On Verifying Resource Contracts using Code Contracts

Rodrigo Castaño, Diego Garbervetsky,
Jonathan Tapicer, Edgardo Zoppi
Departamento de Computación. FCEyN. UBA
Buenos Aires, Argentina
{rcastano, diegog, jtapicer, ezoppi}@dc.uba.ar

Juan Pablo Galeotti
Saarland University
Saarbrücken, Germany
galeotti@cs.uni-saarland.de

In this paper we present an approach to check resource consumption contracts using an off-the-shelf static analyzer. We propose a set of annotations to support resource usage specifications, in particular, dynamic memory consumption constraints. Since dynamic memory may be recycled by a memory manager, the consumption of this resource is not monotone. The specification language can express both memory consumption and lifetime properties in a modular fashion. We develop a proof-of-concept implementation by extending CODE CONTRACTS' specification language. To verify the correctness of these annotations we rely on the CODE CONTRACTS static verifier and a points-to analysis. We also briefly discuss possible extensions of our approach to deal with non-linear expressions.

1 Problem statement

Design by contract [24] 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 [23, 25], 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 [10]. Nevertheless, there has been noticeable progress in techniques that compute symbolic resource usage [10, 4] and complexity [19] upper-bounds.

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 (i.e., due to memory recycling). For programming languages with automatic memory reclaiming mechanisms (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.

CODE CONTRACTS [15] 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.

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: on

Nazareno Aguirre, Leila Ribeiro (Eds.):
First Latin American Workshop on Formal Methods (LAFM 2013)
EPTCS 139, 2014, pp. 1–15, doi:10.4204/EPTCS.139.1

one hand, they state that a method consumes less memory than a particular bound but, 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 between *temporary* and *escaping* objects. Escaping objects refer to objects that are created in a method but may be used by its callers and, thus, should live longer. Temporary objects can be safely collected once the method finishes its execution. We provide constructs to enable client methods to reclaim some of these objects. For verification purposes, it is important to provide an upper bound to these objects. By doing so, it is possible to use these bounds to verify each method's contract separately.

In order to verify the annotations we instrument the original program, adding special counters and translating the annotations to conventional statements in terms of these counters. In general, we are assuming there is a static verification engine for the language in which the original program was coded. In our implementation, we rely on CLOUSOT [16], the static analyzer used by CODE CONTRACTS, which is based in abstract interpretation [14]. One of the main advantages of CLOUSOT is that it is able to verify programs featuring loops without the need of loop invariants. Its main drawback is that it cannot handle non-linear expressions, which may appear in complex programs. In the paper we discuss some possible approaches to overcome this problem.

All this work has been implemented as a VISUAL STUDIO plugin enabling static verification and run-time checks of memory consumption contracts.

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 contracts supported by CLOUSOT, and how we check object lifetime annotations. In §4 we discuss one possible extension of the checker in order to support polynomial constraints using the BARVINOK calculator. Then, in §5 we present some implementation details. We conclude in §6 and §7 discussing some related and future work.

2 Memory usage annotations

In this section we present our annotation language. Its design was driven by the following considerations:

(i) The annotations 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).

(ii) The annotations should be rich enough to allow client methods to check its own annotations using the callees' resource specifications without losing much precision.

(iii) Both quantitative and lifetime constraints have to be in terms of method parameters and instance variables.

(iv) The mechanism to specify consumption information should maintain certain basic encapsulation properties such as information hiding.

To represent memory recycling due to GC we based our annotation language on a very simple memory model[1]: objects can be annotated as *escaping*, meaning that they may be used by a client method and, therefore, should live longer than the method itself, or not annotated, meaning that they are used only

[1]This model is inspired in the scoped-memory management proposed for Real-Time Java [18], but in this case we just used it as an over-approximation of GC behavior.

for auxiliary computation and are no longer needed at the end of the method execution. Furthermore, we only allow annotations to express properties about objects created in the scope of the method in which they are contained. The annotations are based on escape analysis terminology, escaping objects escape the method's scope, whereas all other objects are captured by the method.

`MemReq` is used to specify an upper bound of the maximum number of *live* objects that were allocated by the method. By live objects we mean the objects that could not be collected by the garbage collector during the execution of the method. In what follows, unless explicitly stated, by memory requirements we refer to `MemReq`. `Esc` specifies an upper bound to the total number of allocated objects escaping the method. 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. We will also use the first interpretation to introduce a compositional verification algorithm.

