Integrating Information Flow Analysis in Unifying Theories of Programming

Chunyan Mu ✉
Department of Computing and Games
Teesside University, UK
Email: c.mu@tees.ac.uk

Guoqiang Li ✉
School of Software
Shanghai Jiao Tong University, China
Email: li.g@sjtu.edu.cn

***Abstract*—This paper presents a formal approach for modelling and reasoning about information flow control in software systems under Hoare and He's Unifying Theories of Programming (UTP). We investigate the problem of integrating information flow control into system design in a unified semantic setting. Our approach can therefore treat information flow analysis and control in various families of specification languages and programming paradigms in a more general way. In addition, we formalise the link between classes of predicates as a paired function which maps set of the predicates from one class into set of the predicates from the other with a concern of flow security preservation. The proposed flow-sensitive combined theories of multiple level classes of predicates can be applied to ensure flow security in different paradigms under stepwise development.**

***Keywords*-information flow, language-based security, formal method, UTP, refinement**

I. Introduction

Traditionally, computer security has been largely enforced at the level of the operating system via access control policies, which are designed to restrict access to information, but cannot regulate information propagation once the information is accessed. *Information flow control* is required to prevent information leakage and violations during user program execution, and to defend against attacks from the software application level. Software applications are typically specified in programming languages, so it is natural to express information flow security policies and enforcement mechanisms at the programming language level, too.

There are a large number of programming languages, which can be classified in terms of the paradigms they support, such as imperative, object-oriented and real-time. Unification of theories study the relations and variations among the paradigms. Specifically, *observations*, denoted by the *alphabet* of the theory, are used to describe how the primitive concepts of the theory relate to the real world; *symbols*, denoted by the *signature* of the theory through a set of operators and atomic components, are used to combine the primitive statements of the theory into more complex descriptions of products; finally, in order to describe the results of a theory, *laws*, described as a set of *healthiness conditions*, are specified to provide mathematical support for the design of programs and prediction of the execution results. Theories are then unified by the sharing elements of their alphabet, signature and laws.

In general, the unifying theories of programming (UTP) [1] provide a modelling framework for different programming paradigms. The uniform underlying modelling framework allows the comparison and combination of constructs, notations and techniques from different theories. Fundamentally, the semantics of UTP theories is denotational, with algebraic (refinement) laws being proved based on the denotational model. Refinement is used as the primary verification technique, and laws enable the application of complex refinement strategies. The UTP studies the hierarchy structures of such theories and their relationships to provide a foundation for analysis of semantic features of programming languages, towards their combination and unification. Such a theory explores the underlying principle and combines theories of one or more programming languages at different concretisation levels and provides a deeper understanding in a more general setting.

The notion of secure information flow specifies the security requirements of the system such that secret information does not influence publicly observable information. An ideal flow policy called Non-interference (NI) [2] is a guarantee that no information about the sensitive inputs can be obtained by observing a program's public outputs, for any choice of its public inputs. This paper aims to treat information flow analysis of programming languages in the UTP setting. The main contribution of this work is two-fold. First, we have built the framework for reasoning about information flow in the UTP and integrated encoding flow policies into the UTP. Second, we have constructed a flow-security-preserving connection between different levels of predicate classes accounting for the ordering of flow-sensitive predicates.

Integrating flow policies into system design in the UTP setting allows us: to provide a justification of flow control mechanisms in programming languages; and to investigate combined theories of multiple levels of languages for the purpose of secure flow analysis, so that it can be applied to a family of specification languages and programming paradigms. Specifically, we can integrate the control flow of security properties into a formal stepwise development procedure: constructing flow-secure abstract specification which is enforced by the flow-healthy semantic rules, then performing a series of flow-security-preserving stages with increasing strength conditions.

II. Preliminaries

This section presents preliminary material of the UTP in Section II-A, and reviews approaches to static information flow analysis, particularly typing systems, in Section II-B.

A. Unifying Theories of Programming

Unifying Theories of Programming [1] is a mathematical framework for describing and unifying semantic descriptions of programming languages and modelling notations within the same descriptive environment of the alphabetised relational calculus. A UTP theory consists of an *alphabet* of variable names, a *signature* of language constructs (syntax), and a set of *constraints* called healthiness conditions. Relations are encoded by alphabetised *predicates* that contain additional information about the relation's alphabet. *Predicates* are used to describe the observable behaviour of the executions of the program, and therefore give a formal meaning to it. In general, predicates are also used to specify the requirements to describe the desired behaviours of the system.

1) Alphabet and observable behaviours: The alphabet of each theory contains variables relevant to the description of its programs and paradigm. An *alphabet A* is a collection of names referring to variables whose values are relevant to characterise system behaviour within a given paradigm. An *observation* is an interpretation or a state of a program. An observation of an execution of a particular program will give both the values observed before and after. Particularly, undecorated variables are used to record the initial value, and dashed variables are used to store intermediate or final observations. For instance, $(x = 6 \ \wedge \ x' = 3 \ \wedge \ y = 3 \ \wedge \ y' = 3)$ is an observation of a particular run of the program $x := x - y$. The underlying UTP theory selects the appropriate and relevant subset of variables to represent intended behaviours.

Specifically, one can view the alphabet as a collection of information objects. Information flows among the objects during the execution of a program. Observable behaviours thus present the flows and can cause information leakage in terms of a given flow policy.

2) Signature and predicates: UTP theories are characterised by subsets of predicates that describe the possible observations that can be made regarding program behaviour. The signature of a theory captures the language syntax. The meaning of every program is given as a predicate restricted to the selected alphabet. Those predicate sets can be specified by healthiness conditions, and interpreted independently as a separate closed theory. Healthiness conditions formalise constraints on the semantic model. Only predicates that satisfy the healthiness conditions of a theory are considered as valid models of computations within that theory. It is generally acknowledged that we can integrate the flow policy into the healthiness conditions of a theory to ensure the model behaviours being both secure and healthy.

3) Linking theories: A theory of programming is developed through a series of Refinement is a central concern of the UTP. A program P is refined by a program Q, written as $P \sqsubseteq Q$, iff $Q \Rightarrow P$ for all possible values of the variables in the alphabet. A set of theories with the refining ordering can thus form a hierarchy structure. The hierarchy structure of a set of theories and their relationships provides a foundational for analysis of semantic features of programming languages, towards their combination and unification. The set of predicates defined in theory at each level can be viewed as a subset of those of the previous theory at an upper level. A family of such related sub-theories at different levels of the structure are linked up. Specifically, the link between theories at different levels is defined as a function which maps all predicates from one theory into a subset of the predicates from the other. Galois connection is widely used for justification within UTP theories as a means to enable the description of formal links between a variety of paradigms.

In general, unifying theories can be used to formalise and analyse flows in programming languages and to provide a justification for a variety of theories. The most fundamental problem investigated in this paper is a framework of the unifying theory of programming languages and the corresponding linking theories with a specific concern of information flow analysis and control.

B. Static information flow analysis

Information flow security is concerned with how secure information is allowed to flow through a computer system. The flow is considered *secure* if it accepts a specified policy. A program is considered secure if all the flows in the program satisfy the policy. Information flow policies [2], [3] are end-to-end security policies, which provide more precise control of information propagation than access control models.

The security policies can be categorised into two core principles: confidentiality and integrity. This paper concentrates on confidentiality problems. The *confidentiality* policies require that secret information does not influence publicly observable information. Such policies constrain who can read the data and where the secret data will flow to in the future: information may only flow up the confidentiality lattice. The goal of confidentiality policies is to ensure that secret data does not influence public data. An ideal confidentiality property called *non-interference* (NI) [2] is a guarantee that no information about secret inputs can be obtained by observing a program's public outputs, for any choice of its public inputs. Intuitively, the NI policy requires that low security users should not be aware of the activity of high security users and thus not be able to deduce any information about their behaviours.

