Evaluating Larch/C++ as a Specification Language = A Case Study Using the Microsoft Foundation Class Library

David M. Egle
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Programming Languages and Compilers Commons, and the Systems Architecture Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/91

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
Evaluating Larch/C++ as a Specification Language: A Case Study Using the Microsoft Foundation Class Library

David M. Egle

TR #95-17
July 1995

Keywords: Larch/C++, interface specification, MFC, class library, software engineering, object-oriented programming.


© David M. Egle, 1995. Copies may be made for any educational purpose whatever, provided this copyright notice appears on the copy.

This work was supported in part by NSF Grant number CCR-9503168.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
Evaluating Larch/C++ as a Specification Language: A Case Study Using the Microsoft Foundation Class Library

David M. Egle
Department of Computer Science, Atanasoff Hall
Iowa State University, Ames, Iowa 50014 USA
egle@cs.iastate.edu

July 19, 1995

Abstract

This paper attempts to evaluate Larch/C++, a formal specification language, as a means of more unambiguously documenting the interface specifications of C++ class libraries. In particular, the Microsoft Foundation Class Library is examined to demonstrate some of the advantages and disadvantages of using Larch/C++. Several different examples are presented with analysis. The reader is assumed to have some familiarity with C++ and the idea of formal specification, but need not know the Larch approach to formal specification.

Keywords: Larch/C++, interface specification, software engineering, object-oriented programming
Acknowledgements

I would like to express my sincere gratitude to Dr. Gary Leavens for being my advisor and for the guidance he provided throughout my research. He always had his door open and was willing to help me whenever I needed it.

I am also thankful to Dr. Albert Baker and Dr. Charles Wright for being on my committee and for taking an interest in my work.

Finally, I would like to thank all of my friends for their continued support and encouragement. I especially wish to thank Kishore Dhara for his helpful insights and assistance.
# Contents

1 Introduction 4

2 The Need for Formalization 4
  2.1 Motivation for Using the Microsoft Foundation Class Library 6

3 Background on Larch/C++ 7
  3.1 What is Larch/C++? 7
  3.2 Anatomy of a Function Specification 7
  3.3 CSize: An Example Larch/C++ Class Specification 8
  3.4 CSizeTrait: An Example Larch Shared Language Trait 12

4 The Interface Specifications 13
  4.1 Specification of CObject 14
  4.2 Specification of CException and CMemoryException 16
  4.3 Specification of CObArray 16
  4.4 Specification of CObList 17

5 Analysis of the Specification 17
  5.1 Ambiguities in MFC Documentation Revealed 17
  5.2 Inconsistencies in MFC Documentation Revealed 22

6 Advantages and Disadvantages of Larch/C++ 23
  6.1 Disadvantages of Larch/C++ vs. Informal English 23
  6.2 Advantages of Larch/C++ vs. Informal English 24

7 Conclusions 26

8 Appendix A: Traits 28

9 Appendix B: Specifications 36
1 Introduction

Object-oriented programming (OOP) and object-oriented programming languages (OOPL) have gone from being computer science buzzwords to design and implementation standards in the field of software development. Their popularity has risen due to such advanced features as inheritance, polymorphism, and reusability. Of these, reusability may be the most important. In C++, classes provide a way to accomplish all of the above tasks. However, verifying a class's implementation is correct and that the corresponding documentation is precise and unambiguous is vital in the creation of reusable components. One way to achieve this is by using formal specification techniques.

This paper attempts to evaluate Larch/C++, a formal specification language, as a means of more unambiguously documenting the interface specifications of C++ class libraries. In particular, the Microsoft (R)1 Foundation Class Library (MFC) is used to demonstrate some of the advantages and disadvantages of using Larch/C++. A variety of different types of classes will be examined from the MFC Library. Several different examples are presented with analysis to show the benefits of Larch/C++, as well as its shortcomings.

This paper is organized as follows. Section 2 introduces the MFC Library and provides some background on the need for formalization in specification techniques. Section 3 explains more about Larch/C++ and the Larch Shared Language, and contains an example specification of an MFC Library class. Section 4 shows how different classes were specified using Larch/C++, and section 5 reveals the results of the specification process. Finally, section 6 discusses some of the advantages and disadvantages of using Larch/C++ as a specification language.

2 The Need for Formalization

Developing software is an expensive process, but maintaining it can be even more expensive. A study conducted by the National Bureau of Standards estimated that 60 to 85 percent of the total cost of software is due to maintenance [3]. This is because mistakes not found during operational testing

---

1 Microsoft is a registered trademark of Microsoft Corporation
may need to be fixed at the customer's site, which means every customer becomes a special consideration.

Developers need to build application software quickly and efficiently. This implies the development time should be as short as possible, and that means less time spent programming. If programs can reuse pieces of code (parts of other programs), especially those that have been proven to work correctly, and whose functionality is well understood, then they can be used in the development of new software. To reuse components (implemented as classes in OOP) effectively, a programmer must be able to read and understand the code (if provided) and/or the documentation (if provided) of the software intended for reuse. More than likely, the software will have been written by a different programmer or group of programmers. Many programmers can attest that poor or missing documentation can make this a frustrating and difficult task. The exact effects of invoking a particular function could easily be answered if there was a standard, precise, and accurate way to document the behavior of the function.

In an effort to facilitate reuse, software engineers have been examining formal specification techniques. Such techniques help document the behavior of the code so that programmers will be less confused about the workings of the original code. With more formal specification, ambiguities in the functionality of specific code can be significantly reduced. Such techniques could also aid in proving code is designed and implemented accurately and precisely with respect to the requirements. For example, within the English language, there is often more than one way to interpret a single statement. This is shown through the following example taken from [9]. Three subjects were given the following problem statement.

\[ Construct \ a \ means \ for \ protecting \ a \ small \ group \ of \ human \ beings \ from \ the \ hostile \ elements \ of \ their \ environment. \]

When they had finished constructing their shelter, they examined each other's results and were amazed at the differences. One subject had constructed an igloo, another a castle, and the third a space station. However, each subject had fulfilled the requirements stated in the problem. As informal programming comments are derived from spoken English, it follows that these comments can also have many interpretations. This also gives rise to the problem of assumption. What may seem obvious to one person
may not be to someone else. In the example above, the presenter of the statement may have assumed three different (but similar in style) houses would be built. In programming, what may seem obvious to a C++ class programmer may not be so obvious to a user of the C++ class. For these reasons, programmers are interested in precise ways of documenting classes, formally and unambiguously. Given a specification, a programmer should be able to use the functions without having any knowledge of how the functions are implemented. From another viewpoint, a programmer can implement the functions for some software component without ever knowing how the functions will be used, provided the specifications are adhered to.

2.1 Motivation for Using the Microsoft Foundation Class Library

As the focus of this paper is the evaluation of Larch/C++, a set of C++ classes was needed to demonstrate some of the advantages and disadvantages of the language. As of the writing of this paper, no publicly available Larch/C++ specifications of commercial software were known. Hence, this work is important in the evaluation of Larch/C++.

The MFC Library provides software developers with a set of reusable, C++ components with a detailed interface for applications on the Windows and Windows NT operating systems, UNIX platforms, and the Apple Macintosh. Among the components are such common window-based application features such as scroll bars, form and edit views, print and print previews, dialog boxes, and context-sensitive help. Recent research performed by G. Bhalla & Associates, an independent national research firm, indicates that MFC is the most widely used class library among developers who use class libraries [8]. The survey was conducted by examining developers in the United States, and showed that 58 percent of C/C++ developers use a class library and that three times as many developers used the MFC Library as any other class library. In addition, Microsoft is quickly becoming an industry leader in commercial software development. Because of these reasons, the MFC Library was chosen to demonstrate and evaluate Larch/C++.
3 Background on Larch/C++

While the focus of this paper is not explaining Larch/C++, a good understanding is necessary before the evaluation can be presented.

3.1 What is Larch/C++?

Larch/C++ is an interface specification language designed to provide a more formal, rigorous, and unambiguous way to document C++ classes, more specifically, the class member functions. The Larch approach to specification technique is two-tiered, with each specification having a component on each tier. The two tiers are the the Larch Shared Language (LSL), responsible for defining the underlying abstractions, and a Larch interface language, used to specify the state transformations [4]. Two examples of Larch interface languages are Larch/CLU, which specifies CLU program modules, and Larch/C++, which specifies C++ program modules.

An interface specification language, by definition, is designed to specify the behavior of specific constructs in the programming language. Larch/C++ specifies C++ classes by describing the abstract values of instances of the class and the behavior of the operations of the class when invoked. This aspect of Larch/C++ builds on the work of C.A.R. Hoare on axiomatic semantics and correctness of data representations [6]. Larch/C++ is not intended to completely replace informal English documentation, but adds to it by providing clarity and conciseness. For more on the details of Larch/C++, see [1, 7].

3.2 Anatomy of a Function Specification

In Larch/C++, the behavior of a function is specified using a precondition and postcondition. Syntactically, the interface of a function specification is the same as that of the corresponding C++ function definition. The body of the specification describes the behavior of the function. The precondition is a predicate that must be true for the function to guarantee the postcondition predicate will be true. The precondition follows the Larch/C++ keyword requires, and the postcondition follows the Larch/C++ keyword ensures. If the function has the potential to change anything during its execution, the modifies keyword is used to describe which objects the function is allowed
to mutate. The absence of a modifies clause means the function does not modify anything. (Knowing this, it is very easy to scan a Larch/C++ class interface specification to determine which functions mutate the object.) In the subset of Larch/C++ discussed in this paper, the absence of a requires clause means the function will terminate every time when called with the postcondition true. A function may also include an optional examples section, which contains examples designed to convey the meaning of the function to the reader. Below is an example function specification, followed by an explanation. (Note that in Larch/C++, there is more than one correct way to specify a function, just as there is more than one correct way to specify a class.)

```c++
//A Larch/C++ function specification

int Withdraw(int& source, int amount) {
    requires amount > 0 \ source^ >= amount;
    modifies source;
    ensures source' = source^ - amount \ result = source';
    example source^ = 500 \ amount = 200 \ source' = 300 \ result = 300;
}
```

The function above takes two arguments, a variable `source`, and an `amount` to be withdrawn. The `&` preceding the `source` indicates call-by-reference. The requirements for the function to ensure the postcondition are that `source^`, the value of `source` before the function is invoked, be large enough to cover the `amount` to be withdrawn, and that the `amount` be positive. The postcondition states the value of `source'`, the value of `source` after the function is finished execution, will be updated according to the infix trait function `^=`, which defines integer subtraction. (More information on trait functions will be presented in section 3.4.)

