Using the .NET Profiler API to Collect Object Instances for Constraint Evaluation

Dave Arnold
School of Computer Science
Carleton University
1125 Colonel By Drive
Canada K1S 5B6, Ottawa, ON
darnold@scs.carleton.ca

Jean-Pierre Corriveau
School of Computer Science
Carleton University
1125 Colonel By Drive
Canada K1S 5B6, Ottawa, ON
jeanpier@scs.carleton.ca

ABSTRACT

Evaluating software based constraints at runtime is an important task for both the validation and verification of software. It is not uncommon to encounter constraints that require obtaining the set of all active object instances for a given classifier. When the application under test is being executed on a virtual machine or a managed runtime, it is often difficult, if not impossible, to obtain such a set. We will examine Microsoft's .NET common language runtime, and through the use of the profiler API, provide a concrete mechanism for obtaining the set of live object instances for a given classifier. We will then leverage this set to provide an extension to an existing C# and Object Constraint Language compiler to support the OclAny::allInstances operation.

Keywords
C#, Constraints, Profiler, OCL

1. INTRODUCTION

Software based constraints provide a mechanism for testing software. Such constraints can be expressed using a formal language such as the Object Management Group's Object Constraint Language (OCL) [Omg03a]. Constraints are expressed at the model level in the form of preconditions, postconditions and class invariants [Fra03a, War03a]. In our work, when a model is used to generate source code, the constraints are translated from the model level to the code level. The source code is then compiled through the use of a specialized compiler [Arn04a] to generate executable code. In the case of the C# programming language [Hew02a], this code is executed by Microsoft's Common Language Runtime (CLR) [Hew02b]. The CLR is not a virtual machine, but rather an execution engine. The CLR provides memory management for both allocation and garbage collection. As the CLR abstracts memory management away from the programmer, it is difficult to determine which object instances are allocated and active. The CLR does not provide any feedback to the application being executed about the state of the application's memory. That is, there is no way to determine the object instance information for a given classifier within the containing application.

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

Context

Our paper will present an approach for accessing memory management information from the CLR via the .NET Profiler API [Mic02a]. Our approach will track each object instance of a given classifier from allocation through to garbage collection. We will demonstrate that our profiler, on request from the application being executed, can return the set of all object instances for a given classifier. The object instance set can then be used for various activities including the evaluation of software based constraints.

Organization

The remainder of this paper is organized as follows: Section 2 provides a brief background on the CLR's memory management and garbage collection algorithms. Section 3 presents an unmanaged Component Object Model (COM) component that implements the .NET Profiler API to interface with the CLR and respond to memory allocation events. Section 4 examines how the unmanaged COM component can exchange information with the managed application being profiled. Section 5 provides a concrete use of our method by implementing the previously missing OclAny::allInstances operation in an existing C#/OCL compiler [Arn04a]. Finally, Section 6 presents concluding remarks and areas for future work.

2. CLR MEMORY MANAGEMENT

Instances of classifiers in .NET are allocated from a section of memory known as the managed heap [Stu03a]. The heap is managed because after an application requests memory, the garbage collector takes care of the cleanup. Object instances can be small, containing a few integers, or larger, for example holding a database connection with an extensive amount of state information. Object instances can be self-contained or reference other object instances. The role of the garbage collector is to determine when objects should be collected to free memory for other allocations. The garbage collector fills its role by selecting the object instances that can be deleted. Garbage collection is performed when an application attempts to allocate memory from the managed heap, and the managed heap is too full to complete the request. Managed heaps in the CLR are periodically renewed by identifying dead objects and then fusing contiguous runs of dead objects into blocks of memory to be reallocated. The method used for discovering dead objects is called tracing. Tracing is accomplished by following live references to objects in the managed heap. Once a live reference is encountered it is marked. The garbage collector can then easily determine that any object instance that is not marked can be reclaimed. Live objects are located by looking for heap pointers on all the stacks, in all statically allocated memory, within all object instances, and in a few other CLR data structures [Stu03a]. When a live object is located, the memory that the object points to is examined for additional references (pointers). If more are found they are likewise followed until the entire set of live objects is known. The action of determining the live object set is called "tracing the roots", and results in the transitive closure of the set of live objects.

