Contracts as a support to static analysis of open systems
Work in progress

Nadia Bel Hadj Aissa    Dorina Ghindici    Gilles Grimaud
Isabelle Simplot-Ryl

INRIA/LIFL/Univ. Lille 1
Introduction

Information flow example

Static Analysis

- Family of techniques used to analyse program behaviors and deduce program properties
- The precision of the result depends on the precision of starting hypotheses. For example:
  - Precision of the abstract domains
  - Restriction of the input domains

```c
void m1 (int a) {
    int v;
    if (a > 100)
        v = m2 (a);
    else
        v = m2(a%2);
    m2(a);
    ...
}
```

```c
int m2 (int a) {
    if (a == 1)
        return 0;
    if (a == 0)
        return 2;
    return m3 (a);
}
```
Open Object-Oriented world

Object-Oriented

▶ Virtual invocations ➔ not possible to decide which code will be executed
▶ Except in particular cases
  ▶ Exact types computation
  ▶ Extra-knowledge: call graph, class hierarchy
    (context-sensitive analysis, complete graph unfolding)

Open

▶ New sub-classes
▶ New calling contexts for old methods ➔ may change the hypotheses under which the analysis has been done

Object-Oriented + Open ➔ Highly dynamic
Proposition

Idea
Compositional analysis of methods based on the notion of contract

Major interests:
- To analyse a method when the called methods are not available
- To use contracts when loading a new method:
  - New code must respect required contracts
  - New code uses contracts of old code
- No need to re-analyse old code in new context
Proposition

Idea
Compositional analysis of methods based on the notion of

\textit{contract}

Major interests:

\begin{itemize}
  \item To analyse a method when the called methods are not available
    \begin{itemize}
      \item \textit{dynamic loading}
    \end{itemize}
  \item To use contracts when loading a new method:
    \begin{itemize}
      \item New code must respect required contracts
      \begin{itemize}
        \item already established properties still hold
      \end{itemize}
      \item New code uses contracts of old code
      \begin{itemize}
        \item No need to re-analyse old code in new context
      \end{itemize}
    \end{itemize}
\end{itemize}
Proposition

Idea
Compositional analysis of methods based on the notion of

*contract*

Major interests:

- To analyse a method when the called methods are not available
  - ➤ dynamic loading
- To use contracts when loading a new method:
  - ➤ New code must respect required contracts
    - ➤ already established properties still hold
  - ➤ New code uses contracts of old code
  - ➤ No need to re-analyse old code in new context
Proposition

Idea
Compositional analysis of methods based on the notion of

\textit{contract}

Major interests:

\begin{itemize}
\item To analyse a method when the called methods are not available
\item To use contracts when loading a new method:
  \begin{itemize}
  \item New code must respect required contracts
    \begin{itemize}
    \item already established properties still hold
  \end{itemize}
  \item New code uses contracts of old code
    \begin{itemize}
    \item No need to re-analyse old code in new context
  \end{itemize}
\end{itemize}
\end{itemize}
Principle

```
void m1{
    ...
    m2();
    ...
    m3();
    ...
}

Contracts
adds new contracts/
updates
requires/generates
checks contracts of
overwritten methods

1
requires/generates

2
checks contracts of
overwritten methods

3
adds new contracts/
updates
```
Information flow

Goal
- To detect "illegal" flows between data
- To prove non-interference

Usual solutions
- Well typed program $\Rightarrow$ secure
- Powerful but problems for open systems, extensible, dynamical updates, multi-applications sharing code, different security policies applied to shared code, ... 

Proposition: Dependency calculus
- Computes the "links" between data accessed by a method
- Results can be exploited \textit{a posteriori}
- Contracts make the analysis compositional
Contracts for dependency calculus

What? The method signature is enriched with dependency informations

Guaranty: The method does not produce more dependencies than announced in its contract if used methods respects their own contracts

How? The method contract is computed (or verified) by abstract interpretation of the method code, contracts of called methods are used in the abstract semantics rule

\[
(V, u_n :: \cdots :: u_0 :: s, DPG) \quad C_m \quad \text{invoke } m
\]

- No need to re-analyze called code
- No need to know the complete class hierarchy
Contracts for dependency calculus

What? The method signature is enriched with dependency informations

Guaranty: The method does not produce more dependencies than announced in its contract if used methods respects their own contracts

How? The method contract is computed (or verified) by abstract interpretation of the method code, contracts of called methods are used in the abstract semantics rule

\[
\frac{(\mathcal{V}, u_n :: \cdots :: u_0 :: s, DPG)}{(\mathcal{V}, ret :: s, DPG \oplus C_m)} \quad \text{invoke } m
\]

- No need to re-analize called code
- No need to know the complete class hierarchy
Contracts management

Inheritance

- Contracts of new methods must be *compatible* with the contracts of overwritten methods and interface contracts (Lattice structure)

- When analyzing a call to a method \( m \) of an object \( o \), the static type of \( o \) can be used to find the contract of \( m \)

Missing contracts in the base

- Given by the user:
  - For native methods ➟ trusted base
  - For conceptions reasons ➟ verified when the code arrives

- Not available
  - Set to the greatest element of the lattice ➟ respected by any forthcoming contract
  - We are not able to infer the missing contracts yet
WCET in a few words

Prediction of the worst case execution time of a program

- Intra-method analysis
  - Estimation of execution time of basic blocks
  - Bound of the number of iterations
  - ...