### 3.3 CSize: An Example Larch/C++ Class Specification

Next, we examine how to specify a C++ class. Below is the specification of the class CSize, a simple value type of the MFC Library used to hold X and Y screen coordinates. The Larch/C++ specification includes the declaration
of ten public member functions: four constructor functions, four modifying
operators, and two retrieval operators. A user of this class can abstractly
think of a CSize object as having two integer variables: an X coordinate
(named cx) and a Y coordinate (named cy). The function interfaces are
written according to this abstract representation. Two structures that are
going to be used in the C++ header file will also be needed in the specifi-
cation, so we declare them first. Note that such C++ declaration syntax is
part of Larch/C++.

struct tagSize {
    int cx;
    int cy;
};
typedef tagSize SIZE;

struct tagPoint {
    int x;
    int y;
};
typedef tagPoint POINT;

After creating a typedef for BOOL, we begin specifying the class CSize.
The abstract values of objects of the class (cx and cy) are given by the trait
functions defined in CSizeTrait. Trait functions describe the behavior of
functions defined in the interface module by precisely describing the abstract
values[7, p. 3]. The connection is made by the uses clause. (A class can
use more than one trait). The simulates clause states how the abstract
values of a class, such as CSize, can be interpreted as abstract values of the
derived class, in this case tagSize. The trait function totagSize defines this
mapping.

imports pretend_bool;
typedef bool BOOL;

class CSize : public tagSize {
uses CSizeTrait;
simulates tagSize by totagSize;

The constructors for the class are presented below. After examining the first constructor, one might wonder why no initial values are being assigned. This occurs when a constructor cannot state anything about the initializations taking place upon function invocation. By stating the ensures clause as true, no constraints are placed on the post-state values. In the second constructor, the ' following self indicates the post-value of self, i.e., the value of the object being constructed after the function has executed. The following' extracts a tuple of values from the tuple of two objects that is the value of self'. Although it appears the constructors all use the same trait function, they actually use three different trait functions. This is accomplished by trait function overloading. The any keyword used as a parameter to the trait function contained_objects indicates any state can be used, pre or post.

public:
CSize() {
    modifies contained_objects(self, any);
    ensures true;
}

CSize(int initCX, int initCY) {
    modifies contained_objects(self, any);
    ensures self'' = createCSize(initCX, initCY);
}

CSize(SIZE initSize) {
    modifies contained_objects(self, any);
    ensures self'' = createCSize(initSize);
}

CSize(POINT initPt) {
    modifies contained_objects(self, any);
    ensures self'' = createCSize(initPt);
}
Finally, the remaining operations for the class are specified. Within the ensures clauses of the functions, \((\text{self}^\cdot \text{cx})\) is the pre-state data member object \(\text{cx}\). (Recall that an abstract instance of the class \text{CSize} is a container object that contains two integer objects: \(\text{cx}\) and \(\text{cy}\).) To access the pre-state value of the data member, another de-referencing is done, \((\text{self}^\cdot \text{cx})\)\(^\cdot\). To access the post-state value, \((\text{self}'^\cdot \text{cx})\)\(^'\) is used.

```cpp
BOOL operator == (SIZE size) const {
    ensures result = ((\text{self}^\cdot \text{cx})^\cdot = \text{size}.\text{cx}) \\/
                     ((\text{self}^\cdot \text{cy})^\cdot = \text{size}.\text{cy});
}

BOOL operator != (SIZE size) const {
    ensures result = ((\text{self}^\cdot \text{cx})^\cdot \neq \text{size}.\text{cx}) \\/
                     ((\text{self}^\cdot \text{cy})^\cdot \neq \text{size}.\text{cy});
}

void operator += (SIZE size) {
    modifies self;
    ensures (self'.\text{cx})' = (\text{self}^\cdot \text{cx})' + \text{size}.\text{cx} \\/
                        (self'.\text{cy})' = (\text{self}^\cdot \text{cy})' + \text{size}.\text{cy} ;
}

void operator -= (SIZE size) {
    modifies self;
    ensures (self'.\text{cx})' = (\text{self}^\cdot \text{cx})' - \text{size}.\text{cx} \\/
                        (self'.\text{cy})' = (\text{self}^\cdot \text{cy})' - \text{size}.\text{cy} ;
}

\text{CSize} operator + (SIZE size) const {
    ensures fresh(result) \\/
         result.\text{cx} = (\text{self}^\cdot \text{cx})^\cdot + \text{size}.\text{cx} \\/
                        result.\text{cy} = (\text{self}^\cdot \text{cy})^\cdot + \text{size}.\text{cy} ;
}

\text{CSize} operator - (SIZE size) const {
    ensures fresh(result) \\/
         result.\text{cx} = (\text{self}^\cdot \text{cx})^\cdot - \text{size}.\text{cx} \\/
                        result.\text{cy} = (\text{self}^\cdot \text{cy})^\cdot - \text{size}.\text{cy} ;
}
```

```cpp``
As stated earlier, the purpose of trait functions is to assign values and give meaning to the interface specification. Unlike SPECS-C++, another C++ class specification technique which declares abstract functions within a class specification [5], trait functions are declared in a module other than the actual specification interface module. This enforces the two-tiered approach to Larch style specification languages [10]. The benefits of this approach will become apparent in the discussion of the interface specification of the class CObList, which will reuse the trait functions originally defined for CObArray.

3.4 CSizeTrait: An Example Larch Shared Language Trait

An LSL trait specifies the exact meanings of trait functions used in a Larch interface specification. The mathematical abstractions are only used in writing assertions, and are not to be considered executable functions that can be used in programs [7]. Below is the trait defined for use with the CSize interface specification. The introduces section declares the trait functions and their corresponding signatures, and the asserts section defines the exact semantics of the trait functions.

```plaintext
% An LSL trait (Not Larch/C++)
CSizeTrait : trait
includes
   Integer(int for Int),
   tagSize_Trait, tagPoint_Trait, % these are generated by Larch/C++
   % and define sorts tagSize, Val_tagSize,
   % and ConstTagSize.
   tagSize_Trait(CSize for tagSize, Val_CSize for Val_tagSize,
        ConstCSize for ConstTagSize)

introduces
   createCSize : int, int -> Val_CSize
   createCSize : Val_tagSize -> Val_CSize
   createCSize : Val_tagPoint -> Val_CSize
   toTagSize : CSize -> tagSize
```

12
The three trait functions named `createCSize` are used to specify the
abstract tuples that result from invocations of constructor functions. Note
that in Larch/C++, the abstract value of a structure declared in the C++
is a fixed-length tuple with corresponding fields and sorts. For each struct
declared in C++, the trait defining its abstract model is implicitly used in
any interface specification module where the declaration is made [7, pp. 175-
177]. Since the `CSize` struct actually inherits from `tagSize`, the `tagSize`
trait is included in the `CSize` trait. The trait function `totagSize` is used to
map a `CSize` value to a `tagSize` value (see [7] for reasons).

4 The Interface Specifications

The MFC Library contains numerous classes that implement many of the
most widely used data structures in imperative programming. Among the
most fundamental data structures are arrays and linked lists. Arrays and
linked lists are implemented as part of a group of classes in the MFC Library
appropriately named the *collection classes*. This group of classes can be
used for storing sets and sequences of homogeneous objects such as `CString`
and `CObject`. However, a collection of pointers to `CObjects` could be
heterogeneous due to subtyping.

Included with the class software is the *Microsoft Class Libraries Reference*
[2]. By specifying these classes more formally than the reference manual, one
can compare Larch/C++ to existing documentation and evaluate the advantages
and disadvantages of the specification language. Since many of the
classes provided in the MFC Library are directly or indirectly derived from
the `CObject` class (described in Chapter 2 of the Class Libraries Reference),
the specification process begins with `CObject`. The MFC Library also defines simple value type classes which are not derived from `CObject`. Among
these are 

4.1 Specification of CObject

The principal base class for the MFC Library is the class CObject. It serves not only as the base class for many of the MFC Library classes, but as the base class for user defined classes as well. The class contains a variety of virtual functions, as well as many macros. At the time of this paper, there was no mechanism in Larch/C++ to specify macros. A possible way to specify macros has thus been introduced (informally using comments) in the
class definition (see Appendix B, CObject). The specification of macros was done using the specification of functions as a guideline.

The class CObject uses classes CArchive, CArchiveException, CDump-Context, and CFileException. However, these classes were not specified in order to limit the scope of the project. Because of these omissions, the Larch/C++ keyword informally has been used. This allows a specifier to state in English (less formally) what cannot be (or will not be) stated more formally.

An example function specification using the informally keyword, taken from the class specification of CObject, is given below. After the function interface is given, two “spec cases” are presented, each with a requires and ensures clause. The clause

```c++
let ck: CompilerKnowledge be compiler_state_at_definition;
```

needs explanation here. The term compiler_state_at_definition refers to a function that is not currently part of Larch/C++, but may appear in the language in the future. It has been added here because of the need to know what macros exist at the time this class is defined. The function, which returns a value of sort CompilerKnowledge, provides this information. The actual scope of information provided has yet to be determined.

