Practical verification for the working programmer with CodeContracts and Abstract Interpretation (Invited Talk)

Francesco Logozzo

Microsoft Research, Redmond, WA (USA)
logozzo@microsoft.com

CodeContracts provide a language agnostic way to specify and check preconditions, postconditions and object invariants (collectively called contracts [17]). Specifications take the form of calls to static methods of a `Contract` library [7]. The authoring library is available out-of-the-box to all .NET programmers from $v4$.

An example of CodeContracts usage is reported in Fig. 1. The code illustrates the specification and the implementation of a simple string sanitizer, which filters only ASCII letters and converts all the upper cases into lower cases. The sanitizer also returns the number of lower case and upper case letters in the original string. Strings are represented as `char` arrays. The precondition requires the input string to be not null. The postcondition specifies that the counters are non-negative, that the total number of letters is no larger than the length of the original string and the length of returned string is exactly that size. Furthermore the postcondition also promises the caller that all the elements in the result string are lower case ASCII characters.

The implementation of the sanitizer is pretty straightforward. The original string is systematically traversed, and when an ASCII letter is encountered it is copied into a buffer as it is or if it is upper case, converted to a lower case and then stored into the buffer. A priori we do not know the number of non-ASCII characters, thus the temporary buffer is made as large as the original string. However, on loop exit, we exactly know the length of the sanitized string (it is `lower` + `upper`), so a buffer of the right size is allocated, all the sanitized elements are copied into it, and then it is returned.

The CodeContracts static checker (codename Clousot [9]), performs an abstract interpretation of `Sanitize` to verify that the implementation meets its contract (specification). Clousot analyzes methods in isolation using a classical assume/guarantee reasoning. Clousot directly analyzes bytecode, so it is independent of the particular source language [15]. As a matter of fact Clousot users include C# as well as VB programmers. All the internals of the analyzer are hidden to the user, to whom the Clousot is exposed as an extension of the usual development environment (Fig. 2).

From a high point of view, Clousot has three main phases: inference, checking and inter-module propagation. In the inference phase the program is analyzed to infer facts. In the checking phase the facts are used to discharge the proof

```
public char[] Sanitize(char[] str, ref int lower, ref int upper)
{
 Contract.Requires(str != null);

 Contract.Ensures(upper >= 0);
 Contract.Ensures(lower >= 0);
 Contract.Ensures(lower + upper <= str.Length);
 Contract.Ensures(lower + upper == Contract.Result<char[]>().Length);

 Contract.Ensures(
  Contract.ForAll(0, lower + upper, index => 'a' <= Contract.Result<char[]>()[index]));
 Contract.Ensures(
  Contract.ForAll(0, lower + upper, index => Contract.Result<char[]>()[index] <= 'z'));

 upper = lower = 0;

 var tmp = new char[str.Length];

 int j = 0;
 for (int i = 0; i < str.Length; i++)
 {
   var ch = str[i];

   if ('a' <= ch && ch <= 'z') { lower++; tmp[j++] = ch;}
   else if ('A' <= ch && ch<= 'Z'){ upper++; tmp[j++] = (char)(ch | ' ');}
 }

 var result = new char[j];
 for (int i = 0; i < j; i++) { result[i] = tmp[i]; }

 return result;
}
```

Fig. 1. A string sanitizer and its specification with CodeContracts. Clousot, the CodeContracts static checker, proves that the postcondition holds at the end of the method and that no runtime exception is ever thrown. The verification is completely automatic, with Clousot inferring the right loop invariants with no user assistance.

