Proving what programs do not

Bertrand Meyer

Saint Petersburg Software Engineering Seminar
ITMO, 6 April 2012

Plan

1. Previous language approaches
2. New language approach: the Frame Consistency Principle
3. Verifying the frame condition
4. Assessment and open problems
The frame problem

McCarthy & Hayes (1969):

In proving that one person could get into conversation with another, we were obliged to add the hypothesis that if a person has a telephone he still has it after looking up a number in the telephone book. If we had a number of actions to be performed in sequence we would have quite a number of conditions to write down that certain actions do not change the values of certain fluents. In fact with \( n \) actions and \( m \) fluents we might have to write down \( m \cdot n \) such conditions.
Implementation, specification, client

```
a: ACCOUNT
h: PERSON
b: INTEGER
...
h := a.holder
b := a.balance
a.deposit(100)
check
  a.balance = b + 100
  a.holder = h
end
```

```
class ACCOUNT feature
  holder: PERSON
  balance: INTEGER
  deposit (v: INTEGER)
    do
      balance := balance + v
    ensure
      balance = old balance + v
    end
end
```

Client

```
a.holder = h--?
```

Client

```
a.balance = b + 100-1
```

- 2 -

Previous language approaches
Approach 1: make it explicit

```
class ACCOUNT feature
    holder: PERSON
    balance: INTEGER
    ...
    h := a.holder
    b := a.balance
    a.deposit(100)
end
```

```
check
    a.balance = b + 100
    a.holder = h
end
```

```
Client
    
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Note, however...

The explicit approach works well with inheritance, polymorphism & dynamic binding:

Postconditions and-cumulate

Approach 2: “modifies” clause

"The presence of a modifies-clause asserts that only the set of objects described may have their abstract values newly-defined or changed by the function"

Larch C++ manual

http://www.cs.iastate.edu/~leavens/larchc++manual/lcpp_96.html
Eiffel’s **only** (no longer in language)

```eiffel
class ACCOUNT feature
    holder: PERSON
    balance: INTEGER
    branch: BANK_BRANCH

    deposit (v: INTEGER)
        do
            balance := balance + v
            branch.manager.deposit(1)
        ensure
            balance = old balance + v
            only balance
        end
    end
end
```

Difficulties with **modifies** clauses in O-O programs

Aliasing and remote calls make it difficult to verify the properties

```eiffel
class ACCOUNT feature
    holder: PERSON
    balance: INTEGER
    branch: BANK_BRANCH

    deposit (v: INTEGER)
        do
            balance := balance + v
            branch.manager.deposit(1)
        ensure
            balance = old balance + v
            only balance
        end
    end
end
```
Another problem with **modifies** clauses

```plaintext
class LINKED_LIST[G] feature
  first: LINKABLE[G]
  count: INTEGER
  ...
  set_at(x: G; n: INTEGER)
    -- Set content of n-th item to x.
    require
      positive: n > 0
      possible: n <= count
    local
      a: LINKABLE[G]
    do
      from a := first until i := n loop
        a := a.right; i := i + 1
      end
    end
  end
end

... you can't write them!

![Diagram of a linked list]

