Policy-Based Slicing of Hibernate Query Language

Angshuman Jana[1(✉)], Raju Halder[1], Nabendu Chaki[2], and Agostino Cortesi[3]

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

Abstract. This paper introduces a policy-based slicing of Hibernate Query Language (HQL) based on a refined notion of dependence graph. The policies are defined on persistent objects, rather than transient objects, which are stored in an underlying database. We extend the Class Dependence Graph (ClDG) of object-oriented languages to the case of HQL, and we refine it by applying semantics-based Abstract Interpretation framework. This leads to a slicing refinement of HQL programs, producing more precise slices *w.r.t.* policies and we refine by using semantics equivalence, according to the Abstract Interpretation framework.

Keywords: Program slicing · Hibernate Query Language · Dependence graphs · Abstract interpretation

1 Introduction

Program slicing is a widely used static analysis technique which extracts from programs a subset of statements which is relevant to a given behavior [24]. Some of its worth mentioning applications are debugging, testing, code-understanding, code-optimization, etc. [21,23]. Many program slicing algorithms are proposed during last four decades, referring to various language paradigms, such as imperative, object-oriented, functional, logical, etc. [9,22–24]. Recently, [10,25] extended the slicing approaches to the case of applications interacting with external database states. Various forms of program slicing, like static, dynamic, quasi-static, conditioned, etc. are proposed by tuning it towards specific program analysis aim [23].

Hibernate query language (HQL) is an Object-Relational Mapping (ORM) language which remedies the paradigm mismatch between object-oriented languages and relational database models [2]. Various `Session` methods are used to convert transient objects into persistent ones by propagating their states from memory to the database (or vice versa) and to synchronize both states when

© IFIP International Federation for Information Processing 2015
K. Saeed and W. Homenda (Eds.): CISIM 2015, LNCS 9339, pp. 267–281, 2015.
DOI: 10.1007/978-3-319-24369-6_22

a change is made to persistent objects. HQL has gained popularity to software developers in recent years as it provides a unified platform to develop database applications without knowing much details about the database.

Reduction of software failure rate which may happen due to the frequent change of enterprise policies or customers requirements, always remains a serious challenge in software industry [8]. Let us consider an enterprise scenario where new policies are often defined or existing policies are often modified in order to ensure that the real business-data meet business challenges and goals. This forces software developers to maintain and incorporate appropriate changes to the associated program code even after their deployment in real world. Since new or modified policies may also lead to an inconsistent database state, the identification of inconsistent database parts *w.r.t.* the policies is also an important issue. This motivates us to propose a slicing framework for HQL programs, aiming at extracting a subset of program statements affecting enterprise-policies defined on underlying databases. Unfortunately none of the existing approaches are directly applicable to the case of HQL where high-level variables interact with database attributes through hibernate (`Session`) interface – particularly when we slice HQL programs *w.r.t.* the behavior of persistent objects rather than just transient objects.

