Information Flow and Concurrent Imperative Languages [⋆]

Damas P. GRUSKA

Institute of Informatics, Comenius University,
Mlynska dolina, 842 48 Bratislava, Slovakia,
gruska@fmph.uniba.sk.

Abstract. Information flow for concurrent imperative languages is defined and studied. As a working formalism we use UNITY, where programs consist of sets of assignments executed randomly, i.e. without control flow. We study noninterference for programs which reach and do not reach fixed point - a state which is not changed by a subsequent execution. We present logic formulation of noninterference as well as type system for it.

Keywords: information flow, noninterference, concurrent imperative language, fixed point, safety and progress properties, type system

1 Introduction

Several formulations of a notion of a system security can be found in the literature. Many of them are based on a concept of noninterference (see [GM82]) which assumes an absence of any information flow between private and public systems activities. More precisely, systems are considered to be secure if from observations of their public activities no information about private activities can be deduced. This approach has found many reformulations for different formalisms (Process algebras, Petri nets, simple imperative programming languages, see [SM03] for an overview, multi-threaded programs, see, for example, [SS00,VS98], and so on), for different computational models (labeled transition systems, deterministic models and so on) and for the nature or "quality" of observations (see, for example, [BKMR06,Gru07]).

Presence of information flow can be checked statically, i.e. on the program's code or system's description, or dynamically, i.e. at runtime. In the first case there are several methods, which include typing (see, for example [VSI96]), model checking (see, for example, [HN,HB11]).

The aim of the paper is to define and to study information flow in the case of concurrent imperative languages. Concurrency brings several new challenges with respect to sequential or parallel computations. There is no given control flow, there is no scheduler and program may not terminate. This brings new specific problems how to handle information flow. As a working formalism we take

[⋆] Work supported by the grant VEGA 1/1333/12.

UNITY programs with corresponding specification logic (see [CM88]). These programs consist of sets of assignments which are randomly chosen at runtime. Every run of UNITY program is infinite but it can happen that during the run a fixed point is reached, i.e. point when a subsequent computation does not change values of program variables, which somehow corresponds to termination. Moreover, another source of nondeterminism comes from randomly chosen values of variables if they are not specified at the beginning. On the other side, it makes no sense to study covert channels caused by specific scheduler policy since statements to be executed are chosen randomly (see [SS00,VS98]) and there is a natural way how to compose programs and how to study their composition properties. We will study noninterference for two groups of programs - those which always reach fixed point(s) (there might by more then one) and those which do not. We exploit techniques developed for imperative languages as well as for process algebras. Then we characterize some of these noninterference properties by UNITY logic and by type system.

Organization of the paper. In Section 2 we present UNITY semantics, syntax and logic. In Section 3 we present and investigate absence of information flow for programs which reach fixed points (what somehow correspond to termination) and in the next section we concentrate on non-terminating programs. Section 4 is devoted to logical characterizations of noninterference properties and Section 5 to the type system.

2 Unity

In this section we briefly recall a slightly simplified version of UNITY programs and the underlying logic for describing program properties. For more details see [CM88].

2.1 Syntax

UNITY programs consist of three parts: *declare*, *initially* and *assign* sections, while the second one is optional. Note that originally UNITY programs could contain also section *always* but this is optional and does not bring anything new for our purposes. Hence formally:

```
Program program-name
    declare declare-section
    initially initially-section
    assign assign-section
end
```

The *assign* section, which consists of set of statements - assignments, is the crucial part of a program. The statements are separated by □. Each statement consists of list of variables on the left hand side and list of expressions (of the same length) on the right hand side.

Expressions might by simple (examples: in $x, y := 3, z + 2$ on the right hand side there are the simple expressions) or conditional (in $x, y := 3$ if $y > 0 \sim 2$ if $y < 0, 4$ on the right hand side there is a conditional expression, the assignment assigns value 3 to x if $y > 0$, value 2 if $y < 0$ and leaves it unchanged if $y = 0$ and assigns value 4 to y). Formally:

```
assign-section ::= s{□ s}*
s ::= variable-list := expression-list
variable-list ::= variable{, variable}*
expression-list ::= expression{, expression}*
expression ::= simple-expression |conditional-expression
conditional-expression ::= simple-expression
                                    if boolean-expression
                                ~ simple-expression
                                    if boolean-expression
```

We assume that if more boolean expressions within one conditional expression are valuated as true then corresponding simple expressions have to be equal, i.e. every execution of every statement is deterministic.