```c++
virtual void Dump( CDumpContext& dc) const {
    let ck: CompilerKnowledge be compiler_state_at_definition;
    requires usesMacro(A"IMPLEMENT_DYNAMIC(" || className(ck) || A"," ||
    baseClassName(ck) || A")",ck)
    \ usesMacro(A"IMPLEMENT_SERIAL(" || className(ck) || A"," ||
    baseClassName(ck) || A"", || wSchemaNum(ck) || A")",ck);
    ensures informally "the class name along with diagnostic information"
    "of the object is dumped to dc";
    let ck: CompilerKnowledge be compiler_state_at_definition;
    requires not(usesMacro(A"IMPLEMENT_DYNAMIC(" || className(ck) ||
    A"," || baseClassName(ck) ||
    A")",ck))
    \ not(usesMacro(A"IMPLEMENT_SERIAL(" || className(ck) ||
    A"," || baseClassName(ck) ||
    A"", || wSchemaNum(ck) ||
    A")",ck));
    ensures informally "diagnostic information of the object is dumped"
    "to dc";
}
```
The functions new and delete provided by the class CObject [2, p. 475] are designed to reclaim memory storage space, a concept not built-in to in Larch/C++. Our solution was to model the amount of memory available as a specification variable, the idea being that because the actual amount of memory available is not directly accessible, we do not want to enforce the implementation of such a variable declaration by declaring it extern: it is simply there for specification purposes. A user of the class is consequently made aware of the potential for such operations as new to throw memory exceptions. Larch/C++ models a function that throws an exception by making the result of the function the normal result (specified by result), or an exception result, written thrown(T), where T is the exception’s type [7, p. 91]. The resulting function specification contains two specification cases, one specifying the post-condition when sufficient memory is available, and one for specifying the post-condition when insufficient memory exists. This approach to memory was carried through the specification of CObArray and CObList.

4.2 Specification of CException and CMemoryException

The specification of the classes CException and CMemoryException has also been included in this paper (see Appendix B). The class CException is an abstract class, from which other MFC Library exception classes are derived. The class CMemoryException is one such class, containing the specification of a constructor and destructor, which are the only two member functions provided. The pragmatic warning concerning constructor invocation that is present in the class library documentation can also be stated in the Larch/C++ specification by means of C++ style comments. This is appropriate, as such usage information is not part of the behavior of the class.

4.3 Specification of CObArray

In specifying CObArray, we needed to determine how to model abstractly an instance of the CObArray class. A CObArray object should have an upper bound indicating the last valid index into the array, a size value, and, obviously, a mapping of indices to array values. The Larch Shared Language
trait FiniteMap provided the means to accomplish the mapping: the domain being the set of integers, the range being the set of all CObject pointer objects.

The trait functions that give the abstract values were not trivial to write. This is one of the drawbacks of Larch/C++. When no existing trait having the properties desired is available, one must create a new trait. However, after defining the MSArray trait for CObArray, we discovered it could be used successfully to specify the classes CByteArray, CWordArray, CDWordArray, and CStringArray, which were all very similar in design and functionality. All that was required was a substitution of the actual parameters used as sorts to be stored in the array. So even though traits may be difficult to write, they promote reuse.

4.4 Specification of CObList

The specification of CObList followed directly from the specification of CObArray. Since a list is merely a collection of ordered elements, why not model it abstractly as an array? By using the trait MSArray within the trait MSList, it was relatively easy to specify CObList, realizing that insertion into the array need simply to preserve the ordering. Thus, a new list object was modeled abstractly as a tuple created by the MSArray trait function create.

5 Analysis of the Specification

Throughout the specification process, many ambiguities and inconsistencies were discovered in the documentation for the MFC Library [2]. This section reveals some of them.

5.1 Ambiguities in MFC Documentation Revealed

The main advantage of formal specification is the elimination of ambiguities that would otherwise be present in informal specifications. Consider the CObArray class member functions InsertAt [2, pp. 456-457] and SetAtGrow [2, pp. 460-461]. Both of these functions allow the user to assign a pointer
to a CObject to a given index of the array. From the documentation, the parameter nIndex is allowed to be greater than the current upper bound of the array. Consider the following code fragment using InsertAt. (Assume CAccount is a CObject-derived class with a constructor that takes a double as an argument.)

```cpp
CObArray array;
CAccount* p1;

p1 = new CAccount(20.00);
array.Add(p1);
array.InsertAt(4, p1, 1);
```

At this point, what would be the value of array.GetSize()? Normally, it would be assumed that the size of the array is now 2, since 2 array values have been added. However, the size is actually 5. The array elements array[1], array[2], and array[3] have all been assigned NULL automatically. However, there is no mention of this assignment in the informal specification of InsertAt. A similar situation can occur using SetAtGrow. In this case, the user should be forewarned that these assignments occur automatically. In contrast, examine how the Larch/C++ specification handles the function InsertAt for inserting a newElement.

```cpp
void InsertAt(int nIndex, CObject* newElement, int nCount = 1)
    throw(CMemoryException) {
    extern int volatile memory_available;

    // case 1
    requires nCount >= 1 /\ nIndex >= 0 /\ "legalIndex(self",nIndex) /\ metric_available" >= (memOverhead(sizeof(CObPtr)) *
    (nCount + (nIndex - (self^.upperbound + 1))));
    modifies self, memory_available;
    ensures \forall k:int (0 <= k /\ k <= self^.upperbound =>
    (self'[k])' = (self'[k])) /\ 
    \forall i:int (self^.upperbound < i /\ i < nIndex =>
    fresh(self'[i]) /\ (self'[i])' = NULL) /\ 
    \forall j:int (nIndex <= j /\ j <= (nIndex + nCount-1) =>
    fresh(self'[j]) /\ (self'[j])' = newElement)) /\ 
```
self'.size = self'.size +
    (nCount + (nIndex - (self'.upperbound +1))) /
self'.upperbound = self'.size - 1 /\memory_available' = memory_available' -
    (memOverhead(sizeof(CObPtr))*(nCount + (nIndex - (self'.upperbound +1))));

// case 2
requires nCount >= 1 /\ nIndex >= 0 /\ legalIndex(self', nIndex) /\ "(memory_available' >= (memOverhead(sizeof(CObPtr)) * nCount));
modifies self, memory_available;
ensures \forall i:int (self'.upperbound < i /\ i <= (nCount+self'.upperbound) => fresh(self'[i]) /\ self' = insert_at(self', nIndex, newElement, nCount) /\memory_available' = memory_available' -
    (memOverhead(sizeof(CObPtr))* nCount));

// case 3
requires nCount >=1 /\ nIndex >= 0 /\ "legalIndex(self', nIndex) /\ "(memory_available' >= (memOverhead(sizeof(CObPtr)) *
    (nCount + (nIndex - (self'.upperbound +1))));
ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));

// case 4
requires nCount >= 1 /\ nIndex >= 0 /\ legalIndex(self', nIndex) /\ "(memory_available' >= (memOverhead(sizeof(CObPtr)) * nCount));
ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));
}

In the above specification, memory_available is a spec variable that represents the current amount of memory available to be used by the program. Although Larch/C++ does not provide direct support for this concept, it can modeled successfully using this new technique. The function memOverhead states the actual amount of memory used is probably more than sizeof(CObPtr), but how much more is unknown (see Appendix A, MemoryTrait). The reading of the function can be broken into four cases: the first two cases describe the function’s behavior if sufficient memory exists. The next two cases describe the functions behavior when sufficient memory does not exist, in which case an exception is thrown. For the first two cases, based on which other conjuncts of the requires clause are true, the ensures clause states the value of self (and memory available). For the first case, the actual value of self is described in the interface specification. If suffi-
cient memory exists, and the other conjuncts of the second requires clause are true, the user invoking the function is ensured that the resulting array (being represented by self') is defined by the trait function

\[
\text{insert\_at}(\text{self}', \text{nIndex}, \text{newElement}, \text{nCount})
\]

and memory\_available will be decreased appropriately. All that is required now is the examination of the trait function insert\_at (see Appendix A, MSArray). Note that the trait function insert\_at actually “builds” up the new array value from the old array value.

The documentation of CObList functions InsertBefore and InsertAfter [2, pp. 491-492] also falls victim to this type of ambiguity. Below is the interface for the function InsertBefore, which indicates there are two parameters to the function: position and newElement. (The function InsertAfter is set up similarly). The documentation provided in the reference manual simply states that position need only be a POSITION value returned by a previous GetNext, GetPrev, or Find function call. Actually, this is not entirely true, as POSITION values obtained from GetTailPosition, GetHeadPosition, and FindIndex are perfectly legal values of position. But what happens when a function such as Find returns a NULL position value? When given a NULL position, the function InsertBefore puts the item at the beginning of the list, and the function InsertAfter puts the item at the end of the list. Again, this is not clear from the given documentation. However, this information is clearly conveyed in the Larch/C++ interface specification.

```c
POSITION InsertBefore(POSITION position, CObject *newElement)
    throw(CMemoryException) {
        extern int volatile memory\_available;

        let E: MS\_List be {newElement};
        requires ValidPosition(position) \/
            (memory\_available" >= memOverhead(sizeof(CObPtr)));
        modifies self, memory\_available;
        ensures fresh(self'[length(self')] /\n            self' = head\_not\_incl(self", position) ||
            E || tail\_incl(self", position) /\n            memory\_available' = memory\_available" -
                memOverhead(sizeof(CObPtr)) /\n
        return;
    }
```

20
Another ambiguity occurs in the description of legal parameter values. Consider the parameter value `nCount` for the function `InsertAt` of the class `CObArray`. Suppose a programmer has written the following fragment of code.

```cpp
CObArray array;
CAccount* p1;
int i;

p1 = new CAccount(0.00);

...  

