Technology for Inferring Contracts from Code

Francesco Logozzo
Microsoft Research
One Microsoft Way, Redmond, WA, USA
logozzo@microsoft.com

ABSTRACT

Contracts are a simple yet very powerful form of specification. They consists of method preconditions and postconditions, of object invariants, and of assertions and loop invariants. Ideally, the programmer will annotate all of her code with contracts which are mechanically checked by some static analysis tool. In practice, programmers only write few contracts, mainly preconditions and some object invariants. The reason for that is that other contracts are "clear from the code": Programmers do not like to repeat themselves. As a consequence, any *usable* static verification tool should provide some form of contract inference.

Categories and Subject Descriptors

D [**Software**]: Miscellaneous; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers,Programming by contract*

General Terms

Verification

Keywords

Abstract Interpretation,Contracts,Inference

1. CONTRACT INFERENCE

Abstract interpretation [2] provides the theoretical foundations for automatic contracts inference. The contract inference problem is just an abstraction of the trace semantics. For instance, a loop invariant is an abstraction of the states reaching the loop head and an object invariant is an abstraction of all the states reachable in the steady points of an object [8].

1.1 Loop invariants

Abstract interpretation provides an elegant methodology to infer loop invariants. First, set up a sound abstract domain. The abstract domain captures the properties of interest, *e.g.*, the shape of the heap, linear inequalities among program variables [6], or array contents [5]. Soundness guarantees that no concrete behavior is ignored. In practice, the analysis abstract domain is built by composing atomic abstract domains. Second, set up the abstract operations and transfer functions. The abstract operations combine two abstract elements, the transfer functions describe how abstract states are modified by atomic program statements. Third, design convergence operators (widening, narrowing). Convergence operators guarantee that the loop inference process actually terminates.

Finally, the inferred loop invariant is just the abstract element at the loop head computed by the abstract semantics above. In practice, as the loop invariant is mainly used by the tool, we are not interested in having a "nice-looking" invariant.

1.2 Postconditions

Theoretically, an inferred postcondition is similar to a loop invariant: it is just the abstract element at the method return point. However, in practice we'd like to have "nice-looking" and compact postconditions, *e.g.*, without redundant information. At this aim, the postcondition inference proceeds as follows. First, project all the locals from the abstract state — they are not visible to the external callers. Second, ask each atomic abstract domain to serialize its knowledge into a user-readable form — the abstract domains may have a very compact and optimized representation of their elements, not suitable to appear in a contract. Third, remove the contracts that already appear in the source code as postconditions. Fourth, sort and simplify the redundant postconditions.

1.3 Preconditions

We differentiate among *sufficient* and *necessary* preconditions. If valid, a sufficient precondition guarantees the callee is correct, but nothing can be said if it not valid — the callee may or may not be correct. If not-valid, a necessary precondition guarantees the callee is incorrect, but nothing can be said if it is valid. When automatic inference of preconditions is considered, we advocate the inference of necessary precondition. In fact, a sufficient precondition can be too strong for a caller — at worst `false`. On the other hand, a necessary precondition is something that should be satisfied by the caller, otherwise the program will definitely fail later. We designed several algorithms to infer necessary preconditions:

HILT'13, November 12–14, 2013, Pittsburgh, PA, USA.
ACM 978-1-4503-2467-0/13/11.
http://–enter the whole DOI string from rightsreview form confirmation.

atomic, with disjuctions, and for collections [4]. Necessary preconditions can be easily checked to be also sufficient by injecting them and reanalysing the callee [3].

1.4 Object Invariants

We differentiate among *reachable* and *necessary* object invariants. A reachable object invariant characterizes all the fields values that are reachable after the execution of the constructor or any public method in the class [7]. A necessary object invariant is a condition on the object fields that should hold, otherwise there exists a sequence of public method calls causing the object into an error state [1]. Reachable and necessary object invariants are complementary, and both can be used to improve the precision of contract-based static analyzers.

2. CONCLUSIONS

Inferred contracts are vital for the success of verification tools. In our static contract checker, cccheck/Clousot, we spent a large amount of time to implement, refine, and optimize the contract inference algorithms.

3. REFERENCES

[1] M. Bouaziz, L. Logozzo, and M. Fähndrich. Inference of necessary field conditions with abstract interpretation. In *APLAS*, 2012.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*. ACM Press, Jan. 1977.

[3] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, pages 128–148, 2013.

[4] P. Cousot, R. Cousot, and F. Logozzo. Contract precondition inference from intermittent assertions on collections. In *VMCAI'11*, 2011.

[5] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceeding of the* 38^{th} *ACM Symposium on Principles of Programming Languages (POPL 2011)*. ACM Press, Jan. 2011.

[6] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI '09*, 2009.

[7] F. Logozzo. *Modular static analysis of object-oriented languages*. Thèse de doctorat en informatique, École polytechnique, 2004.

[8] F. Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems & Structures*, 35(2):100–142, 2009.