Impact Analysis based on a Global Hierarchical Object Graph

Marwan Abi-Antoun, Yibin Wang,
Ebrahim Khalaj, Andrew Giang,
Vaclav Rajlich

Department of Computer Science
Wayne State University
Detroit, Michigan, USA

Motivation

- Impact Analysis (IA): compute code dependencies then recommend to developers the code they may need to modify,
- Precision is important: when IA tools recommend too much code elements, developers may explore more irrelevant code
- Many static IA tools use Abstract Syntax Tree (AST) dependencies, e.g.:
  - Dependency Graph in Visual Studio Ultimate
  - JRipples plug-in for Eclipse
- Other static analysis approaches: call graphs, program slicing, static execute-after relationships (Toth et al, PPPJ 2010)
- Dynamic analysis can achieve more precision, but may miss dependencies that arise only in other executions

Key idea

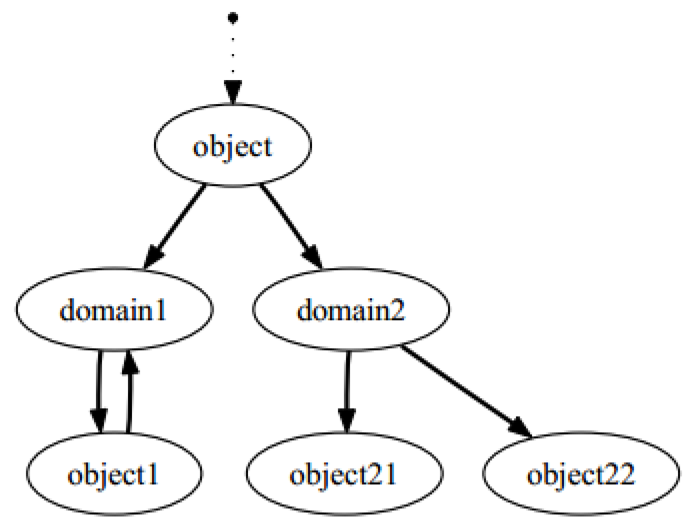
- Within static analysis, explore different part of space:
  - Extract dependencies based on approximating what classes are instantiated at runtime into abstract objects
  - Obtain some precision about shared state
  - Lift that information back to classes
- Underlying analysis: a whole-program static analysis that uses abstract interpretation:
  - extracts a **global points-to graph**
  - enriches graph with usage edges

Contributions

- Definition of ranked dependencies based on abstract interpretation
  - Most important classes related to a class, most important classes behind an interface, etc.
  - Implementation in tool (ArchSummary)
- Evaluation on 2 systems and 5 change tasks
- Comparison between ArchSummary and tool that uses dependencies from AST (JRipples)

Ownership Object Graph (**OOG**)

- A global hierarchical object graph;
- Use **OGraph** as internal representation which has two types of nodes: objects and domains;
- Extracted from code with domain annotations;
- OGraph is input of ArchSummary.