Initially section assigns initial values to (some) variables. It is similar to *assign* section (we use here $=$ instead of $:=$) and we require that if a value of a variable is initialized then it is done uniquely. *Declare* section declares types of variables.

By $P \Box P'$ we denote composition of programs P and P', respectively. The sections of the resulting program are given by unions of corresponding sections of P and P'. It is assumed that there are no conflicts between their declare and *initially* sections, i.e. if a variable is declared in both programs, then it has to have the same type and if a variable is initialized in both programs, then it has to by initialized equally.

2.2 Operational semantics

The UNITY programs are executed in the following way: initial values of variables are given in initial section or they are randomly chosen within appropriate type if they are not initialized. By $Initially(Var)$ we denote set of variables which have initialized values by *initially* section. In general, $\emptyset \subseteq Initially(Var) \subseteq Var$, where Var is the set of all program variables. Then execution proceeds with randomly chosen statement, its execution and again new statement is randomly chosen and executed. This is repeated infinitely many times in such a way that every statement is executed infinitely many times. Hence an underlying semantics is given by the set of infinite sequences (runs) $R, R = R_0, R_1, \ldots$ where $R_i = (R_i.state, R_i.statement)$ i.e. each element of the run contains state - i.e. values of all variables (defined by mapping $v : Var \rightarrow v(dom)$) and the statement which is to be executed as the next. By $R.state$ we will denote the sequence of states of run R, i.e. $R.state = R_0.state, R_1.state, \ldots$. We assume that every statement of the program appears infinitely many times in every run. Note that

$R_{i+1}.state$ is uniquely determined by $R_i = (R_i.state, R_i.statement)$ since execution of any statement is deterministic. $R_0.state$ is an initial state. If initial values of all variables are given in initial section, then every run starts with this unique initial state. Otherwise there are many possible initial states depending on variables which are not initialized. For example, if there is only one boolean variable not initialized, then there are two possible initial states. By initial condition (predicate $Initially$) we mean condition which is given by initial section. Roughly speaking, it is a predicate obtained from *initially* section by replacing := by =, $\Box$ by $\wedge$ and conditional expression $x = e_1$ if $b_1 \sim \ldots \sim e_n$ if b_n is replaced by $(b_0 \Rightarrow (x = e_0)) \wedge \ldots \wedge (b_n \Rightarrow (x = e_n))$. For simplicity we assume that the variables have integer values. Note that considering other types of variables does not bring anything fundamentally new with respect to studied concepts and properties. We use 0 for false and nonzero for true. Formally, a state (we will alternatively use also denotation memory) is defined as mapping υ which assigns integer value to every variable, i.e. $R_i.state \equiv \upsilon : Var \rightarrow Z$. The formal definition of υ is the following:

$$\frac{x \in dom(\upsilon)}{\upsilon \xrightarrow{x:=e} \upsilon[x := \upsilon(e)]}$$

$$\frac{x \in dom(\upsilon)}{\upsilon \xrightarrow{x:=e \text{ if } b} \upsilon[x := \upsilon(e)] \text{ if } b}$$

$$\frac{x \in dom(\upsilon)}{\upsilon \xrightarrow{x:=e \text{ if } b} \upsilon \text{ if } \neg b}$$

$$\frac{\upsilon \xrightarrow{s} \upsilon', \upsilon' \xrightarrow{s'} \upsilon''}{\upsilon \xrightarrow{s.s'} \upsilon''}$$

We write $\upsilon \longrightarrow \upsilon'$ if there exists a sequence of statements $t = s_1 \ldots . s_n$ such that $\upsilon \xrightarrow{t} \upsilon'$.

2.3 Logic

Let p, q are predicates and s statement. The assertion $\{p\}s\{q\}$ denotes that execution of s in any state that satisfies predicate p results in a state that satisfies predicate q.

Let us consider program P which contains at least one statement. We write $s \in P$ if *assign* section of P contains s. We will use standard logical operators as $\vee, \wedge, \neg, \Rightarrow, \ldots$ and moreover, we define UNITY logic operators for safety properties *unless, stable, invariant, constant* and for progress properties *ensures, leads-to.* We start with basic safety property *unless* which is defined as follows.

$$p \ unless \ q \equiv \forall s, s \in P :: \{p \wedge \neg q\}s\{p \vee q\}$$

Derived operators are defined as follows: p is *stable* iff p *unless* $false$, p is *constant* iff both p and $\neg p$ are stable. Predicate p is *invariant* iff it is stable and $Initially \Rightarrow p$.

