

# Data Leakage Analysis of the Hibernate Query Language on a Propositional Formulae Domain

Raju Halder<sup>1(✉)</sup>, Angshuman Jana<sup>1</sup>, and Agostino Cortesi<sup>2</sup>

<sup>1</sup> Indian Institute of Technology Patna, Patna, India  
`{halder, ajana.pcs13}@iitp.ac.in`

<sup>2</sup> Università Ca' Foscari Venezia, Venice, Italy  
`cortesi@unive.it`

**Abstract.** This paper presents an information flow analysis of Hibernate Query Language (HQL). We define a concrete semantics of HQL and we lift the semantics on an abstract domain of propositional formulae. This way, we capture variables dependences at each program point. This allows us to identify illegitimate information flow by checking the satisfiability of propositional formulae with respect to a truth value assignment based on their security levels.

**Keywords:** Hibernate query language · Information flow analysis · Abstract interpretation

## 1 Introduction

Modern database applications are mostly implemented using Object Oriented Programming (OOP) languages supported by relational databases at the back end. Due to paradigm mismatch, the way to access data in object oriented languages is fundamentally different than that in case of relational database languages. Hibernate, an Object Relational Mapping (ORM) framework, mitigates this impedance mismatch problem by replacing direct persistence-related database accesses with high-level object handling functions. Hibernate provides Hibernate Query Language (HQL) which allows SQL-like queries to be written against Hibernate's data objects. Various methods in “**Session**” interface are used to propagate object's states from memory to the database (or vice versa). Hibernate will detect any change made to an object in persistent state and synchronizes the state with the database when the unit of work completes. A HQL query is translated by Hibernate into a set of conventional SQL queries during run time which in turn performs actions on the database. This way, HQL provides a unified platform for the programmers to develop object-oriented applications to interact with databases, without knowing much details about the underlying databases [5, 6, 15].

Secure information flow is comprised of two related aspects: information confidentiality and information integrity. Confidentiality refers to limiting the access and disclosure of sensitive information to authorized users only. For instance,

when we purchase something online, our private data, e.g. credit card number, must be sent only to the merchant without disclosing to any third person during the transmission. Dually, the notion of integrity indicates that data or messages cannot be modified undetectably by any unauthorized person [34].

While access control and encryption prevent confidential information from being read or modified by unauthorized users at source level, they do not regulate the information propagation after it has been released for execution. Confidentiality may be compromised during the flow of information along the control structure of any software systems [29]. Assuming variables ‘h’ and ‘l’ are private and public respectively, the following code fragments depict two different scenarios (explicit/direct flow and implicit/indirect flow) of information leakage:

<pre> 1 := h if(h=0) l:=5; else l:=10; </pre>	<p><b>Explicit/Direct flow</b></p> <p><b>Implicit/Indirect flow</b></p>
---	---

Observe that confidential value in ‘h’ can be deduced by attackers observing ‘l’ on the output channel.

A wide range of language-based techniques are proposed in the past decades to analyze this illegitimate flow in software products [4, 9, 16, 21, 24, 28, 29, 32]. Works in this direction have been starting with the pioneering work of Dennings in the 1970s [13]. As a starting point, the analysis classifies the program variables into various security classes. The simplest one is to consider two: Public/Low (denoted  $L$ ) and Private/High (denoted  $H$ ). Considering a mathematical lattice-model of security classes with order  $L \leq H$ , the secure information flow policy is defined on the lattice: an upward-flow in the lattice is only permissible to preserve confidentiality. Dually, in case of integrity, the lattice-model labels the variables as Tainted (denoted  $T$ ) and Untainted (denoted  $U$ ), and follows a dual flow-policy.

The correctness is guaranteed by respecting the non-interference principle that says “a variation of confidential data does not cause any variation to public data”: *Given a program  $P$  and set of states  $\Sigma$ . The non-interference policy states that  $\forall \sigma_1, \sigma_2 \in \Sigma. \sigma_1 \equiv_L \sigma_2 \implies \llbracket P \rrbracket \sigma_1 \equiv_L \llbracket P \rrbracket \sigma_2$ , where  $\llbracket \cdot \rrbracket$  is semantic function and  $\equiv_L$  represents low-equivalence relation between states.*

Most of the notable works which refer to imperative, object-oriented, functional, database query languages, etc. [8, 18, 21, 24, 27–29] can not be applied directly to the case of HQL due to the presence and interaction of high-level HQL variables and database attributes through **Session** methods. Moreover, as we are interested on persistent data, analyzing object-oriented features of HQL does not meet our objectives neither. Let us illustrate a motivating example depicted in Fig. 1. Two POJO classes  $c_1$  and  $c_2$  correspond to two underlying database-tables by mapping class-fields into table-attributes. In the main method of Service Class **ExClass**, values of the table corresponding to  $c_1$  are used to make a list, and for each element of the list an update is performed on the table corresponding to the class  $c_2$ . Observe that there is an information-flow from confidential (denoted by  $h$ ) to public variables (denoted by  $l$ ). In fact, the confidential database information  $h_1$  which is extracted at statement 15, affects the public view of the database information  $l_2$  at statement 20. This fact is

depicted in Fig. 1(d). The new challenge in this scenario *w.r.t.* state-of-the-art of information leakage detection is that we need to consider both application variables and SQL variables (corresponding to the database attributes).

In this paper<sup>1</sup>, we extend the abstract interpretation-based framework in [34] to the case of HQL, focussing on **Session** methods which act as persistent manager. This allows us to perform leakage analysis of sensitive database information when is accessed through high-level HQL code.

The main contributions in this paper are:

- Defining the concrete and an abstract transition semantics of HQL, by using symbolic domain of positive propositional formulae.
- Analyzing possible information leakage based on the abstract semantics, focussing on variable dependences of database attributes on high-level HQL variables.

The structure of the paper is as follows: Sect. 2 briefly discusses the related works in the literature. In Sect. 3, we define the abstract syntax of HQL in BNF. In Sects. 4 and 5, we formalize the concrete and an abstract transition semantics of HQL, by using the symbolic domain of positive propositional formulae. In Sect. 6, we perform information leakage analysis of programs based on the abstract semantics which captures possible leakage of confidential data. Section 7 concludes the paper.

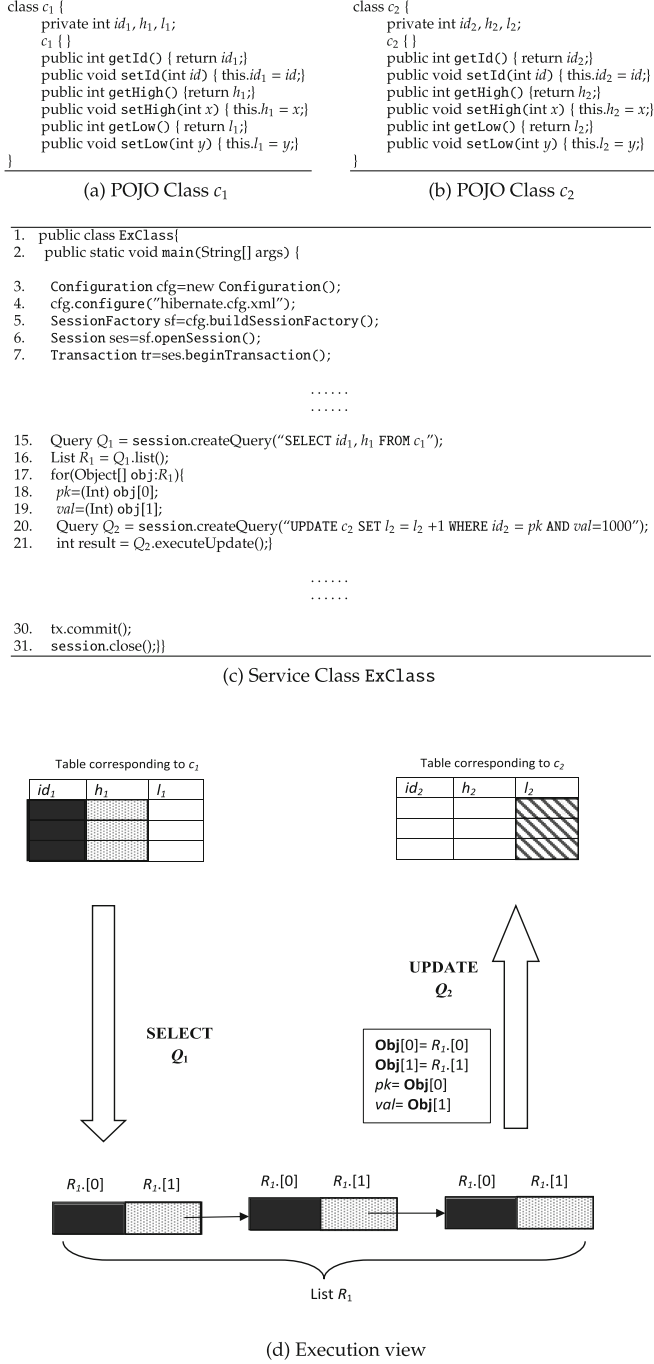
## 2 Related Works

A comprehensive survey on language-based information-flow analysis is reported in [29]. Most popular static analysis techniques are based on type systems [29, 32, 33], dependence graphs [7, 19–21, 23, 24, 26], formal approaches [1, 2, 14, 22, 34, 35], etc. Besides the conservative nature of static analysis, the run-time monitoring systems detect unauthorized information flow dynamically; however, precision of the analysis completely depends on the execution overload, and of course, it is very prone to false negative [3, 31].

