SEIF Project: ResourceContracts.NET

Jonathan Tapicer, Diego Garbervetsky and Martin Rouaux
Departamento de Computación, FCEyN, UBA
Buenos Aires, Argentina
{jtapicer, diegog, mrouaux}@dc.uba.ar

March 4, 2011

Abstract

We present an extension of the specification language of Code Contracts to support resource usage specifications in .NET programs. In particular, we focus on dynamic memory consumption, a resource that can be occupied and released during program execution by a memory manager. We propose a new set of annotations enabling specification of both memory consumption and lifetime properties in a modular fashion. To verify the correctness of these annotations we rely on the Code Contracts static verifier and a points-to analysis. In order to overcome some arithmetic limitations of the engine we incorporate a symbolic calculator capable of dealing with polynomial consumption. This approach was implemented in a tool which fully integrated with the Visual Studio tool-suite, available as an extension, providing facilities such us autocompletion and verification at build time.

1 Problem Statement

Design by contract [14] is a programming discipline that prescribes that software designers should define formal, precise and verifiable interface specifications for software components, extending the ordinary definition of abstract data types with preconditions, postconditions and invariants.

While there has been some success in the adoption of contracts for enforcing functional requirements and design decisions [13, 15], there have not been many signs of their usage to express non-functional requirements such as performance or resource utilization requirements. Possible causes are the inherent difficulty of writing quantitative requirements, the lack of a convenient language to express them and tool support to verify them. However, in many settings it is crucial to enforce the fulfillment of this kind of requirements. Certifying memory consumption is vital to ensure safety in embedded systems; understanding the number of messages sent through a network is useful to detect performance bottlenecks or reduce communication costs, etc. It is well known that inferring, and even checking, quantitative bounds (e.g., resource usage) is difficult [5].

Nevertheless, there has been noticeable progress in techniques that compute symbolic resource usage [5, 2] and complexity [10] upper-bounds.

Code Contracts [8] is a tool that brings the advantages of design-by-contract programming to all .NET based programming languages enabling the use of contracts without requiring a specific compiler. The long term goal of the work presented here is enabling the specification of quantitative constraints such as resource usage and performance requirements in .NET applications using Code Contracts.

In this paper we focus in enforcing dynamic memory consumption contracts. This is a particularly challenging problem because memory footprint does not monotonically increase during program execution. For programming languages with automatic memory reclaiming mechanism (such as .NET based languages), this problem gets even more complex since memory consumption depends on the behavior of both the application and the garbage collector (GC). We believe other quantitative requirements may be computed by using a similar approach.

We present an extension of the Code Contracts annotation language designed to specify the amount of memory consumed by a method. These specifications have two possible interpretations: while they state the program ensures that a method consumes less than a particular bound, once verified, they can be interpreted as well as a precondition stating the system requires at least the amount of memory specified in order to run safely.

The proposed extension also provides means for specifying object lifetimes needed to model object allocation and reclaiming. Roughly speaking, we distinguish *temporary* objects, created by a method (or its callees) for auxiliary calculus, from *residual* objects that may be used by its callers and should live longer. For the latter we provide constructs to enable client methods to reclaim some of these objects.

In order to verify the annotations we rely on Clousot, the Code Contracts static verification engine. We do so by instrumenting the original program with special counters and transforming the memory assertions into equivalent standard Code Contracts assertions in terms of those counters. Some complex quantitative annotations require an arithmetic analysis that is beyond Clousot capabilities. To verify such annotations we integrate the tool with the symbolic calculator Barvinok [7], allowing us to verify specifications featuring polynomials.

All this work has been implemented as a Visual Studio plugin enabling static verification and run-time checks.

The paper is organized as follows: in §2 we introduce a set of annotations to describe memory consumption contracts, objects lifetime information and iteration spaces for loops. In §3 we show how to transform those annotations into code and annotations supported by Clousot, and how we check object lifetime annotations. In §4 we extend the checker to support polynomial constraints using Barvinok. Then, in §5 we present some implementation details. We conclude in §6 and §7 discussing some related and future work.