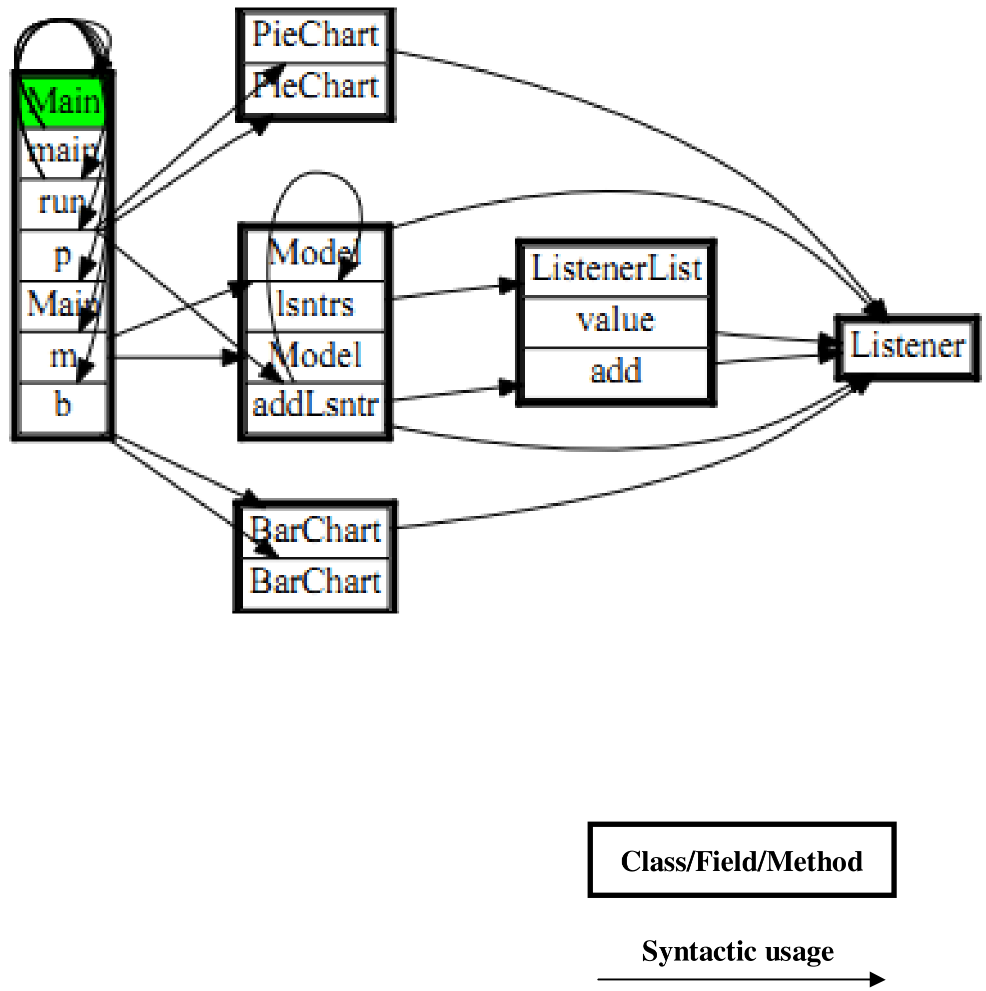


Class and Member Dependency Graph (**CMDG**)

- The dependency graph that JRipples uses;
- CMDG extracts dependencies from the AST;
- Nodes can be types, methods, or fields.