The approach to garbage collection described in the previous paragraph is known as "mark and sweep" collection [Stu03a]. The problem with pure mark and sweep collection is that over time the managed heap becomes fragmented. To avoid heap fragmentation, "compacting collection" is used. Compacting collection removes dead objects and pockets of unallocated memory by sliding live objects down towards the low-address end of each heap segment, and then repairing any dangling pointers with updated values. Such compaction also has the positive side effect of maintaining object creation order, which improves locality of reference.

The expenses of all the object movement can be reduced drastically via an enhancement used by the CLR's garbage collector known as "generational collection" [Stu03a]. When a generational approach is used, object instances are initially allocated in the youngest generation. If they survive past a garbage collection cycle, they are promoted to an elder generation by copying. The refinement of this method over compacting collection is that object instances that are located in the younger generation generally have a shorter survival rate, while objects in the elder generation have a higher survival rate. As object instances are split into different managed heaps, different techniques are used to reclaim memory. The CLR uses a non-copying, non-compacting collector for the elder generations. In the youngest generation, a copying approach is used. The CLR garbage collector is triggered by allocation volume or memory scarcity; when heap resources run low, the roots are traced, and either one or both generations are scavenged for memory. For details on how the CLR garbage collector is implemented see [Stu03a].

Garbage collection is well worth the complexity and the effort [Hil03a]. Garbage collection provides additional application reliability and programmer productivity. However, since garbage collection can be triggered without notice and because the managed application is not notified when garbage collection takes place, it is hard to determine which object instances are active at a given point during execution. We will now examine a method for accessing the managed heaps directly to extract the necessary object instance information.

3. THE PROFILER API

In order to obtain the set of all live object instances, it is obvious that we require a way to get inside the CLR and examine the managed heaps. Unfortunately, the only way to implement such functionality would require modifying the CLR itself. But, modifying the CLR is not a practical solution. Fortunately, for our purposes Microsoft has provided a back door into the inner workings of the CLR. This back door is the profiler API [Hil03a]. The profiler API allows for an external COM component to monitor the execution and memory usage of an application running under the CLR. Normally, the profiler monitors the running application and does not interfere with it. In our approach, we will leverage the profiler API to monitor object instance allocation and garbage collection, and we will return this information to the managed application.

The profiling APIs are implemented via two COM interfaces. One of the interfaces is implemented by the CLR (ICorProfilerInfo), and the other is implemented by the profiler itself (ICorProfilerCallback). The ICoreProfilerCallback interface receives notification from the CLR regarding various events that occur during a managed application's execution. The ICorProfilerInfo interface extracts additional information from the CLR itself.

Initialization

The CLR connects with one profiler at most during its initialization phase [Hil03a]. The profiler must use the Initialize method defined in the ICorProfilerCallback interface to save the ICorProfilerInfo interface pointer so that it can be used to retrieve additional information from the CLR during actual profiling activities. The Initialize method must also register for CLR events that the profiler is interested in. The ICorProfilerCallback interface supports approximately sixty CLR events. To reduce the amount of overhead introduced by the profiler, the profiler specifies which events it is interested in. For our task, we wish to be notified when a new object instance is allocated, when the garbage collector is invoked, and finally we need to be notified when an object reference (pointer) is moved. The last event is required because we will be maintaining a set of pointers to the actual object instance memory locations. Table 1 illustrates the profiler event bit masks we are using. For our purposes of object instance collection and tracking, we only need to implement four of the sixty ICorProfilerCallback events. The following sections will describe each of the methods, and their corresponding implementation details.

Event Mask	Meaning
COR_PRF_MONITOR_ SUSPENDS	GC Notification
COR_PRF_MONITOR_GC	GC Calls
COR_PRF_ENABLE_OBJECT_ ALLOCATED	Object Allocation
COR_PRF_MONITOR_OBJECT_ ALLOCATED	Object Allocation

Table 1. Select Profiling Events

ObjectAllocated

The ObjectAllocated method is invoked by the CLR each and every time memory in the managed heap has been allocated for an object [Hil03a]. The method provides two parameters; the first parameter is a pointer to the managed heap location where the newly allocated object instance is being stored: objectId. The second parameter is a pointer to the class descriptor for the objectId: classId.