2 Memory usage annotations

The design of the annotations language was driven by the following considerations:

(i) The annotations should follow the style of Code Contracts to give users the advantage of having a natural and easy integration with the IDE such as autocompletion and inline documentation.
(ii) They need to provide means to specify that objects are allocated but also potentially reclaimed by the GC in a simple and modular fashion (lifetime information).
(iii) They should be rich enough to allow client methods to check its own annotations using the callees' resource specifications without losing much precision.
(iv) Both quantitative and lifetime constraints have to be in terms of methods parameters and instance variables.
(v) The mechanism to specify consumption information should maintain certain basic encapsulation properties of such us information hiding.

To represent memory recycling due to GC we based our annotation language on a very simple memory model[1] where annotations are used to *only* quantify objects created by the method being specified (or its callees). In this setting, those objects can be *temporary*, used for auxiliary computation and no longer needed at the end of method execution; or *residual*, meaning objects that may be used by a client method and, therefore, should live longer. Using escape analysis terminology, temporary objects are captured by the method whereas residual objects escape its scope.

Figure 1 shows an example exhibiting some of the annotations used to specify memory consumption. They are located under the `Contract.Memory` class as an extension of the available class `Contract` used by Code Contracts.

`Tmp` and `Rsd` are used to specify the amount of *temporal* and *residual* objects consumed by a method respectively. These annotations must be placed at the beginning of a method. They expect a class name and an integer expression which declares the number of objects of that class consumed by the method. Notice that these annotations should be interpreted within the method as an ensures clause stating that the method consumes at most the declared number of objects, but from the client point of view its role is a requires clause demanding that the system needs at least that space for the specified quantity of objects in order to safely run.

In addition to the quantitative expression `Rsd` expects an identifier for tagging this set of objects. The tag is used to specify that those objects belong to a group having similar characteristics in terms of lifetime (e.g., they are part of the same data structure). For instance, the identifier `Contract.Memory.Return` indicates that this set of objects is returned and `Contract.Memory.This` that

[1]This model is inspired in the scoped-memory management proposed for Real-Time Java [9], but in this case we just used it as an over-approximation of GC behavior.

```
public Person[] CreateFamily(List<string> names,
                             string address) {
  Contract.Requires(names.Count > 0);

  Contract.Memory.Rsd<Person[]>(Contract.Memory.Return,1);
  Contract.Memory.Rsd<Person>(Contract.Memory.Return,
                             names.Count);
  Contract.Memory.Rsd<Address>(Contract.Memory.Return,
                               names.Count);
  Contract.Memory.Tmp<AddressValidator>(1);

  Contract.Memory.DestRsd(Contract.Memory.Return);
  Person[] family = new Person[names.Count];

  for (int i = 0; i < names.Count; i++) {
    Contract.Memory.DestRsd(Contract.Memory.Return);
    Person p = new Person();

    Contract.Memory.AddRsd(Contract.Memory.Return,
                           Contract.Memory.This);
    p.SetInfo(names[i], address);
    family[i] = p;
  }
  return family;
}
```

Figure 1: Annotated method

objects may be reachable by the receiver. A developer can define an arbitrary set of identifiers according to hers needs of distinguishing sets of residual objects.

To verify the aforementioned contracts we need to inform the lifetime of every object allocated by the method. To do so, we introduce two new annotations: `DestTmp` and `DestRsd` which should be located before every `new` statement. `DestTmp` declares that an object is temporary and `DestRsd(t)` declares it as residual (living longer that the method itself) and associates the object with one of the tags already mentioned in the contract.

In the case of method invocations we need to figure out the destination of residual objects originated in callees. The annotation `AddTmp(src)` states that callees' residual objects tagged with `src` become temporary in the caller. `AddRsd(dst, src)` states that residual objects tagged with `src` become residual objects identified with `dst`.