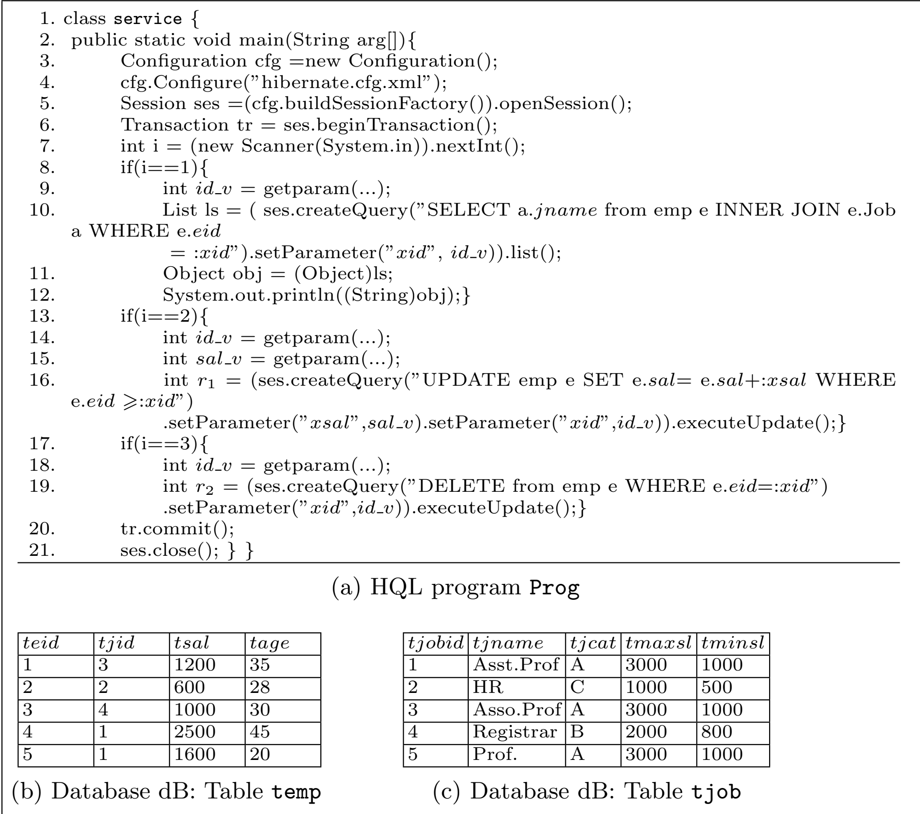
The main contributions in this paper are:

1. We construct syntax-based dependence graph of HQL, considering `(a)` intra-class intra-method dependences, `(b)` intra-class inter-method dependences, `(c)` inter-class inter-method dependences, and `(d)` session-database dependences.
2. We apply semantics-based analysis to refine syntax-based dependence graphs of HQL. To this aim, we define an abstract semantics of Hibernate `Session` methods in a relational abstract domain (the domain of polyhedra).
3. Finally, we propose a policy-based slicing technique on semantically refined dependence graph.

The structure of the rest of paper is as follows: Section 2 describes a running example. In section 3, we describe the syntax-based dependence graphs of HQL. A semantics-based refinement approach of syntax-based dependence graphs is presented in section 4. Section 5 illustrates the running example. Finally, section 6 concludes the work.

2 A Running Example

Consider an enterprise information system where the HQL program `Prog` (Figure 1a) performs three different operations (select, update, delete) on employees information stored in the database dB (Figures 1b and 1c) based on the user choice. Observe that the fields *eid*, *jid*, *sal*, *age* of POJO class `emp` in `Prog` correspond to the attributes *teid*, *tjid*, *tsal*, *tage* of dB table `temp` respectively. Similarly the fields *jobid*, *jname*, *jcat*, *maxsl* of POJO class `Job` in `Prog` correspond to the attributes *tjobid*, *tjname*, *tjcat*, *tmaxsl* of dB table `tjob`



```
1. class service {
2.  public static void main(String arg[]){
3.        Configuration cfg =new Configuration();
4.        cfg.Configure("hibernate.cfg.xml");
5.        Session ses =(cfg.buildSessionFactory()).openSession();
6.        Transaction tr = ses.beginTransaction();
7.        int i = (new Scanner(System.in)).nextInt();
8.        if(i==1){
9.             int id_v = getparam(...);
10.            List ls = ( ses.createQuery("SELECT a.jname from emp e INNER JOIN e.Job
   a WHERE e.eid
                = :xid").setParameter("xid", id_v)).list();
11.            Object obj = (Object)ls;
12.            System.out.println((String)obj);}
13.       if(i==2){
14.            int id_v = getparam(...);
15.            int sal_v = getparam(...);
16.            int r1 = (ses.createQuery("UPDATE emp e SET e.sal= e.sal+:xsal WHERE
   e.eid ⩾:xid")
               .setParameter("xsal",sal_v).setParameter("xid",id_v)).executeUpdate();}
17.       if(i==3){
18.            int id_v = getparam(...);
19.            int r2 = (ses.createQuery("DELETE from emp e WHERE e.eid=:xid")
               .setParameter("xid",id_v)).executeUpdate();}
20.       tr.commit();
21.       ses.close(); } }
```