Specifically, the data manipulated by a program can be typed with security levels [4], [5], which naturally assume the structure of a partially-ordered set as a lattice under certain conditions. Security type systems have been substantially used to formulate the analysis of secure information flow in programs. Sensitive information was stored in programming variables, the powerset of program variables thus forms the universal lattice in terms of the ordering of their security levels. The flow-sensitive types system was defined by a family of inference systems which is forced to satisfy a simple non-interference property. To justify the flow-sensitive

typing systems and to preserve the flow security under related paradigms, a theory is required to do so.

III. Predicate Calculus with Flow Control

This section investigates the problem of formalising and integrating information flow behaviours and policies into the predicate calculus.

A. Observations and alphabet with security types

For our purpose of constructing a UTP theory for integrating secure information flow control, we need to record both the security levels and values of program variables in the observation of the program. We assign each variable in the alphabet a security type, the set of the variables therefore forms a complete lattice $\mathscr{L}_\tau = (L_\tau, \leq_\tau)$ induced by their security types, where L_τ is a set of security levels, $\leq_\tau$ defines the partial ordering of the security levels. We therefore need four names for each variable x in the alphabet: the initial and final values of x will be recorded under name x itself and the dashed name x' respectively, the initial and final security levels of x will be recorded under name x_τ and x'_τ respectively. Without loss of generality, we assume the security level of an undefined variable is $\bot$ (system low). Given an expression e (including boolean expression), $\tau(e) = \bigsqcup_{x \in \mathtt{fv}(e)} \tau(x)$ denotes the security level of e, which is the upper bound of the security levels of the free variables in e. An observation of a single completed execution of a program will therefore give both the values of program variables and the security levels observed, before and after.

In general, a predicate P can be viewed as a set of observations, we write a flow sensitive observation of (a particular run of) P, denoted by ω, as:

$$\omega = \bigwedge_{x \in in\alpha P} (x = v \wedge x' = v' \wedge x_\tau = t \wedge x'_\tau = t') \in P,$$

where $v, v' \in \mathbb{N}$, $t, t' \in L_\tau$.

Example 1: For instance, assume we have two variables x and y, and $\mathscr{L}_\tau = (\{H, L\}, \leq_\tau)$ $(L \leq_\tau H)$, and $\tau(x) = L$, $\tau(y) = H$, then:

$$\begin{aligned} \omega_1 &= x = 6 \wedge x' = 4 \wedge x_\tau = L \wedge\ x'_\tau = H \wedge \\ &\quad\ y = 2 \wedge y' = 2 \wedge y_\tau = H \wedge y'_\tau = H \\ \omega_2 &= x = 6 \wedge x' = 3 \wedge x_\tau = L \wedge\ x'_\tau = H \wedge \\ &\quad\ y = 3 \wedge y' = 3 \wedge y_\tau = H \wedge y'_\tau = H \end{aligned}$$

can be two observations of two particular runs of the program $x := x - y$. Intuitively, the assignment operation causes sensitive information stored in y to leak to the low security variable x so that the security level of x raises up to H after the execution. ■

We consider projections of the observations for the purpose of flow analysis. For instance, consider $t \in L_\tau$, the t-projection of an observation can be considered as: the conjunction of the valuation of variables whose undashed security level is $\leq t$ remains, and all of the valuation of variables whose undashed security level $> t$ have been removed from the original observation. For $t \in L_\tau$, we say two observations are t-equivalent if and only if: the dashed part of their t-projections are equal to each other only if the undashed part of their t-projections are equal to each other.

Definition 1 (t-projection of an observation): Given a predicate P, and a security level $t \in \mathscr{L}_\tau$, the t-projections of an observation $\omega \in P$ is given as:

$$\omega \restriction_t = \bigwedge_{x \in in\alpha P \wedge \tau(x) \leq t} (x = v \wedge x' = v')$$

where $v, v' \in \mathbb{N}$ denote the undashed and dashed valuation of x respectively. In addition, we use $\omega(x)$ $(\omega(x'))$ to denote the undashed (dashed) valuation part of an observation, i.e.,

$$\begin{aligned} \omega \restriction_t (x) &= \bigwedge_{x \in in\alpha P \wedge \tau(x) \leq t} (x = v), \\ \omega \restriction_t (x') &= \bigwedge_{x \in in\alpha P \wedge \tau(x) \leq t} (x' = v'). \end{aligned}$$

Furthermore, we use $P \restriction_t$ to denote the t-projection of predicate P, $P \restriction_t (x)$ and $P \restriction_t (x')$ to denote the undashed and dashed part of t-projection of P respectively.

Definition 2 (t-equivalent observation): Given a predicate P, and a security level $t \in \mathscr{L}_\tau$, let $\omega_1 \in P$ and $\omega_2 \in P$ be any of two observations. We say ω_1 and ω_2 are t-equivalent, denoted by $\omega_1 \approx_t \omega_2$ iff for all $x \in \alpha P$:

$$\omega_1 \restriction_t (x) = \omega_2 \restriction_t (x) \Rightarrow \omega_1 \restriction_t (x') = \omega_2 \restriction_t (x').$$

Definition 3 (t-equivalent predicate): Given two predicates P and Q such that $\alpha P = \alpha Q = A$, and a security level $t \in \mathscr{L}_\tau$, we say P and Q are t-equivalent, denoted by $P \approx_t Q$ iff for all $\omega_P \in P$ and $\omega_Q \in Q$: $\omega_P \approx_t \omega_Q$, i.e.,

$$\forall x \in A.P \restriction_t (x) = Q \restriction_t (x) \Rightarrow P \restriction_t (x') = Q \restriction_t (x').$$

Definition 4 (Flow secure predicate): Given a predicate P and a level $t \in L_\tau$, we say P is *t-flow secure* written as $P \models \phi_t$, *iff*:

$$\forall \omega_1, \omega_2 \in P. \omega_1 \approx_t \omega_2.$$

This condition is quite restrictive, but is intuitive to derive from the original definition of non-interference.

Example 2: Consider again Example 1, for $\tau(y) = H > L$, the presented two observations are with respect to two undashed valuations of $y = 4$ and $y = 3$. Clearly $\omega_1 \not\approx_L \omega_2$, the program does not satisfy the flow security condition ϕ_L and is not flow secure. ■

B. Signatures

The *signature* of a programming theory defines the *syntax* of the programming language, i.e., the meaning of each program in the language as a predicate, with free variables restricted to the alphabet of the language. Table I presents the syntax of the language considered in this paper.

Exp	$e ::= x \mid \mathbb{N} \mid e \oplus e \quad (\oplus = \{+,-,*,/,\%\}, x \in A)$
Bexp	$b ::= \texttt{true} \mid \neg b \mid b \wedge b \mid e \bowtie e \quad (\bowtie = \{>,\geq,<,\leq,=\})$
$x := e$	assignment of the value of expression e to the variable x
$x :\in S$	assignment of an arbitrary value from the set S to the variable x
$x_1,\ldots,x_n := e_1,\ldots,e_n$	concurrently assigning values of $e_1,\ldots,e_n$ to variables $x_1,\ldots,x_n$
$P;Q$	sequential composition: Q is executed after P has terminated
$P \lhd b \rhd Q$	conditional: P is executed if b is true initially, otherwise Q
$P \sqcap Q$	non-determinism: P or Q is executed without specifying which
var x	declaration: introduce a new variable of x
end x	undeclaration: terminate the scope of the variable x
$\mu X \bullet F(X)$	call a recursive procedure which has name X and body $F(X)$

TABLE I
SIGNATURE OF THE LANGUAGE

C. Flow-sensitive relational predicate calculus

We now specify our flow-sensitive predicate calculus for programs in the above language. For a program in such a language, the relevant observations comes in paired-pairs, with one observation of the *values* and the *security types* of all global variables before program execution, and one observation of their *values* and *security types* after program termination. The before and after observation pairs in the language constitutes a *relation*.