The security type system considers various security types (*e.g.*, *low* and *high*) and a collection of typing rules which determine the type of expressions/commands to guarantee a secure information flow [29, 32, 33]. Some of the typing rules from [29] are mentioned below:

- Expression Type:  $\frac{}{\vdash_{exp: high}}$        $\frac{h \notin \text{Var}(exp)}{\vdash_{exp: low}}$
- Explicit-flow Rules:  $\frac{}{\vdash_{exp: pc} \quad [pc] \vdash \quad \vdash_{exp: low}}$
- Implicit-flow Rules:  $\frac{[pc] \vdash h := exp \quad [pc] \vdash \quad c_1 \quad [pc] \vdash \quad c_2}{[pc] \vdash \text{if } exp \text{ then } c_1 \text{ else } c_2}$        $\frac{\vdash_{exp: pc} \quad [pc] \vdash \quad c}{[pc] \vdash \text{while } exp \text{ do } c}$
- Subsumption Rule:  $\frac{[high] \vdash c}{[low] \vdash c}$

<sup>1</sup> This work is a revised and extended version of [10].

Fig. 1. A motivating HQL program  $P$  and its execution view

The notation  $[pc]$  denotes the security context which can be either  $[low]$  or  $[high]$ . According to the subsumption rule, if a program is typable in a high context then it is also typable in a low context. This allows to reset the program security context to low after a high conditional or a loop.

Although type-based approach is provably sound, but a major drawback is the lack of expressiveness. Moreover, it is not flow-sensitive which may produce false alarm. For instance, consider the following code:

```

① if(h=1) then
②     l := 10;
③ else l := 5;
    .....
⑦ l := 0;
⑧ output l;
```

Although the program is secure with respect to the classical noninterference principle as the output is always zero, but the type-based approach produces false alarm according to the implicit-flow rule.

As information flow is closely related to the dependence information of programs, the notion of Program Dependence Graph (PDG) is used widely to capture illegitimate flow in programs [7, 19–21, 24]. As PDGs are flow-sensitive, the analysis improves *w.r.t.* the type-based approach. For instance, in PDG-based approaches, the above code is secure as there is no path  $\textcircled{1} \xrightarrow{*} \textcircled{8}$  in the corresponding PDG. Various extensions of PDG exist, for example System Dependence Graph (SDG) in case of inter-procedural call to capture context-sensitivity, Class Dependence Graph (CIDG) in case of Object-Oriented Languages to capture object-sensitivity on dynamic dispatch, etc [20]. Once the dependence graph of a program is constructed, static analysis is performed on the graph to identify the presence of possible insecure flow. An worth mentioning approach is backward slicing which collects all possible paths (or source-nodes) influencing (directly/indirectly) the observable nodes: to be secure, the levels of variables in a path must not exceed the levels of observable variables in the output-node of that path. In other words, slicing helps to partition any insecure program (as a whole) in to secure and insecure part [7]. Semantics-based improvement (*e.g.* path-conditions) is also proposed to disregard semantically unreachable paths [20].

Approaches based on formal techniques, *e.g.* Abstract Interpretation theory, Hoare Logic, Model Checking, etc. are proposed in [1, 14, 22, 34, 35] to analyze secure information flow in software products. Leino and Joshi [22] first introduced a semantics-based approach to analyzing secure information flow based on the semantic equivalence of programs. [34, 35] defined the concrete semantics of programs and lift it to an abstract domain suitable for flow analysis. In particular, they consider the domain of propositional formula representing variables' dependences. The abstract semantics is further refined by combining with numerical abstract domain which improves the precision of the analysis. A variety of logical forms are proposed to characterize information flow security. Amtoft

and Banerjee [1] defined prelude semantics by treating program commands as prelude transformer. They introduced a logic based on the Abstract Interpretation of prelude semantics that makes independence between program variables explicit. They used Hoare logic and applied this logic to forward program slicing: forward  $l$ -slice is independent of  $h$  variables and is secure from information leakage. Authors in [2] defines a set of proof rules for secure information flow based on axiomatic approach. Recently, [14] proposed a model checking-based approach for reactive systems.

### 3 Syntax of HQL

Syntax of HQL is similar to object oriented constructs along with SQL variants through **Session** objects. The syntactic sets and the abstract syntax of HQL is depicted in Table 1. Like OOP, HQL programs are composed of a set of classes including **main** class. That is, a HQL program  $P$  is defined as  $P = \langle c_{main}, L \rangle$  where  $c_{main} \in \mathbf{Class}$  is the main class and  $L \subset \mathbf{Class}$  are the other classes. Similarly, a class  $c \in \mathbf{Class}$  contains a set of fields and methods, and therefore, is defined as a triplet  $c = \langle \mathbf{init}, F, M \rangle$ , where **init** is the constructor,  $F$  is the set of fields, and  $M$  is the set of member methods.

An additional and attractive feature of HQL is the presence of **Hibernate Session** which provides a central interface between the application and Hibernate and acts as persistence manager. In HQL, an object is transient if it has just been instantiated using the new operator. Transient instances will be destroyed by the garbage collector if the application does not hold a reference anymore. A persistent instance, on the other hand, has a representation in the database and an identifier value assigned to it. Given an object, the **Hibernate Session** is used to make the object persistent. Various methods in **Hibernate Session** are used to propagate object's states from memory to the database (or vice versa).

In abstract syntax, we denote a **Session** method by a triplet  $\langle C, \phi, OP \rangle$  where  $OP$  is the operation to be performed on the database tuples corresponding to a set of objects of classes  $c \in C$  satisfying the condition  $\phi$ . For instance, consider the following update statement which is invoked through a session object '**ses**':

```
Query Q = ses.createQuery('UPDATE std SET rank=rank+1 WHERE mark>500')
```

The abstract syntax of  $Q$  is denoted by

$$\langle C, \phi, OP \rangle = \langle \{\mathbf{std}\}, \mathbf{mark} > 500, \mathbf{rank} = \mathbf{rank} + 1 \rangle$$

The descriptions of  $OP$  in various **Session** methods are as follows:

- $\langle C, \phi, \mathbf{SAVE}(\mathbf{obj}) \rangle = \langle \{c\}, \mathbf{false}, \mathbf{SAVE}(\mathbf{obj}) \rangle$ : Stores the state of the object **obj** in the database table  $t$ , where  $t$  corresponds to the POJO class  $c$  and **obj** is the instance of  $c$ . The pre-condition  $\phi$  is *false* as the method does not identify any existing tuples in the database.

**Table 1.** Abstract syntax of HQL `session` methods

<b>Constants and Variables</b>	
$n \in \mathbb{N}$	Set of Integers
$v \in \mathbb{V}$	Set of Variables
<b>Arithmetic and Boolean Expressions</b>	
$exp \in \mathbb{E}$	Set of Arithmetic Expressions
$exp ::= n \mid v \mid exp_1 \oplus exp_2$ where $\oplus \in \{+, -, *, /\}$	
$b \in \mathbb{B}$	Set of Boolean Expressions
$b ::= true \mid false \mid exp_1 \otimes exp_2 \mid \neg b \mid b_1 \oslash b_2$ where $\otimes \in \{\leq, \geq, ==, >, \neq, \dots\}$ and $\oslash \in \{\vee, \wedge\}$	
<b>Well-formed Formulas</b>	
$\tau \in \mathbb{T}$	Set of Terms
$\tau ::= n \mid v \mid f_n(\tau_1, \tau_2, \dots, \tau_n)$ where $f_n$ is an $n$ -ary function.	
$a_f \in \mathbb{A}_f$	Set of Atomic Formulas
$a_f ::= R_n(\tau_1, \tau_2, \dots, \tau_n) \mid \tau_1 == \tau_2$ where $R_n(\tau_1, \tau_2, \dots, \tau_n) \in \{true, false\}$	
$\phi \in \mathbb{W}$	Set of Well-formed Formulas
$\phi ::= a_f \mid \neg \phi \mid \phi_1 \oslash \phi_2$ where $\oslash \in \{\vee, \wedge\}$	
<b>HQL Functions</b>	
$g(\vec{e}) ::= \text{GROUP BY}(e\vec{x}p) \mid id$ where $e\vec{x}p = \langle exp_1, \dots, exp_n \mid exp_i \in \mathbb{E} \rangle$	
$r ::= \text{DISTINCT} \mid \text{ALL}$	
$s ::= \text{AVG} \mid \text{SUM} \mid \text{MAX} \mid \text{MIN} \mid \text{COUNT}$	
$h(exp) ::= s \circ r(exp) \mid \text{DISTINCT}(exp) \mid id$	
$h(*) ::= \text{COUNT}(*)$ where $*$ represents a list of database attributes denoted by $\vec{v}_d$	
$\vec{h}(\vec{x}) ::= \langle h_1(x_1), \dots, h_n(x_n) \rangle$ where $\vec{h} = \langle h_1, \dots, h_n \rangle$ and $\vec{x} = \langle x_1, \dots, x_n \mid x_i = exp \vee x_i = * \rangle$	
$f(e\vec{x}p) ::= \text{ORDER BY ASC}(e\vec{x}p) \mid \text{ORDER BY DESC}(e\vec{x}p) \mid id$	
<b>Session Methods</b>	
$c \in \text{Class}$	Set of Classes
$c ::= \langle \text{init}, F, M \rangle$ where $\text{init}$ is the constructor, $F \subseteq \text{Var}$ is the set of fields, and $M$ is the set of methods.	
$m_{ses} \in M_{ses}$	Set of Session methods
$m_{ses} ::= \langle C, \phi, OP \rangle$ where $C \subseteq \text{Class}$	
$OP ::= \text{SEL}(f(e\vec{x}p'), r(\vec{h}(\vec{x})), \phi, g(e\vec{x}p))$ $\mid \text{UPD}(\vec{v}, e\vec{x}p)$ $\mid \text{SAVE}(\text{obj})$ $\mid \text{DEL}()$ where $\phi$ represents ‘HAVING’ clause and $\text{obj}$ denotes an instance of a class.	

