

Contracts: Specification and Verification

<http://d3s.mff.cuni.cz>

Department of
Distributed and
Dependable
Systems



Pavel Parízek



FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

Behavior specification using contracts

- Target: program fragment
 - class, object, method (procedure), loop body
- Purpose: define responsibilities
 - Implementation (provider, method, object)
 - Client (caller method, another component)
- Method contract
- Object contract

Method contract

- Precondition
 - Specifies constraints on parameter values and valid states of a target object
 - Logic formula that must hold **at the entry** to the method
 - “caller responsibility”
- Postcondition
 - Specifies constraints on the return value and side effects
 - Captures relation between the initial and final state of the method
 - Logic formula that must hold **at the exit** from the method
 - “implementation responsibility”

Method contract: example

- Program

```
public class ArrayList {  
    public void add(int index, Object obj) {  
        ...  
    }  
    public int size() { ... }  
}
```

- Textual documentation

“Value of the `index` parameter has to be greater than or equal to zero.
Successful call of `add` increases the size of the array by one.”

- Formal contract

```
public void add(int index, Object obj)  
    requires index >= 0;  
    ensures size = old(size) + 1;  
    { ... }
```

Object contract

- Object invariant
 - Specifies valid object states (e.g., values of fields)
 - Logic formula that must hold at the entry and exit of each method defined for the object

How to define contracts

- Three ways
 - Source code comments
 - Explicit annotations
 - Built-in language constructs
- Contract specification languages
 - Spec#, JML, Dafny, Code Contracts, Viper, ...

Spec#

- Programming system
 - Developed by Microsoft Research
 - <https://www.microsoft.com/en-us/research/project/spec/>
- Main components
 - Programming language
 - Extension of C# with contracts
 - Spec# compiler
 - Inserts run-time checks for contracts into the code
 - Verifier: Boogie

Spec# language

```
class ArrayList {  
    public virtual object Insert(int index, object value)  
    {  
        requires 0 <= index && index <= Count;  
        ensures value == this[index];  
        ensures Count == old(Count) + 1;  
        ensures result == old(this[index]);  
        ...  
        int i = count;  
        while (i >= index)  
            loop invariant i >= index - 1;  
        {  
            data[i+1] = data[i];  
            i--;  
        }  
    }  
}
```

precondition

postcondition

return value

initial value

must hold before and after each iteration

JML: Java Modeling Language

- Contract definition language for Java
 - <http://www.eecs.ucf.edu/~leavens/JML/index.shtml>
- Differences from Spec#
 - Contracts defined in source comments
 - No built-in Java language constructs
 - Example

```
/*@
  @ requires E1;
  @ ensures E2;
  @*/
public int doSmth() { ... }
```
- Verification tool: ESC/Java2
 - <http://kindsoftware.com/products/opensource/ESCJava2/>

Advanced features of Spec# and JML

- Exceptional behavior
 - Constraints on the resulting state when an exception is thrown inside the method
- Model fields (“ghost”)
 - Abstract fields visible only in the contracts
- Quantifiers (\exists, \forall)
 - Spec#: `Exists` and `Forall`
- Behavioral subtyping
 - Inheritance of contracts
- Frame conditions
 - List of fields which the method can modify

Verification of program against contracts



Verification of program against contracts

- Goal
 - Checking consistency between the method's implementation and its contract
 - φ : precondition \wedge implementation \rightarrow postcondition
- Target: Spec#
 - Boogie program verifier, SMT solver Z3

Verifying Spec# contracts with Boogie

- Input

- Spec# program (C# annotated with contracts)
- Set of axioms that describe semantics of Spec#

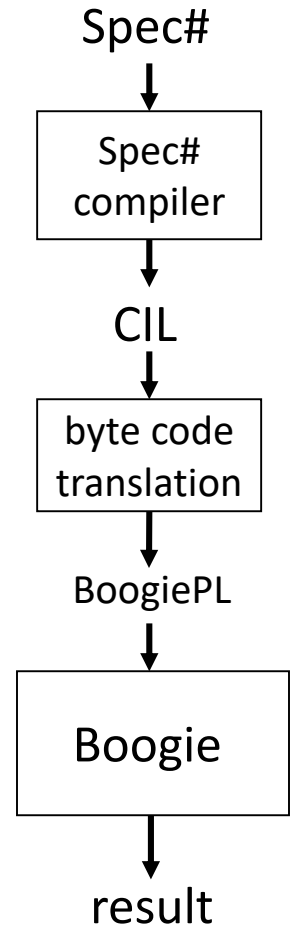
- Axioms

- Semantics

- Type system (subtyping)
- Size of constants

- Examples

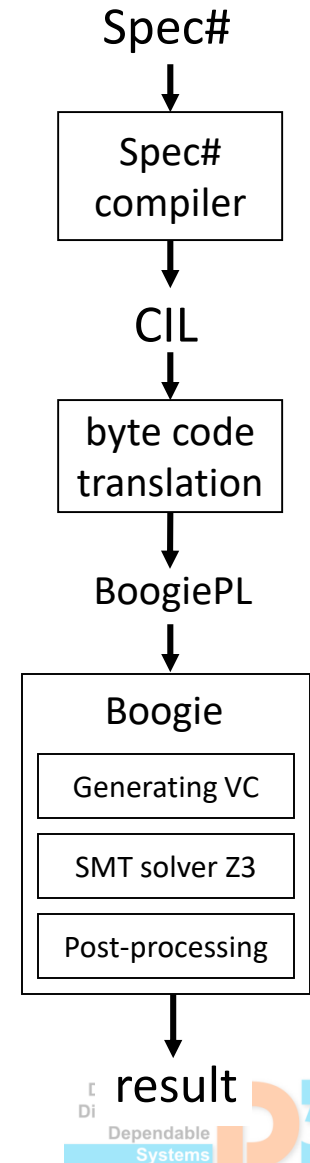
- All classes are subtypes of `System.Object`
- `forall T:type . T <: superclass(T)`



Verifying Spec# contracts with Boogie

- Algorithm

- Translate Spec# program into BoogiePL
- Generate verification condition (VC) from the BoogiePL program
- Run the SMT solver on the VC
 - Result: “no error found” or counterexample
- Post-processing of the result
 - Mapping counterexample back to the source language (Spec#)



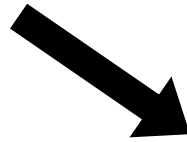
Running example

```
int M(int x)
  requires 100 <= x;    // precondition
  ensures result == 0;  // postcondition
{
  while (0 < x)
    invariant 0 <= x;   // loop invariant
  {
    x = x - 1;
  }
  return x;
}
```

Example program in Spec# taken from:
M. Barnett and R. Leino. Weakest-Precondition of Unstructured Programs.
PASTE 2005, ACM Press

Translation from Spec# to BoogiePL

```
int M(int x)
  requires 100 <= x;    // precondition
  ensures result == 0;  // postcondition
{
  while (0 < x)
    invariant 0 <= x;  // loop invariant
  {
    x = x - 1;
  }
  return x;
}
```



```
Start:  assume 100 <= x;    // precondition
        goto Head;
Head:   assert 0 <= x;      // loop invariant
        goto Body, After;
Body:   assume 0 < x;      // loop guard
        x := x - 1;
        goto Head;
After:  assume not(0 < x);  // neg loop guard
        r := x;            // return
        assert r = 0;      // postcondition
        goto ;
```


- Program structure
 - A program is a set of basic blocks (label, statements)
 - Successor blocks are targets of the `goto` statement
- Semantics
 - Program defines a large set of execution traces
 - State = values of all variables + program counter
 - Arbitrary initial values of all program variables
- Important statements
 - `goto label1, label2` → non-deterministic choice
 - `goto ;` → the execution trace terminates successfully
 - `assume E` → filters out execution traces not satisfying E
 - `assert E` → if E is *false*, then a trace ends with an error

Generating verification condition (VC)

- Construction of an acyclic program (AP)
 - Eliminating loops (back edges in control-flow)
- Transforming into an acyclic passive program (APP)
 - No assignments allowed in APP
- Generating verification condition from the APP