- Inter-method analysis: end-to-end timing behavior
  - Usually: for each method invocation, the algorithm is called recursively. The WCET calculus is propagated from the leaves of the call graph to the root
  - For polymorphic calls

- RT contracts: for each method $\text{WCET}(m) \leq \text{deadline}(m)$
**WCET in a few words**

```
public void f (P o) {
    ...
    o.m();
    ...
}
```

Class L

```
f()
```

Class P

```
m()
```

Class X

```
m()
```

Class Z

```
m()
```

Class Y

```
m()
```

25 ms

46 ms

210 ms

FLACOS’07
WCET in a few words

Prediction of the worst case execution time of a program
  ▶ Intra-method analysis
    ▶ estimation of single execution time
    ▶ bound of the number of iterations
    ▶ ...
  ▶ Inter-method analysis: end-to-end timing behavior
    ▶ In closed world: for each method invocation, the algorithm is called recursively. The WCET calculus is propagated from the leaves of the call graph to the root
    ▶ For polymorphic calls
      \[
      WCET(C.m) = \max_{c' \subseteq C} W(C'.m)
      \]
  ▶ RT contracts: for each method \( WCET(m) \leq \text{deadline}(m) \)
Example

```c
void A()
{
    if (exp)
        statement;
    else
        B();
}
```

We are able to infer contracts for forthcoming methods!

\[
W(A) = W(if) + W(exp) + \max(W(statement), WCET(B))
\]

\[
\text{deadline}(A) \geq W(A) \geq W(if) + W(exp) + \max(W(statement), WCET(B))
\]

- If \( \text{deadline}(A) < W(if) + W(exp) + W(statement) \) then \( A \) is not valid.
- Otherwise the contract \( Cst \geq WCET(B) \) where \( Cst = \text{deadline}(A) - W(if) - W(exp) \) is added to the contract repository.
**Example**

```c
void A() {
    if (exp)
        statement;
    else
        B();
}
```

We are able to infer contracts for forthcoming methods!

\[
W(A) = W(\text{if}) + W(\text{exp}) + \max(W(\text{statement}), WCET(B))
\]

\[
deadline(A) \geq W(A) \geq W(\text{if}) + W(\text{exp}) + \max(W(\text{statement}), WCET(B))
\]

- If \( \text{deadline}(A) < W(\text{if}) + W(\text{exp}) + W(\text{statement}) \) then \( A \) is not valid
- Otherwise the contract \( Cst \geq WCET(B) \) where \( Cst = \text{deadline}(A) - W(\text{if}) - W(\text{exp}) \) is added to the contract repository
Contract management

Contract repository

- Already computed \( WCET \) of methods
- Deadlines of methods
- Contracts for forthcoming methods

A new method \( C.m \)

- Must verify \( W(C.m) \leq \text{Min}_{C \subseteq C'} WCET(C'.m) \)
- Must verify all pending contracts that imply \( C.m \)
  - No need to solve equation \( \Rightarrow \) only replace the unknown by the value of \( W(C.m) \) and check the result
  - When contracts have several unknowns \( \Rightarrow \) first in is right

Reduction of the repository

- Contracts for the same method can be reduced keeping the \( Min \) of deadlines
Contract lookup for $C.m$

Check contracts containing $\text{WCET}(C.m)$ as unknown
- If a contract is not respected, the method is rejected
- If a contract as no more unknown, it is removed
- Contract with remaining unknown are removed from the contract list of $C.m$

Check contracts of super-classes
1. Let $B$ be the direct super-class of $C$
2. If $B$ does not contain a definition of $m$, goto 1 with $B = C''$ if $C''$ is the direct super-class of $B$
3. If $B$ contains a definition of $m$, then
   - If $W(B.m)$ is known, check that $W(B.m) \geq W(C.m)$ and stop
   - Else check that $W(C.m)$ respects the pending contracts that refer to $\text{WCET}(X.m)$, goto 1 with $B = C''$ if $C''$ is the direct super-class of $B$
Conclusion & Perspectives

Conclusion

► Two applications
► Implemented on small embedded systems (Java for IF and Camille for WCET)

Perspectives

► Formalize a general framework
► Cases when contracts of missing code can be inferred