- $\langle \mathbf{C}, \phi, \text{UPD}(\vec{x}, e\vec{x}p) \rangle = \langle \{c\}, \phi, \text{UPD}(\vec{v}, e\vec{x}p) \rangle$ : Updates the attributes corresponding to the class fields  $\vec{x}$  by  $e\vec{x}p$  in the database table  $t$  for the tuples satisfying  $\phi$ , where  $t$  corresponds to the POJO class  $c$ .
- $\langle \mathbf{C}, \phi, \text{DEL}() \rangle = \langle \{c\}, \phi, \text{DEL}() \rangle$ : Deletes the tuples satisfying  $\phi$  in  $t$ , where  $t$  is the database table corresponding to the POJO class  $c$ .
- $\langle \mathbf{C}, \phi, \text{SEL}(f(e\vec{x}p'), r(\vec{h}(\vec{x})), \phi', g(e\vec{x}p)) \rangle$ : Selects information from the database tables corresponding to the set of POJO classes  $\mathbf{C}$ , and returns the equivalent representations in the form of objects.

It is immediate that in case of  $\text{SAVE}()$  the condition  $\phi$  is *false* and  $\mathbf{C}$  is singleton set  $\{c\}$ . As  $\text{UPD}()$  and  $\text{DEL}()$  always target single class, the set  $\mathbf{C}$  is also singleton  $\{c\}$  in those cases. However,  $\mathbf{C}$  may not be singleton in case of  $\text{SEL}()$ .

## 4 Concrete Semantics of HQL

In this section, we define the semantics of HQL by (i) extending the OOP semantics [25] and (ii) defining the semantics of **Session** methods in terms of the semantics of database query languages [17].

### 4.1 Concrete Semantics of OOP [25]

Let **Var**, **Val** and **Loc** be the set of variables, the domain of values and the set of memory locations respectively. The set of environments, stores and states are defined below:

- The set of environments is defined as  $\text{Env} : \mathbf{Var} \longrightarrow \mathbf{Loc}$
- The set of stores is defined as  $\text{Store} : \mathbf{Loc} \longrightarrow \mathbf{Val}$
- The set of states is defined as  $\Sigma : \mathbf{Env} \times \mathbf{Store}$ . So, a state  $\rho \in \Sigma$  is denoted by a tuple  $\langle e, s \rangle$  where  $e \in \mathbf{Env}$  and  $s \in \mathbf{Store}$ .

Some special variables ( $pc$ ,  $V_{in}$ ,  $V_{out}$ ) are used in the subsequent part which represent the following: (i)  $\rho(pc)$  is the program counter; (ii)  $\rho(V_{in})$  is the input value of the current method; (iii)  $\rho(V_{out})$  is the value returned by the current method.

**Constructor and Method Semantics.** During object creation, the class constructor is invoked and object fields are instantiated by input values. Given a store  $s$ , the constructor maps its fields to fresh locations and then assigns values into those locations. Constructor never returns any output.

**Definition 1 (Constructor Semantics).** *Given a store  $s$ . Let  $\{a_{in}, a_{pc}\} \subseteq \mathbf{Loc}$  be the free locations,  $\mathbf{Val}_{in} \subseteq \mathbf{Val}$  be the semantic domain for input values. Let  $v_{in} \in \mathbf{Val}_{in}$  and  $pc_{exit}$  be the input value and the exit point of the constructor.*



**Table 2.** An example class

```

1.  class Demo {
2.      int k;
3.      Demo(int i) {
4.          k = i;
5.      }
6.      int even() {
7.          if(k % 2 == 0)
8.              return 1;
9.          else return 0;
10.     }
11.     int * mul( int j ) {
12.         k = k * j;
13.         return &k;
14.     }
15. }

```

The semantic of the class constructor *init*,  $S[\![init]\!] \in (Store \times Val \rightarrow \wp(Env \times Store))$ , is defined by

$$S[\![init]\!](s, v_{in}) = \{(e_0, s_0) \mid (e_0 \triangleq V_{in} \rightarrow a_{in}, pc \rightarrow a_{pc}) \wedge (s_0 \triangleq s[a_{in} \rightarrow v_{in}, a_{pc} \rightarrow pc_{exit}])\}$$

**Definition 2 (Method Semantics).** Let  $Val_{in} \subseteq Val$  and  $Val_{out} \subseteq Val$  be the semantic domains for the input values and the output values respectively. Let  $v_{in} \in Val_{in}$  be the input values,  $a_{in}$  and  $a_{pc}$  be the fresh memory locations, and  $pc_{exit}$  be the exit point of the method  $m$ . The semantic of a method  $m$ ,  $S[\![m]\!] \in (Env \times Store \times Val_{in} \rightarrow \wp(Store \times Env \times Val_{out}))$ , is defined as

$$S[\![m]\!](e, s, v_{in}) = \{(e', s', v_{out}) \mid (e' \triangleq e[V_{in} \rightarrow a_{in}, pc \rightarrow a_{pc}]) \wedge (s' \triangleq s[a_{in} \rightarrow v_{in}, a_{pc} \rightarrow pc_{exit}]) \wedge v_{out} \in Val_{out}\}$$

*Example 1.* Consider the example of Table 2. The class constructor *Demo()* creates a new environment consists of field  $k$ . The semantics of constructor *Demo()* and the semantics of the methods *even()* and *mul()* are defined below:

$$S[\![Demo()\!](s, i) = \{(e_0, s_0) \mid (e_0 \triangleq k \rightarrow a_{in}, pc \rightarrow a_{pc}) \wedge (s_0 \triangleq s[a_{in} \rightarrow i, a_{pc} \rightarrow 5])\}$$

$$S[\![even()\!](e, s, \emptyset) = \{(e, s', v_{out}) \mid (s' \triangleq s[pc \rightarrow 10]) \wedge (v_{out} = \text{if}(s(e(k)) \% 2) ? 1 : 0)\}$$

$$S[\![mul()\!](e, s, j) = \{(e, s', v_{out}) \mid (s' \triangleq s[e(k) \rightarrow s(e(k)) * j, pc \rightarrow 14]) \wedge v_{out} = e(k)\}$$

Observe that *even()* takes no input and returns an integer value as output, whereas *mul()* takes an integer value as input and returns an address as output.

**Object and Class Semantics.** Object semantics is defined in terms of interaction history between the program-context and the object. A direct interaction takes place when the program-context calls any member-method of the object, whereas an indirect interaction occurs when the program-context updates any address escaped from the object's scope. However, both direct or indirect interaction can cause a change in an interaction state (see Definition 3).

**Definition 3 (Interaction States).** *The set of interaction states is defined by*

$$\Sigma = \mathbf{Env} \times \mathbf{Store} \times \mathbf{Val}_{out} \times \wp(\mathbf{Loc})$$

where  $\mathbf{Env}$ ,  $\mathbf{Store}$ ,  $\mathbf{Val}_{out}$ , and  $\mathbf{Loc}$  are the set of application environments, the set of stores, the set of output values, and the set of addresses respectively.

**Definition 4 (Initial Interaction States).** *Let  $v_{in} \in \mathbf{Val}_{in}$  be an input to the class constructor  $\mathbf{init}$  when creating an object. Let  $s \in \mathbf{Store}$  be a store. Then the set of initial interaction states is defined by*

$$\mathcal{I}_0 = \{ \langle e_0, s_0, \phi, \emptyset \rangle \mid S[\![\mathbf{init}]\!](v_{in}, s) \ni \langle e_0, s_0 \rangle \}$$

Observe that  $\phi$  denotes no output produced by the constructor and  $\emptyset$  represents the empty context with no escaped address.