Figure 1 shows a brief example, written in C#, of the `MemReq` annotation. The `Person` constructor creates a temporary `Logger` which is discarded and not reachable from outside the scope of the method. This object is captured by the method, therefore it is not annotated as escaping.

```
1  class Person {
2    public string FirstName { get; private set; }
3    public string LastName { get; private set; }
4
5    public Person(string firstName, string lastName) {
6      Contract.Memory.MemReq<Logger>(1);
7
8      this.FirstName = firstName;
9      this.LastName = lastName;
10
11     Logger logger = new Logger();
12     logger.logMessage("Person created.");
13   }
14 }
```

Figure 1: Annotated method featuring only captured objects.

In addition to the quantitative expression, `Esc` expects an identifier for tagging this set of escaping 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` indicates that this set of objects may be reachable by the receiver. A developer can define an arbitrary set of identifiers according to her needs of distinguishing sets of escaping objects.

To verify the aforementioned contracts we need to inform the lifetime of every object allocated by the method. To do so, we introduce a new annotation: `DestEsc`, which should be located before every `new` statement. `DestEsc(t)` declares an object as escaping (living longer that the method itself) and associates the object with one of the tags already mentioned in the contract. If the annotation were missing, the object will be considered temporary, and it should not be accessible from outside the scope of the method.

The code of the `Family` constructor, shown in Figure 2, includes an escaping object. As can be seen, the contract of line 8 states that an array of type `Person` is escaping the scope of the method. Instead of

representing the array as one object, we quantify it by its size. Additionally, the `new Person[size];` statement of line 11 is tagged with `DestEsc(Contract.Memory.This)`, indicating that the escaping object will be reachable from outside the scope of the method through the receiver of the method call (in this case, since the method is a constructor, the receiver is the newly created `Family` instance). The method `AddMember` creates an object of type `Person` and includes it in the array of `Family` instances. Since it calls the constructor of `Person`, it requires space for one object of type `Logger` and one of type `Person`. This last object escapes the scope of the method because it is referenced by the instance field `_Members`. Thus, we include the annotation `DestEsc(Contract.Memory.This)` of line 23.

```
1  class Family {
2    private Person[] _Members;
3    public string LastName { get; private set; }
4    public int Size { get; private set; }
5
6    public Family(string lastName, int size) {
7      Contract.Memory.MemReq<Person[]>(size);
8      Contract.Memory.Esc<Person[]>(Contract.Memory.This, size);
9
10     Contract.Memory.DestEsc(Contract.Memory.This);
11     _Members = new Person[size];
12
13     this.LastName = lastName;
14     this.Size = 0;
15   }
16
17   public void AddMember(string firstName) {
18     Contract.Requires(this.Size < _Members.Length);
19     Contract.Memory.MemReq<Person>(1);
20     Contract.Memory.MemReq<Logger>(1)
21     Contract.Memory.Esc<Person>(Contract.Memory.This, 1);
22
23     Contract.Memory.DestEsc(Contract.Memory.This);
24     Person person = new Person(firstName, this.LastName);
25     _Members[this.Size++] = person;
26   }
27 }
```

Figure 2: Annotated method featuring escaping objects.

The example in Figure 3 brings together most of the annotations shown so far. The statement in line 12 shows an example of objects escaping from a method call. In this case, the escaping objects are associated with the returned object. In fact, the `Family` returned will be precisely the one created in that statement. In the case of method invocations we need to figure out the destination of escaping objects originated in callees. The annotation `AddEsc(dst, src)` states that objects escaping the callee, tagged with `src`, become also escaping objects of the caller, but identified with the tag `dst`. All objects escaping a method invocation and missing the corresponding annotation will be considered temporary for the caller and should not be accessible from outside its scope.

It is important to mention the reason for the difference between the total memory requirements and the number of escaping objects. Consider the call to `AddMember`. As specified in Figure 1, a temporary `Logger` will be created by the `Person` constructor. Assuming the contract to be correct, the memory requirements of the callee, in this case, have to be added to the total memory requirements of the method `CreateFamily`, but since this is a temporary object, the memory required for this object can be recycled after the call to `AddMember`. In contrast, escaping objects are accumulative meaning that the caller needs space for each object escaping from `AddMember` at each iteration.

```
1  Family CreateFamily(string lastName, string[] firstNames) {
2    Contract.Memory.MemReq<Family>(1);
3    Contract.Memory.MemReq<Person[]>(firstNames.Length);
4    Contract.Memory.MemReq<Person>(firstNames.Length);
5    Contract.Memory.MemReq<Logger>(1);
6    Contract.Memory.Esc<Family>(Contract.Memory.Return, 1);
7    Contract.Memory.Esc<Person[]>(Contract.Memory.Return, firstNames.Length);
8    Contract.Memory.Esc<Person>(Contract.Memory.Return, firstNames.Length);
9
10   Contract.Memory.DestEsc(Contract.Memory.Return);
11   Contract.Memory.AddEsc(Contract.Memory.Return, Contract.Memory.This);
12   Family family = new Family(lastName, firstNames.Length);
13
14   for (int i = 0; i < firstNames.Length; ++i) {
15     Contract.Memory.AddEsc(Contract.Memory.Return, Contract.Memory.This);
16     family.AddMember(firstNames[i]);
17   }
18
19   return family;
20 }
```

Figure 3: Annotated method featuring most annotations presented.