Definition 5 (Flow-sensitive relations): A flow-sensitive relation is a pair $(\alpha P, c_\tau \vdash P)$, where: $\alpha P = in\alpha P \cup out\alpha P$, in which $in\alpha P$ is a set of undecorated variables standing for initial values and initial security types, and $out\alpha P$ is a set of dashed variables standing for final values and final security types; P is a predicate containing no free variables other than those in αP; $c_\tau \in L_\tau$ records the observation on the security level of the local environment (e.g., a branch of a conditional operator) of P when it starts (in order to eliminate implicit flows from the environment such as a boolean condition); $c_\tau \vdash P$ specifies the predicate calculus of P under counter level c_τ.

We present a flow-sensitive relational predicate calculus of a simple sequential language in Table II, where $\sqcup$ denotes the upper bound. The `Assignment` is the basic action that assigns the evaluation(s) of expression e, and the upper bound of security type(s) of $\tau(e)$ of e and the environment counter type c_τ to the final value(s) and security type(s) of the variable(s) on the left-hand respectively, while variables not mentioned on the left of $:=$ remain unchanged, similar arguments to the operation $:\in$ and concurrent assignment. The `Skip II` is a no-effect command which leaves the values and security types of all the variables unchanged. The `Sequential` composition $P;Q$ describes a program executed by first executing P and then executing Q, after P terminates. In order to record the unobservable intermediate state passed from P to Q, an existential quantification of a set of variables $\vec{v}_0$ is introduced to denote the hidden observation of the intermediate states of variables $\vec{v}$ from P to Q [1]. The `Conditional` $P \lhd b \rhd Q$ describes a program which behaves like P if the initial value of b is `true` and like Q otherwise; the dashed security levels of variables take the upper bound of the level of the counter level c_τ, the boolean condition and the levels of relevant variables after execution of the branch body. The `Non-determinism` $P \sqcap Q$ describes a program executed by either P or Q missing of one will be chosen. The `Declaration` **var** x introduces a new program variable x which grants permission the use of the variable x in the following statements, and the complementary operation `Undeclaration` **end** x terminates the permission region of the use of the variable x. Recursion is modelled by the weakest fixed point $c_\tau \vdash \mu X \bullet F(X)$, the join operator $\bigsqcap$ is applied to the set of all solutions of $[X = c_\tau \vdash F(X)]$ where F is monotonic function from predicates to predicates:

$$\begin{aligned} c_\tau \vdash \mu X \bullet F(X) &\triangleq \bigsqcap\{X \mid X = c_\tau \vdash F(X)\} \\ &= \bigsqcap\{c_{\tau_i} \vdash F^i(X) \mid i = 0,1,\ldots,n\} \end{aligned}$$

where $F^{i+1}(X) = F(F^i(X))$, $F^{n+1}(X) = F^n(X)$, $c_{\tau_{i+1}} = c_{\tau_i} \sqcup c_\tau$ and $c_{\tau_0} = c_\tau$. Specifically, for any variable $x \in A$, consider the initial security type be $x_\tau = t_0 = t$, assume observation (let us focus on security levels) regarding the i^{th} time $(i = 0,1,\ldots,n)$ of applying F be: $\bigwedge_{x\in A}(x_\tau = t_i \wedge x'_\tau = t'_i)$, we have $t_0 = t$, $t_{i+1} = t'_i \sqcup t$, $t'_{n+1} = t'_n$, and we reach the weakest fixed point by applying F n times. Proposition 1 presents a set of algebraic laws for our flow-sensitive predicate calculus: classical laws [1] integrated with flow security environment.

Proposition 1 (Algebraic laws):

L1 $c_\tau \vdash (x := e; x := f(x)) \equiv c_\tau \vdash x := f(e)$
L2 $c_\tau \vdash (P; \mathtt{II}_{\alpha P}) \equiv c_\tau \vdash P = c_\tau \vdash (\mathtt{II}_{\alpha P}; P)$
L3 $c_\tau \vdash (P \lhd b \rhd P) \equiv c_\tau \vdash P$
L4 $c_\tau \vdash (P \lhd b \rhd Q) \equiv c_\tau \vdash (Q \lhd \neg b \rhd P)$
L5 $c_\tau \vdash (P \lhd b \rhd Q) \lhd c \rhd R \equiv c_\tau \vdash P \lhd (b \wedge c) \rhd (Q \lhd c \rhd R)$
L6 $c_\tau \vdash (P \lhd b \rhd (Q \lhd c \rhd R)) \equiv c_\tau \vdash (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$
L7 $c_\tau \vdash (\mathtt{true}; P) \equiv c_\tau \vdash \mathtt{true}$
L8 $c_\tau \vdash (P \sqcap Q) = c_\tau \vdash (Q \sqcap P)$
L9 $c_\tau \vdash (P \sqcap P) \equiv c_\tau \vdash P$
L10 $c_\tau \vdash (P \sqcap (Q \sqcap R)) \equiv c_\tau \vdash ((P \sqcap Q) \sqcap R)$
L11 $c_\tau \vdash (P \sqcap (Q \sqcap R)) \equiv c_\tau \vdash ((P \sqcap Q) \sqcap (P \sqcap R))$

