

Slicing Techniques and Program Calculi in OSS Certification

N. Rodrigues, L. Barbosa, J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

Opencert'07 Workshop
ETAPS'07, Braga , 31 March 2007

The Minho vision

Opencert = Open source software certification

- OSS: source code is available
- Access to implementors design decisions
- How to *understand* them?

Need for

- formal modelling
- formal reverse engineering
- model understanding

Previous projects

National funded projects:

- KARMA — Data cleaning / quality via data reverse engineering (with formal methods)
- PURe — Program Understanding by Reverse Engineering (via reverse program calculation)

Experience

- Access to large (encrypted) data sets
- No access to code (industrial partners not prepared to share code)

OSS regarded as an opportunity to proceed further

Program Understanding

Forward software engineering:

$$Spec \vdash \dots \vdash \llbracket Imp \rrbracket$$

Reverse software engineering:

$$\llbracket Imp \rrbracket \dashv \dots \dashv Spec$$

where $\llbracket \dots \rrbracket$ means *the semantics of...*

Program Understanding

Forward software engineering:

$$Spec \vdash \dots \vdash \llbracket Imp \rrbracket$$

Reverse software engineering:

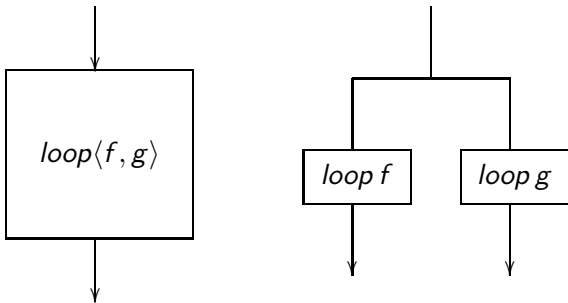
$$\llbracket Imp \rrbracket \dashv \dots \dashv Spec$$

where $\llbracket \dots \rrbracket$ means *the semantics of...*

Fighting “fusion”

Problem

- we are given Imp , not $\llbracket Imp \rrbracket$
- $\llbracket Imp \rrbracket$ can be too complex
- Typically, programmers *fuse code* too early



“Fission for Program Understanding”

- Some kind of preliminary source code analysis should precede formal reverse engineering, leading to code *fission*

$$\llbracket Imp \rrbracket = f(\llbracket Imp_1 \rrbracket, \dots, \llbracket Imp_n \rrbracket)$$

where every Imp_i is a part of Imp (thus smaller).

Need for

- static analysis
- syntactic (or “quasi-semantic”) analysis

Names

Syntactic *ingredient* in every piece of source code:

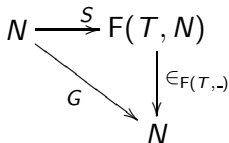
- source code is made of program units
- units are to be referred to
- units need to be named

Programs are, therefore, (typed) collections of (mutually dependent) *name spaces*:

$$\begin{array}{c} \vdots \\ N_i \rightarrow F_i(T_i, N_1, \dots, N_i, \dots, N_n) \\ \vdots \end{array}$$

Membership and slicing

Names can be *membershyped*:



G — a name-to-name relation, or *dependence graph* — is made available for *code slicing*.

Before I hand over to Nuno: names are everywhere, cf.

- memory addresses (heaps, etc)
- primary keys (referential integrity, etc)
- function, procedure identifiers
- mutable variables
- ...