May modify: first.item, first.right.item, first.right.right.item, ...
```

Approach 3: dynamic frames (Kassios)

A frame variable denotes a subset of a program's locations

Framing a set of expressions:

\( f \) **frames** \( x, y, z \)

Modification of a frame:

\( \Delta f \)

Theorem:

\( \Delta f \land g \) **frames** \( D \land \text{disjoint}(f, g) \) \( \Rightarrow D' = D \)

Disadvantage of this notion: need to include frames explicitly
- 3 -

A new language approach: the Frame Consistency Principle

Open vs closed world

The previous techniques are “open-world”: anything that is not explicitly specified as unchanged could change.

Reverse convention, “closed world”: anything that is not explicitly specified as changing is implicitly specified as not changing.

("M" specification language, 1985)
Approach 2: “modifies” clause

“The presence of a modifies-clause asserts that only the set of objects described may have their abstract values newly-defined or changed by the function”

Larch C++ manual
http://www.cs.iastate.edu/~leavens/larchc++manual/lcpp_96.html

More about modifies clauses

Note from an informal survey of some JML code: everything that is mentioned in an “assignable” code appears in the postcondition anyway!
A language convention (informal)

Assume a language with contracts (preconditions, postconditions, invariants) such as Eiffel but also JML, Spec# etc.

Further assume a clear distinction between commands and queries

**Contract-based frame convention**

A routine may only change the value of queries mentioned in the contract

More precise version

**The Frame Consistency Principle**

No routine may affect a system property unless its postcondition or invariant mentions the property or some part of it

The "parts" of a property are its subexpressions and (recursively) the parts of their prefixes

Example: the parts of

\[ f(b).x = c \]

are: \[ f(b), x, c, f(b) \]
Notations

For any program element $p$, we define:

- $p^+$ --- The set of "variables" that $p$ may modify
- $\succ p$ --- The parts of $p$

Example: with the instruction

\[
e := a \ast d
\]

--- Instruction $i$

we have

\[
i^+ = \{e\}
\]

\[
\succ i = \{e, a, d, a \ast d\}
\]

The frame condition

Implementing the Frame Condition principle means verifying, for any routine $r$, the property

\[
\text{body}_r \subseteq \succ \text{contract}_r
\]

or **frame condition**
Applying the frame condition in practice

\[
\text{body}_r \subseteq \rightarrow \text{contract}_r
\]

\[
\begin{align*}
\text{set}_a & \quad \text{do} & \quad b := b + 1 & & \text{a} := b & & \text{end} \\
\text{set}_c & \quad \text{do} & \quad \text{set}_a & & \text{c} := a & & \text{end} & & \text{ensure} & & c = a
\end{align*}
\]

\[
\begin{align*}
\text{body}_{\text{set}_c} & = \{a, b, c\} & & \subseteq & & \rightarrow & & \text{contract}_{\text{set}_c} = \{a, b, c\} \\
\text{contract}_{\text{set}_c} & = \{a, b\} & & \varphi & & \text{Not OK!} & & \text{OK!} \\
\text{>contract}_{\text{set}_c} & = \{a, c\} & & \text{May add} & & b = \text{old} b + 1 & & \text{Or anything citing } b
\end{align*}
\]

New function in \textbf{ANY}

\[
\text{relevant}(x: \text{ANY}): \text{BOOLEAN} \\
\quad -- \text{Is } x \text{ of interest?} \\
\quad \text{do} \\
\quad \quad \text{Result} := \text{True} \\
\quad \text{end}
\]
Citing a variable

set_a
do
  b := b + 1
  a := b
end

set_c
do
  set_a
  c := a
  ensure
  c = a
end

body_{set_c} = \{a, b, c\}

contract_{set_c} = \{a, b, c\}

\subseteq \text{OK!}
\not\subseteq \text{Not OK!}

May add
b = old b + 1

Example: relevant (b)

body_{r} \subseteq >contract_{r}

Example: relevant (b)
Citing a variable

```plaintext
set_a
  do
    b := b + 1
    a := b
  end

set_c
  do
    set_a
    c := a
    ensure
    c = a
  end

body_{set_e} = \{a, b, c\}

contract_{set_e} = \{a, c\}
```

Not OK!

May add

\[ b = \text{old} \ b + 1 \]

Example: `relevant(b)`

Citing several expressions

Because `relevant` takes an argument of type `ANY` we may cite several expressions as a single tuple:

`relevant([b, c, x, f])`
The process for the programmer

1. Write your contracts as you always did.

2. Run the Frame Condition Checker
   (theory for FCC appears next)
   Frame Condition: \( \text{body}_r \subseteq \triangleright \text{contract}_r \)

3. Fix any violations, i.e. element \( x \) of \( \text{body}_r \) not in \( \triangleright \text{contract}_r \), by either:
   - Fixing \( \text{body}_r \) so that it does not change \( x \)
   - Citing \( x \) in postcondition, e.g. by adding \( \text{relevant}(x) \)

You don’t have to be complete!

Example: \( \text{relevant}(b) \)
Can we prove this?

```
class A feature
  a1: B ; a2: A
  a3
  require
    a1.b1 = Current
    a2 = Current
  do
    a1.b3
  ensure
    a1.b1 = Current
    a2 = Current
end
end

class B feature
  b1: A
  b2, whatever: T
  b3
  do
    b2 := whatever
  end
end
```

Change set includes `a1.b2`
Need to add: `relevant(a1.b2)`

What about information hiding?

It just works. Ignore secret features; they can't hurt you.
(Selective exports are a bit more tricky.)