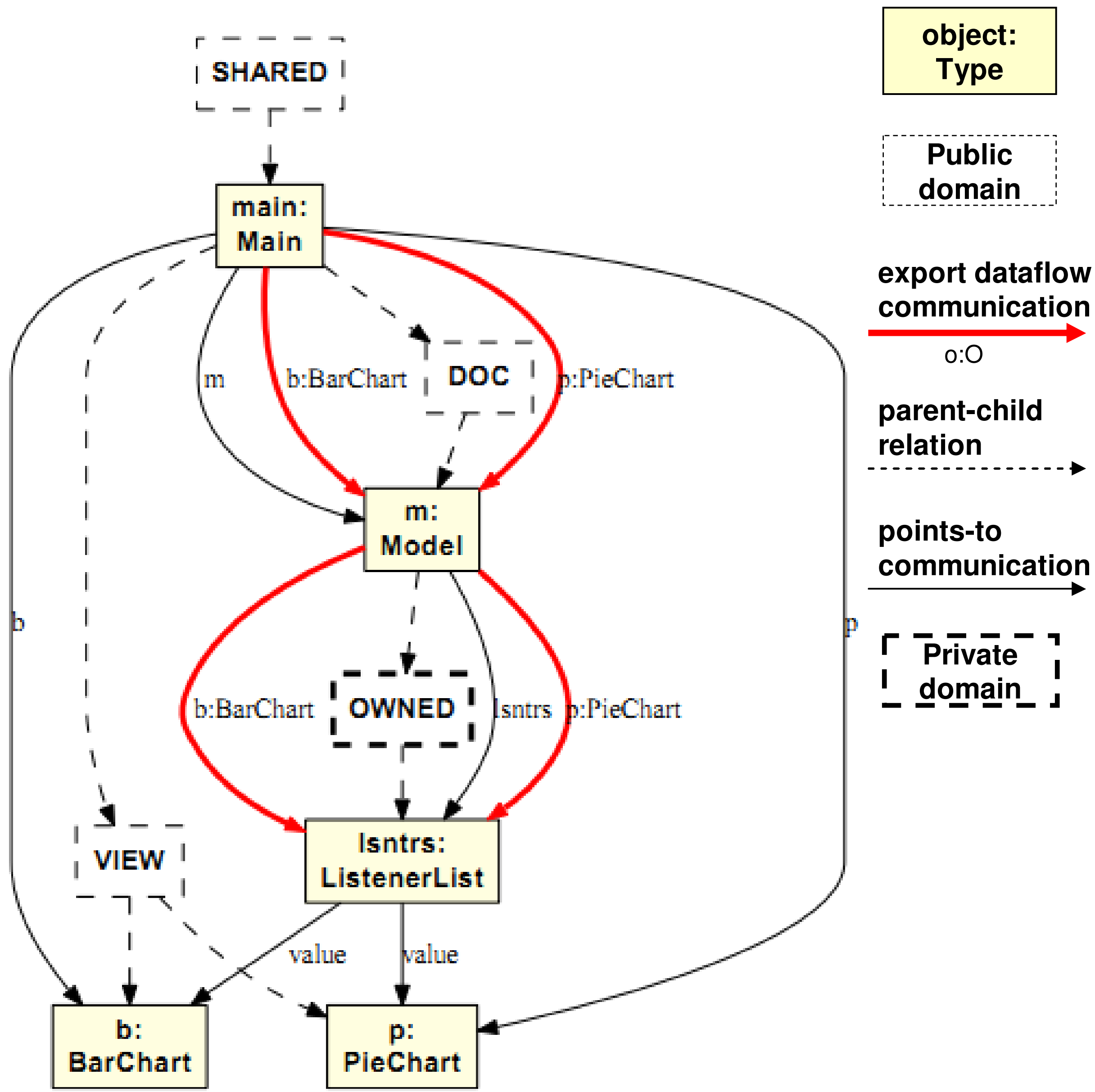
CMDG (in JRipples)

```
interface Listener { }
class ListenerList {
  Listener value;
  void add (Listener el) {…}
}
class BarChart implements Listener { }
class PieChart  implements Listener { }
class Model implements Listener {
  ListenerList lstnrs = new ArrayList<…>();
  void addLsntr(Listener lstnr){
      lsntrs.add(lstnr);  }
}
class Main  {
 Model m = new Model<…> ();
 BarChart b = new BarChart<…> ();
 PieChar p = new PieChart<…> ();
 void run() { m.addLsntr(b); m.addLsntr(p); }
}
```



OGraph

```
interface Listener<owner> { }
class ListenerList<owner, ELTS> {
  Listener<ELTS> value;
  void add (Listener<ELTS> el) {…}
}
class BarChart<owner, M>
implements Listener<owner> { }
class PieChart<owner, M>
implements Listener<owner> { }
class Model<owner, V> implements
Listener<owner> { private domain
OWNED;
  ListenerList< OWNED, V> lstnrs =
new ArrayList<…>();
  void addLsntr(Listener<V> lstnr)
{  lsntrs.add(lstnr);  }
}
class Main  {  domain DOC, VIEW;
 Model< DOC, VIEW> m = new
Model<…> ();
 BarChart< VIEW, DOC> b = new
BarChart<…> ();
 PieChart< VIEW, DOC> p = new
PieChart<…> ();
 void run() { m.addLsntr(b);
m.addLsntr(p); }
}
```