Now we can define two progress properties *ensures, leads-to* (denoted as $\mapsto$).

$$p \textit{ ensures } q \equiv \exists s, s \in P :: \{p \wedge \neg q\}s\{q\} \text{ and } p \textit{ unless } q$$

Leads-to is defined by a finite number of application of the following inference rules. The first inference rule says that ensures is a special case of leads to.

$$\frac{p \textit{ ensures } q}{p \mapsto q}$$

The second inference rule defines transitivity of $\mapsto$.

$$\frac{p \mapsto q, q \mapsto r}{p \mapsto r}$$

The third inference rule defines weakening (W is a set of indexes).

$$\frac{\forall m \in W :: p_m \mapsto q}{(\exists m \in W :: p_m) \mapsto q}$$

By fixed point we mean a state for which it holds that for each statement of the program valuations of variables on the left hand side and right hand side are equal, i.e. any subsequent execution of the program does not change state of the program. Formally,

$$FP \equiv \forall s \in P, \text{ if } s \text{ is "}X := E\text{" then } X = E$$

where X and E is variable and expression lists, respectively. In case of conditional expressions, if neither of conditions is evaluated to true, then the statement does not change the value of variable on the left hand side and we consider this as that the both sides are equal. By $true \mapsto FP$ we will denote that fixed point will be reached.

By v_p we denote that predicate p holds for valuation v. In special case valuation $v_{initially}$ has to satisfy initial condition, i.e. if variable x initializes its value to m than $v_{Initially}(x) = m$.

3 Non-interference for programs with fixed points

Suppose that the set of variables Var is divided into two subsets, namely the set of public (low level) variables Var_L and private (high level) ones Var_H. This division does not vary at runtime. We denote low level variables as $l, l', l_1, \ldots$, high level ones as $h, h', h_1, \ldots$ and constants as $c, c', c_1, \ldots$.

First we define termination-insensitive security (see for example [VS97]) for imperative programs based on an absence of information flow between private a and public variables.

Definition 1. *(BTNI) A deterministic program C satisfies batch-job termination-insensitive noninterference (BTNI) if, for any memory (valuation) υ_1 and υ_2 that agree on public (low) variables, the final memories υ'_1 and υ'_2 produced by running P also agree on public variables if both runs terminate successfully.*

Roughly speaking, BTNI property requires that programs represent functions for which low level outputs do not depend on high level inputs.

Examples of programs which clearly violate BTNI condition are $l := h$ (the explicit information flow) and $l := 1$ if $h = 0$ (the implicit information flow).

We cannot apply BTNI property for UNITY programs from the several reasons. They are undeterministic, there might have several initial states of computations and they do not terminate. So we have to elaborate new noninterference definition for UNITY programs.

First we need some preparatory work. Let υ_1 and υ_2 be valuations. We write $\upsilon_0 =_M \upsilon_1$ iff $\upsilon_0(x) =_M \upsilon_1(x)$ for every variable $x, x \in M$. We will write $\upsilon_0 =_L \upsilon_1$ instead of $\upsilon_0 =_{Var_L} \upsilon_1$ i.e. if valuations υ_0, υ_1 agree on low level variables. From now on we assume that high level variables are not initialized if it is not said otherwise.

Definition 2. *Program P is called fixed point deterministic with respect to M, $M \subseteq Var$ ($P \in FPD(M)$, for short) iff it always reaches fixed point and moreover this is unique for variables from M, i.e. values of variables from M are equal at any fixed point.*

Example 1. Let us consider assignment $l := l + 1$ if $l < l'$. For program P consisting of (only) this assignment and if both values l, l' are initialized then it is $FPD(L)$. If any of them, or both, are not initialized, then it is not $FPD(L)$ despite the fact that it always reaches fixed point but for different runs it can reach different fixed points. On the other side, initialization of all variables is not necessary condition to be fixed point deterministic even with respect to set which does not contain non-initialized variables.

