Research Note
RN/13/01

First International Workshop on Interference and Dependence

21 January 2013

Editor: Kelly Androutsopoulos

Abstract

This technical report consists of a collection of extended abstracts that have been accepted and have been presented at the First International Workshop on Interference and Dependence.

Keynote Talks

Information Erasure: Varieties of Policies and Enforcement Mechanisms

David Sands
Chalmers University of Technology,
Sweden

Sometimes information is made available just for a specific purpose. Once it has fulfilled its purpose there are many good reasons why it should be erased and forgotten. In this talk I will explore the semantics of information erasure, the variety of policies relating to erasure, and static and dynamic mechanisms to enforce such policies. (This talk covers material from joint works with P. Del Tedesco, S. Hunt, and A. Russo.)

Formal verification of program slicing for formally verified loop bound analyser.

David Pichardie
INRIA Rennes
France

Nowadays safety critical systems are validated through long and costly test campaigns. Static analysis is a promising complementary technique that allows to automatically provide guarantees on these critical systems. Significant examples are the state-of-the-art ASTREE static analyzer for C which has proven some critical safety properties for the primary flight control software of the Airbus A340 fly-by-wire system.

Taking note of such a success, the next question is: why do we trust these analyzer that are themselves complex pieces of software ? An analyzer itself can be certified by testing, but exhaustivity cannot be achieved. In this work, we show how mechanized proofs can be used to certify static analyzers or their results.

Worst-case execution time (WCET) estimation tools are other examples of static analysis industrial successes. These complex pieces of software performing tasks such as computation on control flow graphs (CFG) and bound calculation. In this talk, we present a formal verification (in Coq) of a loop bound estimation. It relies on program slicing, followed by a value analysis and a bound calculation. The work has been integrated into the CompCert verified C compiler. Our verified analyses directly operate on non-structured CFG. We extend the CompCert RTL intermediate language with a notion of loop scope (a.k.a. weak topological ordering on CFG) that is useful for intensive reasoning on CFG. The automatic extraction of our loop bound estimation into OCaml yields a program with competitive results, obtained from experiments on a reference benchmark for WCET bound estimation tools.

Chasing infections by unveiling program dependencies

Mila Dalla Preda[1] and Isabella Mastroeni[2]

[1] Dipartimento di Informatica - Univ. di Bologna, Italy
dallapre@cs.unibo.it
[2] Dipartimento di Informatica - Univ. di Verona, Italy
isabella.mastroeni@univr.it

Abstract. Metamorphic malware continuously modify their code, while preserving their functionality, in order to foil misuse detection. The key for defeating metamorphism relies in a semantic characterization of the embedding of the malware into the target program. Indeed, a behavioral model of program infection that does not relay on syntactic program features should be able to defeat metamorphism. Moreover, a general model of infection should be able to express dependences and interactions between the malicious code and the target program. ANI is a general theory for the analysis of dependences of data in a program. We propose an high order theory for ANI, later called HOANI, that allows to study program dependencies. Our idea is then to formalize and study the malware detection problem in terms of HOANI.

1 Introduction

One of the major challenge in computer security is the detection and neutralization of metamorphic malware. A metamorphic malware is a malicious program equipped with a metamorphic engine that takes the malware, or parts of it, as input and morphs it at run-time to a syntactically different but semantically equivalent variant, in order to foil traditional misuse malware detectors. Misuse detectors are syntactic in nature as they identify malware infection by comparing the byte sequence comprising the body of the malware against a signature database [13]. It is exactly this syntactic characterization of the malicious code that makes standard misuse malware detectors so vulnerable to metamorphism. Thus, in order to handle metamorphism a malware detector should be able to recognize the metamorphic variants of a malware, namely the possible evolutions of the malicious code. The metamorphic engine is typically implemented as a set of code obfuscations that preserve program semantics to some extent. Thus, in order to handle metamorphism a malware detector should characterize infection in terms of semantic properties rather than syntactic properties (like signatures). For this reason researchers have started to investigate formal approaches to malware detection where infection is specified in terms of behavioral properties of programs (e.g., [1, 2, 6, 9]). As usual, the efficiency of these approaches is stated in terms of soundness (no false positives) and completeness (no false negatives) properties. In [6] the authors present a general framework based on program semantics and abstract interpretation for proving soundness and completeness of malware detectors in the presence of obfuscations. This semantic model for malware detection implicitly assumes that a malware appends its code and behavior to the one of the target program (the victim) without interacting with it. Hence, this formal model of malware infection is not appropriate for the description and identification of malware whose behaviors interferes with the one of the target program (either with spurious or real dependences added to obstruct program analysis).

In order to develop a more general theory that is able to describe the interactions between the malware and the target program we need a formal framework that is able to describe dependences between fragments of the same program. It is well known that non-interference [12] (NI) provides an ideal theory for reasoning on data dependencies in a program and that abstract non-interference consists in a generalization of the theory weakening the dependency analysis between data [7]. Our idea is to lift the ANI framework on programs and to define a sort of high-order ANI (HOANI) that characterizes dependences and relations among functions, and therefore programs, instead of data. The idea is that we detect infection when the semantics of a program matches the overall semantics of a target program corrupted by a malware. Indeed, if the malware detector could observe differences it would say that the specific malware has not infected the program, since we cannot recognize its semantics in the semantics of the program. This definition of malware detection allows to use HOANI for characterizing both soundness and completeness of the malware detectors, but allows even something more. We can inherits the attacker model and maximal information release characterizations of ANI, which lifted high order and instantiated to the malware detection field seem to provide a way to certify which classes of metamorphic engines do not deceive the detector, and to make a training of the detector depending on the metamorphic engine we aim to unveil. Finally, we prove that HOANI is a generalization of the semantic approach cited above [6] to malware detection since under specific conditions the two approaches collapse.

2 Background

Mathematical notation. If S and T are sets, then $\wp(S)$ denotes the powerset of S and $S \times T$ denotes the Cartesian product of S and T. If $f : S \longrightarrow T$, $Y \subseteq S$, and $X \subseteq T$ then $f(Y) \stackrel{\text{def}}{=} \{f(y) \mid y \in Y\}$ and $f^{-1}(X) \stackrel{\text{def}}{=} \left\{ x \,\middle|\, f(x) \in X \right\}$. We will often denote $f(\{x\})$ as $f(x)$ and use lambda notation for functions. $f{\circ}g \stackrel{\text{def}}{=} \lambda x.\ f(g(x))$. $\langle C, \leq \rangle$ denotes a poset C with ordering relation $\leq$, while $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice C, with ordering $\leq$, *lub* $\vee$, *glb* $\wedge$, top and bottom element $\top$ and $\bot$ respectively. *id* $\stackrel{\text{def}}{=} \lambda x.\ x$ and $\mathbb{T} \stackrel{\text{def}}{=} \lambda x.\ \top$. The point-wise ordered set of monotone functions, denoted $C_1 \xrightarrow{\text{m}} C_2$, is a complete lattice $\langle C_1 \xrightarrow{\text{m}} C_2, \sqsubseteq, \sqcup, \sqcap, \mathbb{T}, \lambda x. \bot \rangle$. $f : C_1 \longrightarrow C_2$ is (completely) additive if f preserves *lub*'s of all subsets of C_1 (emptyset included). Continuity, denoted $\xrightarrow{\text{c}}$, holds when f preserved *lubs*'s of chains. Co-addittivity and co-continuity are dually defined.

Abstract interpretation. We use the framework of abstract interpretation [3, 4] for modeling both the observational capability of malware detector and the invariant properties of metamorphic engines. Abstract interpretation is used for reasoning on *properties* rather than on (concrete) data values. Abstract interpretation is a general theory for deriving sound approximations of the semantics of discrete dynamic systems, e.g., programming languages [3]. Approximation can be equivalently formulated either in terms of Galois connections or closure operators [4]. An *upper closure operator* (uco for short) $\rho : C \to C$ on a poset C, representing concrete objects, is monotone, idempotent, and extensive: $\forall x \in C.\ x \leq_C \rho(x)$. The upper closure operator is the function that maps the concrete values to their abstract properties, namely with the best possible approximation of the concrete value in the abstract domain. Formally, closure operators ρ are uniquely determined by the set of their fix-points $\rho(C)$, for instance $\mathit{Par} = \{\mathbb{Z}, \mathtt{ev}, \mathtt{od}, \varnothing\}$. For upper closures, $X \subseteq C$ is the set of fix-points of $\rho \in \mathit{uco}(C)$ iff X is a *Moore-family* of C, i.e., $X = \mathcal{M}(X) \overset{\text{def}}{=} \{\wedge S \mid S \subseteq X\}$ — where $\wedge\varnothing = \top \in \mathcal{M}(X)$. The set of all upper closure operators on C, denoted $\mathit{uco}(C)$, is isomorphic to the so called *lattice of abstract interpretations of* C [4]. If C is a complete lattice then $\mathit{uco}(C)$ ordered point-wise is also a complete lattice, $\langle \mathit{uco}(C), \sqsubseteq, \sqcup, \sqcap, \top, \mathit{id}\rangle$ where for every $\rho, \eta \in \mathit{uco}(C)$, $\{\rho_i\}_{i\in I} \subseteq \mathit{uco}(C)$ and $x \in C$: $\rho \sqsubseteq \eta$ iff $\forall y \in C.\ \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$; $(\sqcap_{i\in I}\rho_i)(x) = \wedge_{i\in I}\rho_i(x)$; and $(\sqcup_{i\in I}\rho_i)(x) = x \ \Leftrightarrow\ \forall i \in I.\ \rho_i(x) = x$. Given an abstraction, we want also to understand whether the program computes *accurately* on the property. In general, the abstract interpretation framework guarantees that the abstract computation is *sound*, namely we can only lose information by computing on abstract properties. On the other hand, the accuracy of the computation is modeled in terms of *completeness*: an abstract domain is complete for a program if the computation of the program, on the abstract properties, corresponds precisely to the abstraction of the concrete computation. In other words, the abstract domain is as precise as possible with respect to the program to compute. The *best correct approximation* of f is $f^{bca} \overset{\text{def}}{=} \alpha\circ f\circ\gamma$ (or equivalently $\gamma\circ\alpha\circ f\circ\gamma\circ\alpha$). It is known that $f^\sharp$ is sound iff $f^{bca} \sqsubseteq f^\sharp$ and this implies that $\alpha(\mathit{lfp}(f)) \leq \mathit{lfp}(f^{bca}) \leq \mathit{lfp}(f^\sharp)$ [4]. In the following, if $[\![P]\!]$ is specified as fix-point of (a combination of) predicate-transformers $F_P : C \xrightarrow{c} C$, and $\rho \in \mathit{uco}(C)$, we denote by $[\![P]\!]^\rho$ the (fix-point) semantics associated with $F_P^{bca} = \rho\circ F_P\circ\rho$. $[\![P]\!]^\rho$ is the best correct abstract interpretation of P in ρ. In this case $\rho([\![P]\!]) \leq [\![P]\!]^\rho$.

Abstract non-interference. Abstract non-interference (ANI) [7] is a natural weakening of non-interference by abstract interpretation. Suppose the variables of program split in private ($\mathtt{H}$) and public ($\mathtt{L}$). Let $\eta, \rho \in \mathit{uco}(\mathbb{V}^{\mathtt{L}})$ and $\phi \in \mathit{uco}(\mathbb{V}^{\mathtt{H}})$, where $\mathbb{V}^{\mathtt{L}}$ and $\mathbb{V}^{\mathtt{H}}$ are the domains of $\mathtt{L}$ and $\mathtt{H}$ variables. η and ρ characterise the *attacker*. Let $\phi \in \mathit{uco}(\mathbb{V}^{\mathtt{H}})$, which states what, of the private data, can flow to the output observation, the so called *declassification* of ϕ [10]. A program P satisfies ANI, and we write $[\eta]P(\phi \Rightarrow \rho)$, if $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l_1, l_2 \in \mathbb{V}^{\mathtt{L}}$:

$$\eta(l_1) = \eta(l_2)\ \wedge\ \phi(h_1) = \phi(h_2)\ \Rightarrow\ \rho([\![P]\!](h_1, l_1)^{\mathtt{L}}) = \rho([\![P]\!](h_2, l_2)^{\mathtt{L}}). \quad (1)$$

This notion says that, whenever the attacker is able to observe the input property η and the output property ρ, then it can observe nothing more than the property ϕ of private input. It is possible to systematically characterize the most concrete output observation for a program, given the input one [7]. The idea is that of abstracting in the same object all the elements that, if distinguished, would generate a visible flow. On the other hand, we can characterize the maximal information that a program semantics allows to flow, namely which is the most abstract property that needs to be declassified in order to guarantee the non-interference of the program [7].

