Verification of Hibernate Query Language by Abstract Interpretation

Angshuman Jana[1], Raju Halder[1], and Agostino Cortesi[2]

[1] Indian Institute of Technology Patna, India, {ajana.pcs13, halder}@iitp.ac.in
[2] Università Ca' Foscari Venezia, Italy, cortesi@unive.it

Abstract. In this paper, we propose an abstract interpretation framework of Hibernate Query Language (HQL), aiming at automatically and formally verifying enterprise policy specifications on persistent objects which have permanent representation in the underlying database. To this aim, we extend the abstract interpretation approach for object-oriented languages, combined with an abstract semantics of structured query languages.

Key words: Hibernate Query Language, Static Analysis and Verification, Abstract Interpretation

1 Introduction

Hibernate Query Language (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 [1, 2, 8]. HQL is treated as an object-oriented variant of SQL, which allows to represent SQL queries in object-oriented terms and mitigates the paradigm mismatch between object modeling and relational modeling. Hibernate is basically an object-relational mapping tool that simplifies the data creation, data manipulation and data access. Various methods in "`Session`" are used to propagate object's states from memory to the database (or vice versa) and to synchronize both states when a change is made to persistent objects [13]. 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.

It is particularly important, in this context, to provide formal verification methods for behavioral properties like absence of run-time errors, absence of confidential information leakage, etc. Abstract Interpretation [7] is a well-established semantics-based static analysis framework which provides a sound approximation of program semantics focussing on a particular property. The intuition of Abstract Interpretation is to lift the concrete semantics to an abstract domain, by replacing concrete values by suitable properties of interest and simulating the operations in the abstract domain *w.r.t.* their concrete counterparts, in order to ensure the soundness.

F. Logozzo [11] introduced an Abstract Interpretation-based framework of Object-Oriented Programming (OOP) languages, aiming at verifying whether the programs respect the specifications correctly. The framework is used to ensure the class invariant, a property which is valid for all the instances of the class, before and after the execution of any method. Moreover, it can also be used for optimization of the code at class-level. For instance, if a class invariant states that the class will never throw a given exception, then in the corresponding code for throwing/handling the exception can be dropped.

Unfortunately, the usual framework of Abstract Interpretation of OOP languages [11, 12, 3] can not be directly applied to Hibernate Query Language (HQL) if one wants to verify the properties of persistent objects only, rather than transient objects, which have permanent representation in the underlying database. On the other side, the existing work on abstract interpretation of query languages [9] did not consider an access to the database operations through a high-level object-oriented language. The aim of this paper is to fill the gap between these two theories.

As an example, consider the HQL program depicted in Figure 1. The `Session` methods of this program allow to update information (like age and salary) of the employees and to make simple queries to that database [3]. Consider the following enterprise policies given by the following three constraints:

Policy 1: *Employees age should be greater than or equal to 18 and less than or equal to 62.*

Policy 2: *The salary of employees with age greater than 30 should be at least 1500 euro.*

Policy 3: *Employees salary should not be more than three times of the lowest salary.*

Figure 2 depicts different states of the underlying database table when various `Session` methods of the program are executed. For instance, after executing statement 13, a tuple corresponding to the object 'obj' of class `emp` is inserted into the corresponding database state t_1, resulting in a new state t_2. Similarly, update and delete operations on the objects at 14-16 and 19-20, by the corresponding `Session` methods yield the states t_3 and t_5 respectively. Observe that, selection of objects at 17-18 produces the result shown in table t_{sel}, and of course, it does not change the database state (*i.e.* $t_3 = t_4$). The code satisfies policy 1, whereas it does not satisfy policies 2 and 3.

This can be formally and automatically verified by extending the Abstract Interpretation theory to the case of HQL: in general, it can be applied to formally verify some properties of persistent objects which have permanent representation in the underlying relational databases (or to find possible violation of the policy rules), by analyzing the HQL code on non-relational or relational abstract domains [4, 5]. The key point is the formalization of the abstract semantics of `Session` methods relating persistent objects to the database [9].

The structure of the paper is as follows: Sections 2 and 3 recall the basics on the concrete/abstract semantics of query languages and object-oriented languages respectively. We formalize the concrete and abstract semantics of HQL in Sections 4 and 5 respectively, by showing its applications in a simple yet general example. Section 6 concludes.

[3] Observe at program points 13, 14-16, 17-18 that the basic differences between HQL and SQL.

```
class emp {
    private int id, age, dno, sal;
    emp(){ }
    public int getId() { return id;}
    public void setId(int id) { this.id = id;}
    public int getage() {return age;}
    public void setage(int age) { this.age = age;}
    public int getdno() { return dno;}
    public void setdno(int dno) { this.dno = dno;}
    public int getsalary() { return sal;}
    public void setsalary(int sal) { this.sal = sal;}
}
```