Assignment	$c_\tau \vdash x := e \triangleq x' = e \wedge x'_\tau = c_\tau \sqcup \tau(e) \wedge \cdots \wedge z' = z \wedge z'_\tau = z_\tau,\ \ \alpha(x := e) \triangleq A$ $c_\tau \vdash x :\in S \triangleq x' = s \in S \wedge x'_\tau = c_\tau \sqcup \tau(S) \wedge \cdots \wedge z' = z \wedge z'_\tau = z_\tau,\ \ \alpha(x :\in S) \triangleq A$ $c_\tau \vdash x_1,\ldots,x_n := e_1,\ldots,e_n \triangleq x'_1 = e_1 \wedge x'_{1\tau} = c_\tau \sqcup \tau(e_1) \wedge \cdots \wedge z' = z \wedge z'_\tau = z_\tau,\quad \alpha(x_1,\ldots,x_n := e_1,\ldots,e_n) \triangleq A$
Skip II	$c_\tau \vdash \mathtt{II} \triangleq x = x' \wedge x'_\tau = x_\tau \cdots \wedge z = z' \wedge z'_\tau = z_\tau,\ \ \alpha\mathtt{II} \triangleq A$
Sequential	$c_\tau \vdash P(\vec{v}');Q(\vec{v}) \triangleq \exists \vec{v_0}.c_\tau \vdash P(\vec{v_0}) \wedge Q(\vec{v_0})$ *if* $out\alpha P = in\alpha Q = \{\vec{v}'\},\quad in\alpha(P(\vec{v}');Q(\vec{v})) \triangleq in\alpha P,\ out\alpha(P(\vec{v}');Q(\vec{v})) \triangleq out\alpha Q$
Conditional	$c_\tau \vdash P \lhd b \rhd Q \triangleq c_\tau \vdash ((b \wedge P) \vee (\neg b \wedge Q))$ *if* $\alpha b \subseteq \alpha P = \alpha Q$, $\alpha(P \lhd b \rhd Q) \triangleq \alpha P$, where: • $c_\tau \vdash b \wedge P \triangleq c_\tau \sqcup \tau(b) \vdash P$ • $c_\tau \vdash P \vee Q \triangleq ((x' = x_p \wedge \cdots \wedge z' = z_p) \vee (x' = x_q \wedge \cdots \wedge z' = z_q)) \wedge (x'_\tau = (c_\tau \sqcup \tau_{x_p} \sqcup \tau_{x_q}) \wedge \cdots \wedge z'_\tau = (c_\tau \sqcup \tau_{z_p} \sqcup \tau_{z_q}))$ *if* $P = (x' = x_p \wedge x'_\tau = \tau_{x_p} \wedge \cdots \wedge z' = z_p \wedge z'_\tau = \tau_{z_p}),\ \ Q = (x' = x_q \wedge x'_\tau = \tau_{x_q} \wedge \cdots \wedge z' = z_q \wedge z'_\tau = \tau_{z_q})$
Non-determinism	$c_\tau \vdash P \sqcap Q \triangleq c_\tau \vdash P \vee Q$ *if* $\alpha P = \alpha Q,\ \ \alpha(P \sqcap Q) \triangleq \alpha P$
Declaration	$c_\tau \vdash \mathbf{var}\ x \triangleq \exists x \bullet c_\tau \vdash \mathtt{II}_A,\ \ if\ x \in A,\ \ \alpha(\mathbf{var}\ x) \triangleq A \setminus \{x\}$
Undeclaration	$c_\tau \vdash \mathbf{end}\ x \triangleq \exists x' \bullet c_\tau \vdash \mathtt{II}_A,\ \ if\ x' \in A,\ \ \alpha(\mathbf{end}\ x) \triangleq A \setminus \{x'\}$
Recursion	$c_\tau \vdash \mu X \bullet F(X) \triangleq \bigsqcap\{X \mid [X = c_\tau \vdash F(X)]\}$

TABLE II
FLOW-SENSITIVE PREDICATE CALCULUS OF A SEQUENTIAL LANGUAGE

L12 $c_\tau \vdash (P \lhd b \rhd (Q \sqcap R)) \equiv c_\tau \vdash ((P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R))$
L13 $c_\tau \vdash ((P \sqcap Q);R) \equiv c_\tau \vdash ((P;R) \sqcap (Q;R))$
L14 $c_\tau \vdash (P;(Q \sqcap R)) \equiv c_\tau \vdash ((P;Q) \sqcap (P;R))$
L15 $c_\tau \vdash (P \sqcap (Q \lhd b \rhd R)) \equiv c_\tau \vdash ((P \sqcap Q) \lhd b \rhd (P \sqcap R))$

Theorem 1: if $c_{\tau_2} \leq_\tau c_{\tau_1}$, then $(c_{\tau_1} \vdash P) \Rightarrow (c_{\tau_2} \vdash P)$.

Proof: The proof is obtained directly from the definition of flow-sensitive relations (Definition 5) and the predicate calculus presented in Table II. ■

This theorem shows that any predicate is flow secure under a lower environment level if it is flow secure under a higher one.

Theorem 2: $(c_{\tau_1} \vdash P_1) \Rightarrow (c_{\tau_2} \vdash P_2)$ *iff*

$$(c_{\tau_2} \leq_\tau c_{\tau_1}) \wedge (P_1 \Rightarrow P_2).$$

Proof: The proof is obtained by Theorem 1 and the definition of implication. ■

This theorem implies that $c_{\tau_1} \vdash P_1$ is stronger because it has a higher environment security level so its behaviour will be more restricted for flow security concern in addition to the condition of P_1 implying to P_2, and where $c_{\tau_2} \vdash P_2$ is considered flow-secure and satisfied, $c_{\tau_1} \vdash P_1$ will be sure flow-secure and satisfied.

Theorem 3: $(c_{\tau_1} \vdash P_1) \sqcap (c_{\tau_2} \vdash P_2) = (c_{\tau_1} \sqcup c_{\tau_2}) \vdash (P_1 \vee P_2)$.

Proof: The proof is directly obtained from the definition of flow-sensitive relations (Definition 5) and the predicate calculus (non-determinism) presented in Table II. ■

This theorem indicates that the disjunction on a pair of flow-sensitive predicate calculus can be generalised to the union of the predicates with the environment security level being the least upper bound of their security levels.

Theorem 4: $(c_{\tau_1} \vdash P_1) \lhd b \rhd (c_{\tau_2} \vdash P_2) = (\tau(b) \sqcup c_{\tau_1} \vdash P_1) \vee (\tau(b) \sqcup c_{\tau_2} \vdash P_2)$.

Proof: The proof is directly obtained from the definition of flow-sensitive relations (Definition 5) and the predicate calculus (conditional) presented in Table II. ■

This theorem suggests that the conditional operation on a a pair of flow-sensitive predicate calculus can be obtained by taking the union of the two each of which takes their environment level as the least upper bound of the level of the boolean test and that of their original environment.

Theorem 5: For $1 \leq i \leq n$,

$$\bigsqcap_i (\{c_{\tau_i} \vdash P_i\}) = (\sqcup_i c_{\tau_i}) \vdash (\bigvee_i P_i).$$

Proof: The proof of this theorem is obtained by applying Theorem 3 multiple times. ■

Theorem 6 (Monotonicity of the predicate calculus): The relational flow-sensitive predicate calculus specified in Table II is monotone.

Proof: The proof is obtained by applying induction on the relational structure of the predicate calculus. Particularly, for the case of recursion, the sequences of observations regarding security types $t_0, t_1, \ldots$ and $t'_0, t'_1, \ldots$ thus form ascending chains with a weakest fixed point of the calculus on the security lattice $\mathscr{L}_\tau$. ■

Definition 6 (Predicate flow security condition): We say the relational predicate $c_\tau \vdash P$ is *strong flow secure*, written as $P \vdash_s \phi_{c_\tau}$, iff:

(1) for all $x, x', x_\tau, x'_\tau \in \alpha P$: $x'_\tau < c_\tau \Rightarrow x' = x$;
(2) for all $t \in \mathscr{L}_\tau$ and any two observations ω_1 and ω_2 of P: $\omega_1 \approx_t \omega_2$.

In addition, we say $c_\tau \vdash P$ is *weak flow secure* if only condition (2) is satisfied, written as: $P \vdash_w \phi_{c_\tau}$.

Condition (1) ensures that for any variables in P, if its dashed security level is less than the environment security level, then its value must not be changed by P; condition (2) ensures that for any security level $t \in \mathscr{L}_\tau$, if any two observations on undashed variables are t-equivalent, then the two observations

on dashed variables must be also t-equivalent, i.e., the final value of a variable - whose final security level less than or equal to t, must not depend on the initial values of those variables - whose initial security level greater than t.

Theorem 7: $P \vdash_s \phi_{c_\tau} \Rightarrow P \vdash_w \phi_{c_\tau}$.

Proof: Trivial. ■

We write $P \vdash \phi_{c_\tau}$ in general for situations in which "strong" and "weak" do not need to be distinguished.

Theorem 8: If P and Q are flow secure predicates, then $P \vee Q$ and $P \wedge Q$ are flow secure as well, i.e., :

- $P \vdash \phi_{c_\tau} \wedge Q \vdash \phi_{c_\tau} \Rightarrow (P \wedge Q) \vdash \phi_{c_\tau}$.
- $P \vdash \phi_{c_\tau} \vee Q \vdash \phi_{c_\tau} \Rightarrow (P \vee Q) \vdash \phi_{c_\tau}$.