Definition 3. *Program P satisfies fixed point deterministic noninterference (FPDNI, for short) iff it is FPD(L).*

For programs such that $Initially(Var) = L$ property *FPDNI* corresponds to $BTNI$ property (see Definition 1). If some low level variables are not initialized, then it is stronger. Sometimes it might be too strong since it requires that program reaches the same (from low level point of view) fixed point even for low level variables which initial values are chosen randomly. We could weaken *FPDNI* property by requiring that program is deterministic only with respect to initialized low level variables.

Definition 4. *Program P satisfies weak fixed point deterministic noninterference (WFPDNI, for short) iff it is $FPD(Initially(Var) \cap L)$.*

Lemma 1. *Let P satisfy fixed point noninterference then it satisfies also weak fixed point noninterference*

Proof. Clearly, if P satisfies fixed point noninterference then it is fixed point deterministic for all low level variables and hence also for any subset of low level variables. Note that an inverse of the lemma does not hold (see Example 1).

Requiring deterministicity is a strong requirement for concurrent programs. They often lead to different states (results) depending on factors which cannot be predicted (say, random factors). To capture noninterference for such cases requires a different approach. An intruder cannot learn nothing about initial value of high level variable h, if for any computation starting with some value of h there exists a computation starting with different value of h which leads to the same fixed point as regards low level variables. This leads us to the following definition.

Definition 5. *Program P satisfies fixed point noninterference with respect to $M, M \subseteq H$ ($P \in FPNI(M)$, for short) iff it always reaches fixed point and for every valuation v_0 for which it holds $Initially \Rightarrow v_0$ there exists v_0' for which it holds $Initially \Rightarrow v_0'$ such that $v(x) \neq v'(x)$ for every $x \in M$ and $v_0 \rightarrow v_{FP}$, $v_0' \rightarrow v_{FP}'$ and moreover $v_{FP} =_L v_{FP}'$.*

If program has $FPNI(M)$ then, then an intruder cannot learn (exact) value of any of its high level variables from M. For $FPNI(M)$ property we have the following lemma, which say that noninterference for a set of high level variables can be obtained separately from its subsets.

Lemma 2. $FPNI(M_1) \cap FPNI(M_2) = FPNI(M_1 \cup M_2)$.

Proof. By case analysis of all possible runs.

Lemma 3. *Let $M_1 \subseteq M_2$ then $FPNI(M_2) \subseteq FPNI(M_1)$.*

Proof. Directly from definition of $FPNI(M)$ property.

We conclude this section by comparing above proposed noninterference properties by the following proposition.

Proposition 1. *$FPDNI \subset WFPDNI \subset FPNI(M)$ for every $M, M \subseteq H$.*

Proof. Main idea. One part of the proposition comes from Lemma 1. The other part comes from the fact that we can always find different values of high level variables (thanks to their domain and not being initialized as it is assumed at the beginning of this section) for initial states and deterministic programs leads to unique fixed point form low level point of view. The following example shows that also this inclusion is proper.

Example 2. Let us consider program P.

```
Program P
    initially l :=c
    assign $l := l + 1$ if $l < l'$ $\sim h$ if $l \geq l'$
end
```

The program always reaches the fixed point. Then it holds $l = h$ if $c \geq l'$ and $l = l'$ otherwise. Since we do not know an initial value of l' we do not know neither value of h. Hence $P \in FPNI(\{h\})$ but $P \notin WFPDNI$ since the final value of l is not uniquely determined.

4 Non-interference for programs without fixed points

In this section we will deal with programs which do not reach any fixed point(s) and to which the previous definitions of noninterference cannot be applied.

Definition 6. *Let P be program and let R be a run of P such that $R.state = \upsilon_0, \upsilon_1, \ldots$. We say that P satisfies non-fixed point noninterference ($P \in NFPNI$, for short) if for every υ'_0 such that $\upsilon_0 =_L \upsilon'_0$ there exists run R' such that $R'.state = \upsilon'_0, \upsilon'_1, \ldots$ such that $\upsilon_i =_L \upsilon'_i$ for every i.*

$NFPNI$ corresponds to an observer who can see values of public variables at every state but cannot deduce from them values of private variables.

Proposition 2. *Let $P \in NFPNI$, $Initially(Var) = L$, and $true \mapsto FP$ is a property of P. Then $P \in FPNI(M)$ for every $M, M \subseteq H$.*

Proof. Main idea. Let $P \in NFPNI$ and $true \mapsto FP$ is a property of P. Since all low level variables are initialized and we can emulate on low level every two runs then we get that $P \in FPNI(M)$ for every $M, M \subseteq H$.