(a) POJO Class emp

```
public class ExClass{
 public static void main(String[] args) {

  Configuration cfg=new Configuration();
  cfg.configure("hibernate.cfg.xml");
  SessionFactory sf=cfg.buildSessionFactory();
  Session ses=sf.openSession();
  Transaction tr=ses.beginTransaction();

% Creating emp object and stores into database %
  emp obj=new emp( );
  obj.setId(4);
  obj.setage(32);
  obj.setdno(1);
  obj.setsalary(1000);
  ses.save( obj );

% Updating persistent emp objects %
  Query q1 = ses.createQuery("UPDATE emp e SET e.age= e.age+1, e.sal= e.sal + :inc×2 WHERE
                                                                         e.sal > 1600");
  q1.setParameter("inc",100);
  int r1 = q1.executeUpdate();

% Selecting from persistent emp objects %
  Query q2 = ses.createQuery("SELECT e.dno, MAX(e.sal), AVG(DISTINCT e.age) FROM emp e
          WHERE e.sal ≥ 1000 GROUP BY e.dno HAVING MAX(e.sal) < 4000 ORDER BY e.dno");
  List r2 = q2.list();

% Deleting persistent emp objects %
  Query q3 = ses.createQuery("DELETE FROM emp e WHERE e.age > 50");
  int r3 = q3.executeUpdate();

  tr.commit();
  ses.close();}}
```

(b) Service class

Fig. 1: A HQL Program P

2 Semantics of Query Languages

Halder and Cortesi [9] formalized the semantics of query languages. The basic functionality of SQL statements can be stated as "Any SQL statement Q first identifies an active data set from the database using a pre-condition ϕ that follows first-order logic, and then performs the appropriate operations A on the selected data set". Therefore, the abstract syntax of SQL statements is denoted by a tuple $\langle A, \phi \rangle$. For instance, the query "SELECT a_1, a_2 FROM t WHERE $a_3 \leq 30$" is denoted by $\langle A, \phi \rangle$ where A represents the action-part "SELECT a_1, a_2 FROM t" and ϕ represents the conditional-part "$a_3 \leq 30$".

tid	tage	tdno	tsal
1	35	3	1600
2	19	2	900
3	50	3	2550

(a) Orginal Table t_1

tid	tage	tdno	tsal
1	35	3	1600
2	19	2	900
3	50	3	2550
4	32	1	1000

(b) Table t_2: After executing statement 13

tid	tage	tdno	tsal
1	35	3	1600
2	19	2	900
3	51	3	2750
4	32	1	1000

(c) Table t_3: After executing statements 14-16

tid	tage	tdno	tsal
1	35	3	1600
2	19	2	900
3	51	3	2750
4	32	1	1000

(d) Table t_4: After executing statement 17-18 (no change in database)

tdno	MAX(tsal)	AVG(tage)
1	1000	32
3	2750	43

(e) Table t_{sel}: Result of Selection at 17-18

tid	tage	tdno	tsal
1	35	3	1600
2	19	2	900
4	32	1	1000

(f) Table t_5: After executing statements 19-20

Fig. 2: Snapshot of database states after executing various `Session` methods

Table 1 depicts the syntactic sets, the abstract syntax of SQL, the environments and states associated with the SQL programs, and the semantics of SQL statements. Observe that all the syntactic elements in SQL statements (for example, `GROUP BY`, `ORDER BY`, `DISTINCT` clauses, etc) are represented as functions and the semantics are described as a partial functions on the states which specify how expressions are evaluated and instructions are executed. A state in the program is represented by the tuple $(\ell,\ \rho_d)$ where $\ell \in \mathbb{L}$ is a program label and $\rho_d \in \mathfrak{E}_d$ is a database environment. Interested readers may refer to [9] for more details on the semantics of SQL statements.

3 Semantics of Object-Oriented Programming (OOP)

F. Logozzo in [11] formalized the concrete and abstract semantics of object-oriented programming languages as follows. Object-oriented programming languages consist of a set of classes including a main class from where execution starts. Each class contains a set of attributes and a set of methods - called members of the class. Therefore, a program P in OOP is defined as $P = \langle c_{main}, \texttt{L}\rangle$ where `Class` denotes the set of classes, $c_{main} \in \texttt{Class}$ is the main class, $\texttt{L} \subset \texttt{Class}$ are the other classes present in P.

A class $c \in \texttt{Class}$ is defined as a triplet $c = \langle \texttt{init}, \texttt{F}, \texttt{M}\rangle$ where `init` is the constructor, `F` is the set of fields, and `M` is the set of member methods in c.

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 $\texttt{Env} : \texttt{Var} \longrightarrow \texttt{Loc}$
- The set of stores is defined as $\texttt{Store} : \texttt{Loc} \longrightarrow \texttt{Val}$
- A state is denoted by a tuple $\langle e, s\rangle$ where $e \in \texttt{Env}$ and $s \in \texttt{Store}$.

It is assumed that a state contains some special variables $\{pc, V_{in}, V_{out}\} \subseteq \texttt{Var}$, where pc denotes the current program counter, V_{in} denotes the method input variable (if any), and V_{out} denotes the method output variable (if any).

Syntactic Sets

$n : \mathbb{Z}$ (Integer)
$s : \mathbb{S}$ (String)
$k : \mathbb{C}$ (Constants)
$v_d : \mathbb{V}_d$ (Database Variables)
$e : \mathbb{E}$ (Arithmetic Expressions)
$b : \mathbb{B}$ (Boolean Expressions)
$A : \mathbb{A}$ (Action)
$\tau : \mathbb{T}$ (Terms)
$a_f : \mathbb{A}_f$ (Atomic Formulas)
$\phi : \mathbb{W}$ (Pre-condition)
$Q : \mathbb{Q}$ (SQL statements)
$\ell : \mathbb{L}$ (Set of Program Labels)