(a) HQL program Prog

teid	*tjid*	*tsal*	*tage*
1	3	1200	35
2	2	600	28
3	4	1000	30
4	1	2500	45
5	1	1600	20

(b) Database dB: Table temp

tjobid	*tjname*	*tjcat*	*tmaxsl*	*tminsl*
1	Asst.Prof	A	3000	1000
2	HR	C	1000	500
3	Asso.Prof	A	3000	1000
4	Registrar	B	2000	800
5	Prof.	A	3000	1000

(c) Database dB: Table tjob

Fig. 1. Running example

respectively. This mapping is defined in a configuration file of hibernate framework in XML format. Let us suppose that the company decides to introduce a new policy ψ_1 which respects the consistency of employees salary structure, defined below:

> *The salary of employees of age less or equal to 40 cannot have a salary greater than 75% of the maximum salary of the same category.*

The policy is defined on the persistent objects which have permanent representation in the dB. We observe that `Prog` may violate ψ_1 because statement 16 updates employee's salary without checking his/her age.

Our slicing algorithm will allow us to identify the subset of HQL statements responsible for such policy violation.

3 Syntax-Based Dependence Graph of HQL

In this section, we extend the notion of dependence graph representation to the case of HQL. Let us describe its construction using our running example whose

dependence graph is depicted in Figure 2. We consider the following three types of dependences in HQL programs:

Intra-class Intra-method Dependences: These represent the dependences within the same method of a class, and it follows the Program Dependence Graph-based approach [20]. We denote these dependences by dotted edges. Edges 7 - 13, 7 - 17, 7 - 8, 18 - 19, etc. in Figure 2 are of this type.

Intra-class Inter-method Dependences: These represent the dependences between statements of two different methods within same class and are constructed by following System Dependence Graph-based approach [13]. We denote these dependences by long-dash-dotted edges. There is no edge of this type in Figure 2.

Inter-class Inter-method Dependences:

Through Transient Objects. Inter-class Inter-method dependences occur in OOP when a method in one class calls another method in other class. This is done by calling the method through an object of the called-class. Therefore, additional in-parameters corresponding to the object-fields through which the method is called, must be considered [16]. Note that, in this scenario, a constructor-call during object creation is also a part of the graph which follows the same representation as of other inter-class inter-method calls. We denote these dependences by long-dash-dotted edges (as in the case of Intra-class Inter-method Dependences). Edges 3 - 4, 5 - 21, 6 - 20, 5 - 6, 4 - 5, etc. in Figure 2 are of this type. Observe that node 4 calls "`configure()`" method on the object "cfg" which is received from node 3. It configures the "cfg" object using "XML" file and acts as a source for newly-configured "cfg" object. For the sake of simplicity, we do not include here the details of the calling scenario at node 4. Similarly, we hide the details of the calling-scenarios at nodes 5 (which creates the session object by calling `openSession()`), 6 (which creates the transaction object by calling `beginTransaction()`), 20 (when calling `commit()`) and 21 (when calling `close()`) respectively.

Through Session Objects. Various `Session` methods are used to convert objects from transient state to persistent state and to perform various operations, like select, update, delete on the persistent objects in the database. In other words, Hibernate `Session` serves as an intermediate way for the interaction between high-level HQL variables and the database attributes. As creation of a `Session` object implicitly establishes connection with the database, we consider the nodes which create `Session` objects as the sources of the database (hence database-attributes). For instance, in Figure 2 the node 5 acts as a source of `dB`. When `Session` methods (`save(), creatQuery()`) are called through `Session` objects, either a transient object (in case of `save()`) or an object-oriented variant of SQL statement (in case of `createQuery()`) are passed as a parameter. For instance, see nodes 10, 16, 19. The presence of HQL variables in the parameter which have mapping with database attributes leads to a number of dependences

shown by dash-lines between 5 - 10, 5 - 16 and 5 - 19. We call such dependences as session-database dependences. These edges are labeled with used- and defined- HQL variables present in the parameter. For instance, in case of "`obj`" passed to `save()`, all database attributes corresponding to object fields act as defined variables. Similarly, in case of update and delete as parameters in the `createQuery()`, the variables in the WHERE clause act as used variables and the variables in the action part act as defined variables. In case of select, all variables in the parameter act as used variables.