3 The Ingredients

Separability and Program Integration. Let us recall the notions of interleave and separability introduced in [11]. Consider two disjoint sets of variables $X = \{x_1 \dots x_n\}$ and $Y = \{y_1 \dots y_n\}$. We use notation $\bar{x}$ to refer to the tuple $\langle x_1 \dots x_n \rangle$, notation $\mathbf{x}_i$ to refer to the value stored in variable x_i, and notation $\bar{\mathbf{x}}$ to refer to the tuple of values $\langle \mathbf{x}_1 \dots \mathbf{x}_n \rangle$. We define the set of possible states over X and Y as follows:

$$\Sigma_{X:Y} = \left\{ \langle \mathbf{x}_1 \dots \mathbf{x}_n : \mathbf{y}_1 \dots \mathbf{y}_n \rangle \,\middle|\, X = \{x_1 \dots x_n\}, Y = \{y_1 \dots y_n\} \right\}$$

When $Y = \varnothing$ we refer to the set of states over X simply as Σ_X. Every trace $\sigma \in \Sigma^*_{X:Y}$ is of the form $\sigma = \langle \bar{\mathbf{x}}_1 : \bar{\mathbf{y}}_1 \rangle \langle \bar{\mathbf{x}}_2 : \bar{\mathbf{y}}_2 \rangle \dots$ with $\langle \bar{\mathbf{x}}_i, \bar{\mathbf{y}}_i \rangle \in \Sigma_{X:Y}$ for every i. Let ϵ denote the empty trace. We define the projection function $\pi_X : \Sigma_{X:Y} \rightarrow \Sigma_X$ as $\pi_X(\epsilon) = \epsilon$, $\pi_X(\langle \bar{\mathbf{x}}_1 : \bar{\mathbf{y}}_1 \rangle \sigma) = \bar{\mathbf{x}}_1 \pi_X(\sigma)$, and similarly the projection function $\pi_Y : \Sigma_{X:Y} \rightarrow \Sigma_Y$. According to [11] we define function $interleave : \Sigma^*_{X:Y} \times \Sigma^*_{X:Y} \rightarrow \Sigma^*_{X:Y}$ such that $interleave(\sigma_1, \sigma_2) = \sigma$ iff $\pi_X(\sigma) = \pi_X(\sigma_1)$ and $\pi_Y(\sigma) = \pi_Y(\sigma_2)$. A set of traces $\Gamma \in \Sigma_{X:Y}$ satisfies *separability* iff it is closed under interleave, namely if $\forall \sigma_1, \sigma_2 \in \Gamma$ then $interleave(\sigma_1, \sigma_2) \in \Gamma$.

We model program integration as a function $\mathfrak{I} : \mathbb{P} \times \mathbb{P} \longrightarrow \mathbb{P}$ that given two programs combines them into one. Let $\mathit{Var}(P)$ denote the variables of program P. We interpret the notions of interleaving and separability in the context of program integration.

Definition 1 *An integration function $\mathfrak{I} : \mathbb{P} \times \mathbb{P} \longrightarrow \mathbb{P}$ satisfies separability if for every pair of programs Q and T with disjoint variables, i.e., $\mathit{Var}(Q) \cap \mathit{Var}(T) = \varnothing$, the set of traces $[\![\mathfrak{I}(Q,T)]\!] \in \wp(\Sigma^*_{\mathit{Var}(Q):\mathit{Var}(T)})$ is closed for interleave.*

This means that, when the integration function satisfies separability, the behaviors of programs Q and T are kept separate and independent in the behavior of the integrated program $\mathfrak{I}(Q,T)$. In other words an integration functions satisfies separability when it does not add dependences between the programs it composes. Indeed, when we have separability we believe that it is reasonable to assume that the behavior of $\mathfrak{I}(Q,T)$ restricted to Q coincides exactly with the behaviour of Q, namely that if $\mathfrak{I}$ satisfies separability then $\forall Q, T \in \mathbb{P} :\ \pi_{\mathit{Var}(Q)}([\![\mathfrak{I}(Q,T)]\!]) = [\![Q]\!]$.

The Malware Detection Problem. A malware detector can be modeled as a function $D : \mathbb{P} \times \mathbb{P} \to \{true, false\}$ that decides whether a program is infected with a malware or a metamorphic variant of it. Given $M, P \in \mathbb{P}$ we denote with $M \hookrightarrow P$ the fact that program P is infected with malware M. An ideal metamorphic malware detector should be both *sound* (never erroneously claim that a program is infected) and *complete* (detect all metamorphic variants), namely it should satisfy the following:

$$D(M,P) = true \;\;\Leftrightarrow\;\; \exists M' \text{ metamorphic variant of } M : \; M' \hookrightarrow P$$

The weaker notions of relative soundness/completeness are used to certify soundness and completeness of a given malware detector wrt a class of obfuscations [6] .

Definition 2 *Let $\mathbb{O}$ be a set of obfuscations. A malware detector D is sound for $\mathbb{O}$ if $D(P,M) = true \;\Rightarrow\; \exists \mathcal{O} \in \mathbb{O} : \mathcal{O}(M) \hookrightarrow P$. A malware detector D is complete for $\mathbb{O}$ if $\forall\, \mathcal{O} \in \mathbb{O} : \; \mathcal{O}(M) \hookrightarrow P \;\Rightarrow\; D(P,M) = true$.*

In the following we formalize the notion of infection in terms of program integration: $M \hookrightarrow P$ iff $\exists\, T.\; [\![P]\!] = [\![\mathfrak{I}(M,T)]\!]$. Hence, the integration function $\mathfrak{I}$ models infection (we may have different infection functions). For instance, if the malware is a standard file infector appending its code to a target file, then the integration is simply the concatenation of the codes involved and it would be modeled by an integration function that satisfies separability.

Higher-order Abstract Noninterference. In order to model non-interference in *code transformations* such as code obfuscation and metamorphism, we consider an higher-order version of ANI, where the objects of observations are programs instead of values. Hence, we have a part of a program (semantics) that can change and that is not observable, and the environment which remains the same up to an observable property. The function analyzed by HOANI is a program integration function, which takes the two parts of the program and provides a program as result. The output observation is the best correct approximation of the resulting program. Consider programs $P \in \mathbb{P}$ and the corresponding semantics, i.e., $[\![P]\!]$. Hence, we define

$$\eta([\![P_1]\!]) = \eta([\![P_2]\!]) \wedge \; \phi([\![Q_1]\!]) = \phi([\![Q_2]\!]) \;\Rightarrow \rho([\![\mathfrak{I}(Q_1,P_1)]\!]) = \rho([\![\mathfrak{I}(Q_2,P_2)]\!]) \qquad (2)$$

Note that, the abstractions can be any abstract property on programs. In the following, we consider HOANI for a particular family of abstractions, and in particular for semantics' *bca*. In other words, if we have $\rho \in uco(\wp(\Sigma))$, then we consider $\rho^\rho \in uco(\wp(\Sigma) \xrightarrow{\text{m}} \wp(\Sigma))$ such that $\rho^\rho \overset{\text{def}}{=} \lambda f.\; \rho f \rho$ [5]. By noting that, $\rho^\rho([\![P]\!]) = [\![P]\!]^\rho$ (defined in Sect. 2), we can rewrite Eq. 2 in the following HOANI notion:

$$[\![P_1]\!]^\eta = [\![P_2]\!]^\eta \wedge \; [\![Q_1]\!]^\phi = [\![Q_2]\!]^\phi \implies [\![\mathfrak{I}(Q_1,P_1)]\!]^\rho = [\![\mathfrak{I}(Q_2,P_2)]\!]^\rho \qquad (3)$$

Example 1. Consider the following program fragments, where $10! = 3628800$:

$$Q_1 : \left[\begin{array}{l} \mathit{prod} = 1; x = 1; \\ \textbf{while}\ x < 11\ \{ \\ \qquad \mathit{prod} = \mathit{prod} \cdot x; \\ \qquad x = x + 1\}; \end{array}\right. \qquad Q_2 : \left[\begin{array}{l} \mathit{prod} = 10!; x = 11; \\ \textbf{while}\ x > 1\ \{ \\ \qquad x = x - 1; \\ \qquad \mathit{prod} = \mathit{prod}/x\}; \end{array}\right.$$

$$P_1 : \left[\begin{array}{l} \mathit{sum} = 0; x = 1; \\ \textbf{while}\ x < 11\ \{ \\ \qquad \mathit{sum} = \mathit{sum} + x; \\ \qquad x = x + 1\}; \end{array}\right. \qquad P_2 : \left[\begin{array}{l} \mathit{sum} = 55; x = 11; \\ \textbf{while}\ x > 1\ \{ \\ \qquad x = x - 1; \\ \qquad \mathit{sum} = \mathit{sum} - x\}; \end{array}\right.$$

Consider, as $\mathfrak{I}$ ($\mathcal{T} = [\![\mathfrak{I}]\!]$) the integrating algorithm proposed by [8], providing the following resulting programs:

$$\mathfrak{I}(Q_1, P_1) : \left[\begin{array}{l} \mathit{prod} = 1; \mathit{sum} = 0; \\ x = 1; \\ \textbf{while}\ x < 11\ \{ \\ \qquad \mathit{prod} = \mathit{prod} \cdot x; \\ \qquad \mathit{sum} = \mathit{sum} + x; \\ \qquad x = x + 1\}; \end{array}\right. \qquad \mathfrak{I}(Q_2, P_2) : \left[\begin{array}{l} \mathit{prod} = 10!; \mathit{sum} = 55; \\ x = 11; \\ \textbf{while}\ x > 1\ \{ \\ \qquad x = x - 1; \\ \qquad \mathit{prod} = \mathit{prod}/x; \\ \qquad \mathit{sum} = \mathit{sum} - x\}; \end{array}\right.$$

Consider the abstract domain $\iota \in \mathit{uco}(\wp([-\mathfrak{m}, \mathfrak{m}]))$ of limited intervals, where $\mathfrak{m} \in \mathbb{Z}$ is the maximal integer. In this case $\iota(x) = [\min(x), \max(x)]$. Interval analysis is defined in [3], with standard *bca* abstract interpretations for arithmetic operations on intervals: $\odot$, $\oplus$, $\ominus$. Then we have that

$$[\![Q_1]\!]^\iota = [\![Q_2]\!]^\iota \ \wedge\ [\![P_1]\!]^\iota = [\![P_2]\!]^\iota \implies [\![\mathfrak{I}(Q_1, P_1)]\!]^\iota = [\![\mathfrak{I}(Q_2, P_2)]\!]^\iota$$

This HOANI property of the considered integration algorithm tells us that we can vary the involved programs leaving unchanged the variables' intervals without inducing any variation in the interval analysis of the resulting program.

4 Malware detection by unveiling program dependencies

4.1 Abstract noninterference-based malware detector

In this section, we define a notion of malware detector inspired by higher order abstract noninterference, let us call it ANIMD. The idea is that a program P is infected with a possibly metamorphic variant of malware M if it is (semantically) equivalent, at least up to an observation (program analysis), to the integration of a code segment T with the code of the malware M. Formally, given $\rho \in \mathit{uco}(\wp(\Sigma^*_{\mathsf{Var}(P)}))$:

$$\text{ANIMD}_\rho(M, P) = \mathit{true} \quad \Leftrightarrow \quad \exists T \in \mathbb{P} : [\![\mathfrak{I}(M, T)]\!]^\rho = [\![P]\!]^\rho$$

Namely a program P is infected with a malware M if it behaves wrt ρ like a target program T infected with malware M.

Given a metamorphic engine ME we assume that it is possible to identify a semantic property ϕ that is preserved by any code transformation generated by ME, while each transformation changing ϕ cannot be generated by ME. This means that ME can be modeled as a semantic property $\phi \in uco(\wp(\Sigma^*_{\mathsf{Var}(M)}))$ and that the set of all the obfuscating transformations generated by ME can be formalized as follows:

$$\mathbb{O}_\phi = \left\{ \mathcal{O} \,\middle|\, \forall P, Q \in \mathbb{P}.\, [\![P]\!]^\phi = [\![Q]\!]^\phi \;\Leftrightarrow\; P = \mathcal{O}(Q) \right\}.$$

This are exactly all and only the transformations used by the malware equipped with ME as stealthing technique. We can either assume to know this property, or given a set of metamorphic malware variants we can derive it and then use it to model the metamorphic engine (possibly catching also unseen variants). First of all, let us rewrite HOANI in the context of malware detector: by changing the version of the malware, up to an observable property ϕ, the malware detector analysing ρ is not deceived by the differences between the versions and recognize the same infection in both the analyzed programs. Hence, we define HOANI^ϕ_ρ:

$$[\![M]\!]^\phi = [\![M']\!]^\phi \implies [\![\mathfrak{I}(M,P)]\!]^\rho = [\![\mathfrak{I}(M',P)]\!]^\rho \tag{4}$$

At this point we study the precision of the malware detectors based on HOANI in terms of soundness and completeness.

Theorem 2 (Soundness). *Let* $\mathbb{O}_\phi = \left\{ \mathcal{O} \,\middle|\, \forall P, Q \in \mathbb{P}.\, [\![P]\!]^\phi = [\![Q]\!]^\phi \;\Leftrightarrow\; P = \mathcal{O}(Q) \right\}$, *then* ANIMD_ρ *is sound for* $\mathbb{O}_\phi$ *whenever:*

$$[\![M]\!]^\phi = [\![M']\!]^\phi \impliedby [\![\mathfrak{I}(M,P)]\!]^\rho = [\![\mathfrak{I}(M',P)]\!]^\rho \tag{5}$$

Theorem 3 (Completeness). *If* HOANI^ϕ_ρ *holds, then* ANIMD_ρ *is complete for* $\mathbb{O}_\phi$.