Abstract Syntax

$k ::= n \mid s$
$e ::= k \mid v_d \mid op_u\ e \mid e_1\ op_b\ e_2$, where $op_u \in \{+, -\}$ and $op_b \in \{+, -, *, /, \ldots...\}$
$b ::= e_1\ op_r\ e_2 \mid \neg b \mid b_1 \ \vee\ b_2 \mid b_1 \ \wedge\ b_2 \mid true \mid false$, where $op_r \in \{=, \geq, \leq, <, \ldots\}$
$\tau ::= k \mid v_d \mid f_n(\tau_1, \tau_2, ..., \tau_n)$, where f_n is an n-ary function.
$a_f ::= R_n(\tau_1, \tau_2, ..., \tau_n) \mid \tau_1 = \tau_2$, where $R_n(\tau_1, \tau_2, ..., \tau_n) \in \{true, false\}$
$\phi ::= a_f \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x_i\ \phi \mid \exists x_i\ \phi$
$g(\vec{e}) ::= \texttt{GROUP BY}(\vec{e}) \mid id$
$r ::= \texttt{DISTINCT} \mid \texttt{ALL}$
$s ::= \texttt{AVG} \mid \texttt{SUM} \mid \texttt{MAX} \mid \texttt{MIN} \mid \texttt{COUNT}$
$h(e) ::= s \circ r(e) \mid \texttt{DISTINCT}(e) \mid id$
$h(*) ::= \texttt{COUNT}(*)$
$\vec{h}(\vec{x}) ::= \langle h_1(x_1), ..., h_n(x_n)\rangle$, where $\vec{h} = \langle h_1, ..., h_n\rangle$ and $\vec{x} = \langle x_1, ..., x_n\rangle$
$f(\vec{e}) ::= \texttt{ORDER BY ASC}(\vec{e}) \mid \texttt{ORDER BY DESC}(\vec{e}) \mid id$
$A ::= \texttt{SELECT}(f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{e})) \mid \texttt{UPDATE}(\vec{v_d},\ \vec{e}) \mid \texttt{INSERT}(\vec{v_d},\ \vec{e}) \mid \texttt{DELETE}(\vec{v_d})$
$Q ::= \langle A, \phi\rangle \mid Q'\ \texttt{UNION}\ Q'' \mid Q'\ \texttt{INTERSECT}\ Q'' \mid Q'\ \texttt{MINUS}\ Q'' \mid Q';\ Q''$

Database Environment	A database is a set of tables $\{t_i \mid i \in I_x\}$ for a given set of indexes I_x. A database environment is defined as a function ρ_d whose domain is I_x, such that for $i \in I_x$, $\rho_d(i) = t_i$.
Table Environment	A table environment ρ_t for a table t is defined as a function such that $\forall a_i \in attr(t)$, $\rho_t(a_i)=\langle\pi_i(l_j) \mid l_j \in t\rangle$ where π is the projection operator, i.e., $\pi_i(l_j)$ is the i^{th} element of the l_j-th row.
State	The set of states is defined as $\Sigma_d = \mathbb{L} \times \mathfrak{E}_d$ where $\mathfrak{E}_d$ is the set of all database environments.
Semantics	Given a state $(\ell, \rho_d) \in \Sigma_d$, the semantics of SQL statement Q on (ℓ, ρ_d) is defined as $\mathbf{S}_{sql}[\![Q]\!](\ell, \rho_d) = \mathbf{S}_{sql}[\![Q]\!](\ell, \rho_t) = (\ell', \rho_{t'})$ where $\mathbf{S}_{sql}[\![.]\!]$ is the semantic function, $target(Q) = t \in d$, and $\ell' \in \mathbb{L}$ is the label of the successor statement in the program.

Table 1: Syntax and semantics of programs embedding SQL statements

3.1 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. The constructor never returns any output.

Definition 1 (Constructor Semantics). *Given a store s. Let $\{a_{in}, a_{pc}\} \subseteq \texttt{Loc}$ be the free locations, $\texttt{Val}_{in} \subseteq \texttt{Val}$ be the semantic domain for input values. Let $v_{in} \in \texttt{Val}_{in}$ and pc_{exit} be the input value and the exit point of the constructor. The semantic of the class constructor* `init`*, $S[\![\texttt{init}]\!] \in (\texttt{Store} \times \texttt{Val} \to \wp(\texttt{Env} \times \texttt{Store}))$, is defined by:*

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

Definition 2 (Method Semantics). *Let $\texttt{Val}_{in} \subseteq \texttt{Val}$ and $\texttt{Val}_{out} \subseteq \texttt{Val}$ be the semantic domains for the input values and the output values respectively. Let $v_{in} \in \texttt{Val}_{in}$ 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, $S[\![m]\!] \in (\texttt{Env} \times \texttt{Store} \times \texttt{Val}_{in} \to \wp(\texttt{Env} \times \texttt{Store} \times \texttt{Val}_{out}))$, is defined as:*

$$S[\![m]\!](e, s, v_{in}) = \Big\{(e', s', v_{out}) \ \mid\ (e' \triangleq e[V_{in} \to a_{in}, pc \to a_{pc}]) \wedge (s' \triangleq s[a_{in} \to v_{in}, a_{pc} \to pc_{exit}]) \wedge v_{out} \in \texttt{Val}_{out}\Big\}$$

Example 1. Consider the example of Figure 3. The class constructor `Sample()` creates a new environment consisting of field a. The semantics of constructor `Sample()`, semantics of the methods `parity()` and `incr()` are defined below:

```
1. class Sample {
2.   int a;
3.   Sample(int i) {
4.     a = i;
5.   }
6.   int parity() {
7.     if(a % 2 == 0)
8.       return 1;
9.     else return 0;
10.  }
11.  int * incr( int j ) {
12.    a = a + j;
13.    return &a;
14.  }
15. }
```