Our implementation is fairly straightforward: the class descriptor is used along with the previously saved ICorProfilerInfo interface pointer to determine the classifier name. The classifier name is then compared against the set of given classifier names that we are "interested" in. A classifier becomes interesting when the application being profiled notifies us that we will be asked for the classifier's object instance set. Details of how this notification works will be provided in Section 4; for now it suffices that we are only interested in a subset of the list of classifiers. In an effort to reduce the resources needed by the profiler, rather than store all of the object instance information, only instance information for classifiers that are deemed to be interesting is stored.

MovedReferences

The MovedReferences method is invoked by the CLR to notify the profiler that the garbage collector has moved one or more object instance locations [Hil03a]. When this occurs, the objectIds provided by the ObjectAllocated method are no longer valid, as they may no longer point to the correct location within the managed heap. It should be noted, that the object's internal state does not change, just its location within the managed heap. In the context of our profiler, all we are doing is updating our internal arrays to reflect the movements.

ObjectReferences

The ObjectReferences method is called by the CLR once for each object instance that remains in the managed heap after a garbage collection operation has completed [Hil03a]. We use the ObjectReferences method to mark the object instances as un-collected, and the object instances are still kept inside our array of objectIds.

RuntimeSuspendFinished

The CLR calls RuntimeSuspendFinished to notify the profiler that the CLR has suspended all of the threads needed for execution suspension [Hil03a]. One of the reasons for runtime suspension is garbage collection. As the ObjectReferences method will be called for each object instance that survives when the runtime is suspended, we will mark each of the tracked object instances as collected. When the ObjectReferences method calls are complete, the object instances that have survived the garbage collection will be un-collected. Our array will then only contain the objectIds of object instances that are still live.

Intuitively, this may seem like a bad idea because there will be a delay between when we mark all the object instances as collected, and when we realize that a given object instance is live, and needs to be un-collected. The delay is not a problem because the

CLR guarantees all of the ObjectReferences calls will be performed before the CLR's execution threads are restarted.

Summary

By providing a specialized implementation for the preceding four ICorProfilerCallback methods our profiler is able to maintain an internal array of pointers for each instance of the interesting classifiers. Each pointer is a reference to a live object instance on the managed heap. During garbage collection, the runtime is suspended and each object instance we are tracking is marked as collected. Before the runtime is restarted, the CLR provides notification for each object instance that has survived garbage collection. We are then able to un-collect the object instance pointers stored in the array. Finally, should the garbage collector compact the managed heap, our profiler will receive notification so that the heap pointers can be updated accordingly.

We have now explained a COM component that interfaces with the CLR. The component registers for object allocation and garbage collection events. The events are used to maintain an array of currently live object instances for interesting classifiers. The next task, presented in Section 4, is to provide a managed interface into the COM component so that the managed application, which is being profiled, can register its classifiers as being interesting and access the object instance pointer array.

4. GETTING OBJECT INSTANCES

As our COM component is loaded and initialized by the CLR running in an unmanaged memory space, the COM component is unable to call methods that are located inside the managed application. However, managed applications can invoke methods that are exported by a COM component. To allow a managed application to query the array of live object instances for a given classifier, this COM component will have to be able to register a given classifier, request the number of live object instances allocated, and finally be able to move the allocated object instances from the unmanaged COM component into the managed application for inspection. These tasks are implemented via the provision of five methods exported by our COM component. The following sections will discuss each of the five exported methods in detail.

IsOCLProfilerAttached

The first exported method determines if the CLR running the managed application has loaded our profiler. The method name contains the abbreviation OCL, for the Object Constraint Language as our implementation of the described profiler is for use with the OCL. More details of our implementation will be provided in Section 5.

Implementation of this method consists of determining if a global reference to the ICorProfilerCallback interface contains a valid pointer. If a valid pointer is located then the profiler has been loaded correctly, otherwise the profiler is not running. The IsOCLProfilerAttached method is not required, but is used as a safety mechanism to determine if the required profiler functionality exists before the managed application calls the remaining four exported methods.