4.2 Certifying and Training Malware Detectors.

In this section we discuss how we can exploit the HOANI framework in order to understand how we can *certify* the "power" of a malware detector in terms of the classes of metamorphic engines unable to deceive it, and how we can do a *training* of malware detectors starting from a class of obfuscation techniques characterizing a metamorphic engine that we aim to defeat. In this way we could formally understand the relation between the metamorphic invariant property and the analysis performed by the detector.
The ANI framework allows to describe two transformations, one characterizing the most concrete output observation unable to observe interferences, and the other characterizing the maximal property that do not cause interference [7]. We believe that these two transformations, once lifted high order, would provide exactly a way for certifying and training malware detectors. The main difference between ANI and HOANI is that while abstracting data means to consider properties of data, i.e., sets of values; abstracting programs means to consider the best correct approximation of their semantics,

i.e., the abstraction of a function is a more abstract function instead of a set of functions. This difference makes not so immediate to extend ANI from data to functions and requires a deeper analysis of a higher-order notion of abstract non-interference. Note that, because the domain transformers introduced for ANI [7] extended to the definition above of HOANI would generate sets of programs and therefore of semantics (i.e., functions), which in general represent program/semantics properties, we can build a correspondence between semantic properties, i.e., sets of semantic functions, and best correct approximations. In other words, we can always associate a best correct approximation with any set of functions, while we can construct a set of functions corresponding to any given best correct approximation of a given function.
Certification: Given ρ in HOANI we can characterize the maximal amount of information released ϕ. This property ϕ is non-redundant, i.e., any change of ϕ do cause interference, and it is such that when it is invariant then there is no interference in the observation ρ. Hence, if we start from a malware detector ANIMD_ρ, we can characterize the most concrete property ϕ such that ANIMD_ρ is sound and complete for $\mathbb{O}_\phi$. This means that ANIMD_ρ is sound and complete for any metamorphic engine whose code transformations preserves at least ϕ
Training: Given a property ϕ the HOANI framework allows to characterize the most concrete observation unable to observe interferences when the property ϕ is unchanged. In other words, if we start from a set of obfuscations $\mathbb{O}$, whose semantic invariant is the property ϕ then we can characterize the most concrete ρ such that the corresponding ANIMD_ρ is complete for $\mathbb{O}$. Namely, we can modify the observation capability of the malware detector depending in the class of obfuscation we aim to defeat.

4.3 What's new in ANIMD?

In this section we compare the prosed ANIMD with the closest framework of semantic-based malware detectors based on abstract interpretation [6]. The two approaches are clearly related since both model the malware detector as parametric on the program analysis that it is able to perform. Moreover, in both the approaches the malware code has in some way to be separated by the original program and both the approaches characterize classes of obfuscation techniques, those used by a metamorphic engine, in terms of the invariant property left unchanged by the transformations. This means that we can quite easily compare these two approaches. In particular, we show that ANIMD generalize all these aspects by considering the best correct approximation of the program semantics instead of the output abstraction, and by considering a generic integration function instead of the trivial composition of programs. Hence, let us first recall the basic definitions of the first semantic malware detector [6].

Semantic Malware Detector. The idea of [6] is to classify a program P as infected by a possibly metamorphic variant of malware M if there exists a portion of P whose abstract behavior corresponds to the abstract behavior of M. This implicitly assumes that infection does not add dependences between the malware and the target program, namely that the integration function that models infection satisfies separability. Given $\rho \in uco(\wp(\Sigma_{\mathit{Var}(M)}))$, we can rewrite the semantic malware detector of [6] as:

$$\mathrm{SMD}_\rho(M, P) = \mathit{true} \quad \Leftrightarrow \; \exists Q, T \in \mathbb{P} : [\![P]\!] = [\![\Im(Q, T)]\!] \wedge \rho([\![M]\!]) = \rho([\![Q]\!])$$

SMD **vs** ANIMD**.** Observe that SMD_ρ decides infection by comparing the abstraction of the concrete semantics of programs, i.e., $\rho([\![M]\!]) = \rho([\![Q]\!])$, while ANIMD_ρ decides infection by comparing the abstract semantics of programs, i.e., $[\![\Im(M, T)]\!]^\rho = [\![P]\!]^\rho$. The abstraction of the concrete semantics and the abstract computation of the semantics collapse when the abstract domain ρ is complete for the computation of program semantics as shown by the following result.

Lemma 4. *If f is complete for ρ, i.e., $\rho \circ f = \rho \circ f \circ \rho$ then we can apply the fix point Kleene transfer, namely $\rho\, \mathit{lfp} f = \mathit{lfp}\ \rho \circ f \circ \rho$.*

Thus, in order to compare SMD_ρ and ANIMD_ρ we have to assume the completeness of the domain ρ for the semantic computation, i.e., $\forall P \in \mathbb{P} : \rho([\![P]\!]) = \rho(\mathit{lfp} F_P) = \mathit{lfp} \rho \circ F_P \circ \rho = [\![P]\!]^\rho$.

Another difference between SMD and ANIMD is given by the computational domain that they consider: SMD observes properties of the behaviour of the malware, while ANIMD properties of the behaviour of the whole infected program. Thus, in order to understand their relation we define the following domain extension: Given $\rho \in uco(\wp(\Sigma^*_{\mathit{Var}(M)}))$ we denote $\widetilde{\rho} \in uco(\wp(\Sigma^*_{\mathit{Var}(M)})) \times uco(\wp(\Sigma^*_{\mathit{Var}(T)}))$ any abstraction that is ρ on $\mathit{Var}(M)$, i.e., $\widetilde{\rho} = \rho \times \eta$ where $\eta \in uco(\wp(\Sigma^*_{\mathit{Var}(T)}))$.

Theorem 5. *Consider an integration function $\Im$ that satisfies separability, two abstract domains ρ and $\widetilde{\rho}$ that are complete for the computation of program semantics and assume that Equation* 4 *and Equation* 5 *hold, then* $\mathrm{SMD}_\rho(M, P) \Leftrightarrow \mathrm{ANIMD}_{\widetilde{\rho}}(M, P)$.

5 Conclusions and future works

In this work we have begun to investigate the possibility of exploiting the ANI theory for detecting malware infection. To this end we have started to reason on an high order version of the standard ANI framework that allows to reason on dependences and interferences among programs (instead of data). We have formalized the malware detection problem in terms of HOANI and we have proved that the malware detector ANIMD based on HOANI generalizes the semantic malware detector SMD proposed in [6]. An interesting feature of ANIMD is that it is parametric on the infection strategy used by the malware and that it can model possible interactions between the malware and the target program. Another reason that makes our approach promising is the possibility to develop systematic techniques for certifying and training malware detectors. This can be done by lifting high order the ANI transformations that characterize respectively the most concrete output observation unable to detect interferences, and the maximal property that do not cause interference. Indeed, the ability of certifying the precision of a given malware detector, and the possibility of deriving the best malware detector wrt a metamorphic engine are two important challenges in the battle against metamorphic malware. To this end we have to deeply understand and develop the HOANI theory beyond ANIMD.

Based on these results, our goal is to develop a systematic strategy for the design of the best malware detector for a given class of metamorphic code variants. To this end we first need to develop a technique for learning the metamorphic engine ME that has generated the considered malware variants. Next we have to characterize the invariant property ϕ of all the generated variants in order to derive the observation property ρ that characterizes detection for ANIMD_ρ. We believe that this theoretical identification of the program property ρ that the malware detector should consider in order to handle metamorphism for ME can given useful insight in the design and implementation of a malware detector tool able to defeat ME.

References

1. P. Beaucamps, I. Gnaedig, and J. Y. Marion. Behavior abstraction in malware analysis. In *Runtime Verification (RV'10)*, volume 6418 of *LNCS*, pages 168–182, 2010.
2. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, Oakland, CA, USA, 2005.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252, New York, 1977. ACM Press.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282, New York, 1979. ACM Press.
5. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages) (Invited Paper). In *Proc. of the 1994 IEEE Internat. Conf. on Computer Languages* (*ICCL '94*), pages 95–112, Los Alamitos, Calif., 1994. IEEE Comp. Soc. Press.
6. M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5):1–54, 2008.
7. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '04)*, pages 186–197, New York, 2004. ACM-Press.
8. S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345 – 387, 1989.

9. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05)*, volume 3548 of *LNCS*, pages 174–187, 2005.
10. I. Mastroeni. On the rôle of abstract non-interference in language-based security. In K. Yi, editor, *Third Asian Symp. on Programming Languages and Systems (APLAS '05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 418–433, Berlin, 2005. Springer-Verlag.
11. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium in Security and Privacy*, pages 79–93, Los Alamitos, Calif., 1994. IEEE Comp. Soc. Press.
12. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1):5–19, 2003.
13. P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, Boston, MA, USA, 2005.

Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks

Jürgen Graf[1], Martin Hecker[1], Martin Mohr[1], and Benedikt Nordhoff[2]

[1] Karlsruhe Institute of Technology {graf|martin.hecker|martin.mohr}@kit.edu
[2] University of Münster b.n@wwu.de

Abstract. We combine the static analysis techniques of Program Dependence Graphs (PDG) and Dynamic Pushdown Networks (DPN) to improve the precision of interference analysis for multithreaded Java programs. PDGs soundly approximate possible dependence between program points in sequential programs through data and control dependence edges. In a concurrent setting a third category of so-called interference edges captures the potential interferences between memory accesses in different threads. DPNs model concurrent programs with recursive procedures, dynamic thread creation and nested locking. We use a lock-sensitive analysis based on DPNs to remove spurious interference edges, and apply the results to information flow control.

1 Information Flow Control for Multithreaded Java Programs

Information flow control (IFC) analyses check whether information about a programs (secret) input can possibly flow to public output, e.g. if a secret value is printed to console. A program is called *non-interferent*, iff it does not leak secret information. Non-interference of a given program can be verified with a sound static analysis that detects possible information flow through dependencies and interference between program statements using PDGs [1]. If the analysis detects no illegal information flow, it is guaranteed that during execution of the program, an attacker observing the public output will learn nothing about the secret input. However, depending on the precision of the analysis, there may be false alarms, because the analysis reports spurious information flow that is impossible during an actual execution of the program. Thus a main goal of static non-interference analyses is to minimize the number of false alarms, by improving analysis precision.

In the following example we show which false alarms may arise when a multithreaded Java program is analyzed for non-interference. The program in figure 1 does not leak information about the secret value `secret` to a public visible output `println` and should be considered non-interferent: It contains two threads, the main thread t_0 and an instance of `MyThread` t_1. Output to a public visible channel only occurs through the two `println` statements in t_0. At a first glance it may seem possible that the value of `secret` is leaked, because t_1

copies its value to shared variable `x`, but this is not the case. The first `println` statement cannot leak the secret, because it is executed before t_1 starts. Therefore `x` cannot contain the secret value at this time. We call an analysis that can detect the absence of this leak *invocation-sensitive*. The second `println` statement does also not leak the secret, because of the synchronization through lock `l`. The lock `l` is acquired in t_0 before t_1 is started. Due to this the write operation in t_1 can only be executed after t_0 releases `l` again which only happens after the second `println`. An *invocation-* and *lock-sensitive* analysis is able to detect this.

```java
class MyThread extends Thread {
  private Object l;
  private int secret = 42;
  private int x = 0;

  public static void main() {
    Object l = new Object();
    MyThread t1 = new MyThread(l);
    System.out.println(t1.x);
    synchronized (l) {
      t1.start();
      System.out.println(t1.x);
    }
  }
  public MyThread(Object l) {
    this.l = l;
  }

  public void run() {
    synchronized (l) {
      x = secret;
    }
  }
}
```

Fig. 1: A non-interferent multithreaded Java program.

Therefore in practice even the very precise invocation-sensitive PDG-based IFC analysis [1, 2] can only remove the first false alarm and does raise the second one. We were able to remove this false alarm and proof the program non-interferent by incorporating the results of a DPN-based analysis [3–6].

2 Concurrent Program Dependence Graphs

A PDG is a graph that captures the dependencies between statements in a program. Each node corresponds to a statement and potential dependencies between statements are represented by edges. In a sequential program, two statements s_1 and s_2 may either be *data dependent*, when a s_2 uses a value that s_1 has produced, or *control dependent*, when the outcome of s_1 decides if s_2 will be executed. These dependencies are in general a sound overapproximation of all dependencies that may occur during program execution. So whenever two statements are not connected in the PDG, they will never depend on each other during runtime and there is no information flow between them. Figure 2 shows the PDG of the example program with all data and control dependencies that may occur.

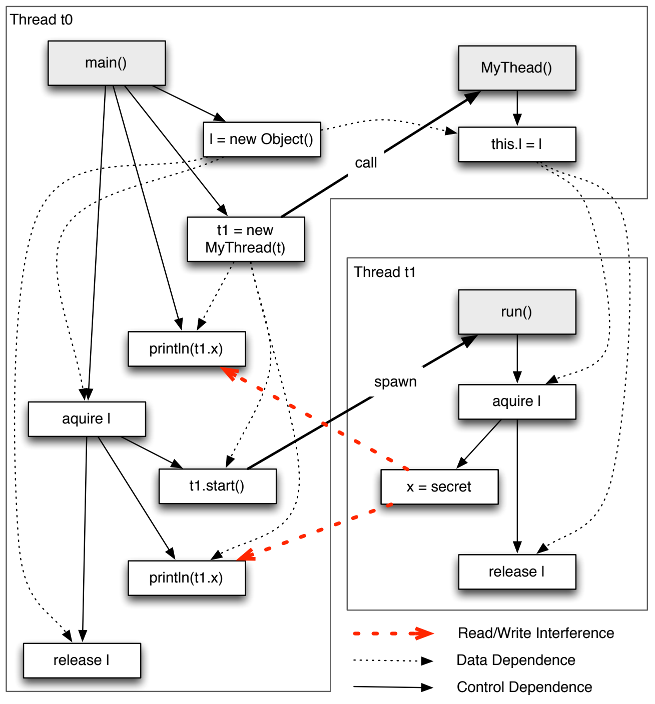

Fig. 2: The concurrent PDG of the program in Figure 1 that contains two spurious interference edges

For concurrent programs, control and data dependencies do not suffice, because they do not capture interference between different threads. Therefore the concurrent PDG contains additional *interference dependence* edges. A write and a read statement from two different threads are connected with an interference dependency, iff the value written may be read by the read statement. The concurrent PDG in Figure 2 shows two interference dependencies between both `println` statements and the statement that writes the value of variable `x`. As previously mentioned, both of these interferences are spurious and can be removed.