Observe that for the sake of simplified representation, we hide the detail calling scenario of "`createQuery()`" by nodes 10, 16, 19. The edges connecting node 5 and dB indicates the propagation and synchronization of memory and database states. Although we connect node 4 with XML file "`hibernate.cfg.xml`" by an edge, it has no role in slicing and we may omit it also.

4 Semantics-Based Refinement of HQL Dependence Graphs

The syntax-based dependence graphs represent dependences based on the syntactic presence of a variable in the definition of another variables. However, this is not the case always if we focus on the actual values instead of variables. For instance, although the expression "$e = x^2 + 4w \bmod 2 + z$" syntactically depends on w, semantically there is no dependency as the evaluation of "$4w \bmod 2$" is always zero. This motivate researchers towards semantics-based dependence computation [12,17].

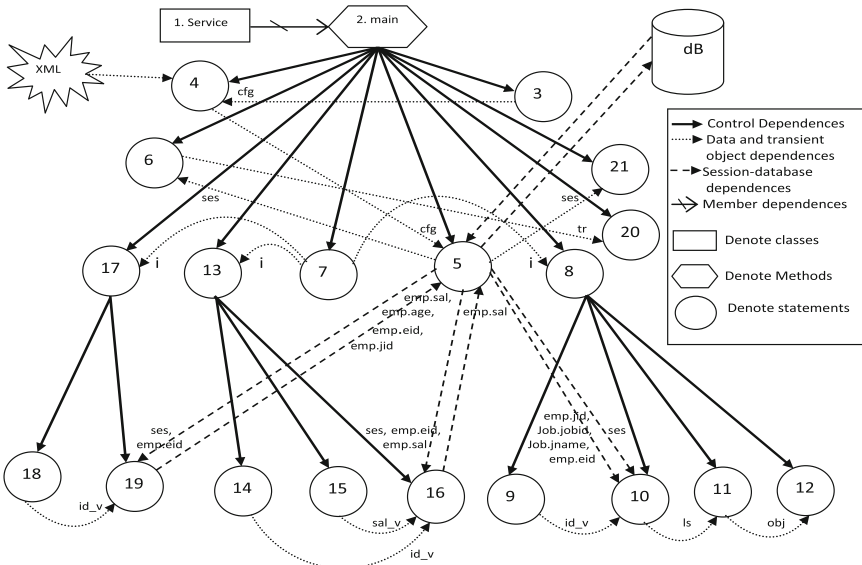


Fig. 2. Dependence Graph of the class "`service`" (Figure 1)

In [12], authors applied three notions: Semantic relevancy, Semantic data-dependences and conditional dependences. This leads to a refinement of syntax-based dependence graphs into more refined semantic-based dependence graphs, producing more precise slices.

The extension of these notions to the case of HQL is not straight-forward because of the existence of transient objects, persistent objects and `Session` methods. Therefore, we have to treat transient objects and persistent objects differently when we apply these semantics-based computations to HQL.