//ASSERT : FLAG == 0

do {
    get_flag(FLAG);
    array.InsertAt(i, p1, FLAG);
    i++;
} while (FLAG != 0);
```

In the above code, the programmer wants to use a do-while loop that keeps inserting values into the array until `FLAG` equals zero. When this happens, the loop invokes `InsertAt` one last time, assigning the `nCount`
parameter the value zero. Since no constraints are placed on the parameter’s value in the documentation (page 457), the programmer may assume this is acceptable. However, this will cause an error, as \textit{nCount} needs to be greater than or equal to 1.

A similar example involves the function \textit{IsKindOf}, a boolean function of the class \texttt{CObject} designed to test if an object corresponds to a given class. A pointer \texttt{pClass} to a \texttt{CRuntimeClass} structure is the parameter to the function. The function returns true if the given object is an object of the specified class or is derived from the specified class. However, no restrictions are made regarding the value of \texttt{pClass} when the function is called, which indicates that any valid parameter assignment is legal. Since the parameter is actually a pointer, its value could be NULL upon function invocation.

5.2 Inconsistencies in MFC Documentation Revealed

Consistency is also difficult to accomplish when writing informal documentation. For example, many of the functions of the class \texttt{CObArray} have an index into the array as a formal parameter. For some of these functions, as in \texttt{GetAt}, it is explicitly stated that the index parameter \textit{nIndex} must be a value greater than or equal to 0 and less than or equal to \texttt{GetUpperBound()}. However, for the function \texttt{InsertAt}, \textit{nIndex} is merely supposed to be an integer index that may be greater than \texttt{GetUpperBound()} [2, p. 457]. Is the user therefore allowed to set \textit{nIndex} to be a value less than 0? Since this parameter has been explained for other functions, is the absence of such an explanation supposed to mean any value of \textit{nIndex} is allowed? In fact, for the documentation of the \texttt{[]} operator, all that is mentioned is that the subscript \textit{nIndex} not be “out of bounds” [2, p. 457], or the “Debug” version of the library “asserts”. A more precise way to state this would be to clearly write the constraints on \textit{nIndex}.

A similar parameter clarification should be added to the documentation of the \texttt{Add} function of \texttt{CObArray}[2, pp. 453]. Since no restriction on the value of \texttt{newElement} is made, and since a value of NULL for \texttt{newElement} is allowed for all other assignment operations, the user might expect to be able to invoke the function \texttt{Add} with such an assignment. However, in this case, \texttt{newElement} is \textit{not} allowed to be NULL. In the documentation given, this restriction is not explicitly stated. In the Larch/C++ specification given below, it is stated by the \texttt{isValid(newElement)} conjunct.
6 Advantages and Disadvantages of Larch/C++

6.1 Disadvantages of Larch/C++ vs. Informal English

Perhaps the greatest disadvantages of Larch/C++ lie not with the interface specification language itself, but in its use of LSL. Recall that the creation of an interface specification module requires the use of trait functions to give meanings and values to an instance of the class being specified. When no existing trait having the properties desired is available, the programmer must create his/her own trait. In this situation, the programmer does not capitalize on the reusability LSL was designed to provide. The specification process can then require the creation of not just one additional module, but potentially many modules. This occurred in the specification of the class CObArray, when a new trait MSArrary had to be created. (However, this trait did use the handbook trait FiniteMap.) This can sometimes be a very time-consuming part of the specification process. With informal documentation, the writer simply needs to state what he/she thinks is relevant to the understanding of the specification. (This, however, is a judgement call that can result in even more misunderstandings.)
Traits may also make the understanding of the interface specification harder than informal English would. Looking again at the COBArray class, the informal documentation provided for the operation InsertAt (with an array as the parameter) is fairly straightforward concerning the results of the function. The trait function InsertArray states the same thing concerning the result, yet it is much harder to understand it based solely on the mathematical equations included in the trait. (This again advocates the need for some form of informal documentation to be presented along with the formal documentation). Programmers who read specifications need to be familiar and comfortable with the mathematical vocabulary (predicate calculus) that is used to to write the trait functions, as well as the interface specification. However, as mathematics is an essential part to the programming process, this concern is not as great as one might think.

Finally, Larch/C++ is still a very “young” specification language, and cannot handle some C++ constructs very eloquently, or even at all. For example, the ability to specify macros was not available at the time of this paper. There was also no way to specify variables that need not be implemented, but were convenient for writing the specifications, such as memory available. This feature, however, was added to the language as a result of this project.

6.2 Advantages of Larch/C++ vs. Informal English

Before looking at the advantages of Larch/C++, let us review the purpose of the language. Larch/C++ is designed to precisely and unambiguously describe the behavior of classes (and their functions). Again, it must be stressed that Larch/C++ is not designed to completely replace English documentation. It would be naive to adopt such a belief. Instead, Larch/C++ should be complementary to English documentation, thereby providing the user with as much information as possible, from formal specifications to examples to informal English. Furthermore, Larch/C++ specifications need not reveal anything about implementation specifics, but can act as a mechanism to achieve information hiding.

By using Larch/C++, the designer of a class usually becomes trained to think more carefully about all possible boundary cases of parameter values to class functions, as well as all possible values of instances of the class. This can surface unrealized possibilities, as presented in the discussion of a NULL.
position to the **CObList** functions **InsertBefore** and **InsertAfter**. In this manner, Larch/C++ aids in the design of an abstract data type. Larch/C++ also allows for the specification of several interfaces for a specific class: public, protected, and private. The designer of such a class can then specify what he/she thinks is important at each interface level for a user of a class.

Larch/C++ also promotes fast specification of similar classes. This can be accomplished two different ways: reuse and inheritance. By reusing old specifications or examining specification examples for which similar concepts (abstract data members) are involved, the specification of a new class can be facilitated a great deal. Inheritance of specifications allows classes that are directly derived from other classes to inherit the specifications for those classes. For example, for the class **CObArray**, we inherit the specification of the class **CObject**. This means that for every every instance of **CObject** used with **CObArray**, it must satisfy the specifications presented in the interface module for **CObject**. By allowing inheritance as C++ does, documentation (like code) need not be repeated for each derived class.

Larch/C++ can be used in formal proofs of correctness of abstract data types. As software projects become larger, correct code and documentation during the early stages of development is critical in order to avoid high maintenance and re-design costs in the future. If, for example, a company relied on the addition of a NULL **CObject** pointer for the **Add** function of the class **CObArray**, and a crash occurs because of it, thousands of dollars could be lost. Although full proofs of correctness may not be practical, Larch/C++ can aid in partial verification of code and provide the more detailed documentation necessary for it.

Larch/C++ also provides a standard for documenting classes. Informal English written by some programmers can become tainted with shorthand notations and abbreviations (acronyms) that only the programmer (or company) may know. Larch/C++ remains the same from person to person and company to company.

In the future, Larch/C++ will be integrated into C++ specification and programming in the following way. From the interface specification module (.lcc), Larch/C++ will generate a C++ header file for the class (.h) and part of the implementation file (.C) [1]. The actual header file for the class may need additional information added to it, but most of it can be obtained by including the .h file. Thus, some of the effort that goes into writing the interface specification can be recovered using Larch/C++.
Conclusions

Larch/C++ is by no means a perfect specification language. In fact, debate still exists in the computer science community on the use of formal specification techniques in general. Many programmers like to provide their own style of informal documentation for their code, and some provide no documentation at all. True, using a formal specification language such as Larch/C++ will consume more time at the beginning of a software project, postponing the arrival of actual working code for a later date than if informal techniques (or none) are used. In the long run, however, the financial savings from reduced maintenance or re-design costs may out-weigh the initial cost in time.