So far, we have been using tags to declare sets of residual objects. This mechanism encompasses information hiding and is sufficient to specify and enforce the quantitative aspects of method consumption. However, to check the validity of annotations concerning objects lifetime, namely `DestTmp` and `DestRsd`, we need to provide the checker with the means to link tags to actual objects. To do that, we introduce the annotation `BindRsd(t, expr)` which connects a tag `t` with a set of objects referred by the path-expression `expr`. For instance. `BindRsd(List, l)` specifies the tag `List` represents all objects reachable from the variable `l`.

It is worth noticing that `AddTmp`, `AddRsd`, `DstTmp`, `DstRsd` and `BindRsd` are internal method annotations, not visible outside the method boundary. In contrast, `Tmp`, `Rsd` and their tags can used by clients.

3 Verifying memory consumption

To automatically check the annotations introduced in the previous section we transform the annotated program into a functionally equivalent program but instrumented into one using only CODE CONTRACTS annotations in such way that a successful verification of the transformed program implies the correctness of the original resource usage annotations.

3.1 Introducing counters and ensure clauses

For every method `m` featuring memory consumption we apply the following procedure: let T be the set of memory lifetime tags appearing in the contract (including one for temporary consumption) and C the set of classes. For each tag `t` $\in T$ and `C` $\in C$ we introduce a counter `C_m_t` which tracks the number of objects of type `C` from `m` that are associated with tag `t`. To keep the counters updated, for each `new C()` statement annotated with `DestTmp` or `DestRsd(t)`, we introduce a statement to increment `C_m_t`. Finally, the memory consumption annotations are transformed into corresponding ensure clauses stating that the associated counters are less than or equal to the specified bounds.

Concretely: `Contract<C>.Rsd(t, e)` is transformed into `Contract.Ensures(C_m_t <= e)`. The same approach applies for temporary consumption contracts.

For the annotations `AddTmp(d)` and `AddRsd(d, s)` the instrumentation consists in adding to the respective local counters the value of the callee counter.

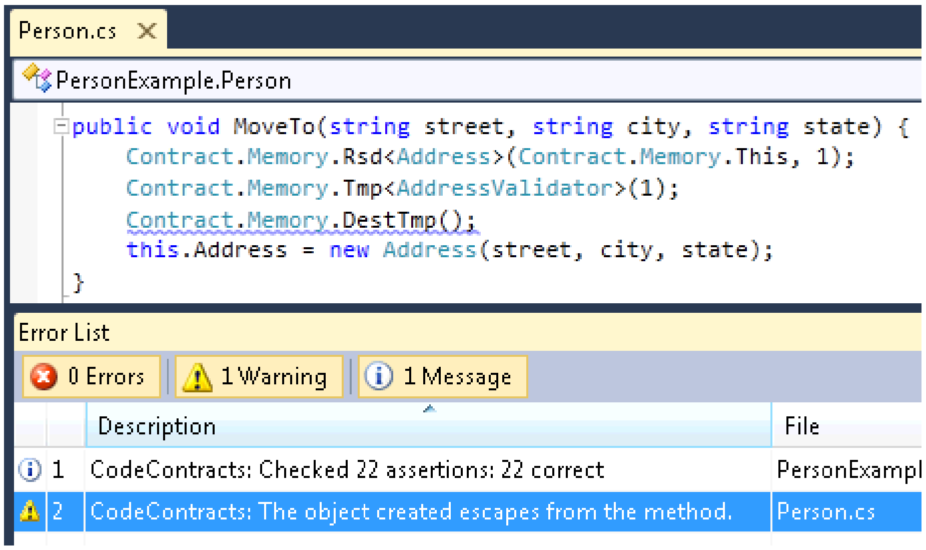
The instrumentation is performed at the IL level and is never read or manipulated by developers. Only for demonstration purposes, in Figure 2 we show a fragment of the instrumented version of the method presented in Figure 1.