*Example 2 (Initial Interaction States).* Consider the example of Table 2. The input to the constructor is  $i$ . Given a store  $s$ , the initial interaction states are

$$\begin{aligned} \mathcal{I}_0 &= \{ \langle e_0, s_0, \phi, \emptyset \rangle \mid S[\![\mathbf{Sample}()]\!](i, s) \ni \langle e_0, s_0 \rangle \} \\ &= \{ \langle e_0, s_0, \phi, \emptyset \rangle \mid (e_0 \triangleq a \rightarrow a_{in}, pc \rightarrow a_{pc}) \wedge (s_0 \triangleq s[a_{in} \rightarrow i, a_{pc} \rightarrow 5]) \} \end{aligned}$$

Observe that the third element in an initial state is  $\phi$  because constructor does not return any value as output. Similarly the fourth element is  $\emptyset$  because no address is escaped from the object's scope after execution of  $\mathbf{sample}()$ .

*Transition Relation.* Let  $\mathbf{Lab} = (\mathbb{M} \times \mathbf{Val}_{in}) \cup \{\mathbf{upd}\}$  be a set of labels, where  $\mathbb{M}$  is the set of class-methods,  $\mathbf{Val}_{in}$  is the set of input values and  $\mathbf{upd}$  denotes an indirect update operation by the context. The transition relation  $\mathcal{T} : \Sigma \rightarrow \wp(\Sigma \times \mathbf{Lab})$  specifies which successor interaction states  $\sigma' = \langle e', s', v', \mathbf{Esc}' \rangle \in \Sigma$  can follow

1. when an object's methods  $m \in \mathbb{M}$  with input  $v_{in} \in \mathbf{Val}_{in}$  is directly invoked on an interaction state  $\sigma = \langle e, s, v, \mathbf{Esc} \rangle$  (**direct interaction**), or
2. the context indirectly updates an address escaped from an object's scope (**indirect interaction**).

**Definition 5 (Direct Interaction  $\mathcal{T}_{dir}$ ).** *Transition on Direct Interaction is defined below:*

$$\begin{aligned} \mathcal{T}_{dir}(\langle e, s, v, \mathbf{Esc} \rangle) &= \{ (\langle e', s', v', \mathbf{Esc}' \rangle, (m, v_{in})) \mid S[\![m]\!](\langle e, s, v_{in} \rangle) \ni \langle e', s', v' \rangle \\ &\quad \wedge \mathbf{Esc}' = \mathbf{Esc} \cup \mathbf{reach}(v', s') \} \end{aligned}$$

where

$$\text{reach}(v, s) = \begin{cases} \text{if } v \in \text{Loc} \\ \{v\} \cup \{\text{reach}(e'(f), s) \mid \exists B. B = \{\text{init}, F, M\}, f \in F, \\ s(v) \text{ is an instance of } B, s(s(v)) = e'\} \\ \text{else } \emptyset \end{cases}$$

*Example 3 (Direct interaction  $\mathcal{T}_{\text{dir}}$ ).* Consider the example of Table 2. The context can invoke any one of the two methods of **Sample** class. Therefore given an interaction state  $\sigma = \langle e, s, v, \text{Esc} \rangle$ , the set of successor interaction states are

$$\begin{aligned} \mathcal{T}_{\text{dir}}(\langle e, s, v, \text{Esc} \rangle) = & \{ (\langle e, s', v', \text{Esc} \rangle, (\text{parity}(), \phi)) \mid S[\llbracket \text{parity}() \rrbracket](\langle e, s, \phi \rangle) \ni \langle e, s', v' \rangle \} \\ & \cup \{ (\langle e, s', v', \text{Esc}' \rangle, (\text{incr}(), j)) \mid S[\llbracket \text{incr}() \rrbracket](\langle e, s, j \rangle) \ni \langle e, s', v' \rangle \} \\ & \wedge \text{Esc}' = \text{Esc} \cup \{v'\} \} \end{aligned}$$

**Definition 6 (Indirect Interaction  $\mathcal{T}_{\text{ind}}$ ).** *Transition on Indirect Interaction is defined below:*

$$\mathcal{T}_{\text{ind}}(\langle e, s, v, \text{Esc} \rangle) = \{ (\langle e, s', v, \text{Esc} \rangle, \text{upd}) \mid \exists a \in \text{Esc}. \text{Update}(a, s) \ni s' \}$$

where  $\text{Update}(a, s) = \{s' \mid \exists v \in \text{Val}. s' = s[a \leftarrow v]\}$

**Definition 7 (Transition Relation  $\mathcal{T}$ ).** *Let  $\sigma \in \Sigma$  be an interaction state. The transition relation  $\mathcal{T} : \Sigma \rightarrow \wp(\Sigma \times \text{Lab})$  is defined as  $\mathcal{T} = \mathcal{T}_{\text{dir}} \cup \mathcal{T}_{\text{ind}}$ , where  $\mathcal{T}_{\text{dir}}$  and  $\mathcal{T}_{\text{ind}}$  represent direct and indirect transitions respectively.*

Let us denote a transition between interaction states  $\sigma_1$  and  $\sigma_2$  by  $\sigma_1 \xrightarrow{\ell} \sigma_2$  where  $\ell \in \text{Lab}$ .

*Objects Fix-point Semantics.* Given a store  $s \in \text{Store}$ , the set of initial interaction states is defined as

$$\mathcal{I}_0 = \{ \langle e_0, s_0, \phi, \emptyset \rangle \mid S[\llbracket \text{init} \rrbracket](v_{\text{in}}, s) \ni \langle e_0, s_0 \rangle, v_{\text{in}} \in \text{Val}_{\text{in}} \}$$

The fix-point trace semantics of **obj**, according to [12], is defined as

$$\mathcal{T}[\llbracket \text{obj} \rrbracket](\mathcal{I}_0) = \text{lfp}_{\emptyset}^{\subseteq} \mathcal{F}(\mathcal{I}_0) = \bigcup_{i \leq \omega} \mathcal{F}^i(\mathcal{I}_0)$$

$$\text{where } \mathcal{F}(\mathcal{I}) = \lambda \mathcal{T}. \mathcal{I} \cup \left\{ \sigma_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \sigma_n \xrightarrow{\ell_n} \sigma_{n+1} \mid \sigma_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \sigma_n \in \mathcal{T} \wedge (\sigma_{n+1}, \ell_n) \in \mathcal{T}(\sigma_n) \right\}$$

*Class Semantics.* A class is nothing but a description of the set of objects. The semantics of a class **c** is defined as

$$S[\llbracket \mathbf{c} \rrbracket] = \cup \{ \mathcal{T}[\llbracket \text{obj} \rrbracket](\mathcal{I}_0) \mid \text{“obj” is an instance of a class } \mathbf{c} \text{ and } \mathcal{I}_0 \text{ is the set of initial interaction states} \}$$

Observe that the semantic definitions of objects and classes aim at verifying invariance properties of classes.

**Object-Oriented Program Semantics.** Let  $P = \langle c_{main}, L \rangle$  be an object-oriented program. Let  $\rightarrow \subseteq (\text{Env} \times \text{Store}) \times (\text{Env} \times \text{Store})$  be a transition relation and  $S_0 \in \wp(\text{Env} \times \text{Store})$  be a set of initial states such that  $\forall \rho_0 \in S_0. \rho_0(\text{currentMethod}) = c_{main}$  and  $\rho_0(pc) = pc_{main}$  where  $pc_{main}$  is the entry point of main method in  $c_{main}$ . The semantic of  $P$  is defined as

$$S[P](S_0) = \text{lfp}_{\emptyset}^{\subseteq} \lambda X. S_0 \cup \{ \rho_0 \rightarrow \rho_1 \rightarrow \dots \rightarrow \rho_n \rightarrow \rho_{n+1} \mid \rho_{n+1} \in (\text{Env} \times \text{Store}) \wedge \rho_0 \rightarrow \rho_1 \rightarrow \dots \rightarrow \rho_n \in X \wedge \rho_n \rightarrow \rho_{n+1} \}$$

## 4.2 Concrete Semantics of HQL

In order to define the semantics of HQL, let us recall the notion of database environment  $\rho_d$  and table environment  $\rho_t$  from [17].

*Database Environment.* We consider a database as a set of indexed tables  $\{t_i \mid i \in I_x\}$  for a given set of indexes  $I_x$ . We define database environment by a function  $\rho_d$  whose domain is  $I_x$ , such that for  $i \in I_x$ ,  $\rho_d(i) = t_i$ .

*Table Environment.* Given a database environment  $\rho_d$  and a table  $t \in d$ . We define  $\text{attr}(t) = \{a_1, a_2, \dots, a_k\}$ . So,  $t \subseteq D_1 \times D_2 \times \dots \times D_k$  where,  $a_i$  is the attribute corresponding to the typed domain  $D_i$ . A table environment  $\rho_t$  for a table  $t$  is defined as a function such that for any attribute  $a_i \in \text{attr}(t)$ ,

$$\rho_t(a_i) = \langle \pi_i(l_j) \mid l_j \in t \rangle$$

where  $\pi$  is the projection operator, i.e.  $\pi_i(l_j)$  is the  $i^{\text{th}}$  element of the  $l_j$ -th row. In other words,  $\rho_t$  maps  $a_i$  to the ordered set of values over the rows of the table  $t$ .