Fig. 3: An example class

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

$$S[\![\texttt{parity()}]\!](e,s,\varnothing) = \Big\{(e,s',v_{out}) \mid (s' \triangleq s[e(pc) \rightarrow 10]) \wedge (v_{out} = \text{if}(s(e(a))\%2)\,?1:0)\Big\}$$

$$S[\![\texttt{incr()}]\!](e,s,j) = \Big\{(e,s',v_{out}) \mid (s' \triangleq s[e(a) \rightarrow s(e(a))+j, e(pc) \rightarrow 14]) \wedge v_{out} = e(a)\Big\}$$

Observe that `parity()` takes no input and returns an integer value as output, whereas `incr()` takes an integer value as input and returns an address as output.

3.2 Object and Class Semantics

The set of interaction states is defined by $\Sigma = \texttt{Env} \times \texttt{Store} \times \texttt{Val}_{out} \times \wp(\texttt{Loc})$ where `Env`, `Store`, $\texttt{Val}_{out}$, and `Loc` are the set of environments, the set of stores, the set of output values, and the set of addresses respectively.

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.

The transition relation $\mathscr{T}$ includes both direct and indirect interactions.

Objects Fix-point Semantics Given a store $s \in \texttt{Store}$, the set of initial interaction states is defined as $\mathcal{I}_0 = \Big\{\langle e_0, s_0, \phi, \emptyset\rangle \mid S[\![\texttt{init}]\!](v_{in}, s) \ni \langle e_0, s_0\rangle, v_{in} \in \texttt{Val}_{in}\Big\}$. The fix-point trace semantics of `obj`, is defined as: $\mathbb{T}[\![\texttt{obj}]\!](\mathcal{I}_0) = \text{lfp}^{\subseteq}_{\emptyset}\mathcal{F}(\mathcal{I}_0) = \bigcup_{i \leq \omega} \mathcal{F}^i(\mathcal{I}_0)$ where

$$\mathcal{F}(\mathcal{I}) = \lambda\mathcal{T}.\,\mathcal{I} \cup \Big\{\sigma_0 \xrightarrow{\ell_0} \ldots \xrightarrow{\ell_{n-1}} \sigma_n \xrightarrow{\ell_n} \sigma_{n+1} \mid \sigma_0 \xrightarrow{\ell_0} \ldots \xrightarrow{\ell_{n-1}} \sigma_n \in \mathcal{T} \wedge (\sigma_{n+1}, \ell_n) \in \mathscr{T}(\sigma_n)\Big\}$$

4 Concrete Semantics of Hibernate Query Language

We are now in position to formalize the concrete and abstract semantics of HQL. We obtain it by (*i*) extending the OOP semantics and (*ii*) defining the semantics of `Session` methods combining with the abstract interpretation of query languages.

4.1 Syntax

Like OOP, a program P in HQL is also defined as $P = \langle c_{main}, \texttt{L}\rangle$ where $c_{main} \in$ `Class` is the main class, $\texttt{L} \subset \texttt{Class}$ are the other classes present in P. Similarly, a class $c \in \texttt{Class}$ is defined as a triplet $c = \langle \texttt{init}, \texttt{F}, \texttt{M}\rangle$ where `init` is the constructor, `F` is the set of fields, and `M` is the set of member methods in c.

An additional and attractive feature of Hibernate is the presence of `Hibernate Session` which provides a central interface between the application and database and acts as a 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) and to synchronize both states when a change is made to persistent objects.

Abstract Syntax of `Session` Methods In abstract syntax, we denote a `Session` method by a triplet $\langle \mathtt{C}, \phi, \mathtt{OP} \rangle$ where `OP` is the operation to be performed on the tuples satisfying ϕ in the database tables corresponding to the set of POJO classes `C`. Four basic `OP` that cover a wide range of operations are `SAVE`, `UPD`, `DEL`, and `SEL`.

– $\left\langle \mathtt{C}, \phi, \mathtt{SAVE(obj)} \right\rangle = \left\langle \{c\}, \mathit{false}, \mathtt{SAVE(obj)} \right\rangle$: 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 ϕ is false as the method does not identify any existing tuples in the database.
– $\left\langle \mathtt{C}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp}) \right\rangle = \left\langle \{c\}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp}) \right\rangle$: Updates the attributes corresponding to the class fields $\vec{v}$ by $\vec{exp}$ in the database table t for the tuples satisfying ϕ, where t corresponds to the POJO class c.
– $\left\langle \mathtt{C}, \phi, \mathtt{DEL}() \right\rangle = \left\langle \{c\}, \phi, \mathtt{DEL}() \right\rangle$: Deletes the tuples satisfying ϕ in t, where t is the database table corresponding to the POJO class c.
– $\left\langle \mathtt{C}, \phi', \mathtt{SEL}\big(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{exp})\big) \right\rangle$: Selects information from the database tables corresponding to the set of POJO classes `C`, and returns the equivalent representations in the form of objects. This is done only for the tuples satifying ϕ'. The descriptions of f, r, h, g, ϕ, etc. are already mentioned in Table 1.

Observe that as `SAVE`(), `UPD`() and `DEL`() always target single class, the set `C` is a singleton $\{c\}$. However, `C` may not be singleton in case of `SEL`(). The syntax is defined in Figure 4.