Proof: The proof is obtained directly from Definition 6. ■

Theorem 9 (Soundness of the flow secure predicate): Every program specified in the flow-sensitive predicate calculus ensured by conditions specified in Definition 6 is flow secure predicate, for $t \in \mathscr{L}$:

$$P \vdash \phi_{c_\tau} \Rightarrow P \models \phi_t.$$

Proof: If P is *skip*, *assignment*, or *non-determinism*, the relevant flow secure rules ensure the security level of each variable after the execution less than or equal to that of before the execution. So clearly the flow security condition is satisfied. The rest of the proof follows from definition of the sequential, conditional and recursion operator presented in Table II and Theorem 3, 8, 6 by induction on the structure of the derivation tree. ■

Example 3: Consider $\alpha P = \{x, y\}$, $\tau_x = H$, $\tau_y = L$, $L \sqsubset H \in \mathscr{L}_\tau$, and $\bot \in \mathscr{L}_\tau$ denotes the system low.

$$\begin{aligned} P &\triangleq \bot \vdash (y := 0 \lhd (x = 0) \rhd y := 1) \\ &\triangleq \bot \vdash (x = 0 \wedge y := 0) \vee (\neg(x = 0) \wedge y := 1) \\ &\triangleq H \vdash y := 0 \vee H \vdash y := 1 \\ &\triangleq (y' = 0 \wedge \tau_y = L \wedge \tau'_y = H) \vee (y' = 1 \wedge \tau_y = L \wedge \tau'_y = H) \\ &\nvdash \phi_\bot. \end{aligned}$$

Note that there is implicit flow introduced by the boolean test and the program is not flow secure. ■

IV. Preserving Flow Security under Refinement Calculus

This section studies the problem of refining the characterisation of the class of the flow secure relations. Refinement guarantees that a refined (concrete) predicate satisfies all the functionality properties of the refining (abstract) one. We have discussed that P is considered flow secure if it satisfies the flow secure condition, which can be enforced by our flow healthy predicate calculus. However, the refining relation cannot guarantee that a refinement of P always preserves the flow security properties: the refined theory might introduce new elements, while the secure flow properties and the enforcing rules depend on the semantics and the flow judging environment.

Example 4: Consider the dining cryptographer problem as an example. Assume three cryptographers c_0, c_1 and c_2 are sharing a meal at a restaurant. At the end of the meal, The cryptographers are told that the meal has been paid by someone, who might be one of the cryptographers or their manager. The cryptographer would like to find out whether their manager paid but respect each other's right to make an anonymous payment.

In the first step, we sketch an event-based abstract model P which produce a decision of "*paid*" (`dec` $= 1$) directly in program `decide` after executing `pay` (`payer`=0,1,2,999 denotes the bill has been paid by cryptographer c_0, c_1, c_2 and the manager respectively):

```
init ≙ payer:=-1; dec:=0;
pay ≙ payer:∈{0,1,2,999};
decide ≙ dec:=1;

P ≙ L ⊢ init; pay; decide;
```

Let the security level of `payer` be H which should be kept secret, and the level of the rest of variables be L and $L \leq_\tau H$. Clearly $P \vdash \phi_L$ since the value of H level variable `payer` (0,1,2) won't affect the final value of `dec` in this abstract model - the waiter has informed them that "the meal has been paid by someone".

The dining cryptographer protocol [6] is proposed to check if the bill has been paid by the manager or by one of the cryptographers but without releasing the identity of the payer. There are two stages performed to solve the problem:

(i) every two cryptographers establish a shared one-bit secret: each of them flips a coin (c_i is used to record the flipping result of cryptographer c_i, $i = 0, 1, 2$), the outcome is only visible to himself and the cryptographer on his right, so each cryptographer can see two outcomes: the one he flipped and the one his left-hand neighbour flipped - this stage is "private";

(ii) each cryptographer publicly announces whether the two outcomes agree or disagree, if the cryptographer is not the payer, he says the truth, otherwise, he states the opposite of what he sees - this stage is "public". When all cryptographers have announced, they count the number of disagrees (recorded in `ctr`). If that number is odd, then one of them has paid (`dec`=1), without directly releasing who is the payer; otherwise the bill has not been paid (`dec`=0).

We now propose a refined model P' to implement a (faulty) variant of the dining cryptographer protocol specified in `decide'` refined from `decide`. Stage (i) is performed once only at the initialisation `init'` (before the bill has been paid), and stage (ii) specified in `decide'` is always run with the secret established during the initialisation. Suppose the faulty protocol is run once before the bill is paid, and once after it is paid.

```
init' ≙ payer,r0,r1,r2,c0,c1,c2:=-1;
        ctr,dec:=0;

flip'0 ≙ (c0 < 0) ▷ (c0 :∈ {0,1}); //c0 flip
flip'1 ≙ (c1 < 0) ▷ (c1 :∈ {0,1}); //c1 flip
flip'2 ≙ (c2 < 0) ▷ (c2 :∈ {0,1}); //c2 flip
pay' ≙ (payer<0) ▷ (payer:∈{999,0,1,2});
```

```
//c0 makes announcement
res'0  ≜ ((payer>0)⊳(r0 := c0 ⊕ c1))⊲(payer=0)
           ⊳(r0 := 1 − (c0 ⊕ c1))
//c1 makes announcement
res'1  ≜ ((payer>=0 ∧ payer<>1)⊳(r1 := c1 ⊕ c2))
           ⊲(payer=1)⊳(r1 := 1 − (c1 ⊕ c2))
//c2 makes announcement
res'2  ≜ ((payer>=0 ∧ payer<>2)⊳(r2 := c2 ⊕ c0))
           ⊲(payer=2) ⊳(r2 := 1 − (c2 ⊕ c0))

//disagreement counting
count' ≜ r0 ≥ 0∧r1 ≥ 0∧r2 ≥ 0 ⊳(ctr:=r0+r1+r2)
 //decision making
decide' ≜ flip'0; flip'1; flip'2;
          res'0; res'1; res'2; count';
          (ctr≥0)⊳(dec:=ctr % 2);

P' ≜ L⊢ init'; decide'; pay'; decide';
```

Intuitively, the faulty protocol will release the identity of the payer since one can deduce it by observing whose announcement differs between the first (prior to payment) run and the second (post-payment) run of stage ii). By Definition 6, $P' \not\vdash \phi_L$: condition (1) is clearly violated since the final value of `dec` depends on that of `counter` and thus on that of the higher security level `payer` which should be kept secret, thus there is information leakage introduced by an implicit flow through boolean condition `(i=payer)`, $(c_0 \geq 0 \wedge c_1 \geq 0 \wedge c_2 \geq 0)$ and `(ctr≥0)` in the refined model, which reveals the information of "the bill being paid or not"; condition (2) is also violated since observations on the executions of the public stage `decide'` are not L-equivalent, when `payer` changes (e.g. executions before paid and after paid by i), the observations on the announcement of the payer (r_i) are different which reveals the identity of the payer. ■

Clearly there is no guarantee that the refinement transformation preserves the proposed flow security properties. Therefore, it is not enough to prove the security property at one level in general.

A. The flow secure refinement calculus

The main goal here is to define a subclass Q of relation P which is ensured to be both flow secure and healthy. To record the observation on initiation and termination of the program, following [1], special boolean variables ok and ok' are introduced to denote the status of the program being started and being terminated respectively. We present the basic definition of *flow secure design* in the refinement calculus as an extension of the definition of *design* [1] as follows.