RegisterObject

RegisterObject is used to inform the attached profiler that the managed application would like to keep track of object instances for the given classifier. Classifiers are provided via the single string parameter to the RegisterObject method. The string should contain a fully qualified classifier name. For example, suppose the class Customer existed in the DaveArnold.Data namespace. The call to RegisterObject would take the following form: *RegisterObject ("DaveArnold. Data.Customer")*.

Each call to the RegisterObject method adds the given classifier name to the list of interesting classifiers. Profiling only begins following the RegisterObject call. In order to achieve accurate object instance information, the RegisterObject calls should be made immediately after the managed application starts.

GetInstanceCount

The GetInstanceCount method takes a single string parameter, and returns an integer value. The parameter is the fully qualified name of the classifier for which the number of live object instances is requested. GetInstanceCount will invoke the garbage collector to determine which object instances are live at the current time. GetInstanceCount will also wait until the thread processing the queue of finalizers has finished. A finalization method can be viewed as a destructor. The finalization queue is the set of instances that have been marked for deletion, but the runtime has yet to execute the finalization method. Depending on the number and complexity of the finalizers to be executed, GetInstanceCount may be computationally intensive. However, the strategy of forcing garbage collection and waiting for finalizer execution, ensures that the return value is always accurate.

StartInstanceCopy & StopInstanceCopy

StartInstanceCopy is used to inform the profiler that the managed application has requested the list of all object instances for a given classifier. StopInstanceCopy informs the profiler that the managed application has received the requested object instance list. The process of transferring the list of object instances from our COM component to a managed application is a non-trivial operation. The following sections will provide rationale for the operation's complexity, and present the technical details of how the transfer process is accomplished.

4.1.1 Direct Access via the Array Pointer

Intuitively, the easiest implementation would have been to pass the fully qualified classifier name to an exported method, and have the method return a pointer to the corresponding array. The managed application could access the array of pointers and de-reference each one for evaluation. The experienced .NET programmer will quickly realize that a managed application cannot take the address or size of a managed type. The reason for this is if the garbage collector is executed and moves the managed object instance, the object instance array and all pointers to the array will become invalid. As we cannot prevent the garbage collector from executing, nor keep a reference to a managed object, another method is required to get the object instance pointers out of the profiler and into the managed application.

4.1.2 Memory Copy

As each array element in the profiler stores a pointer to the managed heap location where the object instance is being stored, the actual bits can be copied to a new location, which is accessible from the managed application. To allow the managed application access to the memory, we will use the managed application to create a new object instance for the given classifier. We will then use the profiler to copy the memory from the existing live object instance to the newly created object instance. The result is that the managed application will create a new object instance for each element in the array stored in the profiler, and then upon creation, the profiler copies the original element's state information to the new object instance. The new object instances can then be used as needed in the managed application. If the new object instances are modified in the managed application, the original instances are not modified. The following code listing presents a C# method that returns an ArrayList of live object instances for the given classifier type, using the previously described operation.

```
1) public static ArrayList GetInstancesFor(string value,
      Type t) {
2)    VerifyProfiler();
3)    lock(typeof(OCLProfilerControl)) {
4)       int count = GetInstanceCount(value);
5)       ArrayList result = new ArrayList();
6)       StartInstanceCopy();
7)       for(int i=0;i<count;i++) {
8)          Object obj = t.InvokeMember(null,
             BindingFlags.DeclaredOnly | BindingFlags.Public |
               BindingFlags.NonPublic | BindingFlags.Instance |
               BindingFlags.CreateInstance, null, null, null);
9)          result.Add(obj);  }
10)      StopInstanceCopy();
11)      return result; } }
```