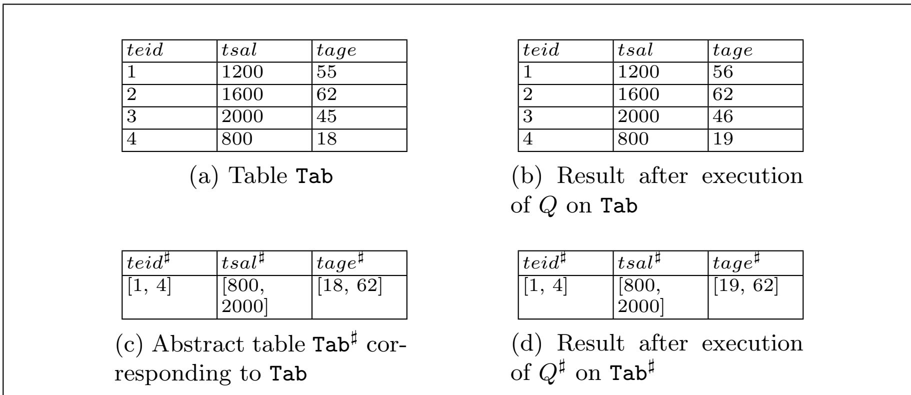
In the rest of this section, we use the following: Let `Var` and `Val` be the set of variables and the domain of values. The set of states is defined as $\Sigma = \texttt{Var} \longmapsto \texttt{Val}_\perp$, where by $\texttt{Val}_\perp$ we denote $\texttt{Val} \cup \{\perp\}$ and $\perp$ is undefined value.

4.1 Semantic Relevancy of HQL Statements *w.r.t* Policies

Consider the following HQL statement on the POJO class `Cls` which corresponds to the database table `Tab` in Figure 3a.

$$\texttt{Q} = \texttt{ses.CreateQuery(UPDATE Cls SET } age = age\texttt{+1 WHERE } age\ \leqslant \texttt{60).executeUpdate()}$$

Consider a company policy ψ_2 (defined on `Tab`) which says that employees ages must belong to the range 18 and 62 (*i.e.* $18 \leqslant tage \leqslant 62$). We denote by σ_D the state of the database D which includes the state of `Tab`. The semantics[1] of `Q` in σ_D, *i.e.* $S[\![\texttt{Q}]\!]\sigma_D$ yields the result shown in Figure 3b. We observe that the policy ψ_2 is satisfied before and after the execution of `Q`, *i.e.* $\psi_2(\sigma_D) = \psi_2(S[\![\texttt{Q}]\!]\sigma_D)$. Therefore, Q is *irrelevant w.r.t.* ψ_2, assuming σ_D is the only state that occurs at the program point of Q.



teid	*tsal*	*tage*
1	1200	55
2	1600	62
3	2000	45
4	800	18

(a) Table `Tab`

teid	*tsal*	*tage*
1	1200	56
2	1600	62
3	2000	46
4	800	19

(b) Result after execution of Q on `Tab`

$teid^\sharp$	$tsal^\sharp$	$tage^\sharp$
[1, 4]	[800, 2000]	[18, 62]

(c) Abstract table $\texttt{Tab}^\sharp$ corresponding to `Tab`

$teid^\sharp$	$tsal^\sharp$	$tage^\sharp$
[1, 4]	[800, 2000]	[19, 62]

(d) Result after execution of $Q^\sharp$ on $\texttt{Tab}^\sharp$

Fig. 3. Concrete and Abstract Query Semantics.

[1] Interested readers may refer [11] for more details on concrete and abstract semantics of query languages.

Although this example is trivial to compute the irrelevancy of Q in concrete domain, in case of very large database (or even when database state depends on run-time inputs) the irrelevancy can be computed in an abstract domain of interest following the Abstract Interpretation framework [6]. The Abstract Interpretation framework is a semantics-based overapproximation technique to infer program's behavior during static time. The aim is to lift concrete semantics to an abstract setting by replacing concrete values by suitable properties of interest, and simulating the concrete operations by sound abstract operations. The mapping between concrete semantics domain ($\mathtt{D}^c$) and abstract semantics domain ($\mathtt{D}^a$) is formalized by Galois Connection $\langle \mathtt{D}^c, \alpha, \gamma, \mathtt{D}^a \rangle$, where $\alpha: \mathtt{D}^c \rightarrow \mathtt{D}^a$ and $\gamma: \mathtt{D}^a \rightarrow \mathtt{D}^c$ represent concretization and abstraction functions respectively.