*Interaction State.* We extend the notion of interaction states of OOP to the case of HQL, considering the interaction of context with **Session** objects. To this aim, we include database environment in the definition of HQL states. The set of interaction states of HQL is, thus, defined by

$$\Sigma = \text{Env} \times \text{Store} \times \mathfrak{E}_d \times \text{Val}_{out} \times \wp(\text{Loc})$$

where  $\text{Env}$ ,  $\text{Store}$ ,  $\mathfrak{E}_d$ ,  $\text{Val}_{out}$ , and  $\text{Loc}$  are the set of application environments, the set of stores, the set of database environments, the set of output values, and the set of addresses respectively.

The set of initial interaction states of HQL is defined by

$$\mathcal{I}_0 = \{ \langle e_0, s_0, \rho_{d_0}, \phi, \emptyset \rangle \mid S[\text{init}](v_{in}, s) \ni \langle e_0, s_0 \rangle \}$$

where  $v_{in} \in \text{Val}_{in}$  is an input to the class constructor **init** when creating an object and  $s \in \text{Store}$  is a store.  $\rho_{d_0}$  is the initial database environment.

The semantics of conventional constructors, methods, objects, classes in HQL are defined in the same way as in the case of OOP. The **Session** methods require

an ‘ad-hoc’ treatment. We define its concrete semantics by specifying how the methods are executed on  $(e, s, \rho_d)$  where  $e \in \mathbf{Env}$  is an environment,  $s \in \mathbf{Store}$  is a store, and  $\rho_d \in \mathfrak{E}_d$  is a database environment, resulting into new state  $(e', s', \rho_{d'})$ . The semantic definitions are expressed in terms of the semantics of database statements **SELECT**, **INSERT**, **UPDATE**, **DELETE** [17].

We use the following functions in the subsequent part:  $map(v)$  maps  $v$  to the underlying database object;  $var(exp)$  returns the variables appearing in  $exp$ ;  $attr(t)$  returns the attributes associated with table  $t$ ;  $dom(f)$  returns the domain of  $f$ .

The semantic function is defined as:

$$S[\llbracket \mathbf{C}, \phi, \mathbf{op} \rrbracket\rrbracket(e, s, \rho_d) = \begin{cases} S[\llbracket \mathbf{C}, \phi, \mathbf{op} \rrbracket\rrbracket(e, s, \rho_{t'}) & \text{if } \exists t_1, \dots, t_n \in dom(\rho_d) : \mathbf{C} = \{c_1, \dots, c_n\} \\ & \wedge (\forall i \in [1 \dots n]. t_i = map(c_i)) \wedge t' = t_1 \times t_2 \times \dots \times t_n. \\ \perp & \text{otherwise.} \end{cases}$$

**Semantics of Session Method UPD().** Consider the **Session** method  $\langle \{c\}, \phi, \mathbf{UPD}(\vec{v}, e\vec{x}p) \rangle$ . The semantics is defined below<sup>2</sup>:

$$S[\llbracket \langle \{c\}, \phi, \mathbf{UPD}(\vec{v}, e\vec{x}p) \rangle \rrbracket\rrbracket$$

$$= \lambda(e, s, \rho_t). \text{ let } c = \langle \mathbf{init}, \mathbf{F}, \mathbf{M} \rangle \text{ such that } map(\mathbf{F}) = attr(t) \text{ and } map(\vec{v}) = \vec{a} \subseteq attr(t) \\ \text{ where } \vec{v} \subseteq \mathbf{F}, \text{ and let } \phi_d = \mathbf{PE}[\llbracket \phi \rrbracket\rrbracket(e, s, \mathbf{F}) \text{ and } e\vec{x}p_d = \mathbf{PE}[\llbracket e\vec{x}p \rrbracket\rrbracket(e, s, \mathbf{F}) \text{ in} \\ \{ \langle e, s, \rho_{t'} \rangle \mid \rho_{t'} \in S[\llbracket \langle \mathbf{UPDATE}(\vec{a}, e\vec{x}p_d), \phi_d \rangle \rrbracket\rrbracket(\rho_t)] \}.$$

The auxiliary function  $\mathbf{PE}[\llbracket X \rrbracket\rrbracket$  (which stands for partial evaluation) is used in the definition above to convert variables in  $X$  into the corresponding database objects. This is defined by

$$\mathbf{PE}[\llbracket X \rrbracket\rrbracket(e, s, \mathbf{F}) = X'$$

$$\text{where } X' = X[x_i/v_i] \text{ for all } v_i \in var(X) \text{ and } x_i = \begin{cases} map(v_i) & \text{if } v_i \in \mathbf{F} \\ E[\llbracket v_i \rrbracket\rrbracket(e, s) & \text{otherwise} \end{cases}$$

*Example 4.* Let us consider a POJO class **std** which corresponds to the database table  $t_1$  depicted in Table 3(a). Consider the following HQL code:

```
Query Q = ses.createQuery("UPDATE std SET rank = rank + 1, mark
                             = mark - 50 * 2 WHERE mark > 500");
int R = Q.executeUpdate();
```

<sup>2</sup> Observe that, for the sake of simplicity, we do not consider here the method **REFRESH()** which synchronize the in-memory objects state with that of the underlying database.

**Table 3.** After execution of the **UPDATE** operation

(a) Table $t_1$				(b) Table $t_2$ : After Updation			
$tsid$	$tmark$	$trank$	$tdno$	$tsid$	$tmark$	$trank$	$tdno$
1	800	5	3	1	700	6	3
2	400	10	2	2	400	10	2
3	600	7	3	3	500	8	3
4	1000	1	1	4	900	2	1

The abstract syntax of the **Session** method above is  $\langle \{c\}, \phi, \text{UPD}(\vec{v}, \vec{exp}) \rangle$ , where

- $\{c\} = \{\mathbf{std}\}$ ,
- $\phi = \text{"mark} > 500\text{"}$ ,
- $\text{UPD}(\vec{v}, \vec{exp}) = \text{UPD}(\langle \text{rank}, \text{mark} \rangle, \langle \text{rank} + 1, \text{mark} - 50 \times 2 \rangle)$

Given the table environment  $\rho_{t_1}$  in Table 3(a), the semantics is defined as:

$$\begin{aligned}
& S[\langle \{\mathbf{std}\}, (\text{mark} > 500), \text{UPD}(\langle \text{rank}, \text{mark} \rangle, \langle \text{rank} + 1, \text{mark} - 50 \times 2 \rangle) \rangle] \\
& = \lambda(e, s, \rho_{t_1}). \text{ let } \mathbf{std} = \langle \mathbf{std}(), \mathbf{F}, \mathbf{M} \rangle \text{ such that } \mathbf{F} = \langle \text{sid}, \text{mark}, \text{rank}, \text{dno} \rangle \text{ and} \\
& \quad \text{map}(\mathbf{F}) = \text{attr}(t) = \langle \text{tsid}, \text{tmark}, \text{trank}, \text{tdno} \rangle \text{ and} \\
& \quad \text{map}(\vec{v}) = \text{map}(\langle \text{rank}, \text{mark} \rangle) = \langle \text{trank}, \text{tmark} \rangle \subseteq \text{attr}(t), \text{ and let} \\
& \quad (\text{tmark} > 500) = \text{PE}[\langle \mathbf{std}.\text{mark} > 500 \rangle](e, s, \mathbf{F}) \text{ and} \\
& \quad \langle \text{trank} + 1, \text{tmark} - 50 \times 2 \rangle = \text{PE}[\langle \text{rank} + 1, \text{mark} - 50 \times 2 \rangle](e, s, \mathbf{F}) \text{ in} \\
& \quad \{ \langle e, s, \rho_{t_2} \rangle \mid \rho_{t_2} \in S[\langle \text{UPDATE}(\langle \text{trank}, \text{tmark} \rangle, \langle \text{trank} + 1, \text{tmark} - 50 \times 2 \rangle), \\
& \quad \quad (\text{tmark} > 500) \rangle](\rho_{t_1}) \}.
\end{aligned}$$

The resulting table environment  $\rho_{t_2}$  is shown in Table 3(b). The semantics of other **Session** methods are in Table 4.

**Fix-Point Semantics of HQL.** Let us define transition relation, considering nondeterministic executions, as  $\mathcal{T} : \mathbf{M}_{\text{ses}} \times \Sigma \rightarrow \wp(\Sigma)$ . This specifies which successor interaction states  $\sigma' = \langle e', s', \rho_{d'} \rangle \in \Sigma$  can follow when a **Session** method  $m_{\text{ses}} = \langle \mathbf{C}, \phi, \mathbf{op} \rangle \in \mathbf{M}_{\text{ses}}$  is invoked on an interaction state  $\sigma = \langle e, s, \rho_d \rangle$ . That is,

$$\mathcal{T}_{\text{ses}}[m_{\text{ses}}](\langle e, s, \rho_d \rangle) = \{ \langle e', s', \rho_{d'} \rangle \mid S[m_{\text{ses}}](\langle e, s, \rho_d \rangle) \ni \langle e', s', \rho_{d'} \rangle \wedge m_{\text{ses}} \in \mathbf{M}_{\text{ses}} \}$$

We now define the transition relation, by considering (i) the direct interaction, when a conventional method is directly invoked, (ii) the session interaction, when a **Session** method is invoked, and (iii) the indirect transition, when context updates any address escaped from the object's scope.