I believe that people interested in documenting C++ class libraries like the MFC Library can obtain some benefit from using Larch/C++. Larch/C++ can be used to specify classes before they are implemented, while they are implemented, or after they are implemented. This paper has presented some of the benefits of specifying classes after they have been implemented and have existing documentation. For programmers intending to build on class libraries, having Larch/C++ specifications as well as informal documentation can be very advantageous. However, I do not believe Larch/C++ is necessary for specifying those highly abstract classes for which English is better suited (such as CObject), nor do I believe Larch/C++ should be the only type of documentation for a class library.

In my opinion, companies producing software can identify with one of three scenarios. When there is too little documentation, projects become harder to complete due to misunderstandings and lack of communication. When there is too much documentation, projects become burdened and software releases get postponed. When there is exactly the right amount of documentation, projects can be completed smoothly and on time. Whether or not Larch/C++ contributes to the “exact” amount of documentation is yet to be seen. I believe that formal specification using Larch/C++ is good for specifying things more precisely and unambiguously, but is too rigorous in some respects. For simple functions and (classes), English documentation (or informal Larch/C++) may be best. For more complicated ones, Larch/C++, augmented with English overviews, may be best. I do believe that by writing specifications more formally using Larch/C++, a programmer becomes trained to think more critically about the functions (or classes)
being specified and/or coded.


8 Appendix A: Traits

MemoryTrait

MemoryTrait : trait
  includes Integer(int for Int)
  introduces
    memOverhead : int -> int
    allocatedSize : void -> int
  asserts
    \forall i : int
    memOverhead(i) \geq i;

InternalStateTrait

InternalStateTrait : trait
  introduces
    ValidInternalState : State -> Bool

CompilerKnowledge

CompilerKnowledge : trait
  includes Set(String[char] for E, Set[String[char]] for C)
  introduces
    compiler_state_at_definition : -> CompilerKnowledge
    compiler_state_at_call_site : -> CompilerKnowledge
    usesMacros : CompilerKnowledge -> Set[String[char]]
    usesMacro : String[char], CompilerKnowledge -> Bool
  asserts
    \forall m: String[char], c: CompilerKnowledge
    usesMacro(m, c) \equiv m \in usesMacros(c);

RunTimeClassTrait

RunTimeClassTrait : trait
  includes CompilerKnowledge
  introduces
    dynamic_class_of : Object -> DynamicClass
to_{DynamicClass} : CRuntimeClass -> DynamicClass
to_{CRuntimeClass} : DynamicClass -> CRuntimeClass
inherits_from : DynamicClass, DynamicClass -> Bool
className : CompilerKnowledge -> String[char]
baseClassName : CompilerKnowledge -> String[char]
wSchemaNum : CompilerKnowledge -> String[char]
runtime_class_of : String[char] -> CRuntimeClass

asserts
forall x, y, z : DynamicClass, b : CRuntimeClass
to_{DynamicClass}(to_{CRuntimeClass}(x)) == x;
to_{CRuntimeClass}(to_{DynamicClass}(b)) == b;
inherits_from(x,x);
inherits_from(x,y) \land inherits_from(y,x) == (x == y);
inherits_from(x,y) \land inherits_from(y,z) => inherits_from(x,z);

\tagSize_Trait
\tagSize_Trait: trait
assumes int
includes MutableObj(int),
ConstObj(int),
NoContainedObjects(Val_{tagSize})
tagSize tuple of cx: Obj[int], cy: Obj[int]
ConsttagSize tuple of cx: ConstObj[int], cy: ConstObj[int]
Val_{tagSize} tuple of cx: int, cy: int
introduces
__ all: tagSize -> tagSize
__ all: ConsttagSize -> ConsttagSize
__ all: Val_{tagSize} -> Val_{tagSize}
contained_objects: tagSize, State -> Set[Object]
contained_objects: ConsttagSize, State -> Set[Object]
__ ! __: tagSize, State -> Val_{tagSize}
__ ! __: ConsttagSize, State -> Val_{tagSize}

asserts
forall s: tagSize, cs: ConsttagSize, vs: Val_{tagSize},
iy, iy: Obj[int],
ciy, ciy: ConstObj[int], st: State
s.all == s;
cs.all == cs;
vs.all == vs;
contained_objects([ix,iy], st) == {widen(ix)} \cup {widen(iy)};
contained_objects([ciy,ciy], st) == {widen(ciy)} \cup {widen(ciy)};
s!st == [(s.cx)!st, (s.cy)!st];
tagPoint_Trait

tagPoint_Trait: trait
assumes int
includes MutableObj(int),
ConstObj(int),
NoContainedObjects(Val_tagPoint)
tagPoint tuple of x: Obj[int], y: Obj[int]
ConsttagPoint tuple of x: ConstObj[int], y: ConstObj[int]
Val_tagPoint tuple of x: int, y: int
introduces
_.all: tagPoint -> tagPoint
_.all: ConsttagPoint -> ConsttagPoint
_.all: Val_tagPoint -> Val_tagPoint
contained_objects: tagPoint, State -> Set[Object]
contained_objects: ConsttagPoint, State -> Set[Object]
_.! _.: tagPoint, State -> Val_tagPoint
_.! _.: ConsttagPoint, State -> Val_tagPoint
asserts
\forall p: tagPoint, cp: ConsttagPoint, vp: Val_tagPoint,
ix, iy: Obj[int],
cix, ciy: ConstObj[int], st: State
p.all == p;
cp.all == cp;
vp.all == vp;
contained_objects([ix,iy], st) == {widen(ix)} \ U {widen(iy)};
contained_objects([cix,ciy], st) == {widen(cix)} \ U {widen(ciy)};
p!st == [(p.x)!st, (p.y)!st];
vp!st == [(cp.x)!st, (cp.y)!st]

CSizeTrait

CSizeTrait : trait
includes Integer(int for Int),
tagSize_Trait, tagPoint_Trait, % these are generated by Larch/C++
% and define sorts tagSize, Val_tagSize, etc.
tagSize_Trait(CSize for tagSize, Val_CSize for Val_tagSize,
ConstCSize for ConsttagSize)
introduces
createCSize : int, int -> Val_CSize
createCSize : Val_tagSize -> Val_CSize
createCSize : Val_tagPoint -> Val_CSize
totagSize : CSize -> tagSize
asserts
\forall x, y, c: CSize, s: Val_tagSize, p: Val_tagPoint
createCSize(x, y) \equiv [x, y];
createCSize(s) \equiv [s.cx, s.cy];
createCSize(p) \equiv [p.x, p.y];
totagSize(c) \equiv [c.cx, c.cy];

CPointTrait

CPointTrait : trait
includes
  Integer(int for Int),
  tagSize_Trait, tagPoint_Trait, \% these are generated by Larch/C++
  % and define sorts tagPoint, Val_tagPoint, etc.
  tagPoint_Trait(CPoint for tagPoint, Val_CPoint for Val_tagPoint,
       ConstCPoint for ConsttagPoint)
introduces
  createCPoint : int, int -> Val_CPoint
  createCPoint : Val_tagSize -> Val_CPoint
  createCPoint : Val_tagPoint -> Val_CPoint
totagPoint : CPoint -> tagPoint
asserts
\forall x, y, c: CPoint, p: Val_tagPoint, s: Val_tagSize
createCPoint(x, y) \equiv [x, y];
createCPoint(s) \equiv [s.cx, s.cy];
createCPoint(p) \equiv [p.x, p.y];
totagPoint(c) \equiv [c.x, c.y];

POSITION

POSITION(E) : trait
introduces
  get_position : E -> POSITION
asserts
  POSITION generated by get_position

MSArray
MSArray(Elem, MS_Arr): trait
  includes FiniteMap(Elem for R, int for D, MSA for M),
  Integer(int for Int), TypedObj(Elem for Loc[T]),
  Set(int, Set[int], int for Int)
MS_Arr tuple of upperbound: int, size: int, map: MSA introduces
  contained_objects : MS_Arr, State -> Set[Object]
  contained_without_indexes: MSA, State, Set[int] -> Set[Object]
create : -> MS_Arr
legalIndex : MS_Arr, int -> Bool
__[___] : MS_Arr, int -> Elem
upperbound : MS_Arr -> int
size : MS_Arr -> int
through : MSA, int, Elem, int -> MSA
build : MSA, int, Elem, int -> MSA
build_offset : MSA, int, int, Elem, int -> MSA
resize : MS_Arr, int -> MS_Arr
growUBound : MS_Arr, int -> int
growSize : MS_Arr, int -> int
insert_at : MS_Arr, int, Elem, int -> MS_Arr
insert_array : MS_Arr, int, MS_Arr -> MS_Arr
removed_at : MS_Arr, int, int -> MS_Arr

asserts
MS_Arr generated by create, insert_at,
  insert_array, removed_at
\forall u, u2, s, s2, i, j, k : int, e : Elem, m, n : MSA, st : State,
  si : Set[int]
  contained_objects([u,s,m], st) ==
      contained_without_indexes(m, st, {} : Set[int]);
  contained_without_indexes({}, st, si) == {};
  contained_without_indexes(update(m,i,e), st, si) ==
      if i \in si then contained_without_indexes(m, st, si) else
      {wid(en(e)) \cup contained_without_indexes(m, st, insert(i,si));
create == [-1, 0, {}];
legalIndex([u, s, m], i) == (0 <= i) \land (i <= u);
([u, s, m])[i] == apply(m, j);
upperbound([u, s, m]) == u;
size([u, s, m]) == s;
through(m, i, e, j) == if (i>j) then m else
    through(update(m, i, e), i+1, e, j);
build(m, i, e, j) == if (i>j) then m else
    build(update(m, i, e), i+1, apply(m, i+1), j);
build_offset(m, i, k, e, j) == if (k>j) then m else

32
build_offset(update(m, i, e), i+1, k+1, apply(m, k+1), j);
resize([u, s, m], i) == if (s = i) then [u, s, m]
    else [i-1, i, build(m, 0, apply(m, 0), i-1)];
growUBound([u, s, m], i) == if (i > u) then i else u;
growSize([u, s, m], i) == if (i > s) then i else s;
insert_at([u, s, m], i, e, j) == [u+j, s+j, through(build(m, 0, apply(m, 0), i-1),
    i+j, apply(m,i), u+j), i, e, (j+i)-1];
insert_array([u, s, m], i, [u2, s2, m]) == [u+s2, s+s2, build_offset(build_offset(build(m, 0, apply(m,0), i-1),
    i, 0, apply(n,0), u2), u2+i+1, i, apply(m,i), u]);
removed_at([u, s, m], i, j) == [u-j, s-j, build(build(m, 0, apply(m, 0), i-1),i,
    apply(m, i+j), u)];

MSList

MSList(Elem, MS_List): trait
    includes MSArray(PosElem for Elem, MS_List for MS_Arr),
    Integer(int for Int),
    POSITION(Elem for E)
PosElem tuple of pos : POSITION, e : Elem
introduces
    {} : -> MS_List
    __ : Elem -> MS_List
    __ || __ : MS_List, MS_List -> MS_List
    length : MS_List -> int
    IsEmpty : MS_List -> Bool
    position_at : MS_List, int -> POSITION
    element_at : MS_List, int -> Elem
    index_at : MS_List, POSITION, int -> int
    element_thru_iter : MS_List, POSITION, int -> Elem
    position_thru_iter : MS_List, Elem, int -> POSITION
    retrieve_at_pos : MS_List, POSITION -> Elem
    ValidPosition : MS_List, POSITION -> Bool
    ValidIndex : MS_List, int -> Bool
    head_incl : MS_List, POSITION -> MS_List
    head_incl_P : MS_List, POSITION, int -> MS_List
    head_not_incl_P : MS_List, POSITION, int -> MS_List
    head_not_incl : MS_List, POSITION -> MS_List
    tail_incl : MS_List, POSITION -> MS_List
    tail_not_incl : MS_List, POSITION -> MS_List
    find : MS_List, Elem, POSITION -> POSITION
asserts
forall L, L1, L2: MS_List, i: int, e:Elem, P: POSITION
{} == create;
{i} == insert_at(create, 0, [get_position(e), e], 1);
L1 || L2 == insert_array(L1, upperbound(L1)+1, L2);
