DATA DEPENDENCIES AND PROGRAM SLICING:
FROM SYNTAX TO ABSTRACT SEMANTICS

Isabella Mastroeni and Damiano Zanardini

PEPM 2008
SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.
SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.

DEPENDENCY: ...defines what *relevant* means.
Slicing vs Dependencies

**Slicing:** ...extracts from programs the statements which are *relevant* for a given behaviour.

**Dependency:** ...defines what *relevant* means.

**Syntactic def-ref:**

\[
\begin{align*}
    & x := y + 2z \\
    & x \text{ depends on } y \text{ and on } z
\end{align*}
\]
SlicIng VS DEPENDENCIES

SlicIng: ...extracts from programs the statements which are relevant for a given behaviour.

DePendencY: ...defines what relevant means.

Syntactic def-ref:

\[
\begin{align*}
  x &:= y + 2z \\
  x &\text{ depends on } y \text{ and on } z \\
  x &:= z + y - y \\
  x &\text{ depends on } y \text{ and on } z
\end{align*}
\]
SLICING: ...extracts from programs the statements which are relevant for a given behaviour.

DEPENDENCY: ...defines what relevant means.

SEMANTIC: \[
\begin{align*}
x := z + y - y \\
x \text{ depends on } z \text{ but it does NOT depend on } y
\end{align*}
\]
SLICING: ...extracts from programs the statements which are relevant for a given behaviour.

DEPENDENCY: ...defines what relevant means.

SEMANTIC:

\[
\begin{align*}
x &:= z + y - y \\
x \text{ depends on } z \text{ but it does NOT depend on } y
\end{align*}
\]

\[
\begin{align*}
x &:= 2y \\
x \text{ depends on } y
\end{align*}
\]
SlicIng VS Dependencies

SliCing: ...extracts from programs the statements which are *relevant* for a given behaviour.

Dependency: ...defines what *relevant* means.

Abstract Semantic (Parity):

\[
\begin{align*}
  x &:= 2y \\
  x &\text{ does NOT depend on } y
\end{align*}
\]
Slicering vs Dependencies

**Slicering:** ...extracts from programs the statements which are *relevant* for a given behaviour.

**Dependency:** ...defines what *relevant* means.

**Abstract Semantic (Parity):**

\[
\begin{align*}
  x &:= 2y \\
  x &\text{ does NOT depend on } y \\
  x &:= 2y + z \\
  x &\text{ depends on } z
\end{align*}
\]
Slicing by means of a calculus for independencies
[Amtoft & Banerjee ’07];
- Syntactic dependencies
- Forward slicing
Slicing by means of a calculus for independencies
[Amtoft & Banerjee ’07];
  ✓ Syntactic dependencies
  ✓ Forward slicing

Abstract dependencies [Rival ’05];
  ✓ Mathematical, set theoretic definition of dependencies;
  ✓ Applied to Alarm diagnosis;
Slicing by means of a calculus for independencies [Amtoft & Banerjee ’07];
- Syntactic dependencies
- Forward slicing

Abstract dependencies [Rival ’05];
- Mathematical, set theoretic definition of dependencies;
- Applied to Alarm diagnosis;

Abstract Slicing [Hong et al. ’05]
- Only for predicate abstractions;
- Considers a subset of possible executions
Consider the complete lattice $< C, \leq, \land, \lor, \bot, \top >$, $A_i \in \text{uco}(C)$

Lattice of Abstract Domains $\equiv$ Lattice $\text{uco}$

$A \equiv \rho(C)$

$< \text{uco}(C), \sqsubseteq, \sqcap, \sqcup, \lambda x. \top, \lambda x. x >$
Consider the complete lattice \(< C, \leq, \wedge, \vee, \bot, \top >\), \(A_i \in \text{uco}(C)\)

Lattice of Abstract Domains \(\equiv\) Lattice uco

\[ A \equiv \rho(C) \]

\[ \langle \text{uco}(C), \subseteq, \cap, \cup, \lambda x. \top, \lambda x. x > \]

\[ A_1 \subseteq A_2 \iff A_2 \subseteq A_1 \]
Consider the complete lattice \(< C, \leq, \land, \lor, \bot, \top >\), \(A_i \in \text{uco}(C)\)