To illustrate, let us consider the abstract domain of intervals. The abstract table $\mathtt{Tab}^\sharp$ corresponding to Tab in the domain of intervals is shown in Figure 3c. The corresponding abstract state which includes $\mathtt{Tab}^\sharp$ is denoted by $\sigma_D^\sharp$. The abstract semantics $S^\sharp[\![\mathtt{Q}^\sharp]\!]\sigma_D^\sharp$ where $\mathtt{Q}^\sharp$ is

$$\mathtt{ses.CreateQuery\ (UPDATE\ Cls}^\sharp\ \mathtt{SET}\ age^\sharp\mathtt{=}age^\sharp\mathtt{+[1,1]\ WHERE}\ age^\sharp \leqslant^\sharp \mathtt{[60,60]).executeUpdate()}$$

yields the abstract result depicted in Figure 3d. We observe that $\psi_2(\sigma_D^\sharp) = \psi_2(S^\sharp[\![\mathtt{Q}^\sharp]\!]\sigma_D^\sharp)$. By following [11], we can prove the soundness, *i.e.*

$$(\psi_2(\sigma_D^\sharp) = \psi_2(S^\sharp[\![\mathtt{Q}^\sharp]\!]\sigma_D^\sharp)) \implies \forall Q \in \gamma(Q^\sharp), \forall \sigma_D \in \gamma(\sigma_D^\sharp):\ \psi_2(\sigma_D) = \psi_2(S[\![\mathtt{Q}]\!]\sigma_D)$$

where γ is the concretization function [6].

There exist various relational and non-relational abstract domain in the abstract interpretation framework. The efficiency and preciseness of static analysis vary depending on the abstract domain chosen [6]. As our objective is to consider enterprise policies on database, we choose a relational abstract domain – the domain of polyhedra [7] – to compute semantic relevancy of HQL code.

Domain of Polyhedra. Convex polyhedra are regions of some n-dimensional space $\mathbb{R}^n$ that are bounded by a finite set of hyperplanes. Let $\vec{x} = \langle x_1, x_2, \ldots, x_n \rangle$ and $\vec{v} = \langle v_1, v_2, \ldots, v_n \rangle \in \mathbb{R}^n$ be the program variables of real numbers and an n-tuple (vector) of real numbers. By $\beta = \vec{v}.\vec{x} \otimes \mathrm{c}$ where $\vec{v} \neq \vec{0}$, $\mathrm{c} \in \mathbb{R}$, $\otimes \in \{=, \geqslant\}$, we denote a set of linear restraints over $\mathbb{R}^n$. Given a set of linear restraints β on $\mathbb{R}^n$, a set of solutions or points $\{\sigma | \sigma \models \beta\}$ defines the polyhedron $\mathtt{P} = (\beta, \mathrm{n})$. We denote this set of points by $\gamma(\mathrm{p})$ [4,7].

Let $\mathfrak{p}$ be a set of all polyhedra on $\mathbb{R}^n$. We define a complete lattice $\mathtt{L}_A = \langle \mathfrak{p}, \sqsubseteq, \emptyset, \mathbb{R}^n, \sqcup, \sqcap \rangle$ where $\emptyset$ and $\mathbb{R}^n$ are bottom and top elements respectively. Given $\mathtt{P}_1, \mathtt{P}_2 \in \mathfrak{p}$: $\mathtt{P}_1 \sqsubseteq \mathtt{P}_2 \Rightarrow \gamma(\mathtt{P}_1) \subseteq \gamma(\mathtt{P}_2)$, $\mathtt{P}_1 \sqcap \mathtt{P}_2 \Rightarrow \gamma(\mathtt{P}_1) \cap \gamma(\mathtt{P}_2)$ and $\mathtt{P}_1 \sqcup \mathtt{P}_2$ returns convex polyhedron hull which is the least polyhedron covering both $\mathtt{P}_1$ and $\mathtt{P}_2$ [4]. Various operations of convex polyhedra are emptiness checking, projection, *etc* [4].