length(L) == size(L);
IsEmpty(L) == (length(L) = 0);
position_at(L, i) == (L[i]).pos;
element_at(L, i) == (L[i]).e;
index_at(L, P, i) == if ValidIndex(L, i)
   then (if (position_at(L, i) = P)
       then i
       else index_at(L, P, i+1))
   else (-1);
element_thru_iter(L, P, i) == if (P = (L[i]).pos)
    then element_at(L, i)
    else element_thru_iter(L, P, i+1);
position_thru_iter(L, e, i) == if ValidIndex(L, i)
    then (if (element_at(L, i) = e)
        then position_at(L, i)
        else position_thru_iter(L, e, i+1))
    else NULL;
retrieve_at_pos(L, P) == element_thru_iter(L, P, 0);
ValidPosition(L, P) == (index_at(L, P, 0) "=" (-1));
ValidIndex(L, i) == (0 <= i) \ (i <= upperbound(L));
head_incl(L, P) == if ValidPosition(L, P)
    then head_incl_P(L, P, 0)
    else {};
head_incl_P(L, P, i) == if (P = position_at(L, i))
    then {element_at(L, i)}
    else {element_at(L, i)} || head_incl_P(L, P, i+1);
head_not_incl_P(L, P, i) == if (P = position_at(L, i))
    then {} else
    {element_at(L, i)} || head_not_incl_P(L, P, i+1);
head_not_incl(L, P) == if ValidPosition(L, P)
    then head_not_incl_P(L, P, 0)
    else {};
tail_incl(L, P) == if ValidPosition(L, P)
    then {retrieve_at_pos(L, P)} ||
        tail_incl(L, position_at(L, (index_at(L, P, 0)+1)))
    else {};
tail_not_incl(L, P) == tail_incl(L, position_at(L, (index_at(L, P, 0)+1)));
find(L, e, P) == if (P = NULL) then position_thru_iter(L, e, 0)
else position_thru_iter(L, e, index_at(L, P, 0));
implies
MS_List generated by {}, {__}, ||
9 Appendix B: Specifications

CSize

struct tagSize {
    int cx;
    int cy;
};
typedef tagSize SIZE;

struct tagPoint {
    int x;
    int y;
};
typedef tagPoint POINT;

imports pretend_bool;

typedef bool BOOL;

class CSize : public tagSize {
    uses CSizeTrait;
    simulates tagSize by totagSize;

public:
    CSize() {
        modifies contained_objects(self, any);
        ensures true;
    }

    CSize(int initCX, int initCY) {
        modifies contained_objects(self, any);
        ensures self'' = createCSize(initCX, initCY);
    }

    CSize(SIZE initSize) {
        modifies contained_objects(self, any);
        ensures self'' = createCSize(initSize);
    }

    CSize(POINT initPt) {
        modifies contained_objects(self, any);
    }
}
ensures self'' = createCSize(initPt);
}

BOOL operator==(SIZE size) const {
  ensures result = ((self'').cx'' = size.cx) /\ 
                   ((self'').cy'' = size.cy);
}

BOOL operator!=(SIZE size) const {
  ensures result = ((self'').cx'' != size.cx) \/
                   ((self'').cy'' != size.cy);
}

void operator+=(SIZE size) {
  modifies contained_objects(self, any);
  ensures (self).cx'' = (self'').cx'' + size.cx /\ 
                    (self).cy'' = (self'').cy'' + size.cy ;
}

void operator-=(SIZE size) {
  modifies contained_objects(self, any);
  ensures (self).cx'' = (self'').cx'' - size.cx /\ 
                     (self).cy'' = (self'').cy'' - size.cy ;
}

CSize operator+(SIZE size) const {
  ensures result.cx = (self'').cx'' + size.cx /\ 
                   result.cy = (self'').cy'' + size.cy ;
}

CSize operator-(SIZE size) const {
  ensures result.cx = (self'').cx'' - size.cx /\ 
                   result.cy = (self'').cy'' - size.cy ;
}

};

CPoint

struct tagSize {
  int cx;
  int cy;
};
typedef tagSize SIZE;

struct tagPoint {
    int x;
    int y;
};
typedef tagPoint POINT;

imports pretend_bool, CSize;
typedef bool BOOL;

class CPoint : public tagPoint {
    uses CPointTrait;
    simulates tagPoint by totagPoint;

public:
    CPoint() {
        modifies contained_objects(self, any);
        ensures true;
    }

    CPoint(int initX, int initY) {
        modifies contained_objects(self, any);
        ensures self' = createCPoint(initCX, initCY);
    }

    CPoint(SIZE initSize) {
        modifies contained_objects(self, any);
        ensures self' = createCPoint(initSize);
    }

    CPoint(POINT initPt) {
        modifies contained_objects(self, any);
        ensures self' = createCPoint(initPt);
    }

    void Offset(int XOffset, int yOffset) {
        modifies contained_objects(self, any);
        ensures (self'.x)' = (self^ .x) + xOffset \!
        (self'.y)' = (self^ .y) + yOffset;
    }
}
void Offset(POINT point) {
    modifies contained_objects(self, any);
    ensures (self'.x)' = (self^.x)' + point.x /
         (self'.y)' = (self^.y)' + point.y;
}

void Offset(SIZE size) {
    modifies contained_objects(self, any);
    ensures (self'.x)' = (self^.x)' + size.cx /
         (self'.y)' = (self^.y)' + size.cy;
}

BOOL operator == (POINT point) const {
    ensures result == ((self^.x)^ == point.x) /
         ((self^.y)^ == point.y);
}

BOOL operator != (POINT point) const {
    ensures result == ((self^.x)^ ~= point.x) /
         ((self^.y)^ ~= point.y);
}

void operator += (SIZE size) {
    modifies contained_objects(self, any);
    ensures (self'.x)' = (self^.x)' + size.cx /
         (self'.y)' = (self^.y)' + size.cy;
}

void operator -= (SIZE size) {
    modifies contained_objects(self, any);
    ensures (self'.x)' = (self^.x)' - size.cx /
         (self'.y)' = (self^.y)' - size.cy;
}

CPoint operator + (SIZE size) const {
    ensures fresh(result) /
        result.x = (self^.x)^ + size.cx /
         result.y = (self^.y)^ + size.cy ;
}

CSize operator - (POINT point) const {
    ensures result.cx = (self^.x)^ - point.x /
         result.cy = (self^.y)^ - point.y ;
}
CPoint operator - (SIZE size) const {
    ensures result.x = (self.x) - size.cx \/
    result.y = (self.y) - size.cy;
}
}

CObject

spec volatile int memory_available;

//define_generator_Macro DECLARE_DYNAMIC(class_name)
// ensures informally "C++ header code necessary for a CObject-derived"
// "class with accessible run-time information is generated."

//define_generator_Macro DECLARE_SERIAL(class_name)
// ensures informally "C++ header code necessary for a CObject-derived"
// "class that can be serialized is generated."

//define_generator_Macro IMPLEMENT_DYNAMIC(class_name, base_class_name)
// ensures informally "C++ code necessary for a dynamic CObject-derived class"
// "with run-time access to the class name and position in"
// "hierarchy is generated"

//define_generator_Macro IMPLEMENT_SERIAL(class_name, base_class_name, wSchema)
// ensures informally "C++ code necessary for a dynamic CObject-derived class"
// "with run-time access to the class name and position in"
// "hierarchy is generated"

//define_function_Macro RUNTIME_CLASS(class_name)
// ensures fresh(*result) \ (*(result = runtime_class_of(class_name)))
// \ result != NULL;

imports pretend_bool, CArchive, CArchiveException, CDumpContext,
CFileException, CMemoryException;

typedef bool BOOL;

// The following class must be defined (for parsing)
class CRuntimeClass;

// The following statement required for parsing
class size_t;
class CObject {
    uses InternalStateTrait, RunTimeClassTrait, CompilerKnowledge,
    NoInformation(CObject), MemoryTrait;

public:
    virtual void AssertValid() const {
        requires ValidInternalState(pre);
        ensures true;

        requires "ValidInternalState(pre);
        ensures liberally false;
    }

    virtual "CObject() {
        modifies self;
        ensures true;
    }

    virtual void Dump( CDumpContext& dc) const {
        let ck: CompilerKnowledge be compiler_state_at_definition;
        requires usesMacro("IMPLEMENT_DYNAMIC(" || className(ck) || A"," ||
            baseClassName(ck) || A")," ||
            wSchemaNum(ck) || A"),ck);
        \/ usesMacro("IMPLEMENT_SERIAL(" || className(ck) || A"," ||
            baseClassName(ck) || A")," ||
            wSchemaNum(ck) || A"),ck);
        ensures informally "the class name along with diagnostic information"
        "of the object is dumped to dc";

        let ck: CompilerKnowledge be compiler_state_at_definition;
        requires not(usesMacro("IMPLEMENT_DYNAMIC(" || className(ck) ||
            A")," || baseClassName(ck) || A")," ||
            wSchemaNum(ck) || A"),ck));
        \/ not(usesMacro("IMPLEMENT_SERIAL(" || className(ck) || A")," ||
            baseClassName(ck) || A")," ||
            wSchemaNum(ck) || A"),ck));
        ensures informally "diagnostic information of the object is dumped"
        "to dc";
    }

    virtual CRuntimeClass* GetRuntimeClass() const {
        let ck: CompilerKnowledge be compiler_state_at_definition;
        requires usesMacro("IMPLEMENT_DYNAMIC(" || className(ck) || A")," ||
            baseClassName(ck) || A"),ck))
        \/ usesMacro("IMPLEMENT_SERIAL(" || className(ck) || A")," ||
            baseClassName(ck) || A")," ||
            wSchemaNum(ck) || A"),ck);
        \/ usesMacro("IMPLEMENT_SERIAL(" || className(ck) || A")," ||
            baseClassName(ck) || A")," ||
            wSchemaNum(ck) || A"),ck);
        \/ usesMacro("IMPLEMENT_DYNAMIC(" || className(ck) || A")," ||
            baseClassName(ck) || A")," ||
            wSchemaNum(ck) || A"),ck);
    }
};
ensures \exists o:Obj<CRuntimeClass>
(o' = to_CRuntimeClass(dynamic_class_of(widen(self))) \/
result = &o);

BOOL IsKindOf(const CRuntimeClass* pClass) const {
  let ck: CompilerKnowledge be compiler_state_at_definition;
  requires (usesMacro(A"DECLARE_DYNAMIC(" || className(ck) || A")",ck) \/
    usesMacro(A"DECLARE_SERIAL(" || className(ck) || A")",ck)) \/
    isValid(pClass);
  ensures result = inherits_from(dynamic_class_of(widen(self)),
    to_DynamicClass(*pClass));
}

BOOL IsSerializable() const {
  let ck: CompilerKnowledge be compiler_state_at_definition;
  ensures result = (usesMacro(A"DECLARE_SERIAL(" ||
    className(ck) || A")",ck) \/
    usesMacro(A"IMPLEMENT_SERIAL(" || className(ck) || A"," ||
    baseClassName(ck) || A"," ||
    wSchemaNum(ck) || A")",ck));
}

virtual void Serialize(CArchive& ar) throw(CMemoryException,
  ArchiveException,
  CFileException) {
  let ck: CompilerKnowledge be compiler_state_at_definition;
  requires usesMacro(A"DECLARE_SERIAL(" || className(ck) || A")",ck) \/
    usesMacro(A"IMPLEMENT_SERIAL(" || className(ck) || A"," ||
    baseClassName(ck) || A"," ||
    wSchemaNum(ck) || A")",ck);