Lattice of Abstract Domains \(\equiv\) Lattice \(\text{uco}\)

\[ A \equiv \rho(C) \]

\[ < \text{uco}(C), \subseteq, \cap, \cup, \lambda x. \top, \lambda x. x > \]

\[ A_1 \subseteq A_2 \iff A_2 \subseteq A_1 \]

\[ \bigcap_i A_i = \mathcal{M}(\bigcup_i A_i) \]
Consider the complete lattice \( < C, \leq, \land, \lor, \bot, \top > \), \( A_i \in \text{UCO}(C) \)

**Lattice of Abstract Domains**: \( \equiv \) Lattice \( \text{UCO} \)

\[
A \equiv \rho(C)
\]

\( < \text{UCO}(C), \subseteq, \cap, \cup, \lambda x. \top, \lambda x. x > \)

\[
A_1 \subseteq A_2 \iff A_2 \subseteq A_1
\]

\[
\bigcap_i A_i = \mathcal{M}(\bigcup_i A_i)
\]

\[
\bigcup_i A_i = \bigcap_i A_i
\]
Consider the complete lattice \(< C, \leq, \wedge, \vee, \bot, T >\), \(A_i \in \text{latt}(C)\)

\[
\text{Lattice of Abstract Domains} \equiv \text{Lattice latt}(\text{UCO}) \equiv \rho(C) \\
\langle \text{UCO}(C), \subseteq, \cap, \cup, \lambda x. T, \lambda x. x \rangle
\]

\(A_1 \subseteq A_2 \iff A_2 \subseteq A_1\) \\
\(\bigcap_i A_i = \mathcal{M}(\bigcup_i A_i)\) \\
\(\bigcup_i A_i = \bigcap_i A_i\)
Consider the complete lattice \(< C, \leq, \land, \lor, \bot, \top >\), \(A_i \in \text{uco}(C)\)

Lattice of Abstract Domains \(\equiv\) Lattice \(\text{uco}\)

\[ A \equiv \rho(C) \]
\[ < \text{uco}(C), \subseteq, \cap, \cup, \lambda x. \top, \lambda x. x > \]

\(A_1 \subseteq A_2 \iff A_2 \subseteq A_1\)

\(\bigcap_i A_i = \mathcal{M}(\bigcup_i A_i)\)

\(\bigcup_i A_i = \bigcap_i A_i\)
Program Dependency Graphs (PDG) are a standard way for modelling dependencies for slicing. They are defined by two kinds of edges \((s_1, s_2)\):

**Control Flow Edge:** \(s_1\) represents a control predicate and \(s_2\) represents a component of the program immediately nested within the predicate \(s_1\);

**Flow Dependence Edge:** \(s_1\) defines a variable \(x\) which is used in \(s_2\) i.e., \(x \in \text{def}(s_1) \cap \text{ref}(s_2)\), and \(x\) is not further defined between \(s_1\) and \(s_2\);
Program Dependency Graphs (PDG) are a standard way for modelling dependencies for slicing. They are defined by two kind of edges \((s_1, s_2)\):

**Control Flow Edge:** \(s_1\) represents a control predicate and \(s_2\) represents a component of the program immediately nested within the predicate \(s_1\);

**Flow Dependence Edge:** \(s_1\) defines a variable \(x\) which is used in \(s_2\), i.e., \(x \in \text{def}(s_1) \cap \text{ref}(s_2)\), and \(x\) is not further defined between \(s_1\) and \(s_2\);

\[
\downarrow
\]

*Flow dependence edges = Direct flows = Def-Ref dependencies*

*Control flow edges = Indirect flows*
SLICING VS DEPENDENCIES

SLICING ⇒ Requires the same I/O behaviour, i.e., no \textit{semantic} dependencies

PDG ⇒ Models \textit{syntactic} dependencies
SLICING VS DEPENDENCIES

SLICING ⇒ Requires the same I/O behaviour, i.e., no semantic dependencies

PDG ⇒ Models syntactic dependencies

↓

There is a clear gap: Semantics vs Syntax
SLICING VS DEPENDENCIES

**SLICING** ⇒ Requires the same I/O behaviour, i.e., no *semantic* dependencies

**PDG** ⇒ Models *syntactic* dependencies

↓

**THERE IS A CLEAR GAP: SEMANTICS VS SYNTAX**

**PDG** ⇒ Generate a slicing considering more dependencies (syntactic)

(SEMANTIC) **SLICING** ⇐ Needs a *weaker* notion of dependence.