Given a complete lattice $\mathtt{L}_C = \langle \wp(\mathtt{Val}), \subseteq, \bot, \mathtt{Val}, \cup, \cap \rangle$ of the concrete domain of values Val, the correspondence between concrete and abstract domains is formalized by Galois Connection $\langle\ \mathtt{L}_C, \alpha, \gamma, \mathtt{L}_A\ \rangle$ where α and γ are abstraction and concretization functions respectively [6].

Abstract Semantics. The transition relation is defined as $\mathscr{T}\colon \texttt{Com} \times \mathfrak{p} \rightarrow \wp(\mathfrak{p})$ where Com is the set of commands and $\mathfrak{p}$ is the set of all polyhedra. It defines the abstract semantics of a command in the domain of polyhedra by specifying how application of a command on a polyhedron results into a set of new polyhedra.

Example 1. **(Assignment)** Given P=(β, n)=({x$\geqslant$ 3, y$\geqslant$ 2}, 2). The transition semantics of assignment statement $x := x{+}y$ is defined as: $\mathscr{T}[\![x := x{+}y]\!]\texttt{P} = \{\texttt{P}'\}$ where $\texttt{P}' = (\{x - y \geqslant 3, y \geqslant 2\}, 2)$.

Example 2. **(Test)** Given P= (β, n)= ({x $\geqslant$ 4, y $\geqslant$ 3}, 2). The transition semantics of boolean expression $x \geqslant 10$ is defined as: $\mathscr{T}[\![x \geqslant 10]\!]\texttt{P} = \{\texttt{P}_T,\ \texttt{P}_F\}$ where

$$P_T = (\{\ x \geqslant 10,\ \ y \geqslant 3\ \},\ 2)\ \text{ and }\ \texttt{P}_F = (\{\ x \geqslant 4,\ \ -x \geqslant -9,\ \ y \geqslant 3\ \},\ 2).$$

We are now in position to define abstract transition semantics of Hibernate Session methods. We denote the abstract syntax of Session methods by triplet ⟨C, ϕ, OP⟩ where OP is the operation to be performed on a set of tuples satisfying ϕ in database tables corresponding to a set of classes C. The condition ϕ is a first order logic formula. Four basic OP that cover a wide range of operations are SAVE, UPD, DEL, and SEL [5].

- $\big\langle \texttt{C}, \phi, \texttt{SAVE(obj)}\big\rangle = \big\langle\{c\}, false, \texttt{SAVE(obj)}\big\rangle$: 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.
- $\big\langle \texttt{C}, \phi, \texttt{UPD}(\vec{x}, e\vec{x}p)\big\rangle{=}\big\langle\{c\}, \phi, \texttt{UPD}(\vec{v}, e\vec{x}p)\big\rangle$: Updates the attributes corresponding to the class fields $\vec{x}$ by $e\vec{x}p$ in the database table t for the tuples satisfying ϕ, where t corresponds to the POJO class c.
- $\big\langle \texttt{C}, \phi, \texttt{DEL}()\big\rangle{=}\big\langle\{c\}, \phi, \texttt{DEL}()\big\rangle$: Deletes the tuples satisfying ϕ in t, where t is the database table corresponding to the POJO class c.
- $\big\langle \texttt{C}, \phi, \texttt{SEL}\big(f(e\vec{x}p'),\ \ r(\vec{h}(\vec{x})),\ \ \phi',\ \ g(e\vec{x}p)\big)\big\rangle$: 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 satisfying ϕ. As of [11], we denote by g, h, r, f, ϕ' the GROUP BY, Aggregate Functions, DISTINCT/ALL, ORDER BY, and HAVING clause respectively.

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().