Definition 7 (Flow secure design): Let P and Q be predicates, let $c_\tau \in \mathscr{L}_\tau$, a flow secure design is a relation in the following form:

$$P \models_{c_\tau} Q \quad \triangleq \quad c_\tau \vdash (P \wedge ok) \wedge (P \vdash_w \phi_{c_\tau}) \;\Rightarrow\; c_\tau \vdash (Q \wedge ok') \wedge (Q \vdash_w \phi_{c_\tau})$$

The predicate $c_\tau \vdash (P \wedge ok) \wedge (P \vdash_w \phi_{c_\tau}) \;\Rightarrow\; c_\tau \vdash (Q \wedge ok') \wedge (Q \vdash_w \phi_{c_\tau})$ specifies a relation such that: if the program starts in a state satisfying flow secure predicate P, it will terminate, and on termination Q will be true and flow secure.

Theorem 10 (Implication of flow secure designs):

$$(P_1 \models_{c_{\tau_1}} Q_1) \Rightarrow (P_2 \models_{c_{\tau_2}} Q_2)$$

iff:

$$(c_{\tau_2} \vdash P_2 \Rightarrow c_{\tau_1} \vdash P_1) \wedge (c_{\tau_2} \vdash (Q_1 \wedge P_2) \Rightarrow c_{\tau_2} \vdash Q_2).$$

This theorem shows that $P_1 \models_{c_{\tau_1}} Q_1$ is stronger because it has a weaker assumption $c_{\tau_1} \vdash P_1$, and where $c_{\tau_2} \vdash P_2$ is considered flow-secure, $c_{\tau_1} \vdash P_1$ will be sure flow-secure. Note that $c_{\tau_2} \vdash P_2 \Rightarrow c_{\tau_1} \vdash P_1$ implies $c_{\tau_1} \leq c_{\tau_2}$ and $P_2 \Rightarrow P_1$ according to Theorem 2. The proof can then be obtained by Theorem 2 and Definition 7.

Theorem 11 (Conditional of flow secure designs):

$$\begin{aligned} & (P_1 \models_{c_{\tau_1}} Q_1) \lhd b \rhd (P_2 \models_{c_{\tau_2}} Q_2) \\ = \;& (P_1 \lhd b \rhd P_2) \models_{c_{\tau_1} \sqcup c_{\tau_2}} (Q_1 \lhd b \rhd Q_2). \end{aligned}$$

This theorem implies that the conditional operation on a pair of flow secure designs can generate a new design with the assumption being the condition on the assumptions of the two, and the commitment being the condition on the commitments of the two, and the environment level being the least upper bound of the environment levels of the two. Proof can be obtained by Definition 7 and Theorem 4.

Theorem 12 (Nondeterminism of flow secure designs):

$$(P_1 \models_{c_{\tau_1}} Q_1) \sqcap (P_2 \models_{c_{\tau_2}} Q_2) = (P_1 \wedge P_2) \models_{c_{\tau_1} \sqcup c_{\tau_2}} (Q_1 \vee Q_2).$$

This theorem suggests that the disjunction of two flow secure designs can be generalised to the design with the conjunction of the assumptions of the two as its assumption (stronger) and the union of the commitments of the two as its commitment (weaker) and the least upper bound of the environment levels as its environment level. The proof of this theorem is directly obtained by Definition 7 and Theorem 3.

Similar results can be obtained regarding disjunction and conjunction of a set of flow secure designs, shown in Theorem 13 and 14.

Theorem 13 (Disjunction of flow secure designs):

$$\prod_i (P_i \models_{c_{\tau_i}} Q_i) = \bigwedge_i P_i \models_{\sqcup c_{\tau_i}} \bigvee_i Q_i.$$

Theorem 14 (Conjunction of flow secure designs):

$$\bigsqcup_i (P_i \models_{c_{\tau_i}} Q_i) = \bigvee_i P_i \models_{\sqcup c_{\tau_i}} (\bigwedge_i (P_i \Rightarrow Q_i)).$$

Example 5: Consider again Example 4, clearly $P \not\models_L P'$ and P' is not a flow secure refinement of P. ■

B. Building connections between theories

Theories are identified primarily by their set of predicates. We propose to define a link between theories as a paired functions which maps all predicates from one into those of another with a concern of flow security property. Such a link can be used to reveal significant underlying structure of the theories to compare, and to enforce no additional leakage being introduced in the refined theories.

A *bijection* is a function that shows the exact reversion between two predicates and thus might lose interesting distinction between them. In general, different predicates should have different expressive power: one (stronger) might contain richer features than another one (weaker). We therefore need a link function which can be used to connect two predicates at different expressive levels rather than exact reversion. Galois connection, represented as a paired function, can be used to build a link to show both the similarity and interesting mathematical distinction of them. We now study linking theories between a relatively abstract one (weaker) and a concrete one (stronger) accounts for *flow-sensitive observations* by building connection between them. Definition 8 reviews the basic definition of Galois connection [7].

Definition 8 (Galois connection): Let $\mathbf{S} = (S, \preceq_S)$ and $\mathbf{T} = (T, \preceq_T)$ be posets, suppose $f_L : \mathbf{S} \to \mathbf{T}$ and $f_R : \mathbf{T} \to \mathbf{S}$, we say the pair (f_L, f_R) is a *Galois connection* between $\mathbf{S}$ and $\mathbf{T}$ if $\forall s \in S$ and $t \in T$:

$$f_L(s) \succeq_T t \quad \Leftrightarrow \quad s \succeq_S f_R(t),$$

or:

$$s \succeq_S f_R(f_L(s)) \quad \Leftrightarrow \quad t \preceq_T f_L(f_R(t)).$$

f_L is called the *left adjoint* of the corresponding f_R, and f_R is the *right adjoint* of f_L.
Intuitively, function f_L can be constructed to map each predicate of the stronger theory to the weaker one, and f_R maps in the opposite direction.

Consider a predicate is a set of observations (executions) satisfying the predicate. To build a link between the weaker and stronger predicate classes accounts for the flow-sensitive observations, we need to study the ordering of them in order to construct appropriate maps between them.

Definition 9 (Ordering of flow-sensitive predicates): Given two predicates P_1 and P_2 with alphabet A, we define:

$$P_1 \leq_{\phi_{c_\tau}} P_2 \triangleq P_1 \models_{c_\tau} P_2.$$

Definition 10 (A link between a pair of theories):
Consider two predicates P_1 and P_2 with alphabet A, let notation $\mathscr{P}(X)$ denote the powerset of set X, then put:

- $\mathbf{P}_1 = (\mathscr{P}(P_1), \subseteq)$: the powerset of the flow-sensitive predicate relations P_1 forms a lattice with a partial ordering on *subset* relation $\subseteq$;
- similarly, we write: $\mathbf{P}_2 = (\mathscr{P}(P_2), \subseteq)$.

Furthermore,

- for $O_1 \subseteq P_1$, we define:

$$f_R(O_1) = \{Y \in P_2 \mid \forall X.(X \in O_1 \to X \models_{c_\tau} Y)\}$$

- for $O_2 \subseteq P_2$, we define:

$$f_L(O_2) = \{X \in P_1 \mid \forall Y.(Y \in O_2 \to X \models_{c_\tau} Y)\}$$

Paired function (f_L, f_R) thus build a link between $\mathbf{P}_2$ and $\mathbf{P}_1$ which accounts for the ordering of flow sensitive predicates.
Essentially, f_R is the concretising function which takes a subset observations O_1 of P_1 and returns all the possible observations of P_2 implied by the observations in O_1 in terms of the ordering of flow-sensitive predicates. More precisely, $f_R(O_1)$ is a subset $\{Y\}$ of P_2 such that for each element X of P_1: $X \models_{c_\tau} Y$. On the other hand, f_L is the abstraction function which takes a subset observations O_2 of P_2 and returns all the possible observations of P_1 implying all observations in O_2 in terms of the ordering of flow-sensitive predicates.