So far, we have been using tags to declare sets of escaping 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 `DestEsc`, we need to provide the checker with the means to link tags to actual objects. To do that, we introduce the annotation `BindEsc(t, expr)` which connects a tag `t` with a set of objects referred by the path-expression `expr`. For instance, in line 6 of Figure 4, `BindEsc(Param, brother)` specifies that the tag `Param` represents all objects reachable from the returning parameter `brother`.

```
1  Person CreateBrothers(string lastName, string name1, string name2, out Person brother) {
2    Contract.Memory.MemReq<Person>(2);
3    Contract.Memory.MemReq<Logger>(1);
4    Contract.Memory.Esc<Person>(Contract.Memory.Return, 1);
5    Contract.Memory.Esc<Person>(Param, 1);
6    Contract.Memory.BindEsc(Param, brother);
7
8    Contract.Memory.DestEsc(Contract.Memory.Return);
9    Person person = new Person(name1, lastName);
10
11   Contract.Memory.DestEsc(Param);
12   brother = new Person(name2, lastName);
13   return person;
14 }
```

Figure 4: Using `BindEsc` to relate tags with path expressions.

It is worth noticing that `AddEsc`, `DestEsc` and `BindEsc` are internal method annotations, not visible outside the method boundary. In contrast, `MemReq`, `Esc` and their tags can be 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 instrumented program in such a way that a successful verification

of the transformed program implies the correctness of the original resource usage annotations. For the case of .NET programs, part of the transformation involves we rely on CODE CONTRACTS' basic annotations. The instrumentation is performed at the IL level and is never read or manipulated by developers.

3.1 Introducing counters and ensure clauses

In order to use standard notation, we transform each memory consumption restriction into assertions in terms of integer counters. With that purpose in mind, we will keep track of one counter for each memory lifetime tag and an additional one for total memory requirements of objects of a particular type.

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 of `m` and C the set of classes. For each tag `t` $\in T$ and `C` $\in C$ we introduce the counter `m_Esc_t_C`. We will also create the counter `m_MemReq_C`. The first counter tracks the number of objects of type `C` escaping from `m` that are associated with tag `t`. In contrast, the latter is incremented with every object creation in `m`, since it represents the total memory requirements of `m` for objects of type `C`. To keep the counters updated, `m_MemReq_C` is incremented before each `new C()` statement in `m`. If the statement were annotated with `DestEsc`, we also introduce a statement to increment `m_Esc_t_C`.

To illustrate the idea consider the example in Figure 5 which allocates two objects of type `A` and performs two method calls. Figure 6 shows the instrumented version where the counters `m_MemReq_A`, `m_Esc_Return_A` and `m_Esc_Param_A` track respectively the amount of objects of type `A` which are required and escaping the method `m`.

```
1  A m(int n, out A p2) {
2    Contract.Memory.MemReq<A>(n + 5);
3    Contract.Memory.Esc<A>(Contract.Memory.Return, 2);
4    Contract.Memory.Esc<A>(Param, 1);
5    Contract.Memory.BindEsc(Param, p2);
6
7    A a1 = new A();
8    Contract.Memory.DestEsc(Param);
9    p2 = new A();
10   A a3 = m1(n);
11   Contract.Memory.AddEsc(Contract.Memory.Return, Contract.Memory.Return);
12   A a4 = m2(n);
13   return a4;
14 }
15 A m1(int m) {
16   Contract.Memory.MemReq<A>(m + 1);
17   Contract.Memory.Esc<A>(Contract.Memory.Return, 1);
18   ...
19 }
20 A m2(int k) {
21   Contract.Memory.MemReq<A>(k);
22   Contract.Memory.Esc<A>(Contract.Memory.Return, 2);
23   ...
24 }
```

Figure 5: A simple example to illustrate the instrumentation process.