3.2 Verifying object lifetime annotations

The instrumentation and verification process assume that object lifetime annotations `DestTmp` and `DestRsd` are correct. To ensure they actually are, we include a lifetime-annotations checker. Figure 3 shows an example where an escaping object is incorrectly declared as temporary. The squiggly highlights the position of that error.

```
public static int Person_CP_rsd_return;
public Person CreatePerson(string name)  {
 ...
 Contract.Ensures(Person_rsd_return <= names.Count);
 Person_CP_rsd_return = 0;
 ...
 for (int i = 0; i < names.Count; i++) {
   Person_CP_rsd_return++;
   Person p = new Person();
   Person_CP_rsd_return += Person_SI_this;
   p.SetInfo(names[i], address);
   ...
```

Figure 2: Fragment of an instrumented version



```
public void MoveTo(string street, string city, string state) {
    Contract.Memory.Rsd<Address>(Contract.Memory.This, 1);
    Contract.Memory.Tmp<AddressValidator>(1);
    Contract.Memory.DestTmp();
    this.Address = new Address(street, city, state);
}
```

```
1  CodeContracts: Checked 22 assertions: 22 correct          PersonExampl
2  CodeContracts: The object created escapes from the method.  Person.cs
```

Figure 3: Lifetime annotations verification result

To perform this verification we rely on a points-to and escape analysis capable of analyzing .NET programs [3]. For every annotated method we run the analysis and build a points-to graph (PTG) which is basically an abstraction of the program heap visible from each method at the end of its execution. To verify the correctness of `DestTmp` annotation, we check whether the node in the PTG, representing objects created at that program point, is not reachable from the global scope, method parameters or the returned object. For `DestRsd` we check if the object escapes and it is reachable through the path expression associated with the tag (i.e., by using `BindRsd`). The correctness of the lifetime information associated to the annotations `AddTmp` and `AddRsd`, where the objects are transferred to the objects designated by the corresponding tags, is also verified in a similar fashion.

4 Verifying complex contracts

Clousot does a very good job in performing automatic verification of contracts, being able to check method featuring loops without demanding loop invariants. However, it has some limitations when dealing with the complex arithmetic required for a quantitative analysis. According to our experiments the current version of Clousot is restrained to contracts having linear integer expressions.

Barvinok [7] is a tool[2] capable of manipulating parametric integer sets and relations. It provides functionality to *count* the number of elements of these sets and for performing *maximization* and *sum* on polynomials over these sets. Given a method featuring a loop (with possible several nested loops)

[2]Available at: http://freshmeat.net/projects/barvinok.

including a `new` statement and a predicate describing its iteration space (i.e. a linear restriction describing the relation between the loop inductive variables and parameters), we can obtain a parametric upper-bound of the number of times the `new` statement is executed. This upper bound is obtained by counting the number of solutions of that iteration space [5]. In a similar fashion, we can deal with polynomial temporary and residual consumptions by applying respectively a symbolic maximization and sum operations over the iteration space.

For those methods whose consumption is beyond the capabilities of Clousot we can use this approach. The price to pay to obtain more precision is the need of a new annotation to specify iteration spaces inside loops: `IterationSpace`. Although this increases the annotation burden, the gain is considerable since it makes possible the verification (and inference) of polynomial consumption. Notice that iteration spaces are a set of linear constraints, amenable to be checked with Code Contracts as well. We think it will be possible to automatically generate these annotations leveraging on Clousot abilities on inferring loop invariants.

```
 1 public List<Person> CreateBigFamily(int n) {
 2   Contract.Requires(n > 0);
 3   Contract.Memory.Rsd<Person>(Contract.Memory.Return,
 4                              n*(n+1)/2);
 5
 6   List<Person> family = new List<Person>(n*(n+1)/2);
 7   for (int i = 1; i <= n; i++) {
 8     Contract.Memory.IterationSpace(1 <= i && i <= n);
 9     for (int j = 1; j <= i; j++) {
10       Contract.Memory.IterationSpace(1 <= j && j <= i);
11       Contract.Memory.DestRsd(Contract.Memory.Return);
12       Person p = new Person();
13       family.Add(p);
14     }
15   }
16   return family;
17 }
```

Figure 4: Using `IterationSpace` to assist the prover

Figure 4 shows a method with a loop and a nested loop inside it. In this case Clousot would not be able to verify the contract. However, using Barvinok and the aid of `IterationSpace` annotations in the loops we can determine the exact number of times that the `new` statement on line 12 is executed and instruct the engine with new knowledge.