Construction of acyclic program

- What must be still checked in AP
 - Loop invariant holds before the loop starts
 - Any iteration does not break the invariant
- Consequence
 - Loop invariant holds at the exit from the loop
- Eliminating loops
 - Abstraction of an arbitrary number of loop iterations
 - Unrolling the loop body

Abstracting loop iterations

```
Start:  assume 100 <= x;  
        assert 0 <= x;      // check loop invariant  
        goto Head;  
  
Head:  havoc x;           // reset variables used in the loop  
        assume 0 <= x;     // assume loop invariant  
        goto Body, After;  
  
Body:  assume 0 < x;  
        x := x - 1;  
        assert 0 <= x;  
        goto ;  
  
After: assume not(0 < x);  
        r := x;  
        assert r = 0;  
        goto ;
```

Unrolling loop body

```
Start:  assume 100 <= x;
        assert 0 <= x;      // check loop invariant
        goto Head;

Head:   havoc x;             // reset variables used in the loop
        assume 0 <= x;      // assume loop invariant
        goto Body, After;

Body:   assume 0 < x;
        x := x - 1;
        assert 0 <= x;      // check loop invariant
        goto ;             // back edge removed

After:  assume not(0 < x);
        r := x;
        assert r = 0;
        goto ;
```

AP: acyclic program

```
Start:  assume 100 <= x;  
        assert 0 <= x;      // check loop invariant  
        goto Head;  
  
Head:  havoc x;             // reset variables used in the loop  
        assume 0 <= x;      // assume loop invariant  
        goto Body, After;  
  
Body:  assume 0 < x;  
        x := x - 1;  
        assert 0 <= x;      // check loop invariant  
        goto ;              // back edge removed  
  
After: assume not(0 < x);  
        r := x;  
        assert r = 0;  
        goto ;
```

Transforming into acyclic passive programs

- Passive program
 - No destructive update allowed
- Two steps
 - Rewrite into a single-assignment form
 - Removing all assignment statements

Rewriting into single-assignment form

```
Start:  assume 100 <= x0;  
        assert 0 <= x0;  
        goto Head;  
Head:   skip; // "havoc x1" not necessary anymore  
        assume 0 <= x1;  
        goto Body, After;  
Body:   assume 0 < x1;  
        x2 := x1 - 1;  
        assert 0 <= x2;  
        goto ;  
After:  assume not(0 < x1);  
        r1 := x1;  
        assert r1 = 0;  
        goto ;
```


Rewriting into single-assignment form

- Problem
 - Join points (after choice)

```
x0 := ...;  
if (E) { x1 := ... }  
else { x2 := ... }
```

Rewriting into single-assignment form

- Problem

- Join points (after choice)

```
x0 := ...;  
if (E) { x1 := ... }  
else { x2 := ... }
```

- Solution (ϕ -functions)

```
x0 := ...;  
if (E) { x1 := ...; x3 := x1 }  
else { x2 := ...; x3 := x2 }
```

Removing assignment statements



```
Start:  assume 100 <= x0;  
        assert 0 <= x0;  
        goto Head;  
  
Head:  skip;  
        assume 0 <= x1;.  
        goto Body, After;  
  
Body:  assume 0 < x1;  
        assume x2 = x1 - 1;  
        assert 0 <= x2;  
        goto ;  
  
After: assume not(0 < x1);  
        assume r1 = x1;  
        assert r1 = 0;  
        goto ;
```

APP: acyclic passive program

```
Start:  assume 100 <= x0;  
        assert 0 <= x0;  
        goto Head;  
  
Head:  skip;  
        assume 0 <= x1;.  
        goto Body, After;  
  
Body:  assume 0 < x1;  
        assume x2 = x1 - 1;  
        assert 0 <= x2;  
        goto ;  
  
After: assume not(0 < x1);  
        assume r1 = x1;  
        assert r1 = 0;  
        goto ;
```

Encoding control flow into logic formula

- Boolean variable B_{ok} is defined for each basic block B
 - $B_{ok} = true \rightarrow$ all possible executions of B and its successors from the current state are correct
- Block equation B_{be} is defined for each basic block B

$$Start_{be}: Start_{ok} \leftrightarrow 100 \leq x0 \Rightarrow (0 \leq x0 \wedge Head_{ok})$$