As there is a correspondence between HQL variables and database attributes, we define the transition semantics $\mathscr{T}_{hql}$ of Hibernate Session methods in terms of the transition semantics $\mathscr{T}_{sql}$ in the corresponding database domain as follows:

$$\mathscr{T}_{hql}[\![\langle \texttt{C}, \phi, \texttt{OP}\rangle]\!] = \begin{cases} \mathscr{T}_{sql}[\![\langle \texttt{T}, \phi_d, \texttt{OP}_d\rangle]\!] \\ \text{if } \texttt{C} = \{c_1, \ldots, c_n\} \wedge (\forall i \in 1 \ldots n.\ t_i = \texttt{Map}(c_i)) \wedge \texttt{T} = \{t_1, t_2, \ldots, t_n\}. \\ \\ \bot \quad otherwise. \end{cases}$$

$\texttt{Map}()$ is a function which maps any HQL component into an equivalent database component involving only database tables and attributes. Thus, $\phi_d = \texttt{Map}(\phi)$, $\texttt{OP}_d = \texttt{Map}(\texttt{OP})$ are the condition-part (WHERE clause) and the database operations (UPDATE, DELETE, INSERT, SELECT) defined on the underlying database.

Example 3. Consider Figure 3 and the following Session method:

m_{upd} ::= ses.CreateQuery (UPDATE Cls SET $sal = sal$+100 WHERE $age \geqslant$30).executeUpdate()

The equivalent abstract syntax is $\langle\{\texttt{Cls}\}, age \geqslant 30, \texttt{UPD}(\langle sal\rangle, \langle sal+100\rangle)\rangle$. The function $\texttt{Map}(m_{upd})$ converts m_{upd} into an equivalent database operation whose abstract syntax is $\langle\{\texttt{Tab}\}, tage \geqslant 30, \texttt{UPDATE}(\langle tsal\rangle, \langle tsal + 100\rangle)\rangle$ where

$$\texttt{Map}(\{\texttt{Cls}\}) = \{\texttt{Tab}\}, \qquad \texttt{Map}(age \geqslant 30) = tage \geqslant 30,$$
$$\texttt{Map}(\texttt{UPD}(\langle sal\rangle, \langle sal + 100\rangle)) = \texttt{UPDATE}(\langle tsal\rangle, \langle tsal + 100\rangle)$$

Let us define abstract transition semantics of four different database operations corresponding to the four aforementioned Session methods in the domain of polyhedra.

1. Update:

$$\mathscr{T}_{hql}[\![\langle\{c\}, \phi, \texttt{UPDATE}(\vec{x}, e\vec{x}p)\rangle]\!]\texttt{P} = \mathscr{T}_{sql}[\![\langle\{t\}, \phi_d, \texttt{UPDATE}(\vec{x}_d, e\vec{x}p_d)\rangle]\!]\texttt{P} = \{\texttt{P}'_T\ ,\ \texttt{P}_F\}$$

where
$\texttt{P}_T = (\texttt{P} \sqcap \phi_d)$.
$\texttt{P}'_T = \mathscr{T}_{sql}[\![\texttt{UPDATE}(\vec{x}_d, e\vec{x}p_d)]\!]\texttt{P}_T = \mathscr{T}_{sql}[\![\vec{x}_d := e\vec{x}p_d]\!]\texttt{P}_T$.
$\texttt{P}_F = (\texttt{P} \sqcap \neg\phi_d)$.
By the notation $\vec{x}_d{:=}\ e\vec{x}p_d$ we denote $\langle x_1{:=}\ exp_1,\ x_2{:=}\ exp_2,\ \ldots,\ x_n{:=}\ exp_n\rangle$ where $\vec{x}_d = \langle\ x_1,\ x_2,\ \ldots,\ x_n\rangle$ and $e\vec{x}p_d = \langle\ exp_1,\ exp_2,\ \ldots,\ exp_n\rangle$, which follow transition semantic definition for assignment statement.

Example 4. Consider table Tab in Figure 3a. The abstract representation of Tab in the form of polyhedron is:

$$\texttt{P} = \langle\{teid \geqslant 1, -teid \geqslant -4, tsal \geqslant 800, -tsal \geqslant -2000, tage \geqslant 18, -tage \geqslant -62\}, 3\rangle$$

Consider the following Session method defined on class Cls:

m_{upd} ::= ses.CