Inference of Necessary Field Conditions with Abstract Interpretation

Mehdi Bouaziz[1], Francesco Logozzo[2], Manuel Fähndrich[2]

[1] École normale supérieure, Paris, France
[2] Microsoft Research, Redmond, WA, USA

Tenth Asian Symposium on
Programming Languages and Systems

December 12, 2012 – Kyoto, Japan

Design by Contract

is a programming methodology which systematically requires the programmer to provide contracts (preconditions, postconditions, object invariants) at design time.

Design by Contract

is a programming methodology which systematically requires the programmer to provide contracts (preconditions, postconditions, object invariants) at design time.

- allow automatic generation of documentation,
- amplify the testing process,
- enable assume/guarantee reasoning for static program verification.

Design by Contract: Example

```
public class Person {
  private readonly string Name;
  private readonly JobTitle JobTitle;

  public Person(string name, JobTitle jobTitle)  {
    Contract.Requires(jobTitle != null && name != null);

    this.Name = name;
    this.JobTitle = jobTitle;
  }

  private void ObjectInvariant() {
    Contract.Invariant(this.Name != null);
    Contract.Invariant(this.JobTitle != null);
  }

  public string GetFullName()  {
    if (this.JobTitle != null)
      return string.Format("{0} ({1})", PrettyPrint(this.Name), this.JobTitle.ToString());
    return PrettyPrint(this.Name);
  }

  public string PrettyPrint(string s)  {
    Contract.Requires(s != null);
    Contract.Ensures(Contract.Result<string>() != null);
    // ...
  }
}
```

Design by Contract: Dream and Reality

PL designer dream: the programmer provides sufficient contracts for all the methods and all the classes; a static verifier leverages them to prove the program correctness.

Design by Contract: Dream and Reality

PL designer dream: the programmer provides sufficient contracts for all the methods and all the classes; a static verifier leverages them to prove the program correctness.

Reality:

- the PL or the programming environment does not support contracts: the programmer use non-contract checks on input parameters/fields, unexploitable by a static analyzer,
- the program is only partially annotated,
- the programmer thinks that some contracts are obvious,
- the provided contracts are too weak.

Design by Contract: Dream and Reality

PL designer dream: the programmer provides sufficient contracts for all the methods and all the classes; a static verifier leverages them to prove the program correctness.

Reality:

- the PL or the programming environment does not support contracts: the programmer use non-contract checks on input parameters/fields, unexploitable by a static analyzer,
- the program is only partially annotated,
- the programmer thinks that some contracts are obvious,
- the provided contracts are too weak.

Solution: Inference!

Contract Inference

By abstract interpretation:

- Postconditions
- Preconditions [Cousot Cousot Logozzo 10] [Cousot Cousot Fähndrich Logozzo 13]

Works well!

Contract Inference

By abstract interpretation:

- Postconditions
- Preconditions [Cousot Cousot Logozzo 10] [Cousot Cousot Fähndrich Logozzo 13]

Works well!

- Object invariants
  Class-Level Modular Analysis [Logozzo 03]

Brittle!

Class-Level Modular Analysis

Fixpoint characterization of the invariant:

$$I = \bigsqcup_{\mathsf{c} \in Constrs} \mathsf{s}[\![\mathsf{c}]\!] \sqcup \bigsqcup_{\mathtt{m} \in Methods} \mathsf{s}[\![\mathtt{m}]\!](I)$$

Class-Level Modular Analysis: Example

```
public class Person {
  private readonly string Name;
  private readonly JobTitle JobTitle;

  public Person(string name, JobTitle jobTitle)  {
    Contract.Requires(jobTitle != null && name != null);

    this.Name = name;
    this.JobTitle = jobTitle;
  }

  public string GetFullName()  {
    if (this.JobTitle != null)
      return string.Format("{0}␣({1})", PrettyPrint(this.Name), this.JobTitle.ToString());
    return PrettyPrint(this.Name);
  }

  public int BaseSalary()  {
    return this.JobTitle.BaseSalary;
  }

  public string PrettyPrint(string s)  {
    Contract.Requires(s != null);
    // ...
  }
}
```

$$I_0 = \langle \texttt{Name} \mapsto \mathsf{NN}, \texttt{JobTitle} \mapsto \mathsf{NN} \rangle$$

Class-Level Modular Analysis: Example, constructor added

```
public class Person {
  private readonly string Name;
  private readonly JobTitle JobTitle;

  public Person(string name, JobTitle jobTitle)  {
    Contract.Requires(jobTitle != null && name != null);

    this.Name = name;
    this.JobTitle = jobTitle;
  }

  public Person(string name) {
    Contract.Requires(name != null);

    this.Name = name;
  }

  public string GetFullName()  {
    if (this.JobTitle != null)
      return string.Format("{0} ({1})", PrettyPrint(this.Name), this.JobTitle.ToString());
    return PrettyPrint(this.Name);
  }

  public int BaseSalary()  {
    return this.JobTitle.BaseSalary;
  }
}
```

$$I_1 = \langle \texttt{Name} \mapsto \mathsf{NN}, \texttt{JobTitle} \mapsto \mathsf{T} \rangle$$

Our Solution: Backward Inference of Necessary Conditions

Necessary conditions: properties that should hold on the object fields; if violated, an error will definitely occur.

Our Solution: Backward Inference of Necessary Conditions

Necessary conditions: properties that should hold on the object fields; if violated, an error will definitely occur.

Goal-directed backward interprocedural propagation of potentially failing assertions.

- push assertions that cannot be proven to method entry points (necessary precondition inference [Cousot Cousot Logozzo 10])
- keep those involving private fields
- propagate them to the constructors
- generate an abstract error trace

Backward Inference of Necessary Conditions: Example

```
public class Person {
  private readonly string Name;
  private readonly JobTitle JobTitle;

  public Person(string name, JobTitle jobTitle)  {
    Contract.Requires(jobTitle != null && name != null);

    this.Name = name;
    this.JobTitle = jobTitle;
  }

  public Person(string name) {
    Contract.Requires(name != null);

    this.Name = name;
  }

  public string GetFullName()  {
    if (this.JobTitle != null)
      return string.Format("{0} ({1})", PrettyPrint(this.Name), this.JobTitle.ToString());
    return PrettyPrint(this.Name);
  }

  public int BaseSalary()  {
    return this.JobTitle.BaseSalary;
  }
}
```

$$I_2 = \langle \texttt{Name} \mapsto \mathsf{NN}, \texttt{JobTitle} \mapsto \mathsf{NN} \rangle$$

The algorithm

Result: A necessary condition $\mathcal{I}^*$ on object fields

while *true* **do**
  $\phi \leftarrow \texttt{true}$
  foreach $\texttt{m} \in \texttt{M}$ **do**
    if $\neg cccheck(\texttt{m}, \textbf{out}\ \bar{\texttt{a}})$ **then** // Strengthen precondition and invariant
      $\langle \phi_P, \phi_I \rangle \leftarrow \pi_2(\mathcal{I}(\texttt{m})(\bar{\texttt{a}}))$
      $\texttt{Pre}_\texttt{m} \leftarrow \texttt{Pre}_\texttt{m} \wedge \phi_P$
      $\phi \leftarrow \phi \wedge \phi_I$
    end
  end
  if $\phi = \texttt{true}$ **then**
    break// no change on $\texttt{I}_\texttt{F}$, we are done
  else
    $\texttt{I}_\texttt{F} \leftarrow \texttt{I}_\texttt{F} \wedge \phi$
  end
end
foreach $\texttt{c} \in \texttt{C}$ **do**
  if $\neg cccheck(\texttt{c}, \textbf{out}\ \bar{\texttt{a}})$ **then** // Strengthen the precondition
    $\texttt{Pre}_\texttt{c} \leftarrow \texttt{Pre}_\texttt{c} \wedge \pi_1(\mathcal{I}(\texttt{c})(\bar{\texttt{a}}))$
  end
end

Special Case: Readonly Fields

Restricted to `readonly` fields, the necessary condition inference algorithm gives object invariants after the first iteration of the main loop.

Experiments

We ran `cccheck` on .Net Framework libraries, with:

(BR) object invariant inference disabled;

(NCR) object invariant inference enabled for readonly fields only;

(NC1) object invariant inference enabled for all fields, with the constraint of analyzing every method only once;

(CLMAR) forward class-level modular analysis enabled for readonly fields only.

Results

TIME		(BR)		(NCR)		(NC1)		(CLMAR)	
Library	# Meth.	Checks	Time	Checks	Time	Checks	Time	Checks	Time
mscorlib	22,904	113,551	31:41	113,750	27:36	115,002	32:22	116,116	26:12
Addin	552	4,170	4:15	4,148	4:07	4,295	4:11	4,067	12:55
Composition	1,340	6,228	0:44	6,356	0:46	6,302	0:47	8,095	1:57
Core	5,952	34,324	29:57	36,100	33:50	36,196	34:54	42,602	72:31
Data.Entity	15,239	88,286	23:13	87,743	24:02	91,591	27:59	88,125	43:36
Data.OracleClient	1,961	9,596	2:38	9,738	2:21	9,736	2:26	107,23	3:25
Data.Services	2,448	18,255	6:45	18,518	7:23	18,733	6:54	21,818	24:18
System	15,586	94,038	15:03	93,948	15:15	96,154	15:30	94,008	17:37

PRECISION		(BR)		(NCR)		(NC1)		(CLMAR)	
Library	# Meth.	Checks	Top	Checks	Top	Checks	Top	Checks	Top
mscorlib	22,904	113,551	13,240	113,750	13,084	115,002	11,053	116,116	13,152
Addin	552	4,170	682	4,148	605	4,295	485	4,067	571
Composition	1,340	6,228	909	6,356	791	6,302	743	8,095	885
Core	5,952	34,324	5,323	36,100	4,820	36,196	4,463	42,602	5,715
Data.Entity	15,239	88,286	12,460	87,743	11,719	91,591	15,861	88,125	11,569
Data.OracleClient	1,961	9,596	1,070	9,738	1,025	9,736	887	107,23	1,018
Data.Services	2,448	18,255	3,118	18,518	2,938	18,733	2,749	21,818	2,989
System	15,586	94,038	8,702	93,948	8,644	96,154	10,693	94,008	8,648

Conclusion

- New approach to infer *necessary* field conditions and object invariants
- Eliminates the brittleness of forward object invariant inference caused by changes in the program
- Traces leading to failure give precious hints on finding the origin and explanations of warnings
- Was #1 request of CodeContracts users for readonly fields
- Now in CodeContracts static checker for 1+ year
- Try it yourself: `rise4fun.com/CodeContracts`, `research.microsoft.com/en-us/projects/contracts` (90,000 downloads)