The memory consumption annotations are transformed into corresponding ensure clauses stating that the associated counters are less than or equal to the specified bounds. For instance, the annotation `Contract.Esc<C>(t, e)` is transformed into `Contract.Ensures(m_Esc_t_C <= e)`. The same approach applies for `MemReq` annotations. For `AddEsc(d, s)` annotations, the instrumentation consists in

```
A m(int m, out A p2) {
  Contract.Ensures(m_MemReq_A <= n + 5);
  Contract.Ensures(m_Esc_Return_A <= 2);
  Contract.Ensures(m_Esc_Param_A <= 1);

  m_MemReq_A = 0;
  m_Esc_Return_A = 0;
  m_Esc_Param_A = 0;

  m_MemReq_A += 1;
  A a1 = new A();

  m_MemReq_A += 1;
  m_Esc_Param_A += 1;
  p2 = new A();

  int maxCalls_A = 0;
  int sumCalls_A = 0;

  int call1_diff_A = (n + 1) - 1; // m1_MemReq_A - m1_Esc_A
  maxCalls_A = max(maxCalls_A, call1_diff_A);
  sumCalls_A += 1; // m1_Esc_A
  A a3 = m1(n);

  int call2_diff_A = n - 2;  // m2_MemReq_A - m2_Esc_A
  maxCalls_A = max(maxCalls_A, call2_diff_A);
  sumCalls_A += 2; // m2_Esc_A
  m_Esc_Return_A += 2; // m2_Esc_A
  A a4 = m2(n);

  m_MemReq_A += maxCalls_A + sumCalls_A;
  return a4;
}
```

Figure 6: Instrumented version for our simple example.

adding to the respective local counters the value of the callee counter.

One important aspect that we have overlooked so far is how to handle objects captured by a callee. As we mentioned previously, temporary objects created inside the method and not escaping its scope, can be recycled. Again, consider the example in Figure 5. There `m` invokes two methods: `m1` and `m2`, consuming $MR_1 = m+1$, $E_1 = 1$ and $MR_2 = k$ and $E_2 = 2$ respectively. The objects escaping `m1` and `m2` must be handled by `m` regardless of whether those objects end up escaping `m`. Thus, they will be part of `MemReq`. In addition, each method may create some temporary objects that can be recycled when they finish its execution. Thus, to compute `MemReq` we can consider the expression $\max\{MR_1 + E_2, MR_2 + E_1\}$, representing this idea of reusing of objects[2]. This expression can be rewritten as $\max\{MR_1 - E_1, MR_2 - E_2\} + E_1 + E_2$ and binding the variables m and k with the call argument n we obtain $\max\{n+1-1, n-2\} + 1 + 2 = n + 3$. We handle this computation by including two auxiliary counters per type called `maxCalls` and `sumCalls`. For instance, in the example of Figure 6 we use the variables `maxCalls_A` and `sumCalls_A` to accumulate the result of calls for the type `A`.

For a method featuring loops, the idea is to apply the same procedure over every loop iteration to compute the maximum among all iterations. Again, we use one auxiliary counter for each expression that is computed during the loop. At the end of the loop these counters contain the expression representing the

[2]Our analysis is flow insensitive. This is because is not trivial to deal with the statement ordering in case of loops featuring conditions in the loop body.

consumption for all the iterations. For instance, in Figure 7 we show one fragment of the instrumented version of the method `CreateFamily`. There, the variables `maxCall_Logger` and `maxCall_Person` compute the contribution of the calls in the loop for the objects of type `Logger` and `Person`.

```
1  Family CreateFamily(string lastName, string[] firstNames) {
2    ...
3    int maxCall_Person = 0;
4    int sumCall_Person = 0;
5    int maxCall_Logger = 0;
6    int sumCall_Logger = 0;
7
8    for (int i = 0; i < firstNames.Length; ++i) {
9      int call_diff_Person = 1 - 1; // AddMember_MemReq_Person - AddMember_Esc_This_Person
10     maxCall_Person = max(maxCall_Person, call_diff_Person);
11     sumCall_Person += 1; // AddMember_Esc_This_Person
12     int call_diff_Logger = 1 - 0; // AddMember_MemReq_Logger - AddMember_Esc_This_Logger
13     maxCall_Logger = max(maxCall_Logger, call_diff_Logger);
14     sumCall_Logger += 0; // AddMember_Esc_This_Logger
15
16     family.AddMember(firstNames[i]);
17   }
18
19   CreateFamily_MemReq_Person += maxCall_Person + sumCall_Person;
20   CreateFamily_MemReq_Logger += maxCall_Logger + sumCall_Logger;
21   ...
22 }
```

Figure 7: Fragment of an instrumented version.

3.2 Verifying object lifetime annotations

For the instrumentation and verification process we are assuming that object lifetime annotations, such as `DestEsc`, are correct. To ensure they actually are, we include a lifetime annotations checker.

To perform this verification we rely on a points-to and escape analysis capable of analyzing .NET programs [7]. A points-to and escape analysis is basically a tool that provides an abstraction of the program at a given program location. A typical (finite) representation of a heap is given by a points-to graph (PTG). A PTG is a graph (L, N, E) where a node $n \in N$ represent a set of objects, in general all objects allocated in the same program point, and a edge $(n1, f, n2)$ represents that one of the objects denoted by $n1$ may point-to one of the objects denoted by $n2$ using the field f. L is a mapping from local variables (including parameters) to nodes, which is used to express how the program accesses the heap using those variables. Using a PTG we can query if an object may be reachable from either some variable or from another object. This query can be easily solved by traversing the PTG starting from L, in case of a parameter, or a node n, in case of an object.

