Static Verification for Code Contracts

Manuel Fähndrich

Microsoft Research maf@microsoft.com

Abstract. The Code Contracts project [3] at Microsoft Research enables programmers on the .NET platform to author specifications in existing languages such as C# and VisualBasic. To take advantage of these specifications, we provide tools for documentation generation, runtime contract checking, and static contract verification.
This talk details the overall approach of the static contract checker and examines where and how we trade-off soundness in order to obtain a practical tool that works on a full-fledged object-oriented intermediate language such as the .NET Common Intermediate Language.

1 Code Contracts

Embedding a specification language in an existing language consists of using a set of static methods to express specifications inside the body of methods [4].

```
1  string TrimSuffix(string original, string suffix)
2  {
3    Contract.Requires(original != null);
4    Contract.Requires(!String.IsNullOrEmpty(suffix));
5
6    Contract.Ensures(Contract.Result() != null);
7    Contract.Ensures(!Contract.Result().EndsWith(suffix));
8
9    var result = original;
10   while (result.EndsWith(suffix)) {
11     result = result.Substring(0, result.Length - suffix.Length);
12   }
13   return result;
14 }
```

The code above specifies two preconditions using calls to Contract.Requires and two postconditions using calls to Contract.Ensures. These methods have no intrinsic effect and are just used as markers in the resulting compiled code to identify the preceeding instructions as pre- or postconditions.

2 Verification Steps

Our analysis operates on the compiled .NET Common Intermediate Language [2] produced by the standard C# compiler. The verification is completely modular in that we analyze one method at a time, taking into account only the contracts of called methods. In our example, the contracts of called String methods are:

```
int Length {
  get { Contract.Ensures(Contract.Result<int>() >= 0); }
}

static bool IsNullOrEmpty(string str)
{
  Contract.Ensures( Contract.Result<bool>() == (str == null || str.Length == 0) );
}

bool EndsWith(string suffix)
{
  Contract.Requires( suffix  != null);

  Contract.Ensures( ! Contract.Result<bool>() || value.Length <= this.Length);
}

string Substring(int startIndex, int length)
{
  Contract.Requires(0 <= startIndex);
  Contract.Requires(0 <= length);
  Contract.Requires( startIndex + length <= this.Length);

  Contract.Ensures(Contract.Result<string>() != null);
  Contract.Ensures(Contract.Result<string>().Length == length);
}
```

We factor the code to be analyzed into subroutines: one subroutine per method body, one subroutine for a method's preconditions, and one subroutine for a method's postconditions. The actual code to be analyzed is then formed by inserting calls to appropriate contract subroutines in the method body. In our example, we insert a subroutine call to TrimSuffix's precondition on entry of the method, and a subroutine call to its postcondition on all exits of the method. Additionally, at each method call-site, we insert a call to the precondition subroutine of the called method just prior to the actual call, and a call to the corresponding postcondition subroutine immediately following the call.

The actual contract calls to Contract.Requires or Contract.Ensures turn into either **assert** or **assume** statements depending on their context. Requires on entry of a method turn into **assume** and Ensures on exit of a method turn into **assert**. Conversely, at call-sites, Requires turn into **assert**, and Ensures turn into **assume**.

Conditional branches are expanded into non-deterministic branches with **assume** statements on the outgoing edges. Additional proof obligations for implicit correctness conditions in MSIL, such as null-dereference checks and array bound checks can be automatically inserted into the analyzed code as **assert**s based on user preference.

In this manner, all conditions are simply sequences of MSIL instructions, no different than ordinary method body code, and all assumptions are **assume** statements, and all proof-obligations are **assert** statements.