4.2 Semantics

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 their concrete semantics by specifying how the methods are executed on (e, s, ρ_d) where $e \in \mathtt{Env}$ is an environment, $s \in \mathtt{Store}$ is a store, and $\rho_d \in \mathfrak{E}_d$ is a database environment, resulting in new state $(e', s', \rho_{d'})$. The semantic definitions are expressed in terms of the semantics of database statements `SELECT, INSERT, UPDATE, DELETE` [9].

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*;

Set of Classes
$c \in$ `Class`
$c ::= \langle$`init`$,$`F`$,$`M`$\rangle$
where `init` is the constructor, `F` $\subseteq$ `Var` is the set of fields, and `M` is the set of methods.
Session methods
$m_{ses} \in \mathtt{M}_{ses}$
$m_{ses} ::= \langle \mathtt{C}, \phi, \mathtt{OP}\rangle$ where `C` $\subseteq$ `Class` and ϕ represents '`WHERE' clause.`
`OP` $::=$ `SAVE(obj)`
$\mid$ $\mathtt{UPD}(\vec{v}, \vec{exp})$
$\mid$ `DEL()`
$\mid$ $\mathtt{SEL}\Big(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{exp})\Big)$
where ϕ represents '`HAVING`' clause and `obj` denotes a class-instance.
HQL Programs
$p \in \mathbb{P}$
$p ::= \langle c_{main}, \mathtt{L}\rangle$ where $c_{main} \in$ `Class` is the main class and `L` $\subset$ `Class`.

Fig. 4: Abstract Syntax of `Session` methods and HQL programs

attr(t) returns the attributes associated with table t; *dom(f)* returns the domain of f.

The semantic function $\mathbf{S}_{hql}$, $\mathbf{S}_{hql} \in ((\mathtt{Env}\times\mathtt{Store}\times\mathfrak{E}_d) \rightarrow \wp(\mathtt{Env}\times\mathtt{Store}\times\mathfrak{E}_d))$, for a given session method $m_{ses} = \langle \mathtt{C}, \phi, \mathtt{OP}\rangle$ is defined as:

$$\mathbf{S}_{hql}[\![m_{ses}]\!](e,s,\rho_d) = \begin{cases} \mathbf{S}_{hql}[\![m_{ses}]\!](e,s,\rho_{t'}) \text{ if } \exists t_1,\dots,t_n \in dom(\rho_d):\ \mathtt{C} = \{c_1,\dots,c_n\} \\ \qquad\qquad \wedge(\forall i \in 1\dots n.\ t_i = map(c_i)) \wedge t' = t_1 \times t_2 \times\cdots\times t_n. \\ \\ \bot \quad otherwise. \end{cases}$$

Semantics of Session Method $\langle\{c\}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp})\rangle$. The semantics of $\langle\{c\}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp})\rangle$ is defined as [4]:

$$\mathbf{S}_{hql}[\![\langle\{c\},\phi,\mathtt{UPD}(\vec{v},\vec{exp})\rangle]\!] = \lambda(e,s,\rho_t).\text{let } c = \langle \mathtt{init},\mathtt{F},\mathtt{M}\rangle \text{ such that } map(\mathtt{F}) = attr(t) \text{ and } map(\vec{v}) = \vec{a} \subseteq attr(t) \text{ where } \vec{v} \subseteq \mathtt{F}, \text{ and let } \phi_d = \mathtt{PE}[\![\phi]\!](e,s,\mathtt{F}) \text{ and } \vec{exp}_d = \mathtt{PE}[\![\vec{exp}]\!](e,s,\mathtt{F}) \text{ in } \Big\{\langle e,s,\rho_{t'}\rangle \mid \rho_{t'} \in \mathbf{S}_{sql}[\![\Big\langle \mathtt{UPDATE}(\vec{a},\vec{exp}_d),\phi_d\Big\rangle]\!](\rho_t)\Big\}.$$

The auxiliary function $\mathtt{PE}[\![X]\!]$ (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 $\mathtt{PE}[\![X]\!](e,s,\mathtt{F}) = X'$, where

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

[4] 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.

Example 2. Consider the HQL example in Figure 1. The abstract syntax of the Session method corresponding to the statements 14-16 is $\left\langle \{c\}, \phi, \texttt{UPD}(\vec{v}, \vec{exp})\right\rangle$, where

- $\{\mathrm{c}\}= \{\texttt{emp}\}$,
- ϕ= "$\texttt{emp}.sal > 1600$",
- $\texttt{UPD}(\vec{v}, \vec{exp})$=$\texttt{UPD}\big(\langle age, sal\rangle,\ \langle age+1, sal+:inc\times 2\rangle\big)$