Line 2 begins by ensuring that the profiler has been loaded and attached. If the profiler is not available an exception will be thrown. Line 3 creates a mutual-exclusion lock on the remainder of the method. Such a lock is required along with various critical sections in the profiler to prevent new object instances from being allocated during the copy process. In addition, a critical section in the profiler prevents the garbage collector from executing until the copy process has completed. For implementation details regarding threading see [Arn04a]. Line 4 uses the previously defined GetInstanceCount method to determine how many object instances will need to be copied. GetInstanceCount also triggers the garbage collector and finalization process so that the object instance array contains accurate information. Line 6 informs the profiler that the next object instances that we create will be copies of existing ones, and not regular object instances. Lines 7 through 9 create a new object instance for each existing instance. The creation process invokes the ObjectAllocated method in the profiler. Instead of executing the normal behaviour of adding another object instance to the given classifier’s array as previously described, the following behaviour is executed. Based on the number of instances copied since the call to StartInstanceCopy, the profiler is able to determine which element of the array to copy. The profiler then uses the saved ICorProfilerInfo interface pointer to determine the size of the object instance via the GetObjectSize method. Finally, the profiler copies the bits used to store the object instance located at the previously determined array index to the location where the newly created objectId has been located in the managed heap. Once the copy process is completed, the profiler increments an internal counter so that the next array index is used on subsequent calls to ObjectAllocated. Line 9 adds the newly created object instance, which is now a copy of the original one, to the result list. Finally, line 10 informs the profiler that any newly created objects are no longer copies of existing ones.

Summary

Calling the GetInstancesFor method shown above will return an ArrayList that contains a copy of every live object instance matching the fully qualified classifier name provided to the method call. The object instance copies can be used for any activity without affecting the original live object instances.

We have now defined a CLR profiler to track object instance allocation and garbage collection, and have created a connection to the managed application being profiled so that the live object instance set can be accessed. The following section will examine a concrete example of how the profiler and corresponding connection can be used to aid in the execution of software based contracts.

5. EVALUATING THE OCL IN C#

The Object Constraint Language (OCL) [Omg03a] is a constraint specification language with precise semantics [War03a]. More specifically, OCL expressions evaluate without side effects. This means, the state of a system can never change due to the evaluation of an OCL expression. The OCL is not a programming language. It is not possible to write logic or flow control statements in the OCL. A process or thread cannot be created, and only query operations may be called. A query operation is an operation that does not produce side effects. As the OCL is a modeling language by definition, its expressions are not directly executable. The OCL is a strongly typed language. Each OCL expression has a type. To be well formed: every OCL expression must conform to the OCL type rules [Omg03a].

We have integrated OCL version 2.0 assertions into the C# programming language. To support this addition, a specialized compiler has been developed that compiles C# source code along with OCL assertions to provide software based constraint evaluation [Arn04a].

OCL Integration

Keeping with C#'s design goal of simplicity [Tru02a], we used a straightforward notation that allows for maximum flexibility. Our experience has indicated that some developers prefer to inline the OCL in close proximity to the corresponding C# element. Other developers prefer to keep the OCL in a separate repository that is linked to the corresponding C# elements at compile-time. We support both approaches.

OCL blocks are denoted by the "OCL" keyword. The keyword is immediately followed by an opening square bracket. Following the opening bracket, a series of C# style literal strings specify the OCL constraint. A closing square bracket is required to denote the end of the OCL block.

Class invariants can be assigned to any C# structure or class. Preconditions and postconditions can be applied to any C# method, constructor, destructor, delegate, property, or indexer. Our specialized compiler can be configured to enforce, use, or ignore the OCL contracts.

Compilation

Our specialized OCL/C# compiler is based on the Mono C# compiler [Xim04a]. The Mono C# compiler is an open source C# compiler, which allowed us to directly integrate OCL constraints into the core of the compiler. In order to allow for the OCL blocks as defined in the previous section to be processed by the C# compiler, we need to augment the C# grammar accordingly. The grammar modification is straightforward. C# defines the notion of attributes [Hew02a]. Attributes can be placed on any programming element in any order and represent additional metadata for the given programming element. Grammatically, our OCL blocks behave like C# attributes. Our C# grammar modification consists of adding a rule that states that wherever attributes can be specified, zero or more OCL blocks can be specified immediately before the attributes. OCL blocks are defined separately from attributes to allow enforcement of their usage by our compiler.

We run the C# compiler until mid-way through the semantic pass. The C# compiler is then stopped so that each of the OCL constraints can be processed. We now need to convert each OCL constraint into a C# assertion. To accomplish this, we go through each operation attached to each structure or class. When a method, delegate, property, or indexer is encountered the following steps are executed[1].