Theorem 15: Link function (f_L, f_R) defined in Definition 10 forms a Galois connection between $\mathbf{P}_2$ and $\mathbf{P}_1$.

Proof: By definition of Galois connection, proving (f_L, f_R) is a Galois connection between $\mathbf{P}_1 = (\mathscr{P}(P_1), \subseteq)$ and $\mathbf{P}_2 = (\mathscr{P}(P_2), \subseteq)$ is equivalent to proving the following equivalence:

$$\forall O_2 \subseteq P_2, O_1 \subseteq P_1 : f_L(O_2) \supseteq O_1 \textit{ iff } O_2 \supseteq f_R(O_1).$$

So:

$$\begin{aligned}
f_L(O_2) \supseteq O_1 \quad &\Leftrightarrow \quad \{X | \forall Y.(Y \in O_2 \to X \models_{c_{\tau_1}} Y)\} \supseteq O_1 \\
&\Leftrightarrow \quad (\forall X \in O_1)(\forall Y \in O_2) \to X \models_{c_{\tau_1}} Y \\
&\Leftrightarrow \quad (\forall Y \in O_2)(\forall X \in O_1) \to X \models_{c_{\tau_1}} Y \\
&\Leftrightarrow \quad \{Y | \forall X.(X \in O_1 \to X \models_{c_{\tau_1}} Y)\} \subseteq O_2 \\
&\Leftrightarrow \quad f_R(O_1) \subseteq O_2
\end{aligned}$$

Therefore, the connection function pair (f_L, f_R) between $\mathbf{P}_2$ and $\mathbf{P}_1$ forms a Galois connection. ■

Next step we study the preservation of the ordering of flow sensitive observations under a set of linked theories connected by a set of linked functions constructed in terms of Definition 10.

Lemma 1: Consider three posets:

$$\mathbf{P}_1 = (\mathscr{P}(P_1), \subseteq), \ \mathbf{P}_2 = (\mathscr{P}(P_2), \subseteq), \ \mathbf{P}_3 = (\mathscr{P}(P_3), \subseteq).$$

Let (f_L, f_R) and (g_L, g_R) be the connection functions between $\mathbf{P}_2$ and $\mathbf{P}_1$, and between P_3 and P_2, specified in Definition 10 respectively. Then the paired function $(f_L \circ g_L, g_R \circ f_R)$ is a Galois connection between $\mathbf{P}_3$ and $\mathbf{P}_1$.

Proof: The proof is obtained directly from Theorem 15 and the composition property of Galois connections. ■

Theorem 16: Consider two theories connected by Galois connection specified in Definition 10, for any $t \in \mathscr{L}$: $\mathbf{P}_2(f_L, f_R)\mathbf{P}_1$, we have:

$$P_1 \models \phi_t \Rightarrow P_2 \models \phi_t.$$

Proof: By Definition 10, 9, and 7, this can be reduced to prove: $c_\tau \vdash (P_1 \wedge ok) \wedge P_1 \vdash_w \phi_{c_\tau} \Rightarrow c_\tau \vdash (P_2 \wedge ok') \wedge P_2 \vdash_w \phi_{c_\tau}$. By Theorem 9, we obtain: $P_1 \models \phi_t \Rightarrow P_2 \models \phi_t$. ■
Theorem 16 implies that the refinement calculus function specified by (f_L, f_R) preserves the flow security condition, i.e., if P_1 is flow secure and $\mathbf{P}_2(f_L, f_R)\mathbf{P}_1$, then the relevant concrete one P_2 satisfies the flow security condition as well.

Theorem 17: Assume $\mathbf{P}_n(f_L, f_R)\mathbf{P}_{n-1} \ldots \mathbf{P}_2(h_L, h_R)\mathbf{P}_1$, where $n \in \mathbb{N}$, paired functions (f_L, f_R), $\ldots$, (h_L, h_R) are the Galois connections build by Definition 10 between $\mathbf{P}_2$ and $\mathbf{P}_1$, $\mathbf{P}_3$ and $\mathbf{P}_2$, and so on, for $t \in \mathscr{L}$ then:

$$P_1 \models \phi_t \Rightarrow P_n \models \phi_t.$$

Proof: The proof follows directly from Lemma 1 and Theorem 16 by induction. ■

Theorem 17 shows that our refinement calculus, formalised as a series of paired function from beginning specification to final design, are guaranteed to preserve the flow security condition.

Example 6: Consider again Example 4, we now propose refined model P'' to implement the idea of the correct (original) dining cryptographer protocol.

$$\texttt{P''} \triangleq L\vdash \texttt{init'; pay'; decide';}$$

By Definition 6, P'' does not satisfy condition (1) due to the implicit flow through boolean condition `(i=payer)`; P'' also violates condition (2), since observations on L-level variables including `dec`, c_i (i=0,1,2), r_i (i=0,1,2), `ctr` are affected by the initial value of H-level variable `payer`. There is information leakage from the initial value of `payer` to the observer - one can at least deduce whether $\texttt{payer} \geq 0$ or not, i.e., whether the bill has been paid or not. Therefore, $P'' \not\vdash \phi_L$, and by Definition 7: $P'' \not\models_L P$. From the view of connections between P and P'', we are not able to find (f_L, f_R) to build flow secure link between the observations of P and P'' based on Definition 10. ■

Example 7: Let us take a further thought of Example 6. Assume the bill has been paid by one of the three successfully (i.e., $\texttt{payer}:\in 0,1,2$), and we only allow `dec` to be observable, observations on L-level observable variables `dec` is not affected by the initial value of H-level variable `payer`, then condition (2) is satisfied: $P'' \vdash_w \phi_L$. Under such assumptions (preconditions), we are able to build a link (f_L, f_R) between P and P'' based on Definition 10, since only `dec` is included in the observations and thus $X \models_{c_\tau} Y$ would be satisfied for any observation X in P and observation Y in P''. The identity of payer (0 or 1 or 2) keeps secret if the bill has been paid in this case, which meets the intuition of the dining cryptographer protocol. ■

Inspired by the consideration in Example 7, it would be interesting to consider the observability of variables to provide more specific flow analysis as an extension of our framework.

V. Related Work

The notion of secure information flow specifies the security requirements of the system where should be no information flow from the confidential data to the observer. This paper relates to the topic of information flow control in formal programming languages.

Denning and Denning [8] first use program analysis to investigate if the information flow properties of a program satisfy a specified multi-level security policy. Security type systems had been substantially used to formulate the analysis of secure information flow in programs. Hunt and Sands [9] presented a flow sensitive type system for program in a simple While language for multi-level security. Sensitive information was stored in programming variables, the powerset of program variables forms the universal lattice. A family of inference systems was developed to be forced to satisfy a simple non-interference property. Their following work [10] showed how flow-sensitive multi-level security typing can be achieved in polynomial time. In addition to type-based treatments of secure information flow analysis for programs, Clark *et. al* presented a flow logic approach in [11]. Hammer and Snelting [12] presented an approach for information flow control in program analysis based on program dependence graphs (PDG). Based on [12], [13] extended the PDG-based flow analysis by incorporating refinement techniques via path conditions to improve the precision of the flow control. Such PDG-based information flow control is more precise but more expensive than type-based approaches. These works did not include treatments on specification language and secure abstraction refinements. Absolute information-flow properties, such as non-interference, are rarely satisfied by real programs. *Declassification* [14] and endorsement which are also known as *downgrading*, allow high-level security information to be used in low-level security contexts. There is no confidential information flows from a restricted execution environment to a public one without having been properly declassified.