**SLICING PARAMENTRIC ON THE CHOSEN DEPENDENCE NOTION!**
**A logic for (in)Dependencies**

Formalization of notion of (in)dependence \([x \propto y] [\text{Amtoft & Banerjee '04}]:

\[
G \vdash \{T_0^\# \} \ x := e \ \{T^\#\}
\]

if \(\forall [y \propto w] \in T^\# \). \((x \neq y \Rightarrow [y \propto w] \in T_0^\#)\)

\((x = y \Rightarrow (w \notin G \land \forall z \in \text{FV}(e). [z \propto w] \in T_0^\#))\)

\[
G_0 \vdash \{T_0^\#\} s_1 \{T^\#\} \quad G_0 \vdash \{T_0^\#\} s_2 \{T^\#\}
\]

\[
G \vdash \{T_0^\#\} \text{if } e \text{ then } s_1 \text{ else } s_2 \{T^\#\}
\]

if \(G \subseteq G_0 \land (w \notin G_0 \Rightarrow \forall x \in \text{FV}(e). [x \propto w] \in T_0^\#)\)

\[
G_0 \vdash \{T^\#\} s \{T^\#\}
\]

\[
G \vdash \{T^\#\} \text{while } e \text{ do } s \{T^\#\}
\]

if \(G \subseteq G_0 \land (w \notin G_0 \Rightarrow \forall x \in \text{FV}(e). [x \propto w] \in T^\#)\)
A logic for (in)Dependencies

Formalization of notion of (in)dependence \([x \times y] [\text{Amtoft & Banerjee '04}]:\)

\[
G \vdash \{T_0^\#\} x := e \{T^\#\}
\]

if \(\forall [y \times w] \in T^\#. (x \neq y \Rightarrow [y \times w] \in T_0^\#)\)

\((x = y \Rightarrow (w \notin G \land \forall z \leadsto e. \quad [z \times w] \in T_0^\#))\)

\[
G_0 \vdash \{T_0^\#\} s_1 \{T^\#\} \quad G_0 \vdash \{T_0^\#\} s_2 \{T^\#\}
\]

\[
G \vdash \{T_0^\#\} \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \{T^\#\}
\]

if \(G \subseteq G_0 \land (w \notin G_0 \Rightarrow \forall x \leadsto e. \quad [x \times w] \in T_0^\#)\)

\[
G_0 \vdash \{T^\#\} s \{T^\#\}
\]

\[
G \vdash \{T^\#\} \textbf{while } e \textbf{ do } s \{T^\#\}
\]

if \(G \subseteq G_0 \land (w \notin G_0 \Rightarrow \forall x \leadsto e. \quad [x \times w] \in T^\#)\)
**Semantic (Abstract) Dependencies**

\[ z := w + y + 2x^2 - w \]

**Syntactic dep.** A variable is a free variable in the expression assigned to \( z \)?

\[ \implies z \text{ depends on } \{w, y, x\}! \]
**Semantic (Abstract) Dependencies**

$$z := w + y + 2x^2 - w$$

**Semantic dep.** By varying the *value* of a variable does the expression change?

$$\varNA e \iff \exists \sigma_1, \sigma_2. \forall y \neq x. \sigma_1(y) = \sigma_2(y) \land [e](\sigma_1) = [e](\sigma_2)$$

$$\Rightarrow z \text{ depends on } \{y, x\}!$$
**Semantic (Abstract) Dependencies**

\[ z := w + y + 2x^2 - w \]

**Abstract Semantic dep.** By varying the *property* of a variable does the *property* of the expression change?

\[ x \not\sim_N e \iff \exists \sigma_1, \sigma_2. \forall y \neq x. \rho(\sigma_1(y)) = \rho(\sigma_2(y)) \land \rho([e](\sigma_1)) = \rho([e](\sigma_2)) \]

⇒ If we consider *Parity* \( z \) depends on \( \{y\} \)!
\textbf{SEMANTIC (ABSTRACT) DEPENDENCIES}

\[ z := w + y + 2x^2 - w \]

\textbf{Abstract Semantic dep.} By varying the \textit{property} of a variable does the \textit{property} of the expression change?

\[ x \leadsto e \iff \exists \sigma_1, \sigma_2. \forall y \neq x. \rho(\sigma_1(y)) = \rho(\sigma_2(y)) \land \rho([e](\sigma_1)) = \rho([e](\sigma_2)) \]

\[ \Rightarrow \text{If we consider Sign } z \text{ depends on } \{y, x\}! \]
We have two kinds of dependencies:

- Data dependencies (Assignments);
- Control dependencies (Control structures)
PRUNING DEPENDENCIES

We have two kinds of dependencies:

- Data dependencies (Assignments) ⇒ Direct flows;
- Control dependencies (Control structures) ⇒ Indirect flows
PRUNING DEPENDENCIES

We have two kinds of dependencies:

- Data dependencies (Assignments) ⇒ Direct flows;
- Control dependencies (Control structures) ⇒ Indirect flows

We propose a PRUNING of data dependencies!