Now we can formulate a compositional property for two UNITY programs which are $NFPNI$.

Proposition 3. *Let $P, P' \in NFPNI$. Then it holds that $P \Box P' \in NFPNI$*

Proof. Sketch. Program $P \Box P'$ is obtained by set unions (with same preliminary requirements on conflicts - see Section 1 and under our assumption that variables from Var_H are not initialized). Hence if we have a run of $P \Box P'$ then it corresponds to interleaving of executions of assignments of P and P'.

Special case of unity programs are those which are low level deterministic, i.e. there is no choice between statements which change low level variables. For such programs it is meaningful to define stronger variant of *NFPNI* property which requires that all runs agree on low level variables. Formally:

Definition 7. *Let P be program and let R be a run of P such that $R.state = \upsilon_0, \upsilon_1, \ldots$. We say that P satisfies strong non-fixed point noninterference ($P \in SNFPNI$, for short) if for every υ'_0 such that $\upsilon_0 =_L \upsilon'_0$ then for every run R' such that $R'.state = \upsilon'_0, \upsilon'_1, \ldots$ it holds $\upsilon_i =_L \upsilon'_i$ for every i.*

Lemma 4. *$SNFPNI \subseteq NFPNI$. Moreover, if P is low level deterministic then $P \in SNFPNI$ iff $P \in NFPNI$.*

Proof. Sketch. The first part of the proof is straightforward. If P is low level deterministic, since there is no choice between runs from low level point of view and there exists at least one which is identical (from low level of view) to given run, we have that $P \in SNFPNI$ iff $P \in NFPNI$.

5 Logic characterization of noninterference

In this section we will define characterization of some noninterference properties by UNITY logic. Let P be a program, $l_1, \ldots, l_n$ are its (all) low level variables and υ is its valuation. By υ_K, where $K = (k_1, \ldots, k_n)$, we denote predicate $\upsilon(K) \equiv (\upsilon(l_1) = k_1) \wedge \ldots \wedge (\upsilon(l_n) = k_n)$.

Proposition 4. *Let P be a program and υ, υ' its two valuations. Then P has FPDNI property iff $(true \mapsto FP) \wedge ((\upsilon_{FP} \wedge \upsilon'_{FP} \wedge \upsilon(K) \wedge \upsilon(K')) \Rightarrow (K = K'))$.*

Proof. Sketch. Let P be a program with FPDNI property. The property requires that it reaches fixed point ($true \mapsto FP$) and that valuations on any fixed points are equal with respect to low level variables. This property is expressed by $(\upsilon_{FP} \wedge \upsilon'_{FP} \wedge \upsilon(K) \wedge \upsilon(K')) \Rightarrow (K = K')$.

Proposition 5. *Let P be a program. Then P has NFPNI property iff $((\upsilon_1 =_L \upsilon_2) \wedge (\upsilon_1 \mapsto \upsilon'_1)) \Rightarrow (\upsilon_2 \mapsto \upsilon'_2) \wedge (\upsilon'_1 =_L \upsilon'_2)$ for some υ'_2.*

Proof. Main idea. $NFPNI$ property requires that for every run and every its state there exists a run with the same initial state (up to values of high level variables) which reaches the same states (up to values of high level variables).

Proposition 6. *Let P be low level deterministic. Then $P \in SNFPNI$ iff $\upsilon =_L \upsilon'$ is stable for every pair of valuations υ, υ' of P.*

Proof. Main idea. Since P is low level deterministic every two runs are equal for low level variables, i.e. every two valuations which once become equal for low level variables remain so, i.e. equality is stable.

6 Type system

Before we define type system which can capture NFPNI property, we need some preparatory work. We begin with program equivalency. Note that due to mechanism of choosing statements for execution, we do not need to work with bisimulation or other branching sensitive relations - at any point of any computation every statements could be executed as the next one.

Definition 8. *We say that programs P and P' are equivalent (denoted $P \simeq P'$), if for every run R of P there exists a run R' of P' such that $R_i.state = R'_i.state$ for every i and vice versa.*

Two equivalent programs exhibit exactly the same behaviour and are indistinguishable from semantical point of view.