Giffhorn [2] proposes an invocation-sensitive but lock-insensitive may-happen-in-parallel (MHP) analysis that keeps track of thread creation and invocation through a dataflow analysis on the control flow graph. This algorithm is able to detect that the first `println` statement may not happen in parallel with the write operation on `x`, because the second thread t_1 has not been started at this time. The second interference dependence however is not removed, because the algorithm does not consider locking.

3 Dynamic Pushdown Networks

In order to achieve lock-sensitivity, we model concurrent Java programs using Dynamic Pushdown Networks (DPN) [3–6]. DPNs can precisely model concurrent programs with dynamic thread creation, unbounded recursion, synchronization via well-nested locks and finite abstractions of thread-local and procedure-local state. *Execution trees* [5] allow us to represent all the DPN's lock-sensitive executions using tree-automata. This allows to check for reachability of configurations with tree-regular properties e.g. calculating MHP information. Note that in the presence of locking MHP is not a sound criteria to remove interference. In fact Giffhorn [2] defines that two statements may-happen-in-parallel iff there exists two executions in which they are executed in opposite order. Recent extensions of DPN-analysis [6] allow to iterate the execution tree based technique and check whether critical configurations can be reached from other configurations while retaining a tree-regular property. In particular, we can check whether there exists an execution that executes the write to `x` first, followed by one of the `println` statements without an intervening killing of `x`. Since this is not the case, the DPN-based analysis will remove the spurious second interference edge.

4 Implementation and Future Work

We have integrated the DPN based interference Analysis in the tool *Joana*[7]. Joana, based on the Wala framework, implements a flow-, object-, context-sensitive IFC Analysis based on PDGs. Within the RS^3 priority program, we plan to integrate further analysis technique in order to further improve the tools precision. In particular, we want to improve on the lock-detection analysis, which in Java is difficult since any object can function as a lock. We plan to use path conditions and linear invariants to detect further spurious information flow. Analyses based on PDGs are typically whole-program analyses. In order to deal with software consisting of several components, we implement modular PDGs and methods for plugin-time analysis.

Acknowledgments. This work was funded by the DFG under the project IFC for Mobile Components in the priority program RS^3 (SPP 1496).

References

1. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security **8**(6) (December 2009) 399–422 Supersedes ISSSE and ISoLA 2006.
2. Giffhorn, D.: Slicing of Concurrent Programs and its Application to Information Flow Control. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)

3. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR 2005. Volume 3653 of LNCS. Springer (2005)
4. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: Proceedings of the 21st International Conference on Computer Aided Verification. CAV '09, Berlin, Heidelberg, Springer-Verlag (2009) 525–539
5. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation. VMCAI'11, Berlin, Heidelberg, Springer-Verlag (2011) 199–213
6. Nordhoff, B., Lammich, P., Müller-Olm, M.: Iterable forward reachability analysis of Monitor-DPNs. Submitted for publication (2012)
7. Programming Paradigms Group, KIT: Joana (Java Object-sensitive ANAlysis) tool. http://joana.ipd.kit.edu

Semantic-based Slicing through the use of Program Contracts

Daniela da Cruz

Departamento de Informática
Universidade do Minho
Braga, Portugal

Abstract. This extended abstract revisits the idea of slicing programs based on their axiomatic semantics, rather than using criteria based on control/data dependencies. It is shown how the forward propagation of preconditions and the backward propagation of postconditions can be combined in a new slicing algorithm that is more precise than the existing specification-based algorithms. The algorithm is based on (i) a precise test for removable statements, and (ii) the construction of a *slice graph*, a program control flow graph extended with semantic labels. It improves on previous approaches in two aspects: it does not fail to identify removable statements; and it produces the smallest possible slice that can be obtained (in a sense that will be made precise). Although only the basic concept of assertion-based slicing is present in this abstract, other topics will be discussed along the presentation, such as: comparison with other semantic-based slicing algorithms; tools that implement such algorithms; and future work.

1 Introduction

Program slicing [1] is a well-established activity in software engineering. For instance it plays an important role in program comprehension, since it allows software engineers to focus on the relevant portions of code (with respect to a given criterion). The basic idea is to isolate a subset of program statements that: either directly or indirectly contribute to the values of a set of variables at a given program location; or are influenced by the values of a given set of variables.

Other statements are considered spurious with respect to the given criterion and can be removed, enabling engineers to concentrate on the analysis of just the relevant ones. The first approach corresponds to *backward* forms of slicing, whereas the second corresponds to *forward* slicing.

Work in this area has focused on the development of progressively more effective, useful, and powerful slicing techniques, and has led to the use of these techniques in many application areas including program debugging, software maintenance, software reuse, and so on. See for instance [2] for a fairly recent survey of the area.

Program verification is an apparently unrelated activity whose goal is to establish that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behaviour of the implementation matches that of the specification (this is usually called the *functional* behaviour of the program), and moreover the program does not 'go wrong', for instance no errors occur during evaluation of expressions (the so-called *safety* behaviour). Modern program verification systems are based on algorithms that examine a program and generate a set of *verification conditions* that are sent to an external theorem prover for checking. If all the conditions generated from a program can be proved, then the program is guaranteed to be correct with respect to the specification.

There are several points of contact between slicing and verification: first, traditional syntactic slicing, applied a priori, facilitates the verification of large programs. Secondly, and this is what concerns us in this paper, it makes sense to slice programs based on semantic, rather than syntactic, criteria, and the contracts used in DbC and program verification are excellent candidates for such criteria. A third point (see Section 6) is that there is evidence that this kind of slicing can also be of help in the verification of large programs.

The expression "assertion-based slicing" is used here to refer to slicing methods based on the axiomatic semantics of programs, taking as criteria assertions (preconditions and/or postconditions) annotated in the programs. This includes *precondition-based* slicing, *postcondition-based* slicing, and *specification-based* slicing. The latter expression has been used in previous work when both a precondition *and* a postcondition (i.e. a specification) are given as criteria.

The paper introduces new ideas which allow the development of an algorithm for specification-based slicing that improves on previous algorithms in two aspects: the identification of sequences of statements that can be safely removed from a program (without modifying its semantics), and the selection of the biggest set of such sequences. Note that removable sequences may overlap, so this is not a trivial problem. This algorithm produces minimal slices (in the sense that will be made precise afterwards).

Structure of the Paper Section 2 introduces the simple language considered in the paper, and sets down the definitions of weakest precondition and strongest postcondition. Section 3 formalizes the basic notions of precondition-, postcondition- and specification-based slicing used in the rest of the paper. The following sections introduce a new test for identifying removable chunks of code (Section 4), and a graph-based algorithm for actually computing the best possible slices of a program with respect to a given specification (Section 5). The paper ends in Section 6 with a summary and some topics that will be explored in the talk.

$$
\begin{array}{llll}
\textbf{Exp[int]} & \ni & e ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid \\
 & & & -e \mid e + e \mid e - e \mid e * e \mid e\,\mathsf{div}\,e \mid e\,\mathsf{mod}\,e \\
\textbf{Exp[bool]} & \ni & b ::= & \mathsf{true} \mid \mathsf{false} \mid e = e \mid e < e \mid e \leq e \mid e > e \mid \\
 & & & e \geq e \mid e \neq e \mid b \wedge b \mid b \vee b \mid \neg\, b \\
\textbf{Assert} & \ni & A ::= & \mathsf{true} \mid \mathsf{false} \mid e = e \mid e < e \mid e \leq e \mid e > e \mid \\
 & & & e \geq e \mid e \neq e \mid A \wedge A \mid A \vee A \mid \neg\, A \mid \\
 & & & A \rightarrow A \mid \forall\, x.\, A \mid \exists\, x.\, A \\
\textbf{Comm} & \ni & C ::= & \textbf{skip} \mid x := e \mid \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S \mid \\
 & & & \textbf{while}\ b\ \textbf{do}\ \{A\}\ S \\
\textbf{Prog} & \ni & S ::= & C \mid C\,;\,S \\
\textbf{Spec} & \ni & P ::= & \{A\}\ S\ \{A\}
\end{array}
$$

Fig. 1. Language syntax

2 The Language, Weakest Preconditions and Strongest Postconditions

To illustrate the ideas the syntax in Figure 1 for a core imperative language will be used. Programs are non-empty sequences of commands; Specifications are programs annotated with preconditions and postconditions.

Please note that the choice of language is not important, and the ideas discussed scale up to realistic languages; the only crucial requirements are the existence of an axiomatic semantics (definitions of weakest precondition and strongest postcondition), and an external proof tool that is capable of reasoning about the data structures that are present in the language. To illustrate the ideas presented along the paper it will be used a very simple language with integer variables only; the syntax of assertions (used as preconditions, postconditions, and loop invariants) is obtained as an extension of boolean expressions with first-order quantification.

The notions of weakest precondition and strongest postcondition are certainly among the most important and popular in programming semantics. For such a simple language, there is a nice symmetry between them, and both can be used to calculate proof obligations (usually called *verification conditions*) when verifying the correctness of programs. The former is however much more widely used in verification tools, because of the absence of quantifiers. The definition of both notions for our language is given in Figure 2.

Notation Let $S = C_1\,;\,\ldots\,;\,C_n$, $1 \leq k \leq n$, and $1 \leq i \leq j \leq n$. It will be used the following notation for the weakest precondition of a suffix of S; the strongest postcondition of a prefix of S; and the sequence obtained by removing a subsequence of S.

- $\overline{\mathsf{wp}}_k(S, Q) \doteq \mathsf{wp}(C_k\,;\,C_{k+1}\,;\,\ldots\,;\,C_n, Q)$
- $\overline{\mathsf{wp}}_{n+1}(S, Q) \doteq Q$
- $\overline{\mathsf{sp}}_0(S, P) \doteq P$
- $\overline{\mathsf{sp}}_k(S, P) \doteq \mathsf{sp}(C_1\,;\,\ldots\,;\,C_{k-1}\,;\,C_k, P)$

$$
\begin{aligned}
\mathsf{wp}(\mathbf{skip}, Q) &= Q \\
\mathsf{wp}(x := e, Q) &= Q^x_e \\
\mathsf{wp}(C_1; C_2, Q) &= \mathsf{wp}(C_1, \mathsf{wp}(C_2, Q)) \\
\mathsf{wp}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, Q) &= (b \to \mathsf{wp}(C_t, Q)) \\
&\quad \wedge\ (\neg\, b \to \mathsf{wp}(C_f, Q)) \\
\mathsf{wp}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\, C, Q) &= I \\
\\
\mathsf{sp}(\mathbf{skip}, P) &= P \\
\mathsf{sp}(x := e, P) &= \exists\, v.\, P^x_v \wedge x == e^x_v \\
\mathsf{sp}(C_1; C_2, P) &= \mathsf{sp}(C_2, \mathsf{sp}(C_1, P)) \\
\mathsf{sp}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, P) &= \mathsf{sp}(C_t, b \wedge P) \\
&\quad \vee\ \mathsf{sp}(C_f,\, \neg\, b \wedge P) \\
\mathsf{sp}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\, C, Q) &= I \wedge\, \neg\, b
\end{aligned}
$$

Fig. 2. Definition of weakest precondition and strongest postcondition. Q^x_e denotes the result of substituting e for x in Q; I is a loop invariant.

– $\mathsf{remove}(i, j, S) \doteq$
  $\begin{cases} \mathbf{skip} \text{ if } i = 1 \text{ and } j = n, \\ C_1\,;\, \ldots\,;\, C_{i-1}\,;\, C_{j+1}\,;\, \ldots\,;\, C_n \text{ otherwise.} \end{cases}$

3 Assertion-based Slices

In this section the different notions of precondition-based slicing, postcondition-based slicing and specification-based slicing are formalized. A program S' is a *specification-based slice* of S if it is a *portion* of S (a syntactic notion, also known as a *reduction* of S) and moreover S can be *refined* to S' with respect to a given specification (a semantic notion). The notions of precondition-based and postcondition-based slice can be defined as special cases of this notion.

Definition 1 (Assertion-based slices). *Let S be a correct program with respect to the specification consisting of precondition P and postcondition Q. The program S' is said to be*

– *a* specification-based slice *of S with respect to (P, Q), written $S' \lhd_{(P,Q)} S$, if $S' \preceq S$ (S' is a portion of S) and S' is also correct with respect to (P, Q);*
– *a* precondition-based slice *of S with respect to P if $S' \lhd_{(P,\mathsf{sp}(S,P))} S$;*
– *a* postcondition-based slice *of S with respect to postcondition Q if $S' \lhd_{(\mathsf{wp}(S,Q),Q)} S$.*

Incidentally, note that the names precondition- and postcondition-based slice are not entirely adequate for describing the notions known under these names. They would more accurately be described as *condition-based forward and backward slice*, respectively, which not only establishes a correspondence with the two classic approaches to syntactic slicing, but also highlights the fact that conditions may be propagated even when preconditions or postconditions are not present. An example is a program that starts with an assignment $x := c$ with c a constant. Even without an informative precondition, forward slicing will use the information $x = c$, calculated as the strongest postcondition of the command, and then propagated forward.

In this paper it is proposed a solution to the problems raised by previous versions of specification-based slicing algorithms, in the form of an optimal slicing algorithm. The algorithm builds on two basic ideas, that will be explained in the next two sections. In abstract terms, any slicing algorithm based on the axiomatic semantics of programs must be able to

1. Identify subprograms that can be removed from the program being sliced, without modifying its semantics. More concretely, given a program $S = C_1 \,;\, \ldots \,;\, C_n$ with specification (P, Q), some test is used to allow the algorithm to decide if $\mathsf{remove}(i, j, S) \triangleleft_{(P,Q)} S$ holds. It was shown that the algorithm of [3] fails to identify some subprograms; In Section 4 it will be shown that using preconditions and postconditions *simultaneously* allows for a precise identification of removable subprograms.
2. Select, among the subprograms identified as removable, the combination that produces the smallest sliced program. In Section 5 it will be shown that this may be reduced to a graph problem that can be solved by applying standard algorithms.