5 Implementation details

We developed a Visual Studio extension[3] that lets developers write memory consumption contracts as they do with Code Contracts and verify them using its static verifier or run-time checker. The only prerequisite for the plug-in is having Code Contracts installed, all the other tools used by the memory

[3] Available at: http://lafhis.dc.uba.ar/resourcecontracts.

contracts checker are packaged in the plugin. We use the Common Compiler Infrastructure (CCI) [1] for code analysis and instrumentation.

The Code Contracts static checker is invoked by Visual Studio after each compilation. In order to transform the code before checking it, we use a wrapper that performs the required instrumentation and then invokes the actual checker. When the plug-in is installed it modifies the Code Contracts configuration in order to setup this wrapper. So far, we have not found a better way to ensure that our tool is invoked before the Code Contracts checker.

6 Related work

Recently, there were relevant advances in resource analysis for imperative and functional programs [6, 4, 5, 2, 11, 10, 12]. For lack of space we will briefly refer to some of them which are focused in verification of annotated programs.

The work in [6] proposes a type system to statically check linear size annotations (Presburger's formulas) in a functional fragment of a Java-like language. This approach allows specifications of the number of preexistent objects released by a method but it requires complex aliasing annotations. We prefer a coarse grained approach demanding less and easier to infer annotations.

Closer to our approach, [4] defines an annotation language based on JML that can be used to annotate Java bytecode. This language is limited and does not contemplate the specification of lifetime information. In [11] the authors present a verification system for C-like programs using recursion as the only iteration mechanism. Similar to ours they use contracts and program instrumentation techniques using a non-specialized verifier. Their system supports `free` statements which in principle enables a more precise reasoning. However, according to our experience, verifying non-linear consumption in those systems is extremely hard because of the need of machinery capable of dealing with lower and upper bounds.

7 Conclusions and Future Work

In this work we presented an extension of Code Contracts language to specify and verify the memory consumption of .NET programs. The tool integrates with Visual Studio enabling autocompletion, inline documentation, static verification and run-time checking as Code Contracts does.

As a future work, we would like to enhance the usability of the tool by automatically inferring quantitative and lifetime annotations. In this setting developers would only need to specify complex or hard-to-infer annotations, not worrying about annotations that can be easily inferred. In this matter we plan to port our previous work on inference of memory consumption for Java [5] to .NET and extend other tools capable of inferring resource usage (e.g., [10]) in order to make them capable of dealing with dynamic memory usage.

References

[1] Common Compiler Iinfrastructure. http://cciast.codeplex.com/.

[2] E. Albert, S. Genaim, and M.Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *ISMM*, pages 129–138. ACM, 2009.

[3] M. Barnett, M. Fändrich, D. Garbervetsky, and F. Logozzo. Annotations for (more) precise points-to analysis. In *IWACO'07*, Berlin, Germany, jul 2007.

[4] G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In *SEFM*, pages 86–95, 2005.

[5] V. Braberman, F Fernández, D. Garbervetsky, and S Yovine. Parametric prediction of heap memory requirements. In *ISMM'08*. ACM, jun 2008.

[6] W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, 2005.

[7] P. Clauss, F.J. Fernandez, D. Garbervetsky, and S. Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *TVLSI*, 17(8):983–996, 2009.

[8] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC 2010*, pages 2103–2110. ACM, 2010.

[9] James Gosling and Greg Bollella. *The Real-Time Specification for Java.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[10] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI'10*, pages 292–304. ACM, 2010.

[11] G. He, S. Qin, C. Luo, and W.N. Chin. Memory Usage Verification Using Hip/Sleek. *ATVA'09*, pages 166–181.

[12] J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. *Programming Languages and Systems*, pages 287–306, 2010.

[13] G.T. Leavens, K. Rustan M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA'00*, pages 105–106, 2000.

[14] Bertrand Meyer. *Object-oriented Software Construction.* Series in Computer Science. Prentice-Hall International, New York, 1988.

[15] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, Hemel Hempstead, 1992.