The next lemma allow us to simplify program syntax while increasing number of statements.

Lemma 5. *For every program P there exists equivalent program P' such that it contains only one conditional expression on the right side of every assignment.*

Proof. Sketch. We replace (split) every assignment with more conditional expressions with a set of assignments with only one (of) conditional expression. It is easy to prove that resulting program is equivalent to the original one.

Lemma 6. *Let $P \simeq P'$ and $P \in NFPNI$ then also $P' \in NFPNI$.*

Proof. Sketch. Two equivalent programs exhibit the same semantical behaviour and hence satisfy the same semantically defined properties.

Lemma 7. *Let $P \in NFPNI$ iff $P' \in NFPNI$ where P' is obtained from P by replacing every statement of P of the form $x_1, \dots, x_n := e_1, \dots, e_n$ by the statement $x_1 := e_1 \Box \dots \Box x_n := e_n$ (we will call P' sequentialization of P).*

Proof. Sketch. The difference between P and P' is that the first one can do parallel assignments while the second one cannot, but it can still emulate any parallel assignments by a sequence of sequential ones.

Let us consider the following type system for UNITY programs. Here c stands for a constant (it is a special case of expression e), *op* for an operational symbol which is allowed in expressions. We use two types *High* and *Low*.

$$\frac{}{\vdash h : High} \qquad \frac{}{\vdash l : Low}$$

$$\frac{}{\vdash c : Low} \qquad \frac{\vdash e_1 : T, \vdash e_1 : T}{\vdash e_1 \; op \; e_2 : T}$$

$$\frac{\vdash e : Low}{\vdash e : High} \qquad \frac{}{[High] \vdash h := e}$$

$$\frac{\vdash e : Low}{[Low] \vdash l := e} \qquad \frac{\vdash e : Low, \vdash b : Low}{[Low] \vdash l := e \text{ if } b}$$

$$\frac{\vdash e : T, \vdash b : T}{[High] \vdash h := e \text{ if } b} \qquad \frac{[High] \vdash s}{[Low] \vdash s}$$

$$\frac{[Low] \vdash s \text{ for every } s \in P}{[Low] \vdash P}$$

Type system for UNITY programs.

Proposition 7. *(Soudness) Let P be a UNITY program. If typing judgment $[Low] \vdash P'$ can be derived where P' is sequentialization of P then $P \in NFPNI$.*

Proof. Main idea. Thanks to Lemmas 5,6 and 7 we can assume that P does not contain more conditional expressions on the right hand side and only one variable on left hand side. If typing judgment $[Low] \vdash P'$ can derived then there are no assignments to low level variable which right hand side depend on high level variable. Hence if we have a run $\upsilon_o, \upsilon_1, \dots$ then regardless on values of high level variables there exists run $\upsilon'_o, \upsilon'_1, \dots$ such that they agree on low level variables.

Example 3. Let us consider program P and P' which consists of one statement $h := 1 if l = 1$ and $l := 1 if h = 1$, respectively. Then we have $[Low] \vdash P$ but $[Low] \vdash P'$ cannot be derived.

We cannot obtain complete type system since we would need to differ between high type expressions which are constant and which really depend on high level variables. For example, $l := h - h$ and $l := c$ if $l = h \sim c$ if $l \neq h$ is not low level assignments while "semantically" it is since a value of l does not depend on a value of h.

Note that the similar type systems can be applied also for other above defined properties as *SFPNI* and *FPDNI.* Moreover if program contains, for example, statements of type $h := c$ for every high level variable then initial values of those variables cannot be deduced from any fixed point. In this way we could extend the presented type system by introducing new type *Const* to cover this case.

7 Conclusions and further work

In this paper we have defined and studied information flow for concurrent imperative languages. We had to deal with four challenges - an absence of control flow, possibly more initial states, an absence of a scheduler which schedules which statement is to be executed as the next and with possibly nonterminating behaviour. Each of thess properties causes that, in general, programs cannot be treated as functions (they are not deterministic and do not output "final values"). Our working formalism - UNITY programs and UNITY logic are very simple but still can capture various noninterference properties. On the other hand we can exclude covert channels produced by specific scheduler policies.