For instance, consider the Figure 8. The first PTG corresponds to the exit point of the `Person` constructor. A dotted circle represents a parameter or load node meaning objects created outside the scope of the method under analysis. In this case, the node models the object represented by the parameter `this`. A solid circle represents an object created during the execution of the method (or some of its callees). In this case, it models the `logger` object. The second PTG corresponds to the exit point of `AddMember`. In this case we can see that this method does not allocate objects but makes the object pointed by `person` reachable from `this`.

Our analysis computes a PTG for each annotated method. This PTG is an abstraction of the program heap visible from each method at the end of its execution. To verify that objects not annotated are in

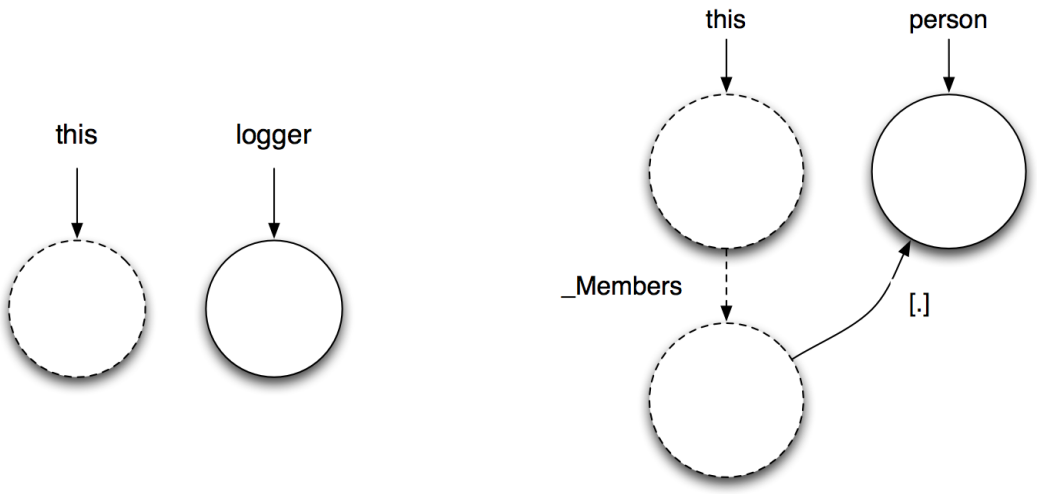

Figure 8: PTGs for the constructor of `Person` and the `AddMember` method of `Family`.

fact captured by the method, 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 `DestEsc` we check if the object escapes and it is reachable through the path expression associated with the tag (i.e., by using `BindEsc`, as we do in the example of Figure 5). The correctness of the lifetime information associated to the annotation `AddEsc`, where the objects are assigned to the corresponding tag, is also verified in a similar fashion.

The points-to analysis we use builds a conservative abstraction. That means that it may produce a false alarm and state that an object may escape when it does not. To overcome this issue, we provide the annotation `DestLocal` for new statements which tells the verification engine to ignore the result of the escape analysis for that allocation.

4 Discussion

4.1 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.

One possible approach to improve our analysis is relying on theorem provers. Fully automatic and complete decision procedures for non-linear constraints is impossible, one needs to sacrifice completeness or automation. Nevertheless, modern SMT solvers like Z3 [9] started to provide some automatic (but incomplete) support for polynomials. Thus, it can be possible to use Z3 to verify this resource consumption expression. The problem with this approach is that the user would need to provide loop invariants describing how the consumption is evolving during loops. This task could be very complex and error-prone.

Another approach is trying directly to infer the consumption in those cases. In [10, 17] we propose a technique to automatically infer memory consumption bounds on Java like programs. Essentially the technique works as follows: given a method featuring a loop (with possible several nested loops)

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. In a similar fashion, we can deal with polynomial temporary and escaping consumptions by applying respectively a symbolic maximization and sum operations over the iteration space [11]. BARVINOK [13] is a tool[3] 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.

For those methods whose consumption is beyond the capabilities of CLOUSOT we may 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 and could even be inferred using CLOUSOT ability for inferring loop invariants.