⇒

STILL WE LOSE SOMETHING ABOUT CONTROL DEPENDENCIES!
PRUNING DEPENDENCIES

We have two kind of dependencies:

- Data dependencies (Assignments) ⇒ Direct flows;

- Control dependencies (Control structures) ⇒ Indirect flows

\[
\text{if } (y + 2x \text{ mod } 2) == 0 \text{ then } w := 0 \text{ else } w := 0
\]

⇒ The guard **DOES NOT DEPEND** on \(x\): **OK**

⇒ The variable \(w\) **DOES NOT DEPEND** on \(y\): **No!**
The definition of narrow deps contains quantifiers on variables and states. This means that, even abstracting states, the number of comparisons between $\rho \langle [e] \sigma_1 \rangle$ and $\rho \langle [e] \sigma_2 \rangle$ may be huge or infinite if the domain is non-trivial.

Yet, we observe that:

- Some states are not possible at a given program point.
- What is computed in a broader state can be valid in narrower states (monotonicity).

\[
\rho([e][\top]) \leq U \iff \rho([e][\text{even}]) \leq U \land \rho([e][\text{odd}]) \leq U
\]
DERIVING ABSTRACT DEPENDENCIES (2)

A systematic way to go through the (variables and states)-space:

- incrementally find the set $X$ of variables which are enough to determine the value of $e$
- $X$ determines $e$ if any change to other variables can be ignored (needs to go into the state space)
A systematic way to go through the (variables and states)-space:

- incrementally find the set $X$ of variables which are enough to determine the value of $e$
- $X$ determines $e$ if any change to other variables can be ignored (needs to go into the state space)
A systematic way to go through the (variables and states)-space:

- incrementally find the set $X$ of variables which are enough to determine the value of $e$
- $X$ determines $e$ if any change to other variables can be ignored (needs to go into the state space)
Deriving abstract dependencies (2)

A systematic way to go through the (variables and states)-space:

- Incrementally find the set \( X \) of variables which are enough to determine the value of \( e \).
- \( X \) determines \( e \) if any change to other variables can be ignored (needs to go into the state space).
DERIVING ABSTRACT DEPENDENCIES (2)

A systematic way to go through the (variables and states)-space:

✓ incrementally find the set $X$ of variables which are enough to determine the value of $e$
✓ $X$ determines $e$ if any change to other variables can be ignored (needs to go into the state space)
Deriving abstract dependencies (2)

A systematic way to go through the (variables and states)-space:

- incrementally find the set $\mathbf{X}$ of variables which are enough to determine the value of $\epsilon$
- $\mathbf{X}$ determines $\epsilon$ if any change to other variables can be ignored (needs to go into the state space)
Another application: simplifying a domain in order to remove dependencies on a set of variables

Basically, systematically removing from $\rho$ the abstract values which are responsible for the distinguishability of two states.
Abstract Non-Interference

[Giacobazzi & Mastroeni '04]
Abstract Non-Interference

[Giacobazzi & Mastroeni ’04]

Diagram:
- Secret H
- Public L
- SW
- Observer: ρ
- External observer
Abstract Non-Interference

[Giacobazzi & Mastroeni ’04]
Certifying programs for ANI

We certify the security degree of programs relatively to an output observation [Giacobazzi & Mastroeni].

We can derive the certification inductively on the syntax of programs [Giacobazzi & Mastroeni ’04].

**Problem:** This system of rules has a *semantic rule!*
CERTIFYING PROGRAMS FOR ANI

We certify the security degree of programs relatively to an output observation [Giacobazzi & Mastroeni].

We can derive the certification inductively on the syntax of programs [Giacobazzi & Mastroeni ’04].

**Problem:** This system of rules has a semantic rule!

We can avoid the semantic rule by computing **abstract dependencies** for assignment!
CONCLUSIONS

- We provide an insight on the strong relation between slicing and dependency;

- A new point of view: Slicing parametric on a notion of dependency;

- Still we are not able to get the most precise semantic slicing;

- Still there is a lot of work to do towards a real implementation.