Given the table environment ρ_{t_2} in Figure 2(b), the semantics is:

$$\mathbf{S}_{hql}[\![\big\langle\{\texttt{emp}\},\ (\texttt{emp}.sal > 1600),\ \texttt{UPD}\big(\langle age, sal\rangle,\ \langle age+1, sal+:inc\times 2\rangle\big)\big\rangle]\!] =$$
$$\lambda(e,s,\rho_{t_2}).\ \text{let } \texttt{emp} = \langle \texttt{emp()}, \texttt{F}, \texttt{M}\rangle \text{ such that } \texttt{F} = \langle id, age, dno, sal\rangle \text{ and}$$
$$map(\texttt{F}) = attr(t) = \langle tid, tage, tdno, tsal\rangle \text{ and } map(\vec{v}) = map(\langle age, sal\rangle) = \langle tage, tsal\rangle \subseteq attr(t),$$
$$\text{and let } \phi_d = (tsal > 1600) = \texttt{PE}[\![(\texttt{emp}.sal > 1600)]\!](e,s,\texttt{F}) \text{ and}$$
$$\vec{exp}_d = \langle tage+1, tsal+100\times 2\rangle = \texttt{PE}[\![\langle age+1, sal+:inc\times 2\rangle]\!](e,s,\texttt{F}) \text{ in}$$
$$\Big\{\langle e,s,\rho_{t_3}\rangle | \rho_{t_3} \in \mathbf{S}_{sql}[\![\big\langle \texttt{UPDATE}(\langle tage, tsal\rangle, \vec{exp}_d),\ \phi_d\big\rangle]\!](\rho_{t_2})\Big\}.$$

Semantics of $\langle \texttt{C}, \phi, \texttt{SEL}(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp}))\rangle$. The semantics of Session method $\langle \texttt{C}, \phi, \texttt{SEL}\big(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp})\big)\rangle$ is defined as:

$$\mathbf{S}_{hql}[\![\langle \texttt{C},\ \phi,\ \texttt{SEL}\big(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp})\big)\rangle]\!] = \lambda(e,s,\rho_t).\ \text{let } \texttt{C} = \{\langle \texttt{init}_i, \texttt{F}_i, \texttt{M}_i\rangle \mid i = 1,\dots,n\},$$
$$\text{and } \texttt{F} = \bigcup_{i=1,\dots,n} \texttt{F}_i,\ \text{and } \langle \vec{exp}'_d,\ \vec{x}_d,\ \phi'_d,\ \vec{exp}_d,\ \phi_d\rangle = \texttt{PE}[\![\langle \vec{exp}',\ \vec{x},\ \phi',\ \vec{exp},\ \phi\rangle]\!](e,s,\texttt{F}),$$
$$\text{and let } \rho_{t'} = \mathbf{S}_{sql}[\![\langle \texttt{SELECT}(f(\vec{exp}'_d),\ r(\vec{h}(\vec{x}_d)),\ \phi'_d,\ g(\vec{exp}_d)), \phi_d\rangle]\!](\rho_t)$$
$$\text{and } (e',s') = \bigsqcup_{\forall l_i \in t'} \mathbf{S}_{hql}[\![\texttt{Object()}]\!](s, \texttt{val}(l_i)) \text{ in } \big\{\langle e', s', \rho_t\rangle\big\}.$$

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

We skip the semantic definition of Session Methods $\langle\{c\}, false, \texttt{SAVE}(\texttt{obj})\rangle$ and $\langle\{c\}, \phi, \texttt{DEL}()\rangle$ for the sake of space.

Fix-point Semantics of Session Objects. Let Env and Store be the set of HQL environments and stores respectively. Let $\mathfrak{E}_d$ be the set of database environments. The set of interaction states of Session objects is defined below:

Definition 3 (Interaction States of Session Objects). *The set of interaction states of Session objects is defined by $\Sigma = \texttt{Env} \times \texttt{Store} \times \mathfrak{E}_d$. Therefore, an interaction state of a Session object is a triplet $\langle e, s, \rho_d\rangle$ where $e \in \texttt{Env}$, $s \in \texttt{Store}$ and $\rho_d \in \mathfrak{E}_d$.*

Because of nondeterministic executions, the transition relation is defined as $\mathscr{T}$: $\texttt{M}_{ses} \times \Sigma \rightarrow \wp(\Sigma)$ specifying which successor interaction states $\sigma' = \langle e', s', \rho_{d'}\rangle \in \Sigma$ can follow when a Session method $m_{ses} = \langle \texttt{C}, \phi, \texttt{op}\rangle \in \texttt{M}_{ses}$ is invoked on an interaction state $\sigma = \langle e, s, \rho_d\rangle$. That is,

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

We denote a transition by $\sigma \xrightarrow{m_{ses}} \sigma'$ when application of a `Session` method m_{ses} on interaction state σ results in a new state σ'.

Let $\mathcal{I}_0$ be the set of initial interaction states. The semantics of `Session` object $\mathtt{obj}_{ses}$ is defined as $\mathbb{T}[\![\mathtt{obj}_{ses}]\!](\mathcal{I}_0) = \mathrm{lfp}^{\subseteq}_{\emptyset}\mathcal{F}(\mathcal{I}_0) = \bigcup_{i \leq \omega} \mathcal{F}^i(\mathcal{I}_0)$, where

$$\mathcal{F}(\mathcal{I}) = \lambda\mathcal{T}.\mathcal{I} \cup \left\{\sigma_0 \xrightarrow{m_0} \ldots \xrightarrow{m_{n-1}} \sigma_n \xrightarrow{m_n} \sigma_{n+1} \mid \sigma_0 \xrightarrow{m_0} \ldots \xrightarrow{m_{n-1}} \sigma_n \in \mathcal{T} \land \sigma_n \xrightarrow{m_n} \sigma_{n+1} \in \mathscr{T}\right\}$$