**Definition 8 (Transition Relation  $\mathcal{T}$ ).** Let  $\sigma \in \Sigma$  be an interaction state. The transition relation  $\mathcal{T} : \mathbf{Lab} \times \Sigma \rightarrow \wp(\Sigma)$  is defined as  $\mathcal{T} = \mathcal{T}_{\text{dir}} \cup \mathcal{T}_{\text{ind}} \cup \mathcal{T}_{\text{ses}}$ , where  $\mathcal{T}_{\text{dir}}$ ,  $\mathcal{T}_{\text{ind}}$  and  $\mathcal{T}_{\text{ses}}$  represent direct, indirect, and session transitions respectively. **Lab** represents the set of labels which include **Session** methods  $\mathbf{M}_{\text{ses}}$ , conventional class methods  $\mathbf{M}$ , and an indirect update operation **Upd** by the context.

**Table 4.** Semantics of Session methods

**The semantics of Session method  $\langle\{c\}, \phi, \text{SAVE}(\text{obj})\rangle$ :**

$$\begin{aligned} & S[\langle\{c\}, \phi, \text{SAVE}(\text{obj})\rangle] \\ &= S[\langle\{c\}, \text{false}, \text{SAVE}(\text{obj})\rangle] \\ &= \lambda(e, s, \rho_t). \text{ let } c = \langle \text{init}, F, M \rangle \text{ such that } \text{map}(F) = \text{attr}(t) = \vec{a}, \text{ and let} \\ & \quad s(e(\text{obj})) = e' \text{ such that } s(e'(F)) = \vec{val}, \text{ in} \\ & \quad \{ \langle e, s, \rho_{t'} \rangle \mid \rho_{t'} \in S[\langle \text{INSERT}(\vec{a}, \vec{val}), \text{false} \rangle](\rho_t) \}. \end{aligned}$$

**The semantics of Session method  $\langle\{c\}, \phi, \text{DEL}()\rangle$ :**

$$\begin{aligned} & S[\langle\{c\}, \phi, \text{DEL}()\rangle] \\ &= \lambda(e, s, \rho_t). \text{ let } c = \langle \text{init}, F, M \rangle \text{ such that } \text{map}(F) = \text{attr}(t) = \vec{a} \text{ and let } \phi_d = \text{PE}[\phi](e, s, F) \\ & \quad \text{in } \{ \langle e, s, \rho_{t'} \rangle \mid \rho_{t'} \in S[\langle \text{DELETE}(\vec{a}), \phi_d \rangle](\rho_t) \} \end{aligned}$$

**The semantics of  $\langle C, \phi, \text{SEL}(f(\vec{exp}'), r(\vec{h}(\vec{x})), \phi', g(\vec{exp})) \rangle$ :**

$$\begin{aligned} & S[\langle C, \phi, \text{SEL}(f(\vec{exp}'), r(\vec{h}(\vec{x})), \phi', g(\vec{exp})) \rangle] \\ &= \lambda(e, s, \rho_t). \text{ let } C = \{ \langle \text{init}_i, F_i, M_i \rangle \mid i = 1, \dots, n \}, \text{ and } F = \bigcup_{i=1, \dots, n} F_i, \text{ and} \\ & \quad \langle \vec{exp}'_d, \vec{x}_d, \phi'_d, \vec{exp}_d, \phi_d \rangle = \text{PE}[\langle \vec{exp}', \vec{x}, \phi', \vec{exp}, \phi \rangle](e, s, F), \text{ and let} \\ & \quad \rho_{t'} = S[\langle \text{SELECT}(f(\vec{exp}'_d), r(\vec{h}(\vec{x}_d)), \phi'_d, g(\vec{exp}_d)), \phi_d \rangle](\rho_t) \text{ and} \\ & \quad (e', s') = \bigsqcup_{l_i \in t'} S[\text{Object}()](s, \text{val}(l_i)) \text{ in } \{ \langle e', s', \rho_{t'} \rangle \}. \end{aligned}$$

Observe that  $\text{val}(l_i)$  converts each tuple  $l_i \in t'$  into input values, and  $S[\text{Object}()](s, \text{val}(l_i))$  invokes the object constructor  $\text{Object}()$  which creates an object by initializing the fields with  $\text{val}(l_i)$ . This is done for all tuples  $l_i \in t'$ , resulting into new  $(e', s')$ .

We denote a transition by  $\sigma \xrightarrow{a} \sigma'$  when application of a label  $a \in \text{Lab}$  on interaction state  $\sigma$  results into a new state  $\sigma'$ .

Let  $\mathcal{I}_0$  be the set of initial interaction states. The fix-point trace semantics of HQL program  $P$  is defined as

$$\mathcal{T}[\![P]\!](\mathcal{I}_0) = \text{lfp}_{\emptyset}^{\subseteq} \mathcal{F}(\mathcal{I}_0) = \bigcup_{i \leq \omega} \mathcal{F}^i(\mathcal{I}_0)$$

$$\text{where } \mathcal{F}(\mathcal{I}) = \lambda \mathcal{T}. \mathcal{I} \cup \{ \sigma_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-1}} \sigma_n \xrightarrow{a_n} \sigma_{n+1} \mid \sigma_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-1}} \sigma_n \in \mathcal{T} \\ \wedge \sigma_n \xrightarrow{a_n} \sigma_{n+1} \in \mathcal{T} \}$$

## 5 Abstract Semantics of HQL

Abstract interpretation [11, 12] provides a general theoretical foundation to specify static analyses, to guarantee their correctness, to tune their precision according to efficiency issues, and to compare and to combine them in a modular

way. It allows to deal separately with concerns that typically interleave, including fix-point algorithms, abstract domains, and termination criteria handled by widening operators. Its theoretical and practical impact has been demonstrated in various application fields, in particular for safety and security analysis.

In [34,35], authors used the Abstract Interpretation framework to define an abstract semantics of imperative languages using symbolic domain of positive propositional formulae in the form

$$\bigwedge_{0 \leq i \leq n, \ 0 \leq j \leq m} \{y_i \rightarrow z_j\}$$

which means that the values of variable  $z_j$  possibly depend on the values of variable  $y_i$ . Later, [18] extends this to the case of structured query languages. The computation of abstract semantics of a program in the propositional formulae domain provides a sound approximation of variable dependences, which allows to capture possible information flow in the program. The information leakage analysis is then performed by checking the satisfiability of formulae after assigning truth values to variables based on their sensitivity levels.

Let  $\mathbf{Pos}$  and  $\mathbb{L}$  be the domain of propositional formulae and the set of program points respectively. An abstract state  $\sigma^\sharp \in \Sigma^\sharp \equiv \mathbb{L} \times \mathbf{Pos}$  is a pair  $\langle \ell, \psi \rangle$  where  $\psi \in \mathbf{Pos}$  represents the variable dependences, in the form of propositional formulae, among program variables up to the program label  $\ell \in \mathbb{L}$ .

Methods in HQL include a set of imperative statements<sup>3</sup>. We assume, for the sake of the simplicity, that attackers are able to observe public variables inside of the main method only. Therefore, our analysis only aims at identifying variable dependences at input-output labels of methods.

The abstract transition semantics of constructors and conventional methods are defined below.

**Definition 9 (Abstract Transition Semantics of Constructor).** *Consider a class  $c = \langle \mathit{init}, F, M \rangle$  where  $\mathit{init}$  is a default constructor. Let  $\ell$  be the label of  $\mathit{init}$ . The abstract transition semantics of  $\mathit{init}$  is defined as*

$$\mathcal{T}^\sharp[\llbracket^\ell \mathit{init}\rrbracket] = \{(\ell, \psi) \rightarrow (\mathit{Succ}(\ell \mathit{init}), \psi)\}$$

where  $\mathit{Succ}(\ell \mathit{init})$  denotes the successor label of  $\mathit{init}$ . Observe that the default constructor is used to initialize the objects-fields only, and it does not add any new dependence.

The abstract transition semantics of parameterized constructors are defined in the same way as of class methods  $m \in \mathbf{M}$ .

**Definition 10 (Abstract Transition Semantics of Methods).** *Let  $U \in \wp(\mathbf{Var})$  be the set of variables which are passed as actual parameters when invoked a method  $m \in \mathbf{M}$  on an abstract state  $(\ell, \psi)$  at program label  $\ell$ . Let  $V \in \wp(\mathbf{Var})$  be*

<sup>3</sup> For a detailed abstract transition semantics of imperative statements, see [34].



the formal arguments in the definition of  $m$ . We assume that  $U \cap V = \emptyset$ . Let  $a$  and  $b$  be the variable returned by  $m$  and the variable to which the value returned by  $m$  is assigned. The abstract transition semantics is defined as

$$\mathcal{T}^\#[\ell m] = \{(\ell, \psi) \rightarrow (\text{Succ}(\ell m), \psi')\}$$

where  $\psi' = \{x_i \rightarrow y_i \mid x_i \in U, y_i \in V\} \cup \{a \rightarrow b\} \cup \psi$  and  $\text{Succ}(\ell m)$  is the label of the successor of  $m$ .