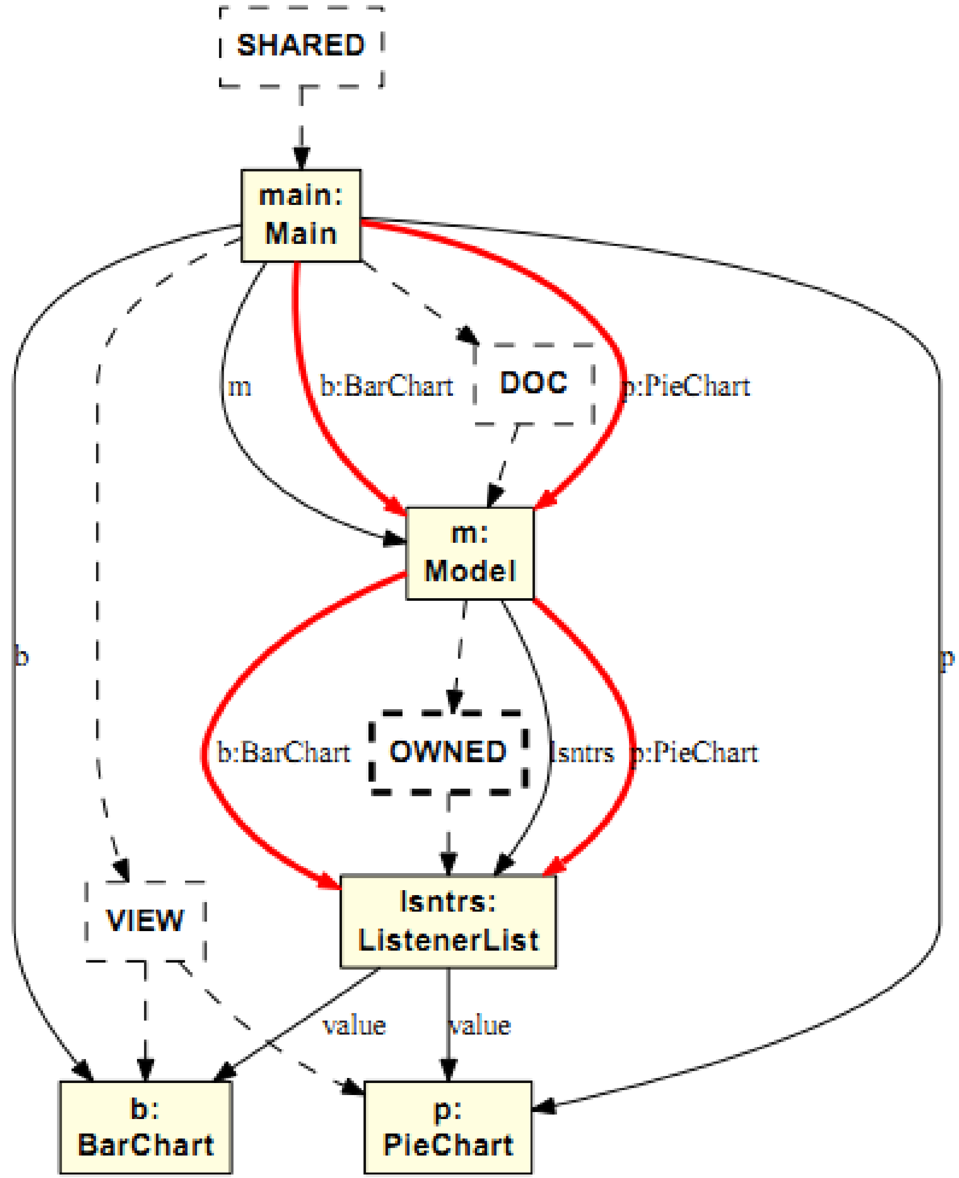


Dependencies from OGraph

- ArchSummary uses OGraph to generate following dependencies:
  - Most Important Classes: MICs
  - Most Important Classes Related to a Class: MIRCs(C)
  - Most Important Classes Behind an Interface: MCBIs(T f)