Method Projected Collecting Semantics. Given a `Session` object trace $\tau = \sigma_0 \xrightarrow{m_1} \sigma_1 \xrightarrow{m_2} \ldots \xrightarrow{m_n} \sigma_n$, where labels m_i $(i = 1, \ldots, n)$ denotes `Session` method $\langle \mathtt{C}_i, \phi_i, \mathtt{op}_i \rangle$. Let $\mathtt{lab}(\tau[i])$ and $\mathtt{State}(\tau[i])$ denote the i^{th} label m_i and the i^{th} state σ_i respectively in a given trace τ. We define the following function which collects all states obtained after performing a specific `Session` method m_i with an operation `op`:

$$g[\![\tau]\!](\mathtt{op}) = \left\{\sigma_i \mid \exists i.\ \mathtt{lab}(\tau[i]) = m_i \text{ with operation } \mathtt{op} \text{ and } \mathtt{State}(\tau[i]) = \sigma_i\right\}$$

Given a set of traces of `Session` objects $\mathcal{T}$. The method projection function over $\mathcal{T}$ is defined as:

$$\mathtt{Projection}[\![\mathcal{T}]\!](\mathtt{op}) = \bigcup_{\tau \in \mathcal{T}} g[\![\tau]\!](\mathtt{op})$$

5 Verifying HQL programs by lifting Semantics from Concrete to Abstract Domains

As it is usual, in the Abstract Interpretation framework, once the concrete semantics is formulated, it can be lifted to an abstract semantics by simply making correspondence of concrete objects (variables values, object instances, stores, states, traces, etc.) into abstract ones representing partial information on them.

Given the set of concrete interaction states Σ. Let $\mathtt{D}^\sharp$ be an abstract domain representing properties of objects fields and database attributes. The concrete powerset domain $\wp(\Sigma)$ can be over-approximated by the abstract domain $\mathtt{D}^\sharp$ following a Galois connection $\langle \wp(\Sigma), \alpha, \gamma, \mathtt{D}^\sharp \rangle$, where α and γ represent abstraction and concretization function respectively. We denote the abstract version [5] `session` methods as $\mathtt{m}^\sharp_{ses} ::= \langle \mathtt{C}^\sharp, \phi^\sharp, \mathtt{OP}^\sharp \rangle$, where

$$\mathtt{OP}^\sharp ::= \mathtt{SEL}^\sharp\Big(f^\sharp(\vec{exp'}^\sharp),\ r^\sharp(\vec{h}^\sharp(\vec{x}^\sharp)),\ \phi^\sharp,\ g^\sharp(\vec{exp}^\sharp)\Big) \mid \mathtt{UPD}^\sharp(\vec{v}^\sharp, \vec{exp}^\sharp) \mid \mathtt{SAVE}^\sharp(\mathtt{obj}^\sharp) \mid \mathtt{DEL}^\sharp()$$

The abstract semantics of $\mathtt{m}^\sharp_{ses}$ is defined in terms of the abstract semantic of $\mathtt{INSERT}^\sharp$, $\mathtt{UPDATE}^\sharp$, $\mathtt{DELETE}^\sharp$, $\mathtt{SELECT}^\sharp$ [9].

Given two abstract states $\sigma^\sharp_1, \sigma^\sharp_2 \in \mathtt{D}^\sharp$, the transition relation in the abstract domain is denoted by $\sigma^\sharp_1 \xrightarrow{\mathtt{m}^\sharp_{ses}} \sigma^\sharp_2$, where the application of $\mathtt{m}^\sharp_{ses}$ on $\sigma^\sharp_1$ results in $\sigma^\sharp_2$. The computation of sound abstract fixed-point trace semantics of session objects in the abstract domain $\mathtt{D}^\sharp$ is straightforward.

[5] The apex $\sharp$ represents an abstract version of the elements in the abstract domain.

A sound abstract projection function "$\texttt{projection}^\sharp$" on a given set of abstract traces $\mathcal{T}^\sharp$ of session object, similarly, collects all the abstract states obtained after performing session methods $\mathtt{m}^\sharp_{ses}$.

In the following example, we show how this can be applied when considering simple abstract domains like intervals, reduced cardinal product, etc. [5] to over-approximate numerical values.

Example 3. Recall the HQL code (Figure 1) and the policies from Section 1.

Policy 1: *Employees age should be greater than or equal to 18 and less than or equal to 62.*
Policy 2: *The salary of employees with age greater than 30 should be at least 1500 euro.*
Policy 3: *Employees salary should not be more than three times of the lowest salary.*

Verifying Policy 1. Let us consider the domain of intervals INT representing properties of numerical values. Since we are interested only on numerical attribute 'age' in the policy, considering objects state and database state, we choose the abstract domain $\mathtt{D}^\sharp = \mathtt{INT} \times \mathtt{INT}$. Intuitively, an element in $\mathtt{D}^\sharp$ is a tuple $\langle [l_1, h_1], [l_2, h_2] \rangle$ where the first component upper-approximates the values taken by the field 'age' in emp class and the second component upper-approximates the values taken by the database attribute 'tage'.

The abstract initial interaction state in the example program is $\sigma^\sharp = \langle \bot, [19, 50] \rangle$ where $\bot$ represents the bottom element in the abstract domain INT. The set of abstract traces of Session object 'ses' in the program is $\mathcal{T}^\sharp = \{ \tau^\sharp \} = \left\{ \sigma^\sharp_0 \xrightarrow{ses.\mathtt{SAVE}^\sharp()} \sigma^\sharp_1 \xrightarrow{ses.\mathtt{UPD}^\sharp()} \sigma^\sharp_2 \xrightarrow{ses.\mathtt{SEL}^\sharp()} \sigma^\sharp_3 \xrightarrow{ses.\mathtt{DEL}^\sharp()} \sigma^\sharp_4 \right\}$ where $\sigma^\sharp_1 = \langle [32, 32], [19, 50] \rangle$ and $\sigma^\sharp_2 = \sigma^\sharp_3 = \langle [32, 32], [19, 51] \rangle$ and $\sigma^\sharp_4 = \langle [32, 32], [19, 50] \rangle$.