Let us classify the high-level HQL variables into two distinct sets:  $\text{Var}_d$  and  $\text{Var}_a$ . The variables which have a correspondence with database attributes belong to the set  $\text{Var}_d$ . Otherwise, the variables are treated as usual variables and belong to  $\text{Var}_a$ . We denote variables in  $\text{Var}_d$  by the notation  $\bar{v}$ , in order to differentiate them from the variables in  $\text{Var}_a$ . This leads to four types of dependences which may arise in HQL programs:  $x \rightarrow y$ ,  $\bar{x} \rightarrow y$ ,  $x \rightarrow \bar{y}$  and  $\bar{x} \rightarrow \bar{y}$ , where  $x, y \in \text{Var}_a$  and  $\bar{x}, \bar{y} \in \text{Var}_d$ .

**Definition of Abstract Transition Function  $\mathcal{T}^\#$  for Session methods.** The abstract labeled transition semantics of various **Session** methods are defined below, where by  $\text{Var}(exp)$  and  $\text{Field}(c)$  we denote the set of variables in  $exp$  and the set of class-fields in the class  $c$  respectively. The function  $\text{Map}(v)$  is defined as:

$$\text{Map}(v) = \begin{cases} \bar{v} & \text{if } v \text{ has correspondence with a database attribute,} \\ v & \text{otherwise.} \end{cases}$$

Notice that in the definition of  $\mathcal{T}^\#$  the notation  $\tilde{v}$  stands for either  $v$  or  $\bar{v}$ . Let  $\text{SF}(\psi)$  denotes the set of subformulas in  $\psi$ , and the operator  $\ominus$  is defined by  $\psi_1 \ominus \psi_2 = \bigwedge (\text{SF}(\psi_1) \setminus \text{SF}(\psi_2))$ .

### The transition semantics for Session method $m_{save}$

$$\begin{aligned} & \mathcal{T}^\#[\ell m_{save}] \\ & \stackrel{def}{=} \mathcal{T}^\#[\ell (\text{C}, \phi, \text{SAVE}(\text{obj}))] \\ & \stackrel{def}{=} \mathcal{T}^\#[\ell (\{c\}, \text{FALSE}, \text{SAVE}(\text{obj}))] \\ & \stackrel{def}{=} \{(\ell, \psi) \xrightarrow{\text{SAVE}} \langle \text{Succ}(\ell m_{save}), \psi' \rangle\} \end{aligned}$$

### The transition semantics for Session method $m_{upd}$

$$\begin{aligned} & \mathcal{T}^\#[\ell m_{upd}] \\ & \stackrel{def}{=} \mathcal{T}^\#[\ell (\text{C}, \phi, \text{UPD}(\bar{v}, e\bar{x}p))] \\ & \stackrel{def}{=} \mathcal{T}^\#[\ell (\{c\}, \phi, \text{UPD}(\bar{v}, e\bar{x}p))] \\ & \stackrel{def}{=} \{(\ell, \psi) \xrightarrow{\text{UPD}} \langle \text{Succ}(\ell m_{upd}), \psi' \rangle\} \\ & \text{where } \psi' = \bigwedge \{ \tilde{y} \rightarrow \bar{z}_i \mid y \in \text{Var}[\phi], \tilde{y} = \text{Map}(y), \bar{z}_i \in \bar{v} \} \cup \end{aligned}$$

$\bigwedge \{ \tilde{y}_i \rightarrow \bar{z}_i \mid y_i \in \mathbf{Var}[\![exp_i]\!], exp_i \in e\vec{x}p, \tilde{y}_i = \mathbf{Map}(y_i), \bar{z}_i \in \vec{v} \} \cup \psi''$   
 and  $\psi'' = \begin{cases} \psi \ominus (\tilde{a} \rightarrow \bar{z}_i \mid \bar{z}_i \in \vec{v} \wedge a \in \mathbf{Var} \wedge \tilde{a} = \mathbf{Map}(a)) & \text{if } \phi \text{ is TRUE by default.} \\ \psi & \text{otherwise} \end{cases}$

### The transition semantics for Session method $m_{del}$

$\mathcal{T}^\#[\![^\ell m_{del}]\!]$   
 $\stackrel{def}{=} \mathcal{T}^\#[\![^\ell(\mathbf{C}, \phi, \mathbf{DEL}())]\!]$   
 $\stackrel{def}{=} \mathcal{T}^\#[\![^\ell(\{c\}, \phi, \mathbf{DEL}())]\!]$   
 $\stackrel{def}{=} \{ \langle \ell, \psi \rangle \xrightarrow{\mathbf{DEL}} \langle \mathbf{Succ}(\ell m_{del}), \psi' \rangle \}$   
 where  $\psi' = \bigwedge \{ \tilde{y} \rightarrow \bar{z} \mid y \in \mathbf{Var}[\![\phi]\!], \tilde{y} = \mathbf{Map}(y), \bar{z} \in \mathbf{Field}(c) \} \cup \psi''$   
 and  $\psi'' = \begin{cases} \psi \ominus (\tilde{a} \rightarrow \bar{z}_i \mid \bar{z}_i \in \vec{v} \wedge a \in \mathbf{Var} \wedge \tilde{a} = \mathbf{Map}(a)) & \text{if } \phi \text{ is TRUE by default.} \\ \psi & \text{otherwise} \end{cases}$

### The transition semantics for Session method $m_{sel}$

$\mathcal{T}^\#[\![^\ell m_{sel}]\!]$   
 $\stackrel{def}{=} \mathcal{T}^\#[\![^\ell(\mathbf{C}, \phi, \mathbf{SEL}(f(e\vec{x}p'), r(\vec{h}(\vec{x})), \phi, g(e\vec{x}p))]\!]$   
 $\stackrel{def}{=} \{ \langle \ell, \psi \rangle \xrightarrow{\mathbf{SEL}} \langle \mathbf{Succ}(\ell m_{sel}), \psi' \rangle \}$   
 where  $\psi' = \bigwedge \{ \tilde{y} \rightarrow \tilde{z} \mid y \in (\mathbf{Var}[\![\phi]\!] \cup \mathbf{Var}[\![\vec{e}]\!] \cup \mathbf{Var}[\![\phi']]\!] \cup \mathbf{Var}[\![\vec{e}']]\!]), z \in \mathbf{Var}[\![\vec{x}]\!], \tilde{y} = \mathbf{Map}(y), \tilde{z} = \mathbf{Map}(z) \} \cup \psi$

## 6 Information Leakage Analysis

We are now in position to use the abstract semantics defined in the previous section to identify possible sensitive database information leakage through high-level HQL variables. After obtaining over-approximation of variable dependences at each program point, we assign truth values to each variable based on their sensitivity level, and we check the satisfiability of propositional formulae representing variable dependences [34].

Since our main objective is to identify the leakage of sensitive database information possibly due to the interaction of high-level variables, we assume, according to the policy, that different security classes are assigned to database attributes. Accordingly, we assign security levels to the variables in  $\mathbf{Var}_d$  based on the correspondences. Similarly, we assign the security levels of the variables in  $\mathbf{Var}_a$  based on their use in the program. For instance, the variables which are used on output channels, are considered as public variables. Observe that for the variables with unknown security class, it may be computed based on the dependence of it on the other application variables or database attributes of known security classes.

Let  $\Gamma : \mathbf{Var} \rightarrow \{L, H, N\}$  be a function that assigns to each of the variables a security class, either public ( $L$ ) or private ( $H$ ) or unknown ( $N$ ).

After computing abstract semantics of HQL program  $P$ , the security class of variables with unknown level ( $N$ ) in  $P$  are upgraded to either  $H$  or  $L$ , according to the following function:

$$\text{Upgrade}(v) = Z \text{ if } \exists (u \rightarrow v) \in \mathcal{S}^\sharp[P]. \Gamma(u) = Z \wedge \Gamma(u) \neq N \wedge \Gamma(v) = N$$

We say that program  $P$  respects the confidentiality property of database information, if and only if there is no information flow from private to public attributes. To verify this property, a corresponding truth assignment function  $\bar{\Gamma}$  is used:

$$\bar{\Gamma}(x) = \begin{cases} T & \text{if } \Gamma(x) = H \\ F & \text{if } \Gamma(x) = L \end{cases}$$

If  $\bar{\Gamma}$  does not satisfy any propositional formula in  $\psi$  of an abstract state, the analysis will report a possible information leakage.