  modifies self, ar;
  ensures informally "the object is read from or written to the "
  "CArchive ar, depending on whether the archive"
  "is loading or storing";
}

void operator delete(void* p) {
  modifies memory_available;
  ensures memory_available' = memory_available" +
    allocatedSize(*p);
}

void* operator new(size_t nSize) throw (CMemoryException) {
extern int volatile memory_available;

requires memory_available >= memOverhead(nSize);
modifies memory_available;
ensures fresh(*result) \ allocatedSize(*result') = memOverhead(nSize) \ memory_available' = memory_available - memOverhead(nSize);

requires memory_available < memOverhead(nSize);
ensures (\exists e:CMemoryException (thrown(CMemoryException) = e));
}

void* operator new(size_t nSize, const char lpszFileName,
    int nLine) throw(CMemoryException) {
extern int volatile memory_available;

requires memory_available >= memOverhead(nSize);
modifies memory_available;
ensures fresh(*result) \ allocatedSize(*result') = memOverhead(nSize) \ memory_available' = memory_available - memOverhead(nSize);

requires memory_available < memOverhead(nSize);
ensures (\exists e:CMemoryException (thrown(CMemoryException) = e));
}

protected:

CObject() {
    modifies self;
    ensures true;
}

private:

// The presence of the private CObject copy constructor guarantees a
// compiler error message if the copy constructor of your class is
// needed but not available

CObject(const CObject& objectSrc) {
    modifies self;
    ensures self' = objectSrc;
}

void operator = (const CObject& src) {
    ensures self' = src;
}
CException

abstract class CException {
    uses NoInformationError(CException), CompilerKnowledge;
    // NOTE: Let ck : CompilerKnowledge be compiler_state_at_definition
    //    usesMacro(A"IMPLEMENT_DYNAMIC(" || enclosingClassName(ck) || "," ||
    //    baseClassName(ck) || "," ||
    //    wSchemaNum(ck)) || "," ck)
    //    must be true to use this class correctly.
};

CMemoryException

imports CException;

class CMemoryException : public CException {
    uses NoInformationSubtype(CMemoryException, CException);
    simulates CException by toSuper;

public:
    CMemoryException() {
        // NOTE: NOT TO BE INVOKED DIRECTLY BY PROGRAMMER
        // Instead, invoke global function AfxThrowMemoryException
        modifies self;
        ensures true;
    }

    ~CMemoryException() {
        // NOTE: NOT TO BE INVOKED DIRECTLY BY PROGRAMMER
        modifies self;
        ensures true;
    }
};

CObArray

imports CObject, CMemoryException;
typedef CObject* COBPtr;

class CObArray : public CObject {
    uses MSArray<Obj<Ptr<Obj<CObject>>> for Elem, CObArray for MS_Arr),
    MemoryTrait;
    invariant ((self\any).upperbound >= (-1)) \ ((self\any).size >= 0) \ ((self\any).size = (self\any).upperbound + 1);

public:
    CObArray() {
        constructs self;
        ensures self' = create;
    }

    "CObArray() {
        modifies contained_objects(self, pre), memory_available;
        ensures trashed(contained_objects(self, pre)) \ memory_available' = memory_available" +
        (memOverhead(sizeof(CObPtr)) * self".size);
    }

    int GetSize() const {
        ensures result = self".size;
    }

    int GetUpperBound() const {
        ensures result = self".upperbound;
    }

    int Add(CObject* newElement) throw(CMemoryException) {
        extern int volatile memory_available;

        requires isValid(newElement) \ (memory_available" >= memOverhead(sizeof(CObPtr)));
        modifies self, memory_available;
        ensures fresh(self'[self".size]) \ (self'[self".size])' = newElement \ self'.upperbound = self".upperbound + 1 \ self'.size = self".size + 1 \ result = self".size \ memory_available' = memory_available" -
        (memOverhead(sizeof(CObPtr))) \ 
        \forall i:int (legalIndex(self", i) => (self'[i]=self"[i]));

        requires "(memory_available" >= memOverhead(sizeof(CObPtr)));
        ensures \(\exists e:CMemoryException (thrown(CMemoryException)= e));
    }
void InsertAt(int nIndex, CObject* newElement, int nCount = 1)
throw(CMemoryException) {
    extern int volatile memory_available;

    requires nCount >= 1 \ nIndex >= 0 \ "legalIndex(self^,nIndex) /
    memory_available^ >= (memOverhead(sizeof(CObPtr)) * 
    (nCount + (nIndex - (self^ . upperbound + 1)))));
    modifies self, memory_available;
    ensures \forall k:int (0 <= k \ k <= self^ . upperbound =>
    (self^ [k])' = (self^ [k]))' /
    \forall i:int (self^ . upperbound < i \ i < nIndex =>
    fresh(self^ [i]) \ (self^ [i])' = NULL) /
    \forall j:int (nIndex <= j \ j <= (nIndex + nCount-1) =>
    fresh(self^ [j]) \ (self^ [j])' = newElement) /
    self^ . size = self^ . size + (nCount + (nIndex - (self^ . upperbound +1))) /
    self^ . upperbound = self^ . size - 1 /
    memory_available' = memory_available^ -
    (memOverhead(sizeof(CObPtr)) * (nCount + (nIndex - (self^ . upperbound+1)))));

    requires nCount >= 1 \ nIndex >= 0 \ "legalIndex(self^,nIndex) /
    memory_available^ >= (memOverhead(sizeof(CObPtr)) * nCount);
    modifies self, memory_available;
    ensures \forall i:int (self^ . upperbound < i \ i <= (nCount+self^ . upperbound) =>
    fresh(self^ [i])) /
    self^ = insert_at(self^, nIndex, newElement, nCount) /
    memory_available' = memory_available^ -
    (memOverhead(sizeof(CObPtr))* nCount);

    requires nCount >=1 \ nIndex >= 0 \ "legalIndex(self^, nIndex) /
    "(memory_available^ >= (memOverhead(sizeof(CObPtr)) * 
    (nCount + (nIndex - (self^ . upperbound + 1)))));
    ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));

    requires nCount >= 1 \ nIndex >= 0 \ legalIndex(self^, nIndex) /
    "(memory_available^ >= (memOverhead(sizeof(CObPtr)) * nCount));
    ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));
}
void RemoveAll() {

extern int volatile memory_available;

modifies self, contained_objects(self, pre), memory_available;

modifies self, memory_available;

ensures self' = create /

\exists e: CMemoryException (thrown(CMemoryException) = e));

requires legalIndex(self^, nStartIndex) \/

(memory_available^ >=

((memOverhead(sizeof(CObPtr)) * (*pNewArray)^.size)));

modifies self, memory_available;

ensures self' = create /

memory_available' = memory_available^ -

((memOverhead(sizeof(CObPtr)) * (*pNewArray)^.size));

ensures self' /= create /


\forall k:int (0 <= k ≤ self^ .upperbound) /


(self' [k] = self [k]) /

\forall i:int (self^ .upperbound < i ≤ nStartIndex) /


(fresh(self' [i]) /


(self'[i])' = NULL) /

\forall j:int (0 <= j ≤ (*pNewArray)^ .upperbound) /


(fresh(self' [(j+self^ .upperbound + 1)]) /


self' [(j+self^ .upperbound + 1)] = (*pNewArray)^ [j]) /


self'.size = self^ .size + (*pNewArray)^ .size /


self'.upperbound = self^ .size - 1 /

memory_available' = memory_available^ -

((memOverhead(sizeof(CObPtr)) * (*pNewArray)^ .size) +


(nStartSize - self^ .upperbound + 1));

ensures (\exists e: CMemoryException (thrown(CMemoryException) = e));

requires legalIndex(self^, nStartIndex) \/

(memory_available^ >=

((memOverhead(sizeof(CObPtr)) * (*pNewArray)^ .size)));

modifies self, memory_available;

ensures self' = create /

memory_available' = memory_available^ -

memory_available^ =

(memory_overhead^ (sizeof(CObPtr)) * (*pNewArray)^ .size));

ensures (\exists e: CMemoryException (thrown(CMemoryException) = e));

void RemoveAll() {

extern int volatile memory_available;

modifies self, contained_objects(self, pre), memory_available;

modifies self, memory_available;

ensures self' = create /

memory_available' = memory_available^ -

memory_available^ =

(memory_overhead^ (sizeof(CObPtr)) * (*pNewArray)^ .size));

ensures (\exists e: CMemoryException (thrown(CMemoryException) = e));

47
void RemoveAt(int nIndex, int nCount = 1) {
    extern int volatile memory_available;
    requires legalIndex(self^, nIndex) \ (1 <= nCount) \ (nCount <= (self^.upperbound - nIndex + 1));
    modifies self, contained_objects(self^, nCount);
    ensures self' = removed_at(self^, nIndex, nCount) \ exists s:Set<Object>
        (\forall i:int ((nIndex <= i \ i < (nIndex + nCount)) =
            widen(self'[i]) \in s \ \text{trashed}(s, post)) /\memory_available' = memory_available' +
            (memOverhead(sizeof(CObPtr)) * nCount);
}

CObject* GetAt(int nIndex) const {
    requires legalIndex(self^, nIndex);
    ensures result = (self^[nIndex])^;
}

CObject*& ElementAt(int nIndex) {
    requires legalIndex(self^, nIndex);
    ensures result = self^[nIndex];
}

void SetAt(int nIndex, CObject* newElement) {
    requires legalIndex(self^, nIndex);
    modifies self;
    ensures (self'[nIndex])' = newElement \ \forall i:int ((legalIndex(self^, i) \ i != nIndex) =>
        (self'[i] = self^[i]));
}

void SetAtGrow(int nIndex, CObject* newElement) throw(CEMemoryException){
    extern int volatile memory_available;
    requires legalIndex(self^, nIndex);
    modifies self;
    ensures ((self'[nIndex])' = newElement \ \forall i:int ((legalIndex(self^, i) \ i != nIndex) =>
        (self'[i] = self^[i]));
}
void SetSize(int nNewSize, int nGrowBy = -1) throw(CMemoryException) {
    extern int volatile memory_available;

    requires 0 <= nNewSize \ nNewSize <= self^.size;
    modifies self, memory_available;
    ensures \forall i:int (0 <= i \ i <= self^.upperbound =>
        self[i] = self[i]); \forall j:int (self^.upperbound < j \ j < nNewSize =>
        fresh(self[j]) \ (self[j])’ = NULL) \ (self^.size = nNewSize \ self^.upperbound = nNewSize - 1) \ memory_available’ = memory_available'' -
        (memOverhead(sizeof(CObPtr)) * (nNewSize - self^.upperbound));
}
requires nNewSize > self^ .size /
   ^(memory_available^ ^ >= (memOverhead(sizeof(CObPtr)) ^ * 
      (nNewSize - self^ ^ .size)));
ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));
}

CObject^ & operator[](int nIndex) {
  requires legalIndex(self^ ^ , nIndex);
  ensures result = self^ [nIndex];
}

CObject^ * operator[](int nIndex) const {
  requires legalIndex(self^ ^ , nIndex);
  ensures result = (self^ [nIndex]^ ^ );
}

void FreeExtra() {
  ensures self' = self^ ^ ;
}

};