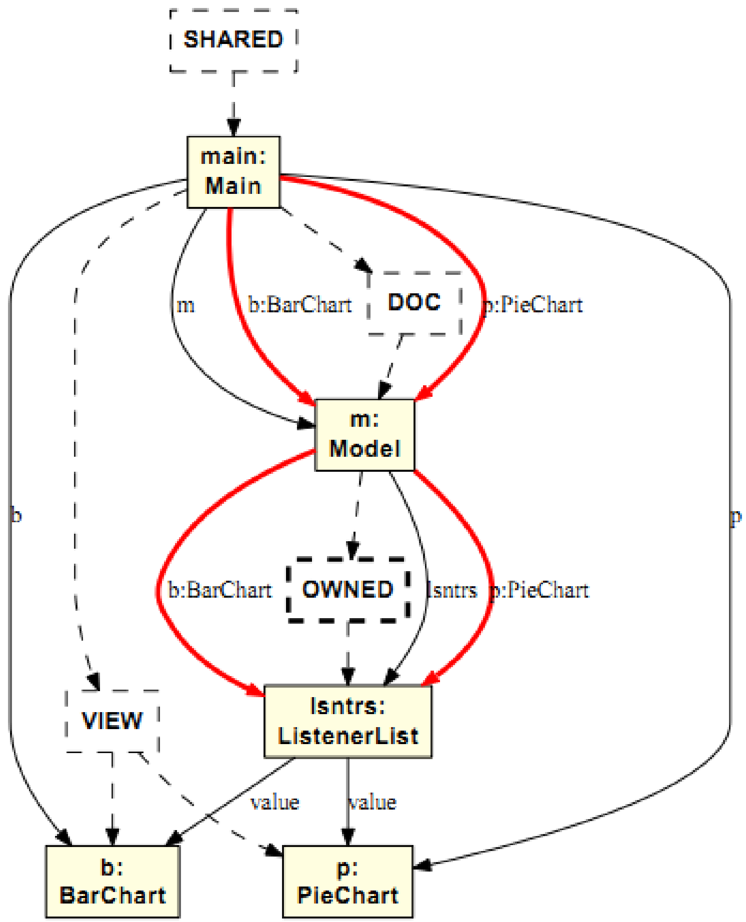
MIRCs(Model)

MIRCs(Model) =
{ Main, ListenerList }



MCBIs(Listener value)

```
interface Listener<owner> { }
class ListenerList<owner, ELTS> {
  Listener<ELTS> value;
  void add (Listener<ELTS> el) {…}
}
class BarChart<owner, M>
implements Listener<owner> { }
class PieChart<owner, M>
implements Listener<owner> { }
class Model<owner, V> implements
Listener<owner> { private domain
OWNED;
  ListenerList< OWNED, V> lstnrs =
new ArrayList<…>();
  void addLsntr(Listener<V> lstnr)
{  lsntrs.add(lstnr);  }
}
class Main  {  domain DOC, VIEW;
  Model< DOC, VIEW> m = new
Model<…> ();
  BarChart< VIEW, DOC> b = new
BarChart<…> ();
  PieChart< VIEW, DOC> p = new
PieChart<…> ();
  void run() { m.addLsntr(b);
m.addLsntr(p); }
}
```



MCBIs(Listener value) =
{ PieChart, BarChart }

Eclipse can show all subtypes of a type, which is:
AllTypes(Listener) =
{ Listener, PieChart, BarChart, Model }

| MCBIs(Listener value) |
  << | AllTypes(Listener) |

Hypothesis

- Dependencies based on abstract interpretation lead to higher effectiveness in impact analysis compared to dependencies based on AST.