The algorithm can be applied to calculate precondition-, postcondition-, and specification-based slices. One concentrate on the latter case, since the first two are particular cases as shown before. Note that the resulting algorithm is *optimal in a relative sense* only. The test for removable subprograms involves first-order formulas whose validity must be established externally by some proof tool. Undecidability of first-order logic destroys any hope of being able to identify every removable subprogram automatically, since some valid formulas may not be proved.

4 Removable Subprograms

One start by considering programs without iteration and postpone the discussion of loops to the end of the section. For such programs the following lemma is straightforward to prove:

Lemma 1. *For every precondition P, postcondition Q, and program S,*

1. $\models P \rightarrow \mathsf{wp}(S, \mathsf{sp}(S, P))$
2. $\models \mathsf{sp}(S, \mathsf{wp}(S, Q)) \rightarrow Q$
3. $\models P \rightarrow \mathsf{wp}(S, Q)$ *iff* $\models \mathsf{sp}(S, P) \rightarrow Q$

Each of the implications mentioned in the third item in fact corresponds to the *verification condition* for the program S: it suffices to check the validity of this condition to ensure that S is correct with respect to the specification (P, Q).

In this section one consider the following problem: given a specification and a program S correct with respect to it, how can it be decided if some subsequence of S can be removed, resulting in a program that is still correct with respect to the specification, i.e. it is a specification-based slice of S? Note that one are not asking if the sequence *should* be sliced (since this could prevent the minimal slice from being obtained); that question is left to the next section.

The following lemma establishes the implications that are valid among the calculated preconditions and postconditions calculated for a subsequence of a given correct program.

Lemma 2. *Let (P, Q) be a specification; $S = C_1 ; \ldots ; C_n$ a program such that $\models P \to \mathsf{wp}(S, Q)$, and i, j, k integers such that $1 \leq i \leq j \leq n$ and $0 \leq k \leq n$. Then*

1. $\models \overline{\mathsf{sp}}_k(S, P) \to \overline{\mathsf{wp}}_{k+1}(S, Q)$
2. $\models \overline{\mathsf{sp}}_{i-1}(S, P) \to \mathsf{wp}(C_i ; \ldots ; C_j, \overline{\mathsf{wp}}_{j+1}(S, Q))$

Observe that $\overline{\mathsf{sp}}_{i-1}(S, P)$ and $\overline{\mathsf{wp}}_{j+1}(S, Q)$ can be seen respectively as the *strongest precondition* and the *weakest postcondition* calculated for the sequence $C_i ; \ldots ; C_j$ w.r.t. the specification (P, Q). Their significance is that, according to the following proposition, they can be used to decide exactly when the sequence $C_i ; \ldots ; C_j$ can be sliced off.

Proposition 1. *In the conditions of the previous lemma,*

$$\models \overline{\mathsf{sp}}_{i-1}(S, P) \to \overline{\mathsf{wp}}_{j+1}(S, Q) \text{ \textit{iff} } \mathsf{remove}(i, j, S) \lhd_{(P,Q)} S$$

Note that the following are implications but not equivalences:

$$\begin{array}{l} \models \overline{\mathsf{wp}}_i(S, Q) \to \overline{\mathsf{wp}}_{j+1}(S, Q) \text{ implies} \\ \models P \to \mathsf{wp}(\mathsf{remove}(i, j, S), Q) \end{array}$$

$$\begin{array}{l} \models \overline{\mathsf{sp}}_{i-1}(S, P) \to \overline{\mathsf{sp}}_j(S, P) \text{ implies} \\ \models P \to \mathsf{wp}(\mathsf{remove}(i, j, S), Q) \end{array}$$

Both conditions would also imply $S' \lhd_{(P,Q)} S$. However, note that the latter conditions are both stronger than the one in the proposition (as a consequence of Lemma 2(1)), which means that using them as tests would not allow for all removable subprograms to be identified.

For commands containing sequences of commands, illustrated here with conditional, the following proposition states that slicing both branches results in a slice of the structured command. It suffices to propagate the postcondition inside both branches, as well as the precondition strengthened with the boolean condition and its negation, respectively.

Proposition 2. *If $S'_t \lhd_{(P \wedge b, Q)} S_t$ and $S'_f \lhd_{(P \wedge \neg b, Q)} S_f$, then*

$$\textbf{if } b \textbf{ then } S'_t \textbf{ else } S'_f \lhd_{(P,Q)} \textbf{if } b \textbf{ then } S_t \textbf{ else } S_f$$

The treatment of loops introduces a few subtleties. First, if S contains loops the implication $\models P \to \mathsf{wp}(S, Q)$ is no longer the only verification condition: other conditions must be introduced related to the preservation of the *loop invariant*, as well as its relation with the loop's desired postcondition (or precondition, if strongest postconditions are used). The notion of refinement, required by the definition of specification-based slicing, will now incorporate the preservation of these additional conditions. Moreover, in a total correctness setting other conditions are involved, regarding the strictly decreasing value of a *loop variant.* Slicing the body of a terminating loop should not result in a non-terminating loop, which is granted by the preservation of the verification conditions involving the loop variant. Full details will be given in the presentation of this paper.

5 Slice Graphs

Below it is introduced the notion of control graph for a program, labeled with respect to a given specification, and the notion of slice graph, in which removable sequences of commands will be associated with edges added to the initial control flow graph.

Definition 2 (Labeled Control Flow Graph). *Given a program S, precondition P and postcondition Q such that $S = C_1 ; \ldots ; C_n$, the* ***labeled control flow graph*** *$LCFG(S, P, Q)$ of S with respect to (P, Q) is a labeled directed acyclic graph (DAG) whose edge labels are pairs of logical assertions on program states. To each command C in the program S one associate an input node $IN(C)$ and an ouput node $OUT(C)$.*

The details about the graph construction can be found in [4].

Informally, the idea is that the label of an edge $C_i \longrightarrow C_j$ represents the strongest postcondition $\overline{\mathsf{sp}}_i(S, P)$ of (the sequence ending with) the command C_i and the weakest precondition $\overline{\mathsf{wp}}_j(S, Q)$ of (the sequence beginning with) the command C_j, calculated from the initial specification (P, Q) taking into account the structure of the program.

It is crucial that sequences that are branches of a conditional are generated using the appropriate strongest postcondition and weakest precondition, in accordance with Proposition 2. The same applies to the body of loop commands. This means that the graph is annotated exactly with the strongest postconditions and weakest preconditions that are calculated recursively throughout the structure of the graph, following the definition of Figure 2. The labelled CFG can thus be seen as a "verification graph" for a program; in particular, the program is correct if $\models P \to \overline{\mathsf{wp}}_1(S, Q)$, where $(P, \overline{\mathsf{wp}}_1(S, Q))$ is the label of the outgoing edge from the *START* node or equivalently if $\models \overline{\mathsf{sp}}_n(S, P) \to Q$, where $(\overline{\mathsf{sp}}_n(S, P), Q)$ is the label of the incoming edge into the *END* node.

An algorithm for constructing the graph could first build the unlabeled graph from the syntax tree of the program, then assign the first component of the labels by traversing the graph from *START* to *END*, and finally assign the second component by traversing the graph in the reverse direction. Note that the label of each edge can be calculated *locally* from the labels of the (one or two) previous edges. In particular, note that for $1 \le k \le n$,

$$\begin{array}{l}\overline{\mathsf{sp}}_k(S,P) = \mathsf{sp}(C_k, \overline{\mathsf{sp}}_{k-1}(S,P)) \\ \overline{\mathsf{wp}}_k(S,Q) = \mathsf{wp}(C_k, \overline{\mathsf{wp}}_{k+1}(S,Q))\end{array}$$

The worst-case execution cost of constructing the graph is apparently linear on the program size. However, weakest preconditions are potentially of exponential size on the length of the program [5], so this is not so. Fortunately this size can be corrected to quadratic (see Section 6).

Definition 3 (Slice Graph). *Consider a program S and a specification (P, Q) such that $\models P \rightarrow \mathsf{wp}(S,Q)$ (in which case one assume loops are not annotated with variants). The* slice graph *$SLCG(S,P,Q)$ of S with respect to (P,Q) is obtained from the labeled control flow graph $LCFG(S,P,Q)$ by inserting additional edges as follows.*

Let $\hat{S} = \hat{C_1}$; ... ; $\hat{C_m}$ be any maximal *sequence of commands in S, i.e. $\hat{S}$ is a branch of a conditional command in S, or the body of a loop command in S, or else $\hat{S} = S$. Then for any two edges $(\hat{C}_{i-1}, \hat{C_i})$ with label $(\overline{\mathsf{sp}}_{i-1}(S,P), \overline{\mathsf{wp}}_i(S,Q))$ and $(\hat{C_j}, \hat{C}_{j+1})$ with label $(\overline{\mathsf{sp}}_j(S,P), \overline{\mathsf{wp}}_{j+1}(S,Q))$ in $LCFG(S,P,Q)$ such that $i < j$, if $\models \overline{\mathsf{sp}}_{i-1}(S,P) \rightarrow \overline{\mathsf{wp}}_{j+1}(S,Q)$,*

- *if $i \neq 1$ or $j \neq m$, an edge $(\hat{C}_{i-1}, \hat{C}_{j+1})$ with label $(\overline{\mathsf{sp}}_{i-1}(S,P), \overline{\mathsf{wp}}_{j+1}(S,Q))$ is inserted;*
- *otherwise if $i = 1$ and $j = m$ a new* **skip** *node is inserted in the graph, together with two edges $(\hat{C}_{i-1}, \textbf{skip})$ and $(\textbf{skip}, \hat{C}_{j+1})$, both with label $(\overline{\mathsf{sp}}_{i-1}(S,P), \overline{\mathsf{wp}}_{j+1}(S,Q))$.*

The time required to insert the additional edges into the graph is again quadratic on the length of the program, since for each sequence of commands it is necessary to generate slicing conditions for every pair of edges such that the first precedes the second in the graph. Note that this presupposes that the external theorem prover checks the validity of formulas in constant time, which is a reasonable assumption since automatic tools are typically used with a time out limit, after which a condition is treated as invalid. Also, the construction depends on the particular external tool used to decide which edges should be inserted, and may in fact result in different graphs if different tools are used.

As an example, Figure 3 shows the slice graph for the program in Listing 1.1 with respect to the specification $(y > 10, x \geq 0)$. It is clear that removable sequences are signaled by the edges (and possibly **skip** nodes) that are added to the initial labeled CFG. The following proposition states that all admissible slices are represented in the slice graph.

Proposition 3. *Let $S' \preceq S$. Then $S' \triangleleft_{(P,Q)} S$ iff the control flow graph $LCFG(S', P, Q)$ is a spanning subgraph (i.e. a subgraph with the same set of nodes) of the slice graph $SLCG(S,P,Q)$.*

```
if (y > 0) then x := 100;
                x := x+50;
                x := x-100
           else x := x-150;
                x := x-100;
                x := x+100
```

Listing 1.1.

Thus for any given sequence of commands σ inside S, the graph contains a path that represents every subsequence σ' of σ such that substituting σ' for σ results in a slice of S. The slice graph represents the entire set of specification-based slices of S, and obtaining the minimal slice is simply a matter of selecting the shortest subsequences using the information in the graph.

Slicing Algorithm The notion of minimal slice *with respect to a given slice graph* is simply given by a read-back from the graph to the program. For each command sequence represented in the graph, one apply a shortest paths algorithm (basically a breadth-first traversal, linear on the size of the graph) to find a minimal slice of that sequence. Nodes that are not traversed correspond to commands that can be removed. This notion of minimality is relative since it is meant with respect to a slice graph: the proof tool may have failed or timed out in checking some valid conditions (and signaling them in the graph); the resulting slice will thus only be as good as the graph.

Slicing a loop involves slicing the loop's body recursively, with respect to the specification $(I, I \wedge \neg b))$. Note however that the usefulness of this approach may be limited without human intervention. If the program is being sliced with a specification that has been weakened with respect to an initial, full specification, it makes sense to weaken the loop invariant accordingly, otherwise slicing the loop may result in no commands being removed at all inside its body.

6 Conclusion

Assertion-based slicing is more powerful and flexible than syntactic slicing, since the criteria can be as expressive as any set of first-order formulas on the initial and final states of the program. One of the first forms of slicing based on program semantics was *conditioned slicing* [6], a form of forward slicing. This was shown to subsume both static and dynamic notions of syntax-based slicing, since the initial state of execution is constrained by a first-order formula that can be used to constrain the set of different admissible initial states to exactly one (corresponding to dynamic slicing), or simply to identify a relevant subset of the state to be used as slicing criterion (as in static slicing). The same applies to backward slicing: using a postcondition as slicing criterion instead of a set of variables is clearly more expressive. Naturally, this expressiveness comes at a cost, since semantic forms of slicing are harder to compute.