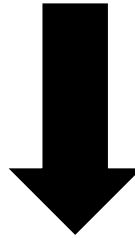
$$Head_{be}: Head_{ok} \leftrightarrow 0 \leq x1 \Rightarrow (Body_{ok} \wedge After_{ok})$$

$$Body_{be}: Body_{ok} \leftrightarrow 0 < x1 \Rightarrow (x2 = x1 - 1 \Rightarrow 0 \leq x2)$$

$$After_{be}: After_{ok} \leftrightarrow \neg(0 < x1) \Rightarrow (r1 = x1 \Rightarrow r1 = 0)$$

Generating verification condition

$Start_{be}: Start_{ok} \leftrightarrow 100 \leq x0 \Rightarrow (0 \leq x0 \wedge Head_{ok})$
 $Head_{be}: Head_{ok} \leftrightarrow 0 \leq x1 \Rightarrow (Body_{ok} \wedge After_{ok})$
 $Body_{be}: Body_{ok} \leftrightarrow 0 < x1 \Rightarrow (x2 = x1 - 1 \Rightarrow 0 \leq x2)$
 $After_{be}: After_{ok} \leftrightarrow \neg(0 < x1) \Rightarrow (r1 = x1 \Rightarrow r1 = 0)$



VC: $Axioms \wedge Start_{be} \wedge Head_{be} \wedge Body_{be} \wedge After_{be} \Rightarrow Start_{ok}$

What does the verification condition mean

a run of the program according to semantics of Spec#

$$Axioms \wedge Start_{be} \wedge Head_{be} \wedge Body_{be} \wedge After_{be} \Rightarrow Start_{ok}$$

postcondition not violated

Contracts and procedure calls

- Idea: use contracts of individual procedures
- Procedure calls

...				assert <i>precondition of M</i>
call M		→		havoc <i>fields modified by M</i>
...				assume <i>postcondition of M</i>

Verification of contracts: limitations

- Incompleteness
 - First-order predicate calculus is semi-decidable
 - Verification tool may run forever on some inputs (programs)
 - Making tools less precise → spurious warnings
- Modular verification
 - Analyze procedures separately (one at a time)
 - Cannot detect errors depending on internal behavior of other procedures (with partial contracts)
 - Better performance and scalability
 - Verification applicable to real-world programs

- Dafny
 - Verification-ready programming language
 - Builds upon the ideas (algorithm) of Spec#
 - <https://dafny.org/>
 - <https://github.com/dafny-lang/dafny>
- VCC: Verifier for Concurrent C
 - <https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/>
 - Target domain: low-level concurrent systems (e.g., OS)
 - Challenge: verify programs with threads and pointers
 - Solution: **object ownership**
 - Thread can write only to objects that it owns in the given state
 - Thread can read only objects that it owns or does not change

- Viper: Verification Infrastructure for Permission-based Reasoning
 - <http://viper.ethz.ch/>
 - Contract language + set of verification tools
 - Limited support for object-oriented programming
 - Features: ownership, **access permissions**
 - Usage: plugin for VSCode, online interface
 - Examples: <http://viper.ethz.ch/examples/>
 - Sorted List (basic access permissions)
 - Linked List (with recursive predicates)

Disclaimer

- Code Contracts
 - Similar definition language
 - Method preconditions and postconditions, invariants
 - Different verification algorithm
 - Mostly based on abstract interpretation (lecture 9)

Further reading

- M. Barnett, K.R.M. Leino, and W. Schulte. **The Spec# Programming System: An Overview**. CASSIS 2004
- M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. **Boogie: A Modular Reusable Verifier for Object-Oriented Programs**. FMCO 2005
- M. Barnett and K.R.M. Leino. **Weakest-Precondition of Unstructured Programs**. PASTE 2005, ACM
- K.R.M. Leino. **Dafny: An Automatic Program Verifier for Functional Correctness**. LPAR 2010
 - <https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/>
- P. Muller, M. Schwerhoff, and A.J. Summers. **Viper: A Verification Infrastructure for Permission-Based Reasoning**. VMCAI 2016