A number of papers addresses flow analysis in specification languages. Iliasov [15] introduced a method for control flow properties in Event-B models. The *flow* analysis in this paper focused on expressions with event ordering and looked at the interference between events introduced by a set of conditions formulated on a machine. It can be used to express flow properties for a model and to verify them using proofs. Bendisposto *et. al* [16] proposed an automatic flow analysis by deriving a flow graph structure from an Event-B model specification. The derived graph contained information about dependence and independence of events which can be used for flow analysis and model comprehension.

Refinement is a process of making an abstract specification more detailed. Jocob [17] first pointed out that secure information flow properties were not preserved by the standard notion of refinement in general. There are a number of papers addressed information flow security and refinement. Heisel *et. al* developed a *condition* for confidentiality-preserving refinement in [18], [19]. The basic idea was that the information allowed to be revealed by the concrete system should also be allowed to be revealed regarding the abstract one. Alur *et. al* [20] presented a framework for preservation of secrecy in *labelled transition system*, and introduced a simulation-based proof technique for preserving secrecy under refinement. Mantel [21] proposed a method for preserving information flow properties under CSP-style refinements regarding an *event system*. The event system was considered as a tuple of a set of input/output events and a set of traces. The idea here was two fold: introducing refinement operators to refine specifications and then constructing secure refined event system based on low-equivalence relations. Mantel showed how tailored refinement operators for information flow properties can construct a refinement in which the resulting refinement preserves the given flow property. Bossi *et. al* [22] studied the problem of preservation of information flow properties under *action refinement* in the context of *process algebra*. Seehusen and Stølen [23] introduced a schema to specify and

preserve secure information flow properties in the semantics of STAIRS [24]. Jürjens [25] presented a framework for preserving secrecy under refinement operators in specification framework FOCUS [26]. In FOCUS, a process was modelled by a total stream-processing function which mapped input streams to sets of output streams. A process was considered preserving the secrecy if without eventually outputting secrets. [25] presented a set of *conditions* w.r.t. FOCUS under which the refinements preserved proposed secrecy properties. Morgan [27] studied enforcement issues under stepwise refinement based on the definition of "ignorance-preserving" refinement. The idea was adding restrictions on adversary's access to higher level information during classical refinement. Mu [28] introduced a more general approach to provide secure flow control in a *specification language*. The presented framework can be used to reason about flow security properties and relevant relations of the stepwise refinement transformations in Event B.

However, those works have not placed the foundations for reasoning about information flow in a unifying theory setting, and can not be deployed over a various families of specification languages and programming paradigms as the approach delineated in this work. Banks and Jocob [29] studied the problem of formalising and reasoning about *confidentiality* property in software design using the UTP. Following this work, [30] proposed a platform to encode confidentiality properties in the *Circus* processes. Comparing with these works, we focused on extending the UTP theory to integrating flow control, results regarding possible operations are formalised, flow security property is naturally coordinated and integrated in the UTP semantics. We also studied the problem of preserving flow security under refining the flow security relations: we extended the definition of *design* to *flow secure design* in the refinement calculus to ensure the flow security condition, and constructed a link between multiple level theories under flow secure refinement relation.

VI. Conclusions

We have presented a unified framework in the UTP setting for specifying and developing flow secure software. Specifically, we formalise and integrate flow security properties with the UTP semantics, present semantic enforcing rules to ensure the flow security condition in an abstract system design. We then present the definition of *flow secure design* in the refinement calculus to specify a subclass which is guaranteed to be both secure and healthy. We also build a link between multi-level theories under refinement transformation from preserving flow security point of view. For future work, it is promising to extend our framework to support probabilistic analysis of the flow security properties. It would also be interesting to consider the observability of variables to provide more specific flow analysis as an extension of our framework.

Acknowledgement

This research is supported by the China National R&D Key Research Program (2019YFB1705703) and the Interdisciplinary Program of SJTU, Shanghai, China (No. YG2019ZDA07).

References

[1] T. Hoare and J. He, *Unifying Theories of Programming*. Prentice Hall Inc., 1998.
[2] J. Goguen and J. Meseguer, "Security policies and security models," in *S & P*, 1982, pp. 11–20.
[3] J. McLean, "Security models and information flow," in *S & P*, Oakland, California, May 1990.
[4] D. E. R. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.
[5] ——, *Cryptography and Data Security*. Addison-Wesley, 1982.
[6] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of Cryptology*, vol. 1, pp. 65–75, 1988.
[7] M. Erne, J. Koslowski, A. Melton, and G. Strecker, "A primer on galois connections," in *York Academy of Science*, 1992.
[8] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.
[9] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL*. ACM Press, January 2006, pp. 79–90.
[10] ——, "From exponential to polynomial-time security typing via principal types," in *ESOP*, 2011, pp. 297–316.
[11] D. Clark, C. Hankin, and S. Hunt, "Information flow for algol-like languages," *Comput. Lang.*, vol. 28, no. 1, pp. 3–28, 2002.
[12] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *Int. J. Inf. Sec.*, vol. 8, no. 6, pp. 399–422, 2009.
[13] M. Taghdiri, G. Snelting, and C. Sinz, "Information flow analysis via path condition refinement," in *FAST*, 2010, pp. 65–79.
[14] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.
[15] A. Iliasov, "On event-b and control flow," School of Computing Science, Newcastle University, Tech. Rep. CS-TR-1159, 2009.
[16] J. Bendisposto and M. Leuschel, "Automatic flow analysis for event-b," in *FASE*, 2011, pp. 50–64.
[17] J. Jacob, "On the derivation of secure components," in *S & P*, 1989, pp. 242–247.
[18] M. Heisel, A. Pfitzmann, and T. Santen, "Confidentiality-preserving refinement," in *CSFW*, 2001, pp. 295–306.
[19] T. Santen, M. Heisel, and A. Pfitzmann, "Confidentiality-preserving refinement is compositional - sometimes," in *ESORICS*, 2002, pp. 194–211.
[20] R. Alur, P. Cerný, and S. Zdancewic, "Preserving secrecy under refinement," in *ICALP*, 2006, pp. 107–118.
[21] H. Mantel, "Preserving information flow properties under refinement," in *S & P*, 2001, pp. 78–92.
[22] A. Bossi, C. Piazza, and S. Rossi, "Action refinement in process algebra and security issues," in *LOPSTR*, 2007, pp. 201–217.
[23] F. Seehusen and K. Stølen, "Maintaining information flow security under refinement and transformation," in *FAST*, 2006, pp. 143–157.
[24] Ø. Haugen and K. Stølen, "Stairs - steps to analyze interactions with refinement semantics," in *UML*, 2003, pp. 388–402.
[25] J. Jürjens, "Secrecy-preserving refinement," in *FME*, 2001, pp. 135–152.
[26] M. Broy and K. Stølen, *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer-Verlag New York, Inc., 2001.
[27] C. Morgan, "The shadow knows: Refinement and security in sequential programs," *Sci. Comput. Program.*, vol. 74, no. 8, pp. 629–653, 2009.
[28] C. Mu, "On information flow control in event-b and refinement," in *TASE*, 2013, pp. 225–232.
[29] M. J. Banks and J. L. Jacob, "Unifying theories of confidentiality," in *UTP*, 2010, pp. 120–136.
[30] ——, "Specifying confidentiality in circus," in *FM*, M. Butler and W. Schulte, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 215–230.