Measures

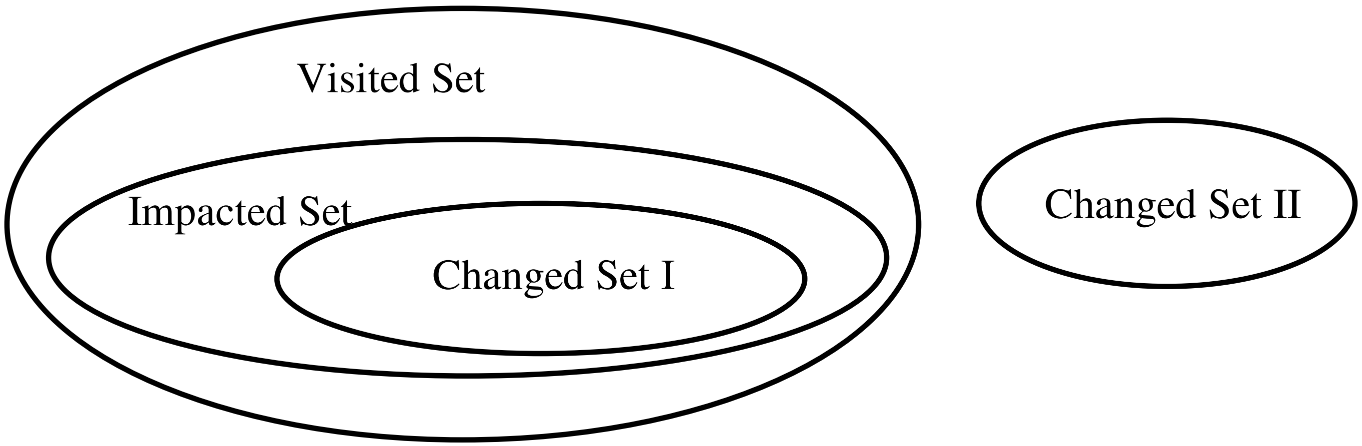
We compare ArchSummary and JRipples for each:

- **Task:** while completing a task, from beginning to end:
  - Distinct Recommended Types (DRT)
  - Number of Visited Types (NVT)
  - Effectiveness
  - MCBIs vs. All_Types
  - MCBIs_Invoked vs. Interfaces_Visited
- **Step:** during a task, each time the tool recommends types to the developer
  - Recommended Types per Step (RTS)

Measures

- **Distinct Recommended Types (DRT)**
  - All types recommended by the tool for a single task
- **Recommended Types per Step (RTS)**
  - Each time the tool recommends types to the developer, it is a step
- **Number of Visited Types (NVT)**
  - All visited types for a single task
- **Effectiveness**
  - Compare the number of really impacted types and NVT to verify the hypothesis for a single task

Measures: Number of Visited Types (NVT) & Effectiveness



$$NVT = |\text{Visited Set}|$$

$$\{\text{Changed Set I}\} = \{\text{Impacted Set}\} \cap \{\text{Changed Set}\}$$

$$\text{Effectiveness} = \frac{|\text{Changed Set I}|}{\text{NVT}} * 100\%$$

Case Study: DrawLets

- Subject System: DrawLets
  - 138 types (115 classes and 23 interfaces), 12 packages
  - 8,800 LOC.
- Task

T5. Implement an “owner” for each figure: An owner is a user who puts that figure onto the canvas, and only the owner is allowed to move and modify it.

Comparative Results – Drawlets

	ArchSummary		JRipples	
T5	DRT	53	DRT	100
	RTS Avg	23	RTS Avg	17
	RTS Max	46	RTS Max	58
	NVT	37	NVT	97
	MCBIs Avg	2.9	AllTypes Avg	6.5
	MCBIs Max	12	AllTypes Max	19
	Effectiveness	35%	Effectiveness	17%
	MCBIs_Invoked	8	Interfaces_Visited	20

Discussion

- Effects of navigation
  – In JRipples: determined by developer’s marks
  – In ArchSummary: determined by MICs, MIRCs, MCBIs that the developer query
- Annotation Overhead
  – Manually adding annotations: 1 hour/KLOC
  – Semi-automated tools

Related Work

- Static analysis [Ren et al., OOPSLA, 2004]
- Dynamic analysis [Law et al., ICSE, 2003]
- Textual information [Poshyvanyk et al., ESE, 2009]
- Mining software repositories [Gethers et al., ICSE, 2012]
- Using OOGs during software evolution [Abi-Antoun and Ammar, WCRE, 2012]

Conclusion

- Following dependencies based on abstract interpretation leads to more effective impact analysis compared to dependencies based on only AST.
- In the future, we plan to explore additional strategies to mine and rank dependencies.