Concurrent imperative languages are something between simple imperative languages and models of concurrency as labeled transition systems or process algebras, as regards treatment of noninterference. If they are "deterministic" and they "terminate" (in a sense) then they could be handled as former ones but if they are undeterministic and/or do not terminate as later ones. Hence we have exploited "noninterference" techniques developed for both imperative languages as well as for process algebras. We have also characterized some of these noninterference properties by UNITY logic and by type system.

Regarding the further work, we consider to exploit an idea studied in process algebra setting, namely property called Non-Deducibility on Composition (NDC for short, see in [FGM03]). Program P would have property NDC if for every program A which changes only high level variables, i.e. υ_K i stable for every K, then $P \Box A \simeq P$.

In [BBM94] for nondeterministic languages authors associate with a program variable x a set of called security variable of x, denoted $\bar{x}$. It is the set of all variables whose values can influence the value of x (directly or undirectly). This approach could be used also for UNITY programs.

Another possible direction of a further research might be quantification of information obtained from low level variables. For example, program $l := 1$

if (h mod 2) $= 1 \sim 0$ if (h mod 2) $\neq 1$ cannot leak exact value of h but it leaks via l parity of h. By similar techniques as it was done in [CHM07] we could try to quantify an amount of information on high level variables obtained from low level ones via information theory.

References

[BBM94] Banatre C., C. Bryce and D. Le Metayer: Compile-time Deytection of Informatioin Flow in Sequential Programs. European Symposium on Research in Computer Security, LNCS 875, Springer, Berlin, 1994.

[BKMR06] Bryans J., M. Koutny, L. Mazare and P. Ryan: Opacity Generalised to Transition Systems. In Proceedings of the Formal Aspects in Security and Trust, LNCS 3866, Springer, Berlin, 2006.

[CHM07] Clark D., S. Hunt and P. Malacaria: A Static Analysis for Quantifying the Information Flow in a Simple Imperative Programming Language. The Journal of Computer Security, 15(3). 2007.

[CM88] Chandy, K. M. and J. Misra: Parallel Program Design. Addison-Wesley 1988.

[FGM03] Focardi, R., R. Gorrieri, and F. Martinelli: Real-Time information flow analysis. IEEE Journal on Selected Areas in Communications 21 (2003).

[HN] Huisman M. and T.M. Ngo: Scheduler-specific Confidentiality for Multi-threaded Programs and Its logic-based Verification. In proceedings of Formal Verification of Object-Oriented Systems (FoVeOos 2011). LNCS, Springer. To appear.

[HB11] Huisman, M. and H.-C. Blondeel: Model-checking Secure Information Flow for Multi-Threaded Programs. In Proceedings of Theory of Security and Applications (Tosca) LNCS 6993, pages 148 - 165 , Springer, 2011.

[GM82] Goguen J.A. and J. Meseguer: Security Policies and Security Models. Proc. of IEEE Symposium on Security and Privacy, 1982.

[Gru09] Gruska D.P.: Quantifying Security for Timed Process Algebras, Fundamenta Informaticae, vol. 93, Numbers 1-3, 2009.

[Gru08] Gruska D.P.: Probabilistic Information Flow Security. Fundamenta Informaticae, vol. 85, Numbers 1-4, 2008.

[Gru07] Gruska D.P.: Observation Based System Security. Fundamenta Informaticae, vol. 79, Numbers 3-4, 2007.

[SM03] Sabelfeld, A and A.C. Myers: Language-Based Information-Flow Security, IEEE Journal on Selected Areas in Communications, 21(1):5-19, January 2003.

[SS00] Sabelfeld, A and D. Sands: Probabilistic Noninterference for Multi-Threaded Programs, Proceedings of the 13th IEEE workshop on Computer Security Foundations, 2000.

[Sch00] Schneider, F. B.: Enforceable Security Policies. ACM Transactions on Information and System Security, Volume 3 Issue 1, Feb. 2000.

[VS98] Volpano D. and G. Smith: Secure Information Flow in a Multi-threaded Imperative Language, Proc. 25th ACM Symposium on Principles of Programming Languages, pp. 355-364, San Diego, CA, Jan 1998.

[VS97] Volpano D. M. and G. Smith: Eliminating Covert Flows with Minimum Typings. In Proc. CSFW, 1997.

[VSI96] Volpano D., G. Smith and C. Irvine: A Sound Type System for Secure Flow Analysis, Journal of Computer Security, Vol. 4, No. 3, pp. 167-187, 1996.