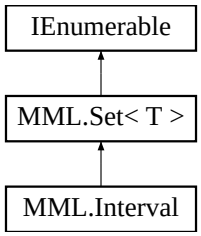
Returns

3.5.4.4 T MML.Sequence< T >.this[int i] [get]

Indexer alias for Item

3.6 MML.Set< T > Class Template Reference

Inheritance diagram for MML.Set< T >:



Public Member Functions

- System.Collections.IEnumerator **GetEnumerator** ()
- Set ()
- Set (T x)
- bool Has (T x)
- bool IsEmpty ()
- bool ForAll (Predicate< T > test)
- bool Exists (Predicate< T > test)
- T AnyItem ()
- delegate bool **OrderType** (T x, T y)
- T Extremum (OrderType order)
- Set< T > Filtered (Predicate< T > test)
- override bool Equals (object obj)
- override int **GetHashCode** ()
- bool IsSubsetOf (Set< T > other)
- bool IsSupersetOf (Set< T > other)

- bool Disjoint (Set< T > other)
- Set< T > Extended (T x)
- Set< T > Removed (T x)
- Set< T > Union (Set< T > other)
- Set< T > Intersection (Set< T > other)
- Set< T > Difference (Set< T > other)
- Set< T > SymDifference (Set< T > other)

Static Public Member Functions

- static Set< T > operator| (Set< T > s1, Predicate< T > p)
- static bool operator== (Set< T > s1, Set< T > s2)
- static bool operator!= (Set< T > s1, Set< T > s2)
- static bool operator<= (Set< T > s1, Set< T > s2)
- static bool operator>= (Set< T > s1, Set< T > s2)
- static Set< T > operator& (Set< T > s1, T t)
- static Set< T > operator/ (Set< T > s1, T t)
- static Set< T > operator+ (Set< T > s1, Set< T > s2)
- static Set< T > operator* (Set< T > s1, Set< T > s2)
- static Set< T > operator- (Set< T > s1, Set< T > s2)
- static Set< T > operator^ (Set< T > s1, Set< T > s2)

Properties

- bool this[T x] [get]
- int Count [get]

3.6.1 Detailed Description

Finite set

Template Parameters

T	Type of set elements

3.6.2 Constructor & Destructor Documentation

3.6.2.1 MML.Set< T >.Set ( )

Create an empty set

3.6.2.2 MML.Set< T >.Set (T *x*)

Create a singleton set

Parameters

x	The only element of the set

3.6.3 Member Function Documentation

3.6.3.1 T MML.Set< T >.AnyItem ( )

Arbitrary element

3.6.3.2 Set<T> MML.Set< T >.Difference (Set< T > *other*)

Set of values contained in this set but not in *other*

3.6.3.3 bool MML.Set< T >.Disjoint (Set< T > *other*)

Do no elements of *other* occur in this set?

3.6.3.4 override bool MML.Set< T >.Equals (object *obj*)

Is *obj* a set with the same elements?

3.6.3.5 bool MML.Set< T >.Exists (Predicate< T > *test*)

Does *test* hold for at least one element?

3.6.3.6 Set<T> MML.Set< T >.Extended (T *x*)

This set extended with *x* if absent

3.6.3.7 T MML.Set< T >.Extremum (OrderType *order*)

Least element with respect to *order* .

3.6.3.8 Set<T> MML.Set< T >.Filtered (Predicate< T > *test*)

Set of all elements that satisfy *test*

3.6.3.9 bool MML.Set< T >.ForAll (Predicate< T > *test*)

Does *test* hold for all elements?

3.6.3.10 bool MML.Set< T >.Has (T *x*)

Is *x* contained?

3.6.3.11 Set<T> MML.Set< T >.Intersection (Set< T > *other*)

Set of values contained in both this set and *other*

3.6.3.12 bool MML.Set< T >.IsEmpty ( )

Is the set empty?

3.6.3.13 bool MML.Set< T >.IsSubsetOf (Set< T > *other*)

Does *other* have all elements of this set?

3.6.3.14 bool MML.Set< T >.IsSupersetOf (Set< T > *other*)

Does this set have all elements of *other* ?

3.6.3.15 static bool MML.Set< T >.operator!= (Set< T > *s1*, Set< T > *s2*) [static]

Are the two objects not equal?