1. Create a C# parse tree for each of the invariants assigned to the class that contains the operation.

2. Create a C# parse tree for each of the preconditions and postconditions assigned to the operation.

3. If a postcondition uses an element that contains the @pre modifier, a local variable is added to the beginning of the operation and is assigned the value of the requested element. The local variable is then used in the postcondition to represent the requested element's value before the operation is executed. The mini C# parse tree for the postcondition is modified to use the local variable, instead of the actual element.

4. If the operation contains either invariants or postconditions, a local variable (result) is added to the beginning of the operation to represent the return value of the operation. If the return type is void, no variable is added. The C# parse tree for the operation is modified so that all return statements are replaced

[1] The following steps do not take into consideration inheritance in order to preserve understandability in the context of this paper.

by an assignment to the local variable and then a jump to the end of the operation. As we need to check invariants and postconditions at the end of the operation, we change the structure of the method to include an area for making the checks, and enforce that all code paths travel through our new area.

5. The OCL specification dictates that the end result of each of the mini C# parse trees is a Boolean constraint. The Boolean constraints are added to the method's parse tree as follows:

(a) Each invariant constraint is placed into the condition section of an if statement and negated. If the if statement evaluates to true, then the invariant has failed. The body of the if statement will generate an assertion. The invariant statements are placed at both the beginning and the end of the operation.

(b) Step (a) is repeated for the preconditions and the if statements are placed immediately after the beginning invariant if statement.

(c) A Boolean constraint is generated to yield the result of the postconditions.

(d) An if statement is created to determine if the postconditions have failed. The body of the if statement will generate an assertion.

6. After the final if statement in the operation, a new return statement is added to return the result variable. If the result variable does not exist, no return statement is added.

Once each operation's C# parse tree has been updated to include the OCL constraints, the C# compiler is restarted. The rest of the semantic analysis is completed on the main C# parse tree, which includes the additions made by the OCL integration. Upon successful completion of the semantic pass, the C# code generator completes the compilation.

allInstances

The OCL defines an allInstances operation on each classifier. The allInstances operation is defined to return a collection of all the object instances defined using the classifier [Omg03a]. The original version of our C#/OCL compiler did not support allInstances because, as already discussed, C# maintains an automatic garbage collector, so it was difficult to determine when an object instance had actually gone out of scope. In addition, there was no mechanism in C# to get the live object instance list.

With our previously discussed method, we can modify the compiler to support the allInstances operation and allow the user more flexibility when defining software contracts. We will discuss the modifications made to the compiler in order to provide this functionality. As the implementation of the allInstances method will require invoking the profiler and incur additional overhead during application execution, we have created a compiler option to enable allInstances support. If an application that uses the allInstances operation is compiled, and the corresponding option is turned off the compiler will issue an error. If the allInstances compiler operation is enabled, but the application being compiled does not make use of the allInstances operation, the compiler will not add profiling code to the application.

The original compiler already has the allInstances operation defined in the lexical analysis and parsing phases. The semantic analyzer has a skeleton method that emits a compiler error, stating that the allInstances operation is not supported. We have replaced the existing method in the semantic analyzer with one that performs the following tasks. The first step is to ensure that the required compiler option has been enabled, if not the compiler issues an error message. Once it has been determined that the allInstances operation is supported, the compiler uses the OCL expression resolution method [Arn05a] to resolve the front part of the expression. Consider the following allInstances expression.

```
Customer.allInstances()->forAll(c : Customer | c.age >= 18)
```

The compiler resolves the front part of the expression, which should result in a classifier. Once the classifier is resolved, the compiler ensures that the classifier is not a primitive type. According to the OCL specification [Omg03a], the allInstances operation is not defined on primitive types. This makes sense because some primitive types are stored as literals or on the stack, and not in the managed heap. In addition, the set of all integers is not really useful from the software constraint point of view. If the classifier is in fact a primitive type, the compiler will issue an error.