Figure 9 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, following the technique presented in [10], using BARVINOK with the aid of `IterationSpace` annotations in the loops, we can determine the exact number of times that the method call on line 17 is executed and instruct the engine with new knowledge.

```
1  public Family CreateBigFamily(int n) {
2      Contract.Requires(n > 0);
3      ...
4      Contract.Memory.Esc<Person>(Contract.Memory.Return, n * (n + 1) / 2);
5      ...
6      Contract.Memory.DestEsc(Contract.Memory.Return);
7      Contract.Memory.AddEsc(Contract.Memory.Return, Contract.Memory.This);
8      Family family = new Family("Doe", n * (n + 1) / 2);
9
10     for (int i = 1; i <= n; i++) {
11         Contract.Memory.IterationSpace(1 <= i && i <= n);
12
13         for (int j = 1; j <= i; j++) {
14             Contract.Memory.IterationSpace(1 <= j && j <= i);
15
16             Contract.Memory.AddEsc(Contract.Memory.Return, Contract.Memory.This);
17             family.AddMember("John");
18         }
19     }
20
21     return family;
22 }
```

Figure 9: Using `IterationSpace` to assist the prover.

4.2 From objects to real memory consumption

Our analysis is focused in quantifying live objects rather than computing the exact memory consumption of a program. Computing real consumption requires to solve challenges posed by particularities of .NET or Java virtual machines like their garbage collectors (GC), representation of arrays and memory fragmentation. Nevertheless, we believe we are still targeting the main problem which is related to the

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

number of objects allocated by a program and how many of them may be released during the execution, in order to reuse the memory.

Our annotations allow the programmer to quantify memory consumption by tags. The tags enable the verifier to take into account that there is a memory manager that collects objects when they are no longer needed. As mentioned, we decided to allow memory reclaiming at the method scope. This is, of course, an approximation of how real garbage collection may perform[4] and other fine-grained mechanisms need to be investigated. We also need to investigate how to specify the memory consumed by the VM itself, which is not visible by looking at the application code.

4.3 About annotations and information hiding

The annotation language enables to aggregate information about objects according to their types. This can be particularly useful when we need approximate real memory consumption as we can use the size of each type to know its required space in memory (modulo some possible fragmentation).

However, the information about types jeopardizes information hiding. Suppose we have a method *m* that requires some auxiliary objects of type `T` perform a task. These objects do not escape the method, but *m* needs to declare that it requires *n* objects of type `T` to run. This information will be propagated to the caller and even to their parents (even the main method!), making the type `T` visible to other fragments of code, and thus breaking information hiding. In our running example, this phenomena occurs with the auxiliary object of type `Logger` used in the `Person` constructor, whose type is propagated to its callers.

One possible solution to overcome this problem is to just quantify objects, regardless of their type, as can be seen in Figure 10. This would make the analysis less useful for computing real memory bounds, but still can be used to compare different implementations of algorithms in terms of their memory usage. Another choice is qualifying the consumption using other kinds of units of measuring, like number of fields, bytes, etc. We need to investigate further to understand which is the right balance between granularity, verifiability, understandability, utility and modularity of contracts.

```
1 Family CreateFamily(string lastName, string[] firstNames) {
2   Contract.Memory.MemReq<object>(2 + 2 * firstNames.Length );
3   Contract.Memory.Esc<object>(Contract.Memory.Return, 1 + 2 * firstNames.Length);
4   ...
5 }
```

Figure 10: Contract hiding type information.

5 Implementation details

We implemented our prototype tool as a VISUAL STUDIO extension[5] 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 contracts checker are packaged in the plugin and relies on the Common Compiler Infrastructure (CCI) [1] for code analysis and instrumentation.

[4] Actually, the analysis over-approximate the behavior of an *ideal* garbage collector. That is a GC that collects objects as soon as they become unreachable.

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

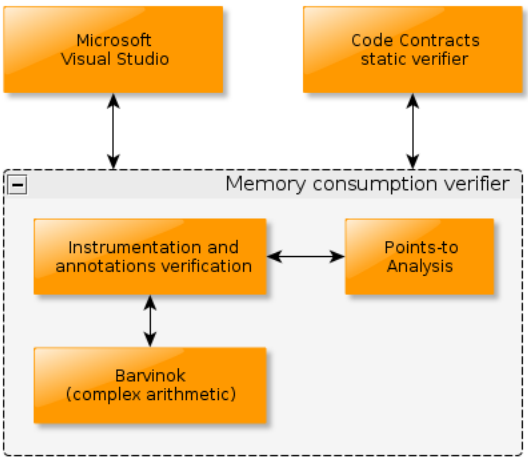

Figure 11: General architecture of the tool.

Figure 11 shows the general architecture of the tool and Figure 12 presents a sequence diagram showing the interaction made by the main module in order to verify resource contracts. The component labeled *Memory consumption verifier* has the same interface as the CODE CONTRACTS verifier, so it can replace it when VISUAL STUDIO invokes it. Internally, the *Memory consumption verifier* uses the described algorithms and tools to do the instrumentation and verification, then it invokes the CODE CONTRACTS verifier and returns to VISUAL STUDIO the verification results.