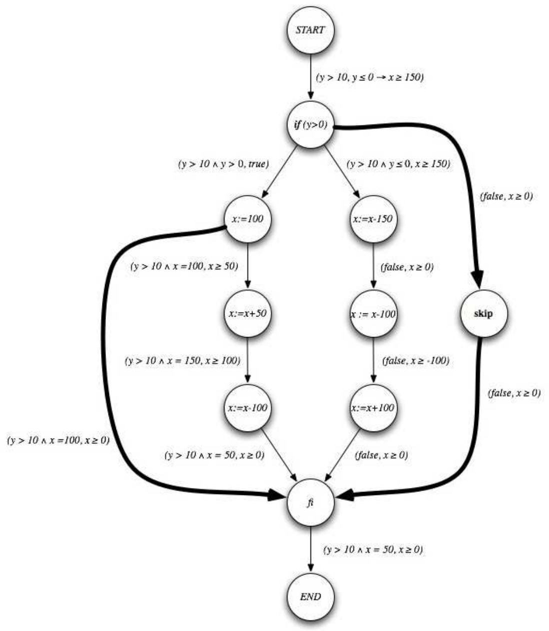

Fig. 3. Example slice graph. Thick lines represent edges that were added to the initial CFG, corresponding to "shortcut" subprograms that do not modify the semantics of the program. These paths have the same source and destination nodes as other longer paths corresponding to removable sequences

GamaSlicer [7] is a tool developed for experimenting the previous versions of specification-based slicing algorithms and also the new algorithm proposed in this paper. The user can choose between different precondition-based, postcondition-based, and specification-based slicing algorithms; the tool also offers standard verification capabilities (verification condition generation) and a visual representation of the LCF graphs introduced in the paper.

While the front-end is meant to allow for experimentation and comparison of different algorithms, one intend to optimize and test the graph-based algorithms with realistic code. One obligatory step will be to calculate weakest preconditions using Flanagan and Saxe's algorithm [5], which avoids the potential exponential explosion in the size of the conditions generated, keeping our algorithm within quadratic time.

Along the presentation, the main topic (assertion-based slicing) will serve as motto to compare this new concept with existing semantic-based slicing algorithms. Also, the adaption of this algorithm to work with passive programs will be discussed, since there are interesting challenges when considering such kind of programs (GamaBoogie was developed to slice Boogie programs in its passive form). Future work and improvements will be discussed: it will be interesting to compare the present approach with the work of Fox and colleagues [8], who introduced the *backward conditioning* technique, based on symbolic execution. The goal of this related approach is to remove from a program statements which, when executed, always lead to the negation of a given postcondition. The interest of this work is that it indicates that the interaction between slicing and verification happens in both directions: verification offers the tools used for implementing assertion-based slicing (of correct programs), but slicing can also be used to facilitate program verification.

Finally, it will be emphasized that the notion of control flow graph labeled with semantic information has a broader usefulness and may have other applications in program analysis, verification, and of course visualization.

References

1. M. Weiser, "Program slicing," in *ICSE '81: Proceedings of the 5th international conference on Software engineering.* Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.
2. B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, 2005.
3. I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon, "Program slicing based on specification," in *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing.* New York, NY, USA: ACM, 2001, pp. 605–609.
4. José Bernardo Barros, Daniela Carneiro da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. "Assertion-based slicing and slice graphs." in *Formal Asp. Comput.*, 24(2):217–248, 2012.
5. C. Flanagan and J. B. Saxe, "Avoiding exponential explosion: generating compact verification conditions," in *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York, NY, USA: ACM, 2001, pp. 193–205.
6. G. Canfora, A. Cimitile, and A. D. Lucia, "Conditioned program slicing," *Information and Software Technology*, vol. 40, no. 11-12, pp. 595–608, November 1998, special issue on program slicing.
7. D. da Cruz, P. R. Henriques, and J. S. Pinto, "Gamaslicer: an Online Laboratory for Program Verification and Analysis," in *proceedings of the 10th. Workshop on Language Descriptions Tools and Applications (LDTA'10)*, 2010.
8. C. Fox, S. Danicic, M. Harman, and R. M. Hierons, "Backward conditioning: A new program specialisation technique and its application to program comprehension," in *IWPC.* IEEE Computer Society, 2001, pp. 89–97.

Efficient Algorithms for Control Closures

Christian Hammer

Saarland University, Saarbrücken, Germany
{lastname}@cispa.uni-saarland.de

Abstract. Recently a unifying theory of control dependence has been developed that handles arbitrary control flow graphs. In particular, it no longer requires a single entry, nor a single exit point of the control flow graph. While the authors presented algorithms that compute weak and strong control closures, representing the transitive hull of control dependence, their worst-case complexity contains third and fourth degree polynomials, which makes them too expensive for realistic programs. This paper presents an algorithm that efficiently computes both weak and strong control dependence, with an amortized worst case complexity in $O(N \log N)$ and $O(N(\log N)^2)$.

Keywords: Control dependency, program slicing

1 Introduction

In program analysis terms, there are two major sources of interference: data dependence and control dependence. There exists a *data dependence* between two operations of a program, if there exists an execution where one operation computes a value that the other uses. This notion is also known as *explicit flow* in the information flow control literature. The other source of interference emerges if one operation of a program controls whether or how often another operation executes, which is represented as a control dependence between those two operations. This notion is called *implicit flow* for information flow analyses. Research proposed several definitions of control dependence, some of which are known to be equivalent, others extensions to programs with more relaxed notions of the control flow graph, which is a typical program representation. On top of this, there is another dimension of control dependence, namely whether it takes program termination into account or ignores it.

Recently, a unifying theory of control dependence was presented [1], which subsumes all the previous definitions and comes in only two variants: termination-sensitive and termination-insensitive. As this theory is based on a general notion of control flow graph, there is hope that these definitions are general enough to be applicable to all programing paradigms.

In contrast to many previous definitions, Danicic et al. define the underlying semantics for control dependence as relations between graphs called weak and strong projection, and show that the graph induced by a slicing criterion (which is a set of vertices) is a weak/strong projection of the original graph iff the induced graph is weakly/strongly closed under their definition of control dependence.

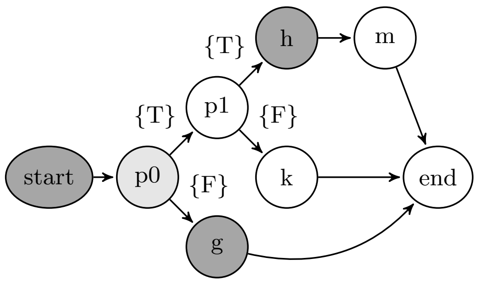

Fig. 1. Example control flow graph, criterion darkly shaded, closure lightly

Danicic et al. [1] present algorithms to compute weak and strong control closures, however their worst case complexity of $O(N^3)$ and $O(N^4)$, where N is the number of operations of a program is prohibitive for program analysis, especially when programs can reach millions of lines of code. In contrast, traditional non-termination-sensitive control dependence on a single-entry-single-exit control flow graph can be computed in almost linear time in the size of the program [2,3]. So in order to make the new theory applicable to realistic software, more efficient algorithms need to be found.

This contributions of this paper are two variants of an algorithm that efficiently computes weak and strong control closures. We conjecture a worst case complexity in $O(N \log N)$ for weak and $O(N(\log N)^2)$ for strong control closures. Therefore, both forms of control closure computation are therefore widely applicable even for large program sizes. This work, however, is still in progress, and do not yet provide a proof of complexity or correctness.

2 Weak and Strong Control Closures

The new forms of control closures are defined on a very general form of the control flow graph $G = (V, E, \beta)$ where (V, E) is a directed graph and the vertex set V is partitioned into predicates P and non-predicates N. The edge labeling function $\beta : E \to \mathcal{P}(T, F)$ has the following properties:

1. $\forall x \in P : \mathit{outdegree}(x) \leq 2 \wedge \forall (x, y) \in E : \beta(x, y) \neq \emptyset$
2. $\forall x \in N : \mathit{outdegree}(x) \leq 1 \wedge \forall (x, y) \in E : \beta(x, y) = \emptyset$
3. $\forall p \in P, (p, y), (p, z) \in E : x \neq y \implies \beta(p, y) \cap \beta(p, z) = \emptyset$
4. A special vertex *end* of out-degree 0 may be present to represent normal termination.

As an example, consider Fig. 1, which is a traditional CFG with distinguished start and end vertices. Danicic et al. show that the traditional control slice of the criterion $\{g, h\}$ is equivalent to the weak control closure with criterion $\{\mathit{start}, \mathit{end}, g, h\}$. This traditional slice would contain both predicates. The control closure of $\{\mathit{start}, g, h\}$, however, only contains $p0$, as only this node determines whether g or h will be executed. Their paper also provides Algorithm 1

Algorithm 1: computing minimal weak control closures

```
Input: slicing criterion $V'$
Result: its minimal weak control closure $X$
$X = V'$;
while $\exists\ (p, v) \in E : V' \to^* p \wedge |\Theta(G, X, v)| = 1 \wedge |\Theta(G, X, p)| \geq 2$ do
    $X = X \cup \{p\}$;
```

to compute control closures, where $\Theta(G, X, v)$ is the set of first-reachable vertices from v to X in G, i.e. the set of vertices in X that are reachable from v in the subgraph of G where all outgoing edges from any vertex in X have been deleted[1]. In particular, we interpret this definition as $\Theta(G, V', v) = 1$ for any $v \in V'$.

In the graph of Fig. 1, for $V' = \{start, g, h\}$ we have $\Theta(G, V', p0) = 2$, as $p0$ can reach both g and h directly, $\Theta(G, V', p1) = 1$ as it can only reach h. As $p0$ is also reachable from $start$, the algorithm adds $p0$ to X. Now $\forall x \in V : \Theta(G, X, x) \leq 1$ and the algorithm terminates.

For $V' = \{start, end, g, h\}$ we have $\Theta(G, V', p0) = 3$, $\Theta(G, V', p1) = 2$, and $\Theta(G, V', k) = 1$. Thus, we add $p1$ to X which yields $\Theta(G, X, p0) = 2$, $\Theta(G, X, p1) = 1$. After adding $p0$ to X the algorithm terminates.

Danicic et al. argue that this algorithm is in $O(N^3)$, as the `while` loop can execute N times, the existence test may check $|E|$ edges, and the computation of Θ is in $O(N)$.

2.1 Strong Control Closure

Based on that definition, algorithm 2 is presented to compute that function. This algorithm is in $O(N^2)$ as it executes at most $|E|$ iterations and needs to check $|E|$ edges in each. Consider Fig. 2, where vertex k has a self-loop. $\Gamma(G, \{start, g, h\})$ removes the edge $(p1, h)$ such that $p1$ becomes a non-predicate. It also removes

[1] [1] states that this subgraph is a CFG, however, it might no longer be connected

Algorithm 2: Original algorithm for computing gamma

```
Input: $V'$
$X = V'$;
while $\exists\ (y, x) \in E : x \in X \wedge y \notin X$ do
    $E = E \setminus (y, x)$;
    if $\exists (y, z) \in E : z \neq x$ then
        $P = P \setminus \{z\}; N = N \cup \{z\}$
    else if $y \in N$ then
        $X = X \cup \{y\}$
$\Gamma = V \setminus X$
```

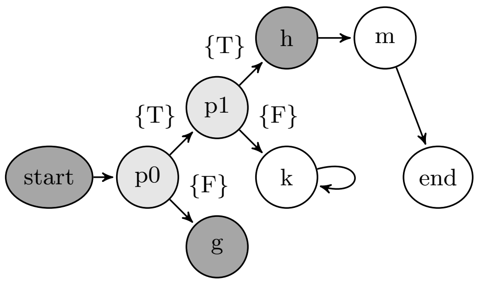

Fig. 2. Example control flow graph, criterion shaded darkly, strong closure shaded

Algorithm 3: computing minimal strong control closures

```
Input: slicing criterion $V'$
Result: its minimal strong control closure $X$
$X = V'$;
while $\exists\ (p,v) \in E : V' \to^* p \wedge |\Theta(G,X,v)| = 1 \wedge v \notin \Gamma(G,X)$
   $\wedge(|\Theta(G,X,p)| \geq 2 \vee p \in \Gamma(G,X))$ do
    $X = X \cup \{p\}$;
```

$(p0, g)$, which makes $p0$ another non-predicate. Then the algorithm terminates and $\Gamma(G, \{start, g, h\} = \{p0, p1, k, m, end\}$.

The original algorithm to compute strong control closures is very similar to the one for weak control closures. Basically, it just requires a new function Γ which is defined as follows:

Definition 1. *Let $G = (V, E)$ be a finite directed graph and let $V' \subseteq V$. We define $\Gamma(G, V')$ to be the set of all $x \in V$ that lie on a complete path in G which does not pass through V'. A complete path is either an infinite path or a finite path whose last vertex is final. A final vertex is either a non-predicate vertex of out-degree 0 or an incomplete predicate, i.e. a predicate whose union of outgoing edge labels is not $\{T, F\}$.*

Consider Fig. 2, where vertex k has a self-loop. In contrast to the weak control closure of $V' = \{start, g, h\}$, the strong control closure contains $p1$, as this predicates whether the program terminates or goes into an infinite loop. Algorithm 3 reflects this by the additional constraints of the `while` loop (as opposed to Algorithm 1.) For the edge $(p1, h)$, we have $h \notin \Gamma(G, V')$ but $p1 \notin \Gamma(G, V')$, which adds $p1$ to X.

3 An Efficient Algorithm

The main ideas behind the efficient algorithm presented in this section are:

1. Computing $\Theta(G, X, \cdot)$ can be done in $O(N)$, i.e. $\Theta(G, X, y)$ can be computed in constant amortized time.