CObList

imports CObject, CMemoryException, pretend_bool;

typedef CObject^ * CObPtr;

// The following class declaration needed for parsing
class POSITION;

class CObList : public CObject {
  uses MSList(Obj_Ptr_Obj_CObject for Elem, CObList for MS_List), MemoryTrait;
  invariant (length(self\any) >=0) /\ (upperbound(self\any) >= (-1));
public:
  CObList(int nBlockSize = 10) {
    constructs self;
    ensures self' = {};
  }

  ~CObList() {
    modifies contained_objects(self, pre);
  }

  int GetIndex(int nIndex) {
    requires legalIndex(self^ ^ , nIndex);
    return nIndex;
  }

  CObject^ * GetElement(int nIndex) {
    requires legalIndex(self^ ^ , nIndex);
    return self^ [nIndex];
  }

  void Insert(int nIndex, CObject^ & newElement) {
    requires legalIndex(self^ ^ , nIndex);
    Element atIndex = self^ [nIndex];
    self^ [nIndex] = newElement;
    ensure insertIndex = atIndex;
  }

  void Remove(int nIndex) {
    requires legalIndex(self^ ^ , nIndex);
    CObject^ * elem = self^ [nIndex];
    self^ [nIndex] = null;
    ensure removeIndex = elem;
  }

  void FreeExtra() {
    ensures self' = self^ ^ ;
  }
}
ensures trashed(contained_objects(self, pre)) /
    memory_available' = memory_available +
    (memOverhead(sizeof(CObPtr)) * length(self'));
}

POSITION AddHead(CObject* newElement) throw(CMemoryException) {
    extern int volatile memory_available;

    requires memory_available >= memOverhead(sizeof(CObPtr));
    modifies self, memory_available;
    ensures fresh(self'[length(self')]) /
    self' = {newElement} || self' /
    memory_available' =
    memory_available' - memOverhead(sizeof(CObPtr)) /
    result = position_at(self', 0);

    requires memory_available < memOverhead(sizeof(CObPtr));
    ensures (\exists l:CMemoryException (thrown(CMemoryException)= e));
}

void AddHead(CObList* pNewList) throw(CMemoryException) {
    extern int volatile memory_available;

    requires memory_available >=
    (memOverhead(sizeof(CObPtr)) * length(*pNewList));
    modifies self, memory_available;
    ensures forall i:int (upperbound(self') < i /
    i <= upperbound(*pNewList') =>
    fresh(self'[i]) /
    self' = (*pNewList') || self' /
    memory_available' = memory_available -
    (memOverhead(sizeof(CObPtr)) * length(*pNewList'));

    requires memory_available <
    (memOverhead(sizeof(CObPtr)) * size(*pNewList'));
    ensures (\exists l:CMemoryException (thrown(CMemoryException)= e));
}

POSITION AddTail(CObject* newElement) throw(CMemoryException) {
    extern int volatile memory_available;

    requires memory_available >= memOverhead(sizeof(CObPtr));
    modifies self, memory_available;
    ensures fresh(self'[length(self')]) /
    self' = self' || {newElement} /

memory_available' =
  memory_available" - (memOverhead(sizeof(CObPtr))) \/
  result = position_at(self', upperbound(self'));

requires memory_available" < memOverhead(sizeof(CObPtr));
ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));

void AddTail(CObList* pNewList) throw(CMemoryException) {
  extern int volatile memory_available;

  requires memory_available" =>
    (memOverhead(sizeof(CObPtr)) * length(*pNewList"));
  modifies self, memory_available;
  ensures \forall i:int (upperbound(self") < i \&\& i <= upperbound(*pNewList") =>
    fresh(self'[i]) \&\& self' = self" || (*pNewList") \&\&
    memory_available' = memory_available" -
    (memOverhead(sizeof(CObPtr)) * length(*pNewList"));

  requires memory_available" <
    (memOverhead(sizeof(CObPtr)) * size(*pNewList"));
  ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));
}

POSITION Find(CObject* searchValue, POSITION startAfter = NULL) const {
  requires isValid(searchValue);
  ensures result = find(self", searchValue, startAfter);
}

POSITION FindIndex(int nIndex) const {
  requires ValidIndex(self", nIndex);
  ensures result = position_at(self", nIndex);

  requires "ValidIndex(self", nIndex);
  ensures result = NULL;
}

CObject*& GetAt(POSITION position) {
  requires ValidPosition(position);
  ensures result = retrieve_at_pos(self", position);
}

CObject* GetAt(POSITION position) const {
  requires ValidPosition(position);

  memory_available" =
  memory_available" - (memOverhead(sizeof(CObPtr))) \/
  result = position_at(self', upperbound(self'));

  requires memory_available" < memOverhead(sizeof(CObPtr));
  ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));
}
ensures result = (retrieve_at_pos(self^, position))`;
`
}

int GetCount() const {
  ensures result = length(self`);
}

CObject*& GetHead() {
  requires "IsEmpty(self`);
  ensures result = element_at(self`, 0);
}

CObject* GetHead() const {
  requires "IsEmpty(self`);
  ensures result = (element_at(self`, 0))`;
}

POSITION GetHeadPosition() const {
  requires "IsEmpty(self`);
  ensures result = position_at(self`, 0);

  requires IsEmpty(self`);
  ensures result = NULL;
}

CObject*& GetNext(POSITION& rPosition) {
  requires ValidPosition(rPosition);
  modifies rPosition;
  ensures result = retrieve_at_pos(self`, rPosition`) /
    (index_at(self`, rPosition`, 0) = upperbound(self`) =>
      rPosition` = NULL /
      "(index_at(self`, rPosition`, 0) = upperbound(self`) =>
        rPosition` = position_at(L, index_at(self`, rPosition`, 0) + 1);"
}

CObject* GetNext(POSITION& rPosition) const {
  requires ValidPosition(rPosition);
  modifies rPosition;
  ensures result = retrieve_at_pos(self`, rPosition`)` /
    (index_at(self`, rPosition`, 0) = upperbound(self`) =>
      rPosition` = NULL /
      "(index_at(self`, rPosition`, 0) = upperbound(self`) =>
        rPosition` = position_at(L, index_at(self`, rPosition, 0) + 1);";
CObject*& GetPrev(POSITION& rPosition) {
    requires ValidPosition(rPosition);
    modifies rPosition;
    ensures result = retrieve_at_pos(self^, rPosition^) /\n        (index_at(self^, rPosition^, 0) = 0) => rPosition' = NULL /\n        "(index_at(self^, rPosition^, 0) = 0) => rPosition' = position_at(L, index_at(self^, rPosition^, 0) - 1);"
}

CObject* GetPrev(POSITION& rPosition) const {
    requires ValidPosition(rPosition);
    modifies rPosition;
    ensures result = retrieve_at_pos(self^, rPosition^) /\n        (index_at(self^, rPosition^, 0) = 0) => rPosition' = NULL /\n        "(index_at(self^, rPosition^, 0) = 0) => rPosition' = position_at(L, index_at(self^, rPosition^, 0) - 1);"
}

CObject*& GetTail() {
    requires "IsEmpty(self^)";
    ensures result = element_at(self^, upperbound(self^));
}

CObject* GetTail() const {
    requires "IsEmpty(self^)";
    ensures result = (element_at(self^, upperbound(self^)))^;
}

POSITION GetTailPosition() const {
    requires "IsEmpty(self^)";
    ensures result = position_at(self^, upperbound(self^));

    requires IsEmpty(self^);
    ensures result = NULL;
}

BOOL IsEmpty() const {
    ensures result = IsEmpty(self^);
}

void RemoveAll() {
    extern int volatile memory_available;
}
modifies self, contained_objects(self, pre), memory_available;
ensures self' = {} \/
  trashed(contained_objects(self, pre)) \/
  memory_available' = memory_available" +
  (memOverhead(sizeof(CObPtr)) * length(self"));

POSITION InsertAfter(POSITION position, CObject *newElement)
  throw(CMemoryException) {
    extern int volatile memory_available;
    let E: MS_List be {newElement};
    requires ValidPosition(position) /
      (memory_available" >= memOverhead(sizeof(CObPtr)));
    modifies self, memory_available;
    ensures fresh(self'[length(self")]) /
      self' = head_incl(self", position) ||
      E || tail_not_incl(self", position) /
      memory_available' = memory_available" - memOverhead(sizeof(CObPtr)) /
      result = position_at(E, 0);
    requires (position = NULL) /
      (memory_available" >= memOverhead(sizeof(CObPtr)));
    modifies self, memory_available;
    ensures fresh(self'[length(self")]) /
      self' = self" || {newElement} /
      memory_available' = memory_available" - memOverhead(sizeof(CObPtr)) /
      result = position_at(, 0);
    requires memory_available" < memOverhead(sizeof(CObPtr));
    ensures (\exists e:CMemoryException (thrown(CMemoryException)= e));
  }

POSITION InsertBefore(POSITION position, CObject *newElement)
  throw(CMemoryException) {
    extern int volatile memory_available;
    let E: MS_List be {newElement};
    requires ValidPosition(position) /
      (memory_available" >= memOverhead(sizeof(CObPtr)));
    modifies self, memory_available;
    ensures fresh(self'[length(self")]) /
      self' = head_not_incl(self", position) ||
      E || tail_incl(self", position) /
      memory_available' = memory_available" - memOverhead(sizeof(CObPtr)) /

result = position_at(E, 0);

requires (position = NULL) /
    (memory_available' >= memOverhead(sizeof(CObPtr)));
modifies self, memory_available;
ensures fresh(self' [length(self' )]) /
    self' = {newElement} || self' /
    memory_available' = memory_available" - memOverhead(sizeof(CObPtr)) /
    result = position_at(self', 0);

requires memory_available" < memOverhead(sizeof(CObPtr));
ensures (\exists e: CMemoryException (thrown(CMemoryException) = e));

CObject* RemoveHead() {
    extern int volatile memory_available;

    requires ~IsEmpty(self");
    modifies self, contained_objects(self, pre), memory_available;
    ensures result = (element_at(self", 0))" /
        self" = tail_not_incl(self", position_at(self", 0))" /
        trashed(self"[0])" /
        memory_available' = memory_available" + memOverhead(sizeof(CObPtr));
}

CObject* RemoveTail() {
    extern int volatile memory_available;

    requires ~IsEmpty(self");
    modifies self, contained_objects(self, pre), memory_available;
    ensures result = (element_at(self", upperbound(self")))" /
        trashed(self"[upperbound(self")])" /
        self" = head_not_incl(self", position_at(self", upperbound(self")))" /
        memory_available' = memory_available" + memOverhead(sizeof(CObPtr));
}

void SetAt(POSITION position, CObject* newElement) {
    requires ValidPosition(position);
    modifies self;
    ensures self" = head_not_incl(self", position) ||
        {newElement} || tail_not_incl(self", position);
}
References