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.

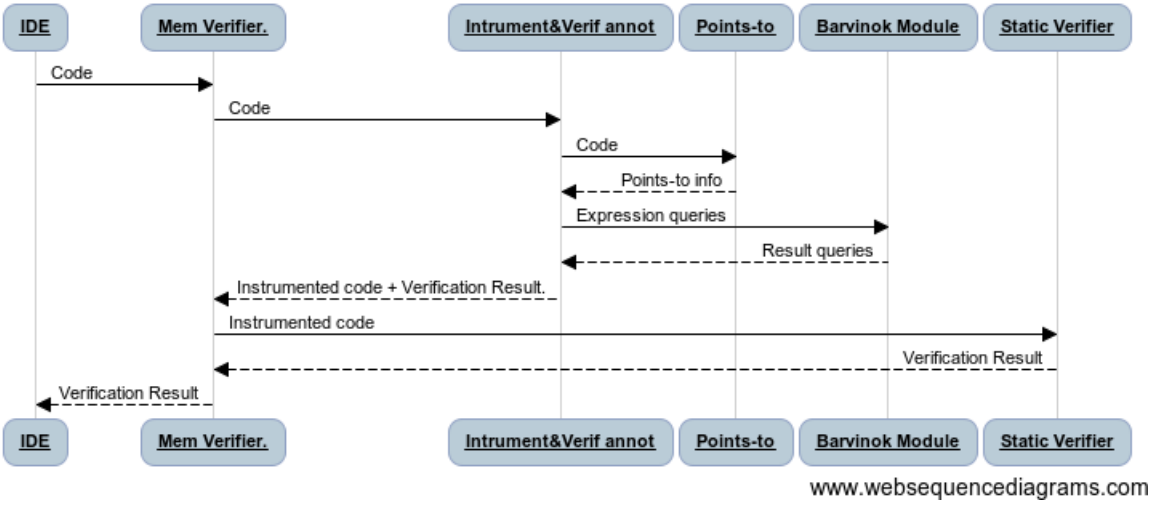

Figure 12: Interaction between main modules.

6 Related work

Recently, there were relevant advances in resource analysis for imperative and functional programs [12, 8, 10, 17, 5, 20, 19, 21]. Most of them are aimed at inferring of resource consumption bounds rather than verifying them. Here will briefly refer to some of them which are focused in verification of annotated programs.

The work in [12] 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 annotations in order to specify potential aliasing relations between expressions in the language. We prefer a coarse grained approach demanding less and easier to infer annotations.

Closer to our approach, [8] 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 [20] 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.

Atkey [6] presents a technique to verify imperative pointer-manipulating languages by embedding a resource logic, which is an extension of separation logic. The technique is inspired in the concept of amortized resource analysis [22]. Essentially, the logic enables the specification of input/output potentials for each method. Input potentials are used to represent the required resources to run the method and output potentials represent the resources that are warranted to be available at the end of its execution. He also describes a proof search procedure that allows generated verification conditions to be discharged while using linear programming to infer consumable resource annotations. In contrast with our approach, his language has an explicit construct to manually deallocate objects and their current logic only allows linear restraints.

More recently, Albert and collaborators [3] presented an approach to verify memory consumption contracts using a verifier called KEY [2]. Essentially, they propose to use their tool COSTA [5] to infer annotations about memory consumption and then apply the solver KEY to verified them. In this case they need to rely on the power of the verifier to deal with the non-linearity of the inferred bounds. COSTA may help by providing some annotation but still the user may need to provide invariants and annotations about the shape and lifetime of objects. In contrast, in our approach we are more focused in automating the process by providing an automatic escape analyzer, using an static analyzer which does not require invariants, and using an external tool to deal with non-linear expressions.

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 address more complex and larger programs. To do so we plan 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 [10] to .NET and extend other tools capable of inferring resource usage (e.g., [19]) in order to make them capable of dealing with dynamic memory usage.

We plan to formally prove that the instrumentation technique preserves the behavior of the original program in terms of memory consumption.

Our current annotations depends on an external points-to analysis that is used to approximate escape analysis information. This analysis must be executed before our verification process. We would like to extend our annotation language to allow also to express directly lifetime properties (while maintaining information hiding) and being able to modularly verify programs without requiring such external tool.

Acknowledgments

This work has been partially funded by CONICET, UBACyT-20020110200075/20020100100813, MinCyT PICT-2010-235/2011-1774/2012-0724, CONICET-PIP 11220110100596CO, MINCYT-BMWF AU/10/19, INRIA Associated Team ANCOME, and LIA INFINIS and MEALS 295261.