According to the abstract projected collecting semantics, we get

$$\begin{aligned}
\mathtt{Projection}^\sharp[\![\mathcal{T}^\sharp]\!](\mathtt{SAVE}^\sharp()) &= \{\sigma^\sharp_1\} = \{\langle [32, 32], [19, 50] \rangle\} \\
\mathtt{Projection}^\sharp[\![\mathcal{T}^\sharp]\!](\mathtt{UPD}^\sharp()) &= \{\sigma^\sharp_2\} = \{\langle [32, 32], [19, 51] \rangle\} \\
\mathtt{Projection}^\sharp[\![\mathcal{T}^\sharp]\!](\mathtt{SEL}^\sharp()) &= \{\sigma^\sharp_3\} = \{\langle [32, 32], [19, 51] \rangle\} \\
\mathtt{Projection}^\sharp[\![\mathcal{T}^\sharp]\!](\mathtt{DEL}^\sharp()) &= \{\sigma^\sharp_4\} = \{\langle [32, 32], [19, 50] \rangle\}
\end{aligned}$$

It is evident that the collecting projected semantics satisfy Policy 1.

Verifying Policy 2. Consider the relational abstract domain of reduced cardinal power $\mathtt{INT}^{\mathtt{INT}}$ where the base INT represents abstract *salary* values in the domain of intervals and the exponent INT represents the abstract *age* values in the domain of intervals [5]. We choose the abstract domain $\mathtt{D}^\sharp = \mathtt{INT}^{\mathtt{INT}} \times \mathtt{INT}^{\mathtt{INT}}$ where the first component in an element of $\mathtt{D}^\sharp$ upper-approximates the values taken by fields 'sal' and 'age' in emp class, whereas the second component upper-approximates the values taken by the database attributes 'tsal' and 'tage'. Following the similar method as in the case of Policy 1, it is immediate to say that the abstract projected collecting states on $\mathtt{SAVE}^\sharp()$ may not satisfy the Policy 2 because the base of the second component in $\sigma^\sharp_1$ has lower limit of salary below 1500 euro with valid age interval in the exponent. Hence, a possible policy violation is detected.

Verifying Policy 3. In this policy, since we are interested only on 'salary', we choose the abstract domain $\mathtt{D}^\sharp = \mathtt{INT} \times \mathtt{INT}$. According to the policy, an abstract state $\langle [h_1, k_1], [h_2, k_2] \rangle$ respects the policy if $k_2 < 3 * h_2$. The analysis says that the projected abstract collecting state $\sigma_2^\sharp$ on $\mathtt{UPD}^\sharp()$ may not satisfy Policy 3. Therefore in this case also a possible policy violation is detected by the analysis.

6 Conclusions

The contribution in this paper is not only the extension of Abstract Interpretation of object-oriented languages to the case of HQL, but also an interesting example of combination of concrete/abstract semantics of different languages for verification purposes. This generic framework can have many applications, *e.g*. formal verification of security issues like database access control, specification-based slicing of HQL programs, language-based information-flow analysis, *etc*.

References

1. Bauer, C., King, G.: Hibernate in Action. Manning Publications Co. (2004)
2. Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications Co. (2006)
3. Bouaziz, M., Logozzo, F., Fähndrich, M.: Inference of necessary field conditions with abstract interpretation. In: Proc. of the 10th Asian Symposium on Programming Languages and Systems (APLAS 2012). pp. 173–189 (2012)
4. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Proc. of the 6th Asian Symposium on Programming Languages and Systems. pp. 3–18. ACM Press, Bangalore, India (2008)
5. Cortesi, A., Costantini, G., Ferrara, P.: A survey on product operators in abstract interpretation. EPTCS 129, 325–336 (2013)
6. Cortesi, A., Halder, R.: Abstract interpretation of recursive queries. In: Proc. of the 9th Int. Conf. on Distributed Computing and Internet Technologies. pp. 157–170. Springer LNCS 7753, India (5–8 Feb 2013 2013)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the POPL'77. pp. 238–252. ACM Press, Los Angeles, CA, USA (1977)
8. Elliott, J., O'Brien, T., Fowler, R.: Harnessing Hibernate. O'Reilly, first edn. (2008)
9. Halder, R., Cortesi, A.: Abstract interpretation of database query languages. Computer Languages, Systems & Structures 38, 123–157 (2012)
10. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of jml: A behavioral interface specification language for java. SIGSOFT Softw. Eng. Notes 31(3), 1–38 (2006)
11. Logozzo, F.: Class invariants as abstract interpretation of trace semantics. Computer Languages, Systems & Structures 35, 100–142 (2009)
12. Logozzo, F.: Practical verification for the working programmer with codecontracts and abstract interpretation. In: Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11). pp. 19–22. Springer-Verlag, Austin, TX, USA (2011)
13. O'Neil, E.J.: Object/relational mapping 2008: Hibernate and the entity data model (edm). In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08). pp. 1351–1356. ACM, New York, NY, USA (2008)
14. Wiśniewski, P., Stencel, K.: Universal query language for unified state model. Fundamenta Informaticae 129(1-2), 177–192 (2014)