Let us illustrate this on the running example program  $P$  in Fig. 1. According to the policy, let the database attribute corresponding to variable  $h_1$  is private, whereas the attributes corresponding to  $id_1$ ,  $id_2$  and  $l_2$  are public. Therefore,

$$\Gamma(\bar{h}_1) = H \text{ and } \Gamma(\bar{id}_1) = \Gamma(\bar{id}_2) = \Gamma(\bar{l}_2) = L$$

For other variables in the program, the security levels are unknown. That is,

$$\Gamma(R_1.[0]) = \Gamma(R_1.[1]) = \Gamma(\text{obj}[0]) = \Gamma(\text{obj}[1]) = \Gamma(pk) = \Gamma(h_2) = N$$

Considering the domain of positive propositional formulae, the abstract semantics yields the following formulae at program point 20 in  $P$ :

$$\begin{array}{llll} \bar{id}_1 \rightarrow R_1.[0]; & \bar{h}_1 \rightarrow R_1.[1]; & R_1.[0] \rightarrow \text{obj}[0]; & R_1.[1] \rightarrow \text{obj}[1]; \\ \text{obj}[0] \rightarrow pk; & \text{obj}[1] \rightarrow h_2; & pk \rightarrow \bar{l}_2; & \bar{id}_2 \rightarrow \bar{l}_2; \quad h_2 \rightarrow \bar{l}_2; \end{array}$$

According to the  $\text{Upgrade}()$  function, the security levels of the variables with unknown security level  $N$  are upgraded as below:

$$\begin{array}{l} \Gamma(R_1.[0]) = L, \Gamma(R_1.[1]) = H, \Gamma(\text{obj}[0]) = L, \Gamma(\text{obj}[1]) = H \\ \Gamma(pk) = L, \quad \Gamma(h_2) = H \end{array}$$

Finally, we apply the truth assignment function  $\bar{\Gamma}$  which does not satisfy the formula  $h_2 \rightarrow \bar{l}_2$ , as  $\bar{\Gamma}(h_2) = T$  and  $\bar{\Gamma}(\bar{l}_2) = F$  and  $T \rightarrow F$  is false.

Therefore, the analysis reports that the example program  $P$  is vulnerable to leakage of confidential database data.

## 7 Conclusions

We proposed a static analysis framework to perform information flow analysis of HQL based on the Abstract Interpretation framework. Our approach captures information leakage on “permanent” data stored in a database when a HQL program manipulates them. This may also lead to a refinement of the

non-interference definition that focusses on confidentiality of the data instead of variables. We are now investigating a possible enhancement of the analysis integrating with other abstract domains. As various aggregate operations are often performed on persistent data in HQL, to consider declassification policies [30] is also our future aim. We are currently working on designing and implementing a prototype based on our proposed approach.

**Acknowledgement.** This work is partially supported by PRIN “Security Horizons” project and by the research grant (SB/FTP/ETA-315/2013) from the Science & Engineering Research Board (SERB), Department of Science and Technology, Government of India. We thank the anonymous reviewers for their valuable comments and suggestions.

## References

1. Amtoft, T., Banerjee, A.: A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Sci. Comput. Program.* **64**, 3–28 (2007)
2. Andrews, G.R., Reitman, R.P.: An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.* **2**, 56–76 (1980)
3. Bao, T., Zheng, Y., Lin, Z., Zhang, X., Xu, D.: Strict control dependence and its effect on dynamic information flow analyses. In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pp. 13–24. ACM Press, Trento (2010)
4. Barbon, G., Cortesi, A., Ferrara, P., Pistoia, M., Tripp, O.: Privacy analysis of android apps: implicit flows and quantitative analysis. In: Saeed, K., Homenda, W. (eds.) *CISIM 2015. LNCS*, vol. 9339, pp. 3–23. Springer, New York (2015)
5. Bauer, C., King, G.: *Hibernate in Action*. Manning Publications Co., Greenwich (2004)
6. Bauer, C., King, G.: *Java Persistence with Hibernate*. Manning Publications Co., Greenwich (2006)
7. Cavadini, S.: Secure slices of insecure programs. In: *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pp. 112–122. ACM Press, Tokyo (2008)
8. Cortesi, A., Dovier, A., Quintarelli, E., Tanca, L.: Operational and abstract semantics of the query language G-log. *Theor. Comput. Sci.* **275**(1–2), 521–560 (2002)
9. Cortesi, A., Ferrara, P., Pistoia, M., Tripp, O.: Datacentric semantics for verification of privacy policy compliance by mobile applications. In: D’Souza, D., Lal, A., Larsen, K.G. (eds.) *VMCAI 2015. LNCS*, vol. 8931, pp. 61–79. Springer, Heidelberg (2015)
10. Cortesi, A., Halder, R.: Information-flow analysis of hibernate query language. In: Dang, T.K., Wagner, R., Neuhold, E., Takizawa, M., Küng, J., Thoai, N. (eds.) *FDSE 2014. LNCS*, vol. 8860, pp. 262–274. Springer, Heidelberg (2014)
11. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the POPL 1977*, pp. 238–252. ACM Press, Los Angeles (1977)
12. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 269–282. ACM Press, San Antonio (1979)

13. Denning, D.E.: A lattice model of secure information flow. *Commun. ACM* **19**, 236–243 (1976)
14. Dimitrova, R., Finkbeiner, B., Kovács, M., Rabe, M.N., Seidl, H.: Model checking information flow in reactive systems. In: Kuncak, V., Rybalchenko, A. (eds.) *VMCAI 2012*. LNCS, vol. 7148, pp. 169–185. Springer, Heidelberg (2012)
15. Elliott, J., O'Brien, T., Fowler, R.: *Harnessing Hibernate*, 1st edn. O'Reilly, Sebastopol (2008)
16. Halder, R.: Language-based security analysis of database applications. In: *Proceedings of the 3rd International Conference on Computer, Communication, Control and Information Technology (C3IT 2015)*, pp. 1–4. IEEE Press (2015)
17. Halder, R., Cortesi, A.: Abstract interpretation of database query languages. *Comput. Lang. Syst. Struct.* **38**, 123–157 (2012)
18. Halder, R., Zanioli, M., Cortesi, A.: Information leakage analysis of database query languages. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC 2014)*, 24–28 March 2014, pp. 813–820. ACM Press, Gyeongju (2014)
19. Hammer, C.: Experiences with PDG-based IFC. In: Massacci, F., Wallach, D., Zannone, N. (eds.) *ESSoS 2010*. LNCS, vol. 5965, pp. 44–60. Springer, Heidelberg (2010)
20. Hammer, C., Krinke, J., Snelting, G.: Information flow control for java based on path conditions in dependence graphs. In: *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, pp. 87–96. IEEE, Arlington (2006)
21. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Secur.* **8**, 399–422 (2009)
22. Joshi, R., Leino, K.R.M.: A semantic approach to secure information flow. *Sci. Comput. Program.* **37**(1–3), 113–138 (2000)
23. Krinke, J.: Information flow control and taint analysis with dependence graphs. In: *Proceedings of the Third International Workshop on Code Based Software Security Assessments (CoBaSSA)*. Technical report TUD-SERG-2007-023, Vancouver, Canada, Delft University of Technology, pp. 6–9 (2007)
24. Li, B.: Analyzing information-flow in java program based on slicing technique. *SIGSOFT Softw. Eng. Notes* **27**, 98–103 (2002)
25. Logozzo, F.: Class invariants as abstract interpretation of trace semantics. *Comput. Lang. Syst. Struct.* **35**, 100–142 (2009)
26. Mantel, H., Sudbrock, H.: Types vs. PDGs in information flow analysis. In: Albert, E. (ed.) *LOPSTR 2012*. LNCS, vol. 7844, pp. 106–121. Springer, Heidelberg (2013)
27. Myers, A.C.: Jflow: practical mostly-static information flow control. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999)*, January 20–22 1999, pp. 228–241. ACM Press, San Antonio (1999)
28. Pottier, F., Simonet, V.: Information flow inference for ML. *ACM Trans. Program. Lang. Syst.* **25**, 117–158 (2003)
29. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE J. Sel. Areas Commun.* **21**, 5–19 (2003)
30. Sabelfeld, A., Sands, D.: Declassification: dimensions and principles. *J. Comput. Secur.* **17**, 517–548 (2009)
31. Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: *Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007*, pp. 203–217. IEEE Computer Society, Washington DC (2007). <http://dx.doi.org/10.1109/CSF.2007.20>

32. Smith, G.: Principles of secure information flow analysis. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (eds.) *Malware Detection. Advances in Information Security*, vol. 27, pp. 291–307. Springer, Nov Smokovec (2007)
33. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. *J. Comput. Secur.* **4**, 167–187 (1996)
34. Zanioli, M., Cortesi, A.: Information leakage analysis by abstract interpretation. In: Černá, I., Gyimóthy, T., Hromkovič, J., Jefferey, K., Královič, R., Vukolić, M., Wolf, S. (eds.) *SOFSEM 2011. LNCS*, vol. 6543, pp. 545–557. Springer, Heidelberg (2011)
35. Zanioli, M., Ferrara, P., Cortesi, A.: Sails: static analysis of information leakage with sample. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC 2012)*, pp. 1308–1313. ACM Press, Trento (2012)

Transactions on Large-Scale Data- and

Knowledge-Centered Systems XXIII

Selected Papers from FDSE 2014

Hameurlain, A.; Küng, J.; Wagner, R.; Dang, T.K.; Thoai,  
N. (Eds.)

2016, IX, 125 p. 50 illus. in color., Softcover

ISBN: 978-3-662-49174-4