```
class A feature
  a1: B ; a2: A
  a3
  require
    a1.b1 = Current
    a2 = Current
  do
    a1.b3
  ensure
    a1.b1 = Current
    a2 = Current
end
end

class B feature
  b1: A
  b2, whatever: T
  b3
  do
    b2 := whatever
  end
end
```

Need to prove its own frame conditions!

Now OK as it is
Some invariant properties are listed

```java
class A feature
    a1: B; a2: A
    a3
        require
            a1, b1 = Current
        do
            a2 = Current
            a1, b3
        ensure
            a1, b1 = Current
            a2 = Current
end end
```

```java
class B feature
    b1: A
    b2, whatever: T
    b3
        do
            b2 := whatever
        end
end
```

We would have to prove this, as part of the Frame Condition, if `a1, b1` and `a2` were not listed in the postcondition.

Can also be written

```
a1, b1 = old a1, b1
a2 = old a2
```

So let's do it!

---

Adjustment to frame condition: `old`

The appearance of an expression

`old e`

must not cause the inclusion of `e` in `contract_r`

We simply consider that `e` is not a subexpression of `old e`

Examples:

- `f = old e`
- `f = old e + 1`
- `e = old e`
- `e = old e + 1`
Overview of the requirements task

Verifying the frame condition

Where we are

We have the language design (or non-design) and the rule for the programmers: the frame condition.

We now need to devise the theory that will enable the construction of a Frame Condition Checker.

In the frame condition

\[ \text{body}_r \subseteq \text{>contract}_r \]

>contract$_r$ is easy to determine

What remains is to compute

\[ \text{body}_r \]

for any routine $r$

Minor detail: in O-O, $\text{body}_r$ is often infinite. But wait...
Ensuring the Frame Condition

The practical task is to compute

\[ p^\rightarrow \]

for any construct \( p \)

(including \( \text{body}_r \) for a routine \( r \))

---

Universe of discourse

What are the elements of the sets of interest

\[ p^\rightarrow \]

\[ \triangleright p \]

\(?\)

**Pointer Path Expression**: finite sequence of attribute names, written with dots:

\[ x_1 \cdot x_2 \cdot x_3 \cdot \ldots \cdot x_n \]

Denotes a path starting from current object
Pointer Path Expressions: semantics

- \( x_1 \circ x_2 \circ x_3 \circ \ldots \circ x_n \)

- Path starting from current object
- All the corresponding functions are partial, but we rely on type checking to assume that any expression we use is meaningful
- References can be void, but we don’t mind

The change set can be infinite

Change set: \( \{ \text{first.right, first.right.right, first.right.right.right,...} \} \)

```
class LINKED_LIST[G] feature
    first: LINKABLE[G]
    count: INTEGER
    ...
    set_at(\( x: \text{LINKABLE}[G]; n: \text{INTEGER} \))
    -- Set content of \( n \)-th item to \( x \).
    require
        positive: \( n > 0 \)
        possible: \( n \leq \text{count} \)
    local
        a: \text{LINKABLE}[G]
    do
        from a := first until i = n loop
            a := a.right; i := i + 1
        end
        a.put_right(\( x \))
    end
```
Terminology

If $e = a \cdot b \cdot c \cdot d$, its prefixes (the first three proper prefixes) are the expressions

- $a$
- $a.b$
- $a.b.c$
- $a.b.c.d$

Canonical forms

In the frame condition $body^\rightarrow_r \subseteq >contract^\rightarrow_r$:

- $body^\rightarrow_r$ may be infinite
- If $a$ may change, then $a.x$ may change for any $x$!

**Canonical form** of a change set $body^\rightarrow_r$:

- remove any element $e$ such that a proper prefix of $e$ is also in the set

**Theorem**: the change set has a finite set of prefixes (i.e. the canonical form of a change set is finite)

**Proof**: by König’s lemma
Revised Frame Condition

\[ \text{body}_p \subseteq \text{Prefixes} (\text{contract}_p) \]

Computing change sets \( p' \): easy cases

\[
\begin{align*}
[v := e] & \rightarrow = \{v\} \\
[i1; i2] & \rightarrow = i1 \cup i2 \\
[\text{if } c \text{ then } i1 \text{ else } i2 \text{ end}] & \rightarrow = i1 \cup i2 \\
[\text{from } i \text{ until } c \text{ loop } b \text{ end}] & \rightarrow = i \cup b \\
[\text{create } v] & \rightarrow = \{v\} \\
[\text{require } pre \text{ do } b \text{ ensure } post \text{ end}] & \rightarrow = b \\
\quad r & \rightarrow = \text{decl}_r \\
\end{align*}
\]
\[ p \mapsto: \text{the Qualified Call case} \]

The aliased call conjecture:

\[ [a.r] \mapsto = a.[r \mapsto] \]

Aliases of \( a \):

\( \text{Current} \)

- Aliases of \( a \): \( b, c, d, e \)

\( r \mapsto \): Objects modified by \( r \) when applied to this object

- 4 -

Assessment
Assessment

For programmers, simpler than all the techniques proposed so far - assuming they write contracts already

Relies on extensive static analysis; feasibility remains to be demonstrated

Closely connected to the alias calculus

Usual problem of dealing with advanced language constructs, such as agents and exceptions