3.6.3.16 static Set<T> MML.Set< T >.operator& (Set< T > *s1*, T *t*) [static]

Operator equivalent for extended operation

See Also

Extended(T)

>

3.6.3.17 static Set<T> MML.Set< T >.operator* (Set< T > *s1*, Set< T > *s2*) [static]

Operator equivalent for intersection operation

See Also

Intersection(Set<T>)

>

3.6.3.18 static Set<T> MML.Set< T >.operator+ (Set< T > *s1*, Set< T > *s2*) [static]

Operator equivalent for union operation

See Also

Union(Set<T>)

>

3.6.3.19 static Set<T> MML.Set< T >.operator- (Set< T > *s1*, Set< T > *s2*) [static]

Operator equivalent for difference operation

See Also

Difference(Set<T>)

>

3.6.3.20 static Set<T> MML.Set< T >.operator/ (Set< T > *s1*, T *t*) [static]

Operator equivalent for removed operation

See Also

Removed(T)

>

3.6.3.21 static bool MML.Set< T >.operator<= (Set< T > *s1*, Set< T > *s2*) [static]

Operator equivalent for IsSubsetOf operation

See Also

IsSubsetOf(Set<T>)

>

3.6.3.22 static bool MML.Set< T >.operator== (Set< T > *s1*, Set< T > *s2*) [static]

Are the two objects equal? (Operator is equivalent to the |=| alias in Eiffel)

3.6.3.23 static bool MML.Set< T >.operator>= (Set< T > *s1*, Set< T > *s2*) [static]

Operator equivalent for IsSupersetOf operation

See Also

IsSupersetOf(Set<T>)

>

3.6.3.24 static Set<T> MML.Set< T >.operator^ (Set< T > *s1*, Set< T > *s2*) [static]

Operator equivalent for symmetric difference operation

See Also

SymDifference(Set<T>)

>

3.6.3.25 static Set<T> MML.Set< T >.operator| (Set< T > *s1*, Predicate< T > *p*) [static]

Operator equivalent for filtered operation

See Also

Filtered(Predicate<T>)

>

3.6.3.26 Set<T> MML.Set< T >.Removed (T *x*)

Current set with *x* removed if present

3.6.3.27 Set<T> MML.Set< T >.SymDifference (Set< T > *other*)

Set of values contained in either this set or *other* , but not in both

3.6.3.28 Set<T> MML.Set< T >.Union (Set< T > *other*)

Set of values contained in either this set or *other*

3.6.4 Property Documentation

3.6.4.1 int MML.Set< T >.Count [get]

Cardinality

3.6.4.2 bool MML.Set< T >.this[T x] [get]

Indexer alias for Has

Index

Bibliography

[1] N. Polikarpova, C. A. Furia, and B. Meyer, "Specifying Reusable Components," in *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, G. T. Leavens, P. O'Hearn, and S. Rajamani, eds., vol. 6217 of *Lecture Notes in Computer Science*, pp. 127–141. Springer, August, 2010.

[2] T. Widmer, "Reusable Mathematical Models," Master's thesis, ETH Zurich, 2004. `http://se.inf.ethz.ch/old/projects/tobias_widmer/Thesis-Widmer2004.pdf`.

[3] `http://research.microsoft.com/en-us/projects/contracts/`. [Online; accessed 30-October-2012].

[4] `http://dsa.codeplex.com`. [Online; accessed 30-October-2012].

[5] `http://research.microsoft.com/en-us/projects/pex/`. [Online; accessed 30-October-2012].

[6] `http://se.inf.ethz.ch/student_projects/elena_mokhon/`. [Online; accessed 30-October-2012].

[7] `https://code.vis.ethz.ch/svn/mbc`. [Online; accessed 30-October-2012].

[8] M. Barnett, M. Fähndrich, P. de Halleux, F. Logozzo, and N. Tillmann, "Exploiting the synergy between automated-test-generation and programming-by-contract.," in *ICSE Companion*, pp. 401–402. IEEE, 2009. `http://dblp.uni-trier.de/db/conf/icse/icse2009c.html#BarnettFHLT09`.

[9] `http://quickgraph.codeplex.com/`. [Online; accessed 30-October-2012].