2.1 Heap Abstraction

Next, the code is transformed into a scalar program by abstracting away the heap. This is the step where we allow some assumptions and approximations that are not safe in general in order to obtain a practical analysis that does not over-burden the programmer. First, we assume that memory locations not explicitly aliased by the code under analysis are non-aliasing. This is clearly an optimistic assumption, but works very well in practice. Second, we guess the set of heap locations that are modified at call-sites (we don't require programmers to write heap modification clauses). Our guesses are often conservative, but may be optimistic if our non-aliasing assumptions are wrong. These assumptions allow us to compute a value numbering for all values accessed by the code, including heap accessing expressions. We also introduce names for uninterpreted functions marked as [Pure] by the programmer. This provides reasoning over abstract predicates. Finally, abstracting the heap also removes old-expressions in postconditions that refer to the state of an expression at the beginning of the method.

To compute the value numbering, we break the control flow of the analyzed code into maximal tree fragments. The root of each tree fragment is a join point (or the method entry point) and is connected by edges to predecessor leafs of other tree fragments.

The set of names used by the value numbering is unique in each tree fragment. Edges connecting tree leafs to tree roots contain a set of assignments effectively rebinding value names from one fragment to the names of the next. The resulting code is in mostly passive form, where each instruction simply relates a set of value names. This form is ideal for standard abstract interpretation based on numerical domains. The assignments on rebinding edges between tree fragments provide a way to transform abstract domain knowledge prior to the join from one set of value names to the next, so that the join can operate on a common set of value names. The rebindings act as a generalization of ϕ-nodes. In contrast to ϕ-nodes which provide a join for each value separately, our rebindings form a join for the entire state simultaneously, which is crucial to maintain relational properties.

2.2 Abstract Interpretation Fixpoints

On the scalar program, we compute abstract program invariants for each program point based on standard abstract interpretation fixpoint techniques [1]. Our motivation to use abstract interpretation rather than theorem proving techniques is to enable programmers to use static verification without requiring them to write loop invariants. It also provides control over cost/precision trade-offs. We use a variety of novel domains such as Pentagons, Disintervals, and Subpolyhedra [5] to deal with relations that arise in practice. We also lift these domains over sequences in order to deal with universally quantified properties.

For each **assert** statement in the code, we attempt to discharge the proof obligation using the computed fixpoint at that program point. If the abstract

state is strong enough to imply the obligation, the obligation is discharged. Otherwise, we attempt to discharge it using an additional backward analysis.

2.3 Weakest Precondition Analysis

If the abstract state at an **assert** is too weak to imply the proof obligation, we transform the obligation using weakest preconditions into obligations for all predecessor program points and attempt to use the abstract state at those points to discharge them. This approach is good at handling disjunctive invariants which our abstract domains typically don't represent precisely. E.g., an **assert** after a join point may not be provable due to loss of precision at the join. However, the abstract states at the program points just prior to the join may be strong enough to discharge the obligation. This backwards analysis discharges an obligation if it can be discharged on all paths leading to the assertion. It thus acts as a form of on-demand trace partitioning.

3 Conclusion

For the example code, our verification discharges 5 implicit non-null obligations on the receiver of the calls to EndsWith, Substring, and Length. It also discharges all preconditions of these methods as well as the postconditions of TrimSuffix.

Our tools have been available to the general public since March 2009 and the response has been very positive. We received much useful feedback that has been incorporated back into the tools. We believe that our approach is viable and that we have made good progress towards our goal of enabling non-verification experts to start writing specifications and use tools to enforce better programming discipline. Still, much work remains to be done on the static verification with respect to better scalability, precision, and automation.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. ACM Press (Jan 1977)
2. ECMA: Standard ECMA-355, Common Language Infrastructure (Jun 2006)
3. Fähndrich, M., Barnett, M., Logozzo, F.: Code Contracts (Mar 2009), `http://research.microsoft.com/contracts`
4. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 2103–2110. ACM, New York, NY, USA (2010)
5. Laviron, V., Logozzo, F.: Subpolyhedra: A (more) scalable approach to infer linear inequalities. In: VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 229–244. Springer-Verlag, Berlin, Heidelberg (2009)