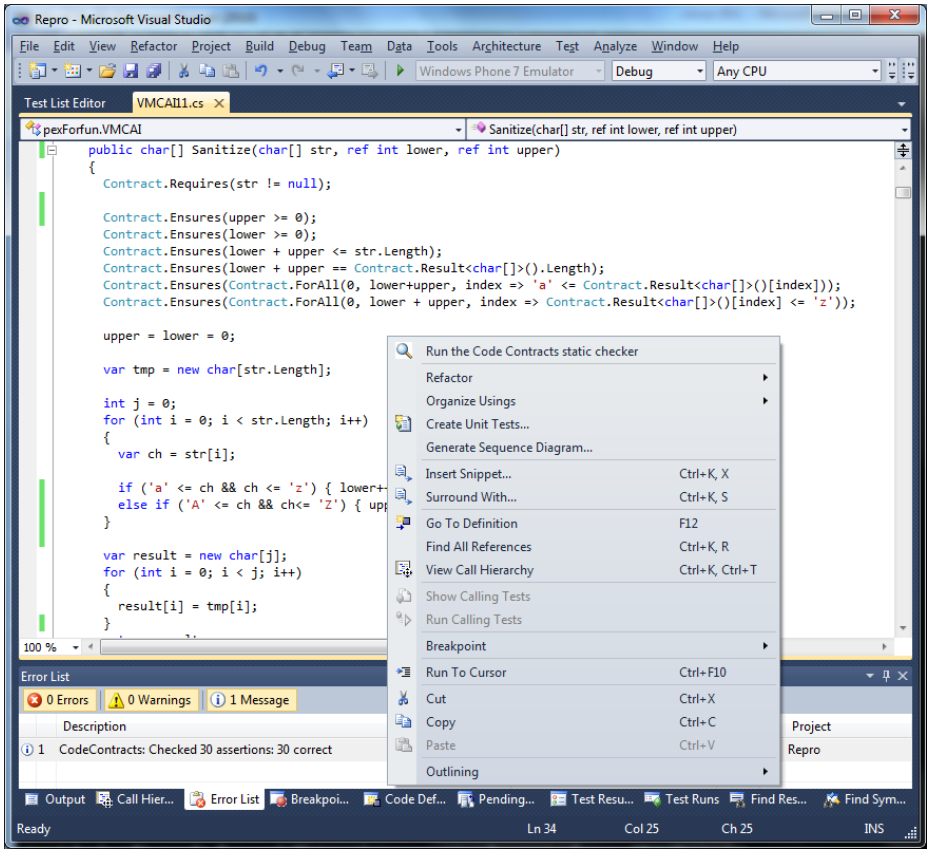

Fig. 2. A screenshot illustrating the user experience with Clousot in Visual Studio. The user can run the analyzer on the whole project, or she can opt for a selective method/class verification with a simple right click. The output is reported in the usual `ErrorList` window, the same where *e.g.* the compiler provides its output.

obligations. There are two kinds of proof obligations: explicit (assertions from contracts) and implicit (assertions from the semantics of the language). If the checking phase is inconclusive, the analysis is refined by using a more precise abstract domain and/or a backward goal-directed reasoning. In the inter-module propagation, inferred contracts are propagated to the callers [5].

Unlike previous approaches based on weakest precondition calculus (*e.g.* [12, 3, 1, 2, 11]), Clousot is based on abstract interpretation [4]. This provides us several advantages. First, the analyzer is automatic: Loop invariants are automatically inferred, without requiring the programmer to provide (for instance) trivial loop invariants. In the example above *all* the invariants are inferred without user interaction. Similarly, the underlying abstract domains are of infinite height and width, providing a stronger expressivity than the domains used for

instance in predicate abstraction. For instance in the example the non-trivial loop invariant `lower` + `upper` == `j` is automatically discovered by the analysis. Second, the analyzer is performing: The trade-off cost/precision can be finely tuned by adjusting the precision of the underlying abstract domains. Third, the analysis is predictable: by performing chaotic iterations following the program structure, the analysis mimics the intentions of the programmer, so that causes of false positive can be (more) easily found. Furthermore, termination is guaranteed so that annoying causes of analysis non-determinism such as unlucky infinite quantifier instantiation are ruled out. There are some drawbacks though. The main one is that Clousot verification focuses only on a certain class of properties, forgetting for instance general universal quantified or existential quantifiers. Clousot instead contains abstract domains to check common contracts involving: non-nullness [8], linear arithmetic [16, 14, 13] and numerical properties in general [10], simple facts over arrays and containers [6] and un-interpreted facts.

References

1. M. Barnett, B.-Y.E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*.
2. G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK - a tool for validation of security and behaviour of Java applications. In *FMCO'06*.
3. P. Chalin, J.R. Kiniry, G.T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO'05*, LNCS 4111, pages 77–101. Springer, 2006.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*. ACM Press.
5. P. Cousot, R. Cousot, and F. Logozzo. Contract precondition inference from intermittent assertions on collections. In *VMCAI'11*. Springer-Verlag, 2011.
6. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM POPL'11*. ACM Press, January 2011.
7. M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *ACM SAC'10*, 2010.
8. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM OOPSLA*, 2003.
9. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS'10*, LNCS. Springer-Verlag, 2010.
10. P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code in .NET. In *OOPSLA'08*. ACM Press, 2008.
11. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07*.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM PLDI'02*.
13. V. Laviron and F. Logozzo. Refining abstract interpretation-based static analyses with hints. In *APLAS'09*.

14. V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI'09*.
15. F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC'08*.
16. F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *ACM SAC'08*.
17. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.