Contracts: Specification and Verification

http://d3s.mff.cuni.cz

Pavel Parízek

Department of Distributed and Dependable Systems

CHARLES UNIVERSITY IN PRAGUE
faculty of mathematics and physics
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**
  ```java
  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**
  ```java
  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, Code Contracts, ...
Spec#

- Programming system
  - Developed by Microsoft Research

- Main components
  - Programming language
    - Extension of C# with contracts
  - Spec# compiler
    - Inserts run-time checks for contracts into the code
  - Verifier: Boogie
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

Spec# language

Contracts: Specification and Verification

Pavel Parízek
JML: Java Modeling Language

- Contract definition language for Java

- 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
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 (∃, ∀)
  - Spec#: Exists and Forall
- Behavioral subtyping
  - Inheritance of contracts
- Frame conditions
  - List of fields which the method can modify
Verification of program against contracts

• Goal
  - Checking consistency between the method’s implementation and its contract
    - $\phi$: precondition $\land$ 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#)

```
Spec# -> Spec# compiler
              ^      
              |      
              v      
CIL         

byte code translation

BoogiePL     

Generating VC
SMT solver Z3
Post-processing

result
```
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;
}
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;
    x := x - 1;
    goto Head;

After: assume not(0 < x); // neg loop guard
    r := x;
    assert r = 0;
    goto ; // return

    // postcondition
Program structure
- A program is a set of basic blocks (label, statements)
- Successor blocks are targets of the \texttt{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
- \texttt{goto} \texttt{label1, label2} $\Rightarrow$ non-deterministic choice
- \texttt{goto} ; $\Rightarrow$ the execution trace terminates successfully
- \texttt{assume} \ E $\Rightarrow$ filters out execution traces not satisfying \( E \)
- \texttt{assert} \ E $\Rightarrow$ if \( E \) is \texttt{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

\textbf{Start:} \hspace{1em} assume \texttt{100 <= x;}
\hspace{1em} \textbf{assert 0 <= x;} \quad \text{\texttt{// check loop invariant}}
\hspace{1em} goto Head;

\textbf{Head:} \hspace{1em} \texttt{havoc x;} \quad \text{\texttt{// reset variables used in the loop}}
\hspace{1em} assume \texttt{0 <= x;} \quad \text{\texttt{// assume loop invariant}}
\hspace{1em} goto Body, After;

\textbf{Body:} \hspace{1em} assume \texttt{0 < x;}
\hspace{1em} x := x - 1;
\hspace{1em} \textbf{assert 0 <= x;}
\hspace{1em} goto ;

\textbf{After:} \hspace{1em} assume \texttt{not(0 < x);} 
\hspace{1em} r := x;
\hspace{1em} \textbf{assert r = 0;}
\hspace{1em} goto ;
Unrolling loop body

Start: assume 100 \leq x;
assert 0 \leq x;  // check loop invariant
goto Head;

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

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

After: assume not(0 < x);
r := x;
assert r = 0;
goto ;
**AP: acyclic program**

**Start:**
\[
\text{assume } 100 \leq x; \\
assert 0 \leq x; \quad // \text{check loop invariant} \\
goto \text{Head};
\]

**Head:**
\[
\text{havoc } x; \quad // \text{reset variables used in the loop} \\
\text{assume } 0 \leq x; \quad // \text{assume loop invariant} \\
goto \text{Body, After};
\]

**Body:**
\[
\text{assume } 0 < x; \\
x := x - 1; \\
assert 0 \leq x; \quad // \text{check loop invariant} \\
goto ; \\
\]

**After:**
\[
\text{assume } \neg(0 < x); \\
r := x; \\
assert r = 0; \\
goto ;
\]
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 ;
Problem

Join points (after choice)

\[ x_0 := \ldots; \]
\[ \text{if} \ (E) \ \{ \ x_1 := \ldots \} \]
\[ \text{else} \ \{ \ x_2 := \ldots \} \]

Q: how to solve this problem?
Problem

- Join points (after choice)

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

Solution

```plaintext
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 ;
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 $B_{\text{ok}}$ is defined for each basic block $B$
  - $B_{\text{ok}} = \text{true} \implies \text{all possible executions of B and its successors from the current state are correct}$

- Block equation $B_{\text{be}}$ is defined for each basic block $B$

\[
\begin{align*}
\text{Start}_{\text{be}} &: \quad \text{Start}_{\text{ok}} \iff 100 \leq x0 \implies (0 \leq x0 \land \text{Head}_{\text{ok}}) \\
\text{Head}_{\text{be}} &: \quad \text{Head}_{\text{ok}} \iff 0 \leq x1 \implies (\text{Body}_{\text{ok}} \land \text{After}_{\text{ok}}) \\
\text{Body}_{\text{be}} &: \quad \text{Body}_{\text{ok}} \iff 0 < x1 \implies (x2 = x1 - 1 \implies 0 \leq x2) \\
\text{After}_{\text{be}} &: \quad \text{After}_{\text{ok}} \iff \neg(0 < x1) \implies (r1 = x1 \implies r1 = 0)
\end{align*}
\]
Generating verification condition

\[
\begin{align*}
\text{Start}_{be} & : \quad \text{Start}_{ok} \iff 100 \leq x0 \Rightarrow (0 \leq x0 \land \text{Head}_{ok}) \\
\text{Head}_{be} & : \quad \text{Head}_{ok} \iff 0 \leq x1 \Rightarrow (\text{Body}_{ok} \land \text{After}_{ok}) \\
\text{Body}_{be} & : \quad \text{Body}_{ok} \iff 0 < x1 \Rightarrow (x2 = x1 - 1 \Rightarrow 0 \leq x2) \\
\text{After}_{be} & : \quad \text{After}_{ok} \iff \neg(0 < x1) \Rightarrow (r1 = x1 \Rightarrow r1 = 0)
\end{align*}
\]

\[
\text{VC: } \text{Axioms} \land \text{Start}_{be} \land \text{Head}_{be} \land \text{Body}_{be} \land \text{After}_{be} \Rightarrow \text{Start}_{ok}
\]
What does the verification condition mean

A run of the program according to semantics of Spec#

\[ \text{Axioms} \land \text{Start}_{be} \land \text{Head}_{be} \land \text{Body}_{be} \land \text{After}_{be} \Rightarrow \text{Start}_{ok} \]

Postcondition not violated
Contracts and procedure calls

- Idea: use contracts of individual procedures

- Procedure calls

  ```
  ... call M ... 
  assert precondition of M 
  havoc fields modified by M 
  assume postcondition of M 
  ```
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
Tools

- Spec#

- VCC: 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
Code Contracts

- Similar definition language
  - Method preconditions and postconditions, invariants

- Different verification algorithm
  - Mostly based on abstract interpretation (lecture 9)

- You will see more today during the labs
Further reading