Algorithm 4: computing Θ

```
Input: slicing criterion $V'$
Result: $\Theta(G, V', \cdot)$
foreach $n \in V'$ do
    remove all outgoing edges of $n$
foreach $vertex\ finishing\ in\ a\ DFS$ do
    if $vertex \in V'$ then
        $fr = fr \cup (vertex \mapsto \{vertex\})$;
        $\Theta(G, V', vertex) = 1$;
    else
        $nodes = \emptyset$;
        foreach $n \in succ(vertex)$ do
            $nodes = nodes \cup fr(n)$;
        $fr = fr \cup (vertex \mapsto nodes)$;
        $\Theta(G, V', vertex) = |nodes|$;
foreach $set \in stronglyConnectedSets(G)$ do
    $theta = max_{n \in set}(theta(n))$;
    foreach $Node\ node : set$ do
        $\Theta(G, V', node) = theta$;
set the theta of all vertices not reachable from $V'$ to 0;
```

Algorithm 5: computing the component graph

```
Input: Graph $G$
Result: Component Graph $G'$
$G'$ = new DirectedAcyclicGraph();
$componentOf = \emptyset$;
foreach $set \in stronglyConnectedComponents(G)$ do
    create new component node n;
    n.nodes = set;
    add vertex n to $G'$;
    foreach $Node\ node : set$ do
        $componentOf = componentOf \cup (node \mapsto n)$;
foreach $node \in G$ do
    from = componentOf(node);
    foreach $s \in succ(node)$ do
        to = componentOf(s);
        if $from \neq to$ then
            add edge (from, to) to $G'$;
return $G'$;
```

Algorithm 6: computing weak control closures

```
computeTheta($V$);
computeComponentGraph();
compute reverse topological sorting in the component graph;
assign each vertex the corresponding number of previous step;
$pq$ = new PriorityQueue() sorted according to this number;
foreach $n : V$ do
    $n.diff = 0$;
    add $n$ to $pq$;
$X = V$;
while $pq$ is not empty do
    $v$ = remove vertex from $pq$;
    $\Theta^{\delta} = v.diff$;
    foreach $p \in predecessors(v)$ do
        if $p \notin X$ then
            if $\Theta(G, X, p) - \Theta^{\delta} \geq 2 \wedge (\Theta(G, X, v) - \Theta^{\delta} = 1 \vee v \in X)$ then
                $X = X \cup p$;
                $diff = \Theta(G, X, p) - 1$;
                $\Theta(G, X, p) = 1$;
                $p.diff = diff$;
                add $p$ to $pq$;
            else
                $p.diff = \Theta^{\delta}$;
                add $p$ to $pq$;
return criterion;
```

2. If $(x, y) \in E, outdegree(x) = 1$ then $\Theta(G, X, x) = \Theta(G, X, y)$
3. Adding a vertex to X in the computation of Θ can only change (transitive) predecessors of that vertex, so the computation of Θ should be done "backwards".

First, we describe the algorithm to compute $\Theta(G, X, \cdot)$ in linear time (see Algorithm 4). After removing outgoing edges from the criterion nodes, we do a depth first search on the graph, and propagate the first-reachable vertices of V' backwards whenever a vertex has been fully processed (second foreach loop). Then we use the observation that all vertices in a loop of the pruned CFG have the same Θ value, so we set it to the maximum determined in that loop. Finally, vertices not reachable from V' are assigned a Θ of 0, so they will not satisfy the predicate of Algorithm 1.

We use reverse topological order to compute the control closure "backwards", however, reverse topological order is only defined for directed acyclic graphs, so we fold the CFG to its component graph, where all strongly connected components are represented by a single vertex. Since the component graph is acyclic, we can determine its reverse topological order and propagate the order number

Algorithm 7: incrementally computing gamma

```
Input: $update$, the set of vertices to add to $X$
$pq$ = new PriorityQueue() using reverse topological sorting;
foreach $n : update$ do add $n$ to $pq$;
while $pq$ is not empty do
    $v$ = remove vertex from $pq$;
    foreach $edge \in incomingEdgesOf(v)$ do
        $p = edge.from$;
        if $p \notin X$ then
            remove $edge$ from $G^{\Gamma}$;
            if $G^{\Gamma}.outDegreeOf(p) > 0$ then $N = N \cup \{p\}$;
            else if $N.contains(p)$ then
                $X = X \cup p$;
                $\Gamma = \Gamma \setminus p$;
                add $p$ to $pq$;
$\Gamma = \Gamma \setminus update$
```

of the component to all vertices it represents. This means that vertices of one component will be processed with the same priority. Algorithm 5 shows one way to compute a component graph in linear time.

Now we are ready to compute the weak control closure as depicted in Algorithm 6. The vertices of V' are added to a priority queue according to their reverse topological order number. Then we process the vertices in that order, going through all their predecessors to check the predicate defined in the original algorithm (other than reachability from V', which we included in Θ.) However, as we need to account for a changing Θ based on the changes to X, we propagate a difference between a vertex' original value of Θ and the new value based on the changes to X, named Θ^{δ} in the algorithm. Here, we are only interested in propagating the differences in straight-line code. If a $\Theta(G, X, v)$ is not available for a particular X we resort to the value of $\Theta(G, X, v) - \Theta^{\delta}$. When a vertex is added to X we also add it to the priority queue.

3.1 Strong Control Closures

The original algorithms for computing weak and strong control closures only differ in a predicate based on the value of Γ. This is also reflected in our algorithm (see Algorithm 8), which only differs in this predicate and additionally needs to recompute Γ incrementally every time a new vertex is added to X. As the algorithm to compute Γ (depicted in Algorithm 7) modifies the control flow graph, our algorithm clones that graph to G^{Γ}. Again, determining Γ in reverse topological order allows to reduce the complexity class from $O(N^2)$ to $O(N \log N)$, as we have to walk through all vertices in that order. On top of that, this algorithm can be computed incrementally, such that when we add a vertex to X during the computation of the control closure, we only need to update the graph for this single element. This update may, however, modify the complete graph as demonstrated by the inner foreach loop. Yet, once all vertices have been processed, subsequent increments will have no more work to do and thus execute in constant time. So this algorithm has an amortized worst case complexity in $O(N \log N)$.

As mentioned earlier, Algorithm 8 only differs from Algorithm 6 in the predicate and code blocks that compute and update $\Gamma(G, X)$. Based on this observation, we conjecture an amortized complexity of $O(N(\log N)^2)$, as computing an increment of Γ is amortized logarithmic. This algorithm is therefore still fairly scalable.

4 Discussion

This paper presents an idea for two variants of an algorithm to compute weak and strong control closures. While there is still a fair amount of work to be done to show that the presented algorithms and worst case execution times are indeed correct, we are confident that this work sparks discussions in the community as it allows to use the more general algorithms for realistic size applications. This is however, not the end of this research. The new notion would need to be included into slicing algorithms to be useful for general applications like information flow control or program maintenance. In many cases, it might also be desirable to have a traditional relation of control dependence instead of only an algorithm to compute closures, as dependences are more local an thus allow compositional analysis. For the same reason a formulation of procedural analysis for control closures would be imperative.

Acknowledgement. This research was supported by the German Federal Ministry of Education and Research (BMBF) within the Center for IT-Security, Privacy and Accountability (CISPA) at Saarland University, and Deutsche Forschungsgemeinschaft (DFG Grant HA 6869/1-1) within the priority programme 1496 "Reliably Secure Software Systems – RS3".

References

1. Danicic, S., Barraclough, R.W., Harman, M., Howroyd, J.D., Ãkos Kiss, Laurence, M.R.: A unifying theory of control dependence and its application to arbitrary program structures. Theoretical Computer Science 412(49), 6809 – 6842 (2011), `http://www.sciencedirect.com/science/article/pii/S0304397511007377`
2. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1(1), 121–141 (Jan 1979), `http://doi.acm.org/10.1145/357062.357071`
3. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 49–61. POPL '95, ACM, New York, NY, USA (1995), `http://doi.acm.org/10.1145/199448.199462`

Algorithm 8: computing strong control closures

```
computeTheta($V$);
computeComponentGraph();
compute reverse topological sorting in the component graph;
assign each vertex the corresponding number of previous step;
$\mathit{\Gamma} = \emptyset$;
$G^{\mathit{\Gamma}} = G$;
$X^{\mathit{\Gamma}} = V$;
computeGamma($G^{\mathit{\Gamma}}, \mathit{\Gamma}, X^{\mathit{\Gamma}}, V$);
$pq$ = new PriorityQueue() sorted according to this number;
foreach $n : V$ do
    $n.\mathit{diff} = 0$;
    add $n$ to $pq$;
$X = V$;
while $pq$ is not empty do
    $v$ = remove vertex from $pq$;
    $\mathit{\Theta}^{\delta} = v.\mathit{diff}$;
    foreach $p \in predecessors(v)$ do
        if $p \notin X$ then
            if $(\mathit{\Theta}(G, X, p) - \mathit{\Theta}^{\delta} \geq 2 \lor p \in \mathit{\Gamma}) \land v \notin \mathit{\Gamma}$
            $\land(\mathit{\Theta}(G, X, v) - \mathit{\Theta}^{\delta} = 1 \lor v \in X)$ then
                $X = X \cup p$;
                $\mathit{diff} = \mathit{\Theta}(G, X, p) - 1$;
                $\mathit{\Theta}(G, X, p) = 1$;
                computeGamma($G^{\mathit{\Gamma}}, \mathit{\Gamma}, X^{\mathit{\Gamma}}, \{p\}$);
                $p.\mathit{diff} = \mathit{diff}$;
                add $p$ to $pq$;
            else
                $p.\mathit{diff} = \mathit{\Theta}^{\delta}$;
                add $p$ to $pq$;
return $X$;
```

Synchronized Symmetric Model-View-Controller

Michele Sama[1] and Franco Raimondi[2]

[1] PuzzleDev s.n.c.
Via A. Badiali 140, Ravenna, Italy
`m.sama@puzzledev.com`
[2] School of Science and Technology
Middlesex University, London, UK
`f.raimondi@mdx.ac.uk`

Abstract. In the past two decades the Model-View-Controller pattern has been employed successfully in the development of software systems. In this paper we argue that this model may be improved to support the development of applications running on multiple devices, possibly not always connected. Specifically, we introduce the notion of Symmetric Synchronized Model-View-Controller, in which multiple views and controllers are generated from a single model, and we propose the adoption of synchronization mechanisms based on ideas borrowed from the approaches developed for collaborative software (groupware) communication.

1 Introduction

The Model-View-Controller (MVC) pattern (see for instance the standard reference [1] for a detailed description) has been employed successfully in the past two decades for the development of software systems. A number of development frameworks and development environments currently in use support and encourage the adoption of this pattern. Notable examples include Apple XCode for the development of iOS applications and the open source frameworks Django [2] and Spring [3] for the development of Python and Java applications, respectively. Typically, in these frameworks developers specify a model and the code for the views and the controllers can be partially generated in an automatic way. For instance, a web-based application can be generated in Django by describing the model (essentially, an abstraction of the data model), and the corresponding views and controllers are automatically generated for clients using a browser[3].

Due to the increase in the number of mobile platforms, web based applications are required to be able to interact with a variety of clients, in addition to "standard", desktop-based clients. In many circumstances it may be required that mobile clients are also able to work off-line. A typical example are web-based calendars, which can be accessed by a (connected) desktop, and by a a range of

[3] `https://docs.djangoproject.com/en/dev/topics/class-based-views/generic-display/`

native mobile applications that make a local copy of the remote database and present the former when connectivity is not available.

The current trend in industry is to consider the server-side (together with its browser-based client) and the mobile platforms as different applications all together. In fact, in some cases these products are developed by different companies, specialising in different technologies, e.g. Django for the server-side[4], and the iOS SDK for the mobile clients. This means that developers need to work on multiple MVC patterns, one for each platform, which need to be kept aligned when a change is made in the server model, and with the additional issue of synchronization of the various local databases. As exemplified in the following sections, this causes substantial code repetition and makes validation and verification impractical.

In this paper we argue that server and all the mobile clients should be seen as a *single modular application*, in which the application for each platforms *depends* on the applications running in all the other platforms, and this dependency is exploited at the modelling stage to automatically generate code. Our idea is to extend the MVC pattern by introducing the notion of Symmetric Synchronized MVC (SSMVC):

1. Symmetric: extract views and controllers for multiple classes of devices from a single model (the server model).

2. Synchronized: automatically generate code for client-server synchronization using ideas borrowed from Computer Supported Cooperative Work (CSCW, e.g., DiscoTech [4])

We provide the details of this idea in the rest of the paper, which is organised as follows: in Section 2 we describe a scenario that is used throughout the paper as a running (and motivational) example, while our idea is described in Section 3. We provide related work in Section 4 and we conclude in Section 5.

2 A motivational example

In this section we describe a scenario that is obtained from the simplification of a real application: a law firm composed by a number of lawyers manages various customers concurrently. Lawyers operate using a range of devices: desktop PCs, mobile phones, tablets, etc., which sometimes need to work off-line due to lack of connectivity. The two key requirements for this application are: (1) Data must be always available, even when devices are off-line; (2) Data must be synchronized: if a change is made, this should be pushed to all the other devices as soon as a connection is available.

To develop this application, we can start from a framework such as Spring that provides a Java-based back-end and a standard web client implementation

[4] From now on we will omit the fact that server-side development includes also a browser client, when this is clear from the context.

using an MVC pattern. More in detail, the development process is similar to the following:

1. Develop the server-side model using a dedicated tool (this is typically a domain specific language, or a graphical tool).
2. Develop controller and view for the browser-based client: source code for these is normally generated automatically from the client-side model description, and the developers only need to introduce changes for the specific application.
3. Develop server-side views (and, when needed, a controller) for REST (or SOAP) API. These APIs are used by the native mobile clients that need to interact with the "main" server.

This process enables the creation of the web-based application that lawyers can use from their desk. However, in many cases lawyers need to work from remote locations (e.g. court, at clients' residence, etc.), and in many cases connectivity is not available (e.g. in tribunals). Suppose now we want to develop a native Android client to be used in these circumstances. Effectively, this results in the development of a new, separate application that interacts with the previous one by means of REST (or SOAP) APIs. The typical steps would be:

1. Develop the client-side model and persistence model. Android has currently no support for automated model and persistence generation. The implementation is completely left to developers which have to choose among various solutions including the Google App engine [5] and a Sqlite DB.
2. Develop the client-side view and controller (i.e. the user interface). As above, these may be partially generated automatically from the model. item Implement the client-side REST (or SOAP) API to synchronize with the "main" server.