References

[1] *Common Compiler Infrastructure*. Available at `http://cciast.codeplex.com/`.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager et al. (2005): *The key tool*. *Software & Systems Modeling* 4(1), pp. 32–54, doi:10.1007/s10270-004-0058-x.

[3] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle & Guillermo Román-Díez (2012): *Verified Resource Guarantees for Heap Manipulating Programs*. In: *FASE*, pp. 130–145, doi:10.1007/978-3-642-28872-2_10.

[4] Elvira Albert, Samir Genaim & Miguel Gómez-Zamalloa (2009): *Live heap space analysis for languages with garbage collection*. In: *ISMM*, pp. 129–138, doi:10.1145/1542431.1542450.

[5] Elvira Albert, Samir Genaim & Miguel Gómez-Zamalloa (2013): *Heap space analysis for garbage collected languages*. *Sci. Comput. Program.* 78(9), pp. 1427–1448, doi:10.1016/j.scico.2012.10.008.

[6] Robert Atkey (2011): *Amortised Resource Analysis with Separation Logic*. *Logical Methods in Computer Science* 7(2), doi:10.2168/LMCS-7(2:17)2011.

[7] Michael Barnett, Manuel Fändrich, Diego Garbervetsky & Francesco Logozzo (2007): *Annotations for (more) Precise Points-to Analysis*. In: *Proceedings of the 2nd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO'07)*, pp. 11–18. Available at `http://people.dsv.su.se/~tobias/iwaco/p4-barnett.pdf`.

[8] Gilles Barthe, Mariela Pavlova & Gerardo Schneider (2005): *Precise Analysis of Memory Consumption using Program Logics*. In: *SEFM*, pp. 86–95, doi:10.1109/SEFM.2005.34.

[9] Nikolaj Bjorner & Leonardo de Moura (2009): *Z3: An efficient SMT solver*. http://research.microsoft.com/en-us/um/redmond/projects/z3/, doi:10.1007/978-3-540-78800-3_24.

[10] Víctor Braberman, Federico Fernández, Diego Garbervetsky & Sergio Yovine (2008): *Parametric prediction of heap memory requirements*. In: *Proceedings of the 7th international symposium on Memory management*, ACM, pp. 141–150, doi:10.1145/1375634.1375655.

[11] Víctor A. Braberman, Diego Garbervetsky, Samuel Hym & Sergio Yovine (2013): *Summary-based inference of quantitative bounds of live heap objects*. *Submitted*.

[12] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin & Martin Rinard (2005): *Memory Usage Verification for OO Programs*. In: *SAS 05*, doi:10.1007/11547662_7.

[13] Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky & Sven Verdoolaege (2009): *Symbolic polynomial maximization over convex sets and its application to memory requirement estimation*. *TVLSI* 17(8), pp. 983–996, doi:10.1109/TVLSI.2008.2002049.

[14] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points*. In: *POPL 77*, pp. 238–252, doi:10.1145/512950.512973.

[15] Manuel Fähndrich, Michael Barnett & Francesco Logozzo (2010): *Embedded contract languages*. In: *SAC 2010*, ACM, pp. 2103–2110, doi:10.1145/1774088.1774531.

[16] Manuel Fähndrich & Francesco Logozzo (2009): *Clousot: a language agnostic abstract interpretation-based static analyzer for .NET*. http://research.microsoft.com/en-us/people/logozzo/.

[17] Diego Garbervetsky, Sergio Yovine, Víctor A. Braberman, Martín Rouaux & Alejandro Taboada (2011): *Quantitative dynamic-memory analysis for Java*. *Concurrency and Computation: Practice and Experience* 23(14), pp. 1665–1678, doi:10.1002/cpe.1656.

[18] James Gosling & Greg Bollella (2000): *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc.

[19] Sumit Gulwani & Florian Zuleger (2010): *The reachability-bound problem*. In: *PLDI'10*, ACM, pp. 292–304, doi:10.1145/1809028.1806630.

[20] Guanhua He, Shengchao Qin, Chenguang Luo & Wei-Ngan Chin: *Memory Usage Verification Using Hip/Sleek*. *ATVA'09*, pp. 166–181, doi:10.1007/978-3-642-04761-9_14.

[21] Jan Hoffmann & Martin Hofmann (2010): *Amortized resource analysis with polynomial potential*. *Programming Languages and Systems*, pp. 287–306, doi:10.1007/978-3-642-11957-6_16.

[22] Martin Hofman & Stefen Jost (2003): *Static Prediction of Heap Usage for First-Order Functional Programs*. In: *POPL 03*, SIGPLAN, New Orleans, LA, doi:10.1145/640128.604148.

[23] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby & Bart Jacobs (2000): *JML: notations and tools supporting detailed design in Java*. In: *OOPSLA'00*, pp. 105–106, doi:10.1145/367845.367996.

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

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