Once the previously defined classifier resolution and primitive type check are complete, the compiler begins to generate C# code to implement the allInstances operation. The first step is to register the classifier with the profiler upon application startup. This is accomplished by inserting a call to the RegisterObject method at the beginning of the application’s entry point. With classifier registration complete, the compiler then generates code to implement actual retrieval of object instances. The OCL expression is translated into the following C# code.

```
bool result = true;
foreach(Customer c in
  (Set)OCLProfilerControl.GetInstancesFor(“Customer”,
  System.Type.GetType(“Customer”))) {
    result = result & (c.age >= 18);
}
```

The expression above, results in a Boolean value specifying if the OCL constraint is valid or not. The GetInstancesFor method is used to return an ArrayList containing the active object instances of type Customer. The first parameter to the method call is a string literal representing the classifier name, the second parameter is a System.Type object representing our Customer. The type object will be used to dynamically create the Customer copies as previously discussed. The GetInstancesFor method returns an ArrayList, the OCL specification indicates that the allInstances operation returns a Set. The Set type does not exist in the .NET Framework Class Library (FCL). The compiler includes an OCL type library [Arn04a], which defines the OCL Set type [Omg03a]. The Set type contains a conversion operator to convert an ArrayList to a Set. Finally, the foreach C# primitive is used to iterate through each Customer in the Set and determine if the age constraint holds. With the OCL allInstances expression converted to a C# Boolean expression, the C# code can be inserted into the application being compiled as discussed in the previous section.

6. CONCLUDING REMARKS

The following section will look at some areas for future work and recapitulate our approach by discussing how the addition to our existing C#/OCL compiler provides the constraint developer with additional resources for writing accurate and detailed constraints.

Future Work

We have only illustrated how this method can be used to implement software based constraints via the OclAny::allInstances method. It would be interesting to explore other uses for the complete set of live object instances. We are currently exploring how our method can be used in the verification and validation of non-functional requirements.

Conclusion

We have seen how a specialized COM component can be written using the Microsoft .NET Profiler API. The profiler API provides our component with notifications when object instances are being allocated on the managed heap, when the object instances are being moved, and finally when they are collected. Using these notifications we are able to maintain a list of live object instances sorted by the creating classifier. As the COM component runs outside of the managed runtime provided by the CLR, a series of exported methods are required to provide an interface for accessing the live object instance list under the CLR. Using the COM component together with the connecting bridge we are able to extend our existing C#/OCL compiler to provide support for the OclAny::allInstances operation. Such support empowers the software constraint designer with additional resources form which more detailed and accurate software constraints can be devised. Ultimately, allowing the constraint designer to create constraints that are not limited by technical aspects, leads to a more complete and accurate software verification and validation process.

7. ACKNOWLEDGEMENTS

Funding for this work has been generously provided by the Natural Sciences and Engineering Research Council of Canada.

8. REFERENCES

[Arn05a] Arnold, D. Constraints in C# using the OCL 2.0. In proceedings of the 23rd IASTED International Conference on Software Engineering, Innsbruck, February, 2005.

[Arn04a] Arnold, D. C#/OCL Compiler at http://www.ewebsimplex.ca/csocl.

[Fra03a] Frankel, D. Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, New York, 2003.

[Hew02a] Hewlett-Packard, Intel, Microsoft, C# Language Specification. Technical report, ECMA, 2002.

[Hew02b] Hewlett-Packard, Intel, Microsoft, Common Language Infrastructure (CLI). Technical report, ECMA, 2002.

[Hil03a] Hilyard, J. Inspect and Optimize Your Program's Memory Usage with the .NET Profiler API. MSDN Magazine, January, 2003.

[Mic02a] Microsoft, .NET Framework Tool Developer's Guide: Profiling Specification. Technical report, Microsoft, 2002.

[Omg03a] OMG: Response to the UML 2.0 RfP. Technical report, OMG document ad2003-01-16, 2003.

[Stu03a] Stutz, D., Neward, T., and Shilling, G. Shared Source CLI Essentials. O'Reilly & Associates, Sebastopol, 2003.

[Tru02a] Trupin, J. Sharp New Language: C# Offers the Power of C++ and Simplicity of Visual Basic. MSDN Magazine, September, 2002.

[War03a] Warmer, J., Kleppe, A. The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Boston, MA, 2003.

[Xim04a] Ximian Mono Project at http://developer.ximian.com/projects/mono