Notice that all the issues associated with synchronization (e.g. pushing local changes or fetching new data) need to be resolved by the developer on the client-side API.

In addition to mobile phones running Android, a number of lawyers have access to tablets such as the Apple iPad. The implementation of a native iOS application would follow the same steps above using the iOS SDK, but for iOS clients there is another possibility that avoids the need of placing the iOS mobile client on Apple App Store: developing a iOS web app using the Javascript extensions of Safari mobile[5]. This can be considered as a Javascript-based mobile application in which the browser itself act as a sandbox for the application. The interaction with the "main" server in this case requires a server-side MANIFEST file listing all the resources required to execute the application. The browser drives the application life cycle with a sequence of events and allows the Javascript code to read and write external resources such as private property files or databases.

[5] This is the solution currently adopted by `http://www.ft.com`. Web apps can also be implemented using Chrome extensions, see below.

In this particular case, the whole client implementation is embedded in server-side views or files which are loaded by Safari mobile when the application bootstraps. More in detail iOS web apps require to:

1. Develop a Javascript client-side model.
2. Develop all the views and the Javascript controller.
3. Implement a Javacript client for the REST API. (The implementation of a Chrome web extension follows an identical pattern.)

Both the native Android/iOS development and the web app development presented above show that the current implementation strategies lead to a substantial duplication of code that is clearly interdependent, and to a number of inefficiencies:

– The MVC pattern is repeated in all clients.
– A change in the "main" (server-side) model requires the immediate change of all the clients' models, making the code hard to maintain. As an example, suppose that there is a change in the customers' records, e.g. by introducing an additional field in a table. The models of all the mobile applications need to be modified accordingly before being able to interact with the server again.
– For each mobile client, developers have to implement manually all the aspect of synchronization, for each possible platform.
– Testing the application (as a whole) is problematic: for instance, fixtures need to be developed *for each* possible client and for the "main" server.

3 Symmetric Synchronized MVC

In this section we describe SSMVC (Symmetric Synchronized MVC), our proposed extension of the MVC pattern.

MVC-based frameworks ask developers to implement model classes representing persistent stateful objects. Starting from those classes each framework is capable of creating (semi-)automatically the database schema, the REST API, detail and list views on each model. If the original model changes, the framework is also capable of (semi-)automatically updating the code [6]. However, as in the development case, this update operation is repeated on the server and on each client separately.

3.1 Symmetric MVC

One key point of our idea is to consider web-based and mobile clients as a *single application*. Following this idea the same framework that is generating the server-side database, views and controllers can also generate all the client code. Similarly changes in one of the models can be (semi-)automatically and simultaneously updated by the framework itself.

Notice that traditional client applications use API to synchronize with the server. As we stated in Section 2, both server and client endpoints have to

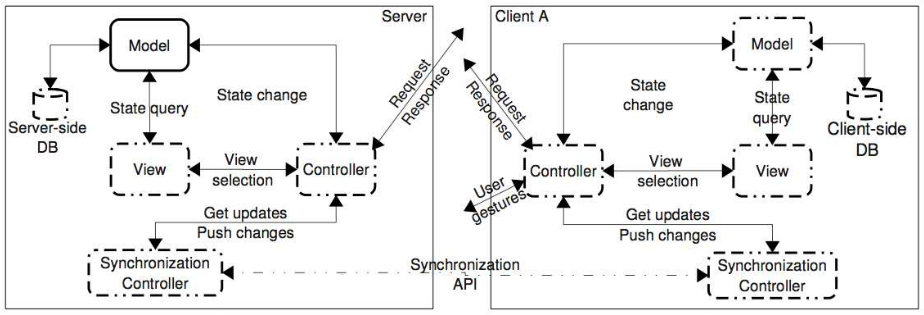

Fig. 1. Symmetric Synchronized MVC

be implemented. However, we argue that both of them can be automatically generated starting from the model.

Figure 1 depicts the SSMVC process, where the model in bold is provided by the user, and all the elements in dashed boxes are (semi-automatically) generated. More in detail, the generation process needs to:

1. extract the client-side model from the server model.
2. generate for each model the database schema and the platform-dependent persistence code (e.g. all the code containing the query to create, save and change models on the database).
3. generate on demand basic views and controller (e.g. a detail view on a model instance or a list view listing all the instances of a certain models).
4. configure a set of synchronization API on the server and their client counterpart to load data from the server or to push changes from the client.

Notice that most of these steps are *already* performed by existing MVC frameworks, but only for a single platform. Table 1 describes which part of the application are automatically generated by applying the MVC pattern to various server and mobile platform: the same steps can be repeated for SSMVC starting from a *single* model.

`Django-jom` is our initial prototype implementation of the SSMVC pattern for the Django framework. A detailed description of this implementation can be found at `https://github.com/msama/django-jom/wiki`. `Django-jom` stands for Django Javascript Object Models. The implementation is a Python Django component that automatically generates all the Javascript code necessary to export existing Django models for the server into Javascript objects for clients. Developers need to specify a descriptor indicating, for each model that they want to be exported, some basic properties, including which fields should be exported and how. Additionally, developers can implement a skeleton with additional prototypes that the Javascript objects should have at runtime. At this stage of the implementation `Django-jom` only handles the Symmetric behaviour. The Synchronization strategy described in the next section is currently being implemented.

Platform	Description
Server	Model: Manually implemented at abstract level. Persistence handled by the framework. View: Partially generated by the framework. HTML and CSS and custom views are manually added. Controller: Generated by the framework. Developer can add custom ones.
Web apps (Chrome/iOS)	Model: Javascript model and database connection are automatically generated. View: Partially generated by the framework. HTML and CSS and custom views are manually added. Controller: Default Javascript controllers are generated by the framework. Developer can add custom ones.
Native (iOS/Android)	Model: SQLite and core data are automatically generated as well as model classes. View: Automatic XML generation (Android); limited by XCode integration (iOS) Controller: Default controllers are generated by the framework. Developer can add custom ones.

Table 1. Automatic code generation in SSMVC

3.2 Synchronized MVC

Synchronizing client-server applications when clients can work both on-line and off-line is non-trivial. Traditional REST (or SOAP) API are designed to support the transfer of data when the client is on-line but they have no support for re-synchronizing clients operating off-line.

Consider for instance two lawyers updating the same data, one online and one off-line. The lawyer operating off-line will change data in the local database but will not be able to push changes to the server. When the lawyer returns on-line, the mobile client has to:

1. trace local changes which have not been pushed
2. pull an update from the server as soon as a connection is re-established
3. resolve possible conflicts on the locally modified data
4. resolve the conflict with a given policy (e.g. ask the user, merge or discard the changes)
5. push the update to the server
6. mark all the local changes as pushed to the server

None of the above features is automatically provided by the existing frameworks. Currently, implementations are left to developers which have to implement this mechanism for each model.

Our proposed Synchronized MVC looks at this problem from another perspective. Modern groupware implement a number of strategies to handle disconnection and reconnection of clients. For instance, the DiscoTech toolkit [4] provides an API to manage *event queues*, both at the client and at the server side. The specific instantiation of a strategy is application-dependent, but the toolkit provides a parametric framework that can be adapted by the application developer to manage the possible events of a particular application. Notice that given the model and given a framework supporting the synchronization issues, all the system code could be automatically generated. Figure 2 provides a high-level overview of the toolkit; we refer to [4] for additional details.

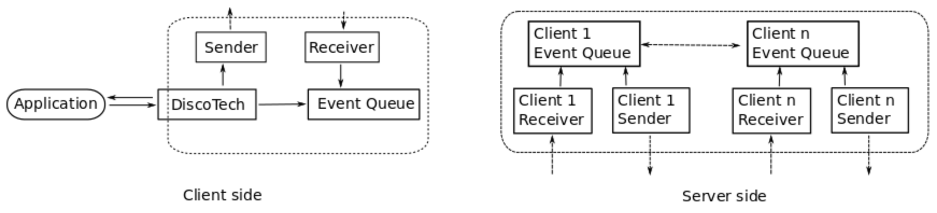

Fig. 2. Overview of the DiscoTech toolkit (from [4])

Conflict resolution is one of the most important aspect in terms of usability and user experience. Various systems could implement different policies according to their needs. For instance in our example we could apply a policy in which junior lawyers cannot override changes made by senior lawyers. The definition of such conflict resolution policies is out of scope for this paper but we remark that the SSMVC can accommodate these policies.

For the sake of completeness, there is an additional issue which needs to be addressed: changes in the model while the system is being used. This is a common issue in every data-driven application. Existing system such as iOS web applications employ a mechanism to check if the database version has changed and to trigger an update script. In SSMVC we can apply a more efficient solution. If the model (and, therefore, the database) changes, we can add the appropriate event to the queue of events, thus distributing it to all the clients as if it were a data change.

4 Related work

The MVC pattern is broadly used in industry and most of the existing web and mobile frameworks adopt it as base for the development. There has been some work in the extension of the MVC pattern. For instance Huang and Zhang [7] add an extra layer using XML and XSL. Their approach goes in the direction of using the MVC pattern for the server side application but does not include mobile clients. Other extensions include Flexible Web-Application Partitioning [8], targeting broswer-based web applications and MVC RIA (Rich Internet Applications) [9]. The latter work suggests splitting the model in client and server models, adopting browser plug-ins for interoperability of code, and mentions the use of a client-side persistence layer. However, the issue of synchronization is not addressed in this work.

More in general, the concept of synchronizing client applications, allowing users to access their data from multiple platforms, is becoming increasingly popular in industry. Server-side Chrome extensions [10] support client synchronization. Off-line changes are synchronized and distributed to all the clients employed by the same user (e.g. see the synchronization mechanism of Google Calendar for web browsers, phones, and tablets). However, there is no coherent solution: we found examples in which clients cannot make modifications if they bootstrap off-line, even when they go back on-line. Our pattern associates synchronization with concepts from CSCW, allowing multi-user concurrent pushes.

Other works exists in literature for mobile client synchronization. Shun at al. [11] propose a cache-like synchronization system. They introduce an intermediate state in which a client operating off-line has to re-synchronize before operating on-line. They also propose a coordination algorithm. The synchronization part of SSMVC is similar in spirit, but instead of proposing a new synchronization algorithm we suggest to use a Git-like protocol, and we also look at the global architecture of an application.

There is a substantial body of work on collaborative applications, see for instance [12] and references therein. The SSMVC pattern makes use of ideas from CSCW to address the issue of synchronization among multiple clients. Overall, however, our focus is different: the applications for which we suggest the use of SSMVC are not necessarily groupware applications (in the sense that multiple users collaborate to achieve a common goal), but could be applications in which a single user access data from a range of devices, not always connected. Additionally, we address the issue of *code dependency* and *code generation* for deployments on heterogeneous platforms, and to the best of our knowledge this is an issue that has not been addressed by the CSCW community.

5 Conclusion

The idea we propose here is a first step in the direction of considering backends, mobile applications and synchronization of user data as components of a *single* distributed and coordinated data-driven system. In this sense the SSMVC pattern that we propose in this paper aims at aiding web and mobile application developers by introducing a "design once and deploy everywhere" principle, with the additional benefits of avoiding code duplication, improving efficiency, testability, and maintainability. We achieve this by exploiting the *dependencies* between the various components of the overall system.

We consider our MVC extension *symmetric* because the same architecture is replicated both on the server and on each client platform. We consider our MVC extension *synchronized* because it gives developers a way to handle on-line and off-line concurrent changes by exploiting synchronization strategies developed for groupware applications that can be automatically generated.

We are currently developing a tool implementing model-to-model transformations to support SSMVC. A preliminary prototype implementing the Symmetric MVC pattern is available at `https://github.com/msama/django-jom/wiki`; in this prototype we generate Javascript code automatically from a server model for the Python-based Django framework. We believe that the SSMVC pattern could substantially improve the current approaches to web system development. We hope that this initial submission will provide feedback and comments on the feasibility of our idea, and suggestions for improvements to the process as a whole and to its single components.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
2. Django: `https://www.djangoproject.com/` Accessed: 26/06/2012.
3. Spring: `http://www.springsource.org/` Accessed: 26/06/2012.
4. Roy, B., Graham, T.C.N., Gutwin, C.: Discotech: a plug-in toolkit to improve handling of disconnection and reconnection in real-time groupware. In Poltrock, S.E., Simone, C., Grudin, J., Mark, G., Riedl, J., eds.: CSCW, ACM (2012) 1287–1296
5. Google App Engine: `http://developer.android.com/training/cloudsync/aesync.html` Accessed: 29/06/2012.
6. South: `http://south.readthedocs.org/` Accessed: 26/06/2012.
7. Huang, S., Zhang, H.: Research on improved MVC design pattern based on struts and xsl. In: Information Science and Engineering, 2008. ISISE '08. International Symposium on. Volume 1. (dec 2008) 451 –455
8. Leff, A., Rayfield, J.: Web-application development using the model/view/controller design pattern. In: IEEE EDOC'01, IEEE (2001) 118–127
9. Morales-Chaparro, R., Linaje, M., Preciado, J., Sánchez-Figueroa, F.: Mvc web design patterns and rich internet applications. Proceedings of the Jornadas de Ingenierıa del Software y Bases de Datos (2007)
10. Chrome web extension: `http://code.google.com/chrome/extensions/storage.html#apiReference` Accessed: 26/06/2012.
11. Shun-Yan, W., Luo, Z., Yong-Liang, C.: A data synchronization mechanism for cache on mobile client. In: WiCOM06. (2006) 1 –5
12. Avgeriou, P., Tandler, P.: Architectural patterns for collaborative applications. Int. J. Comput. Appl. Technol. **25**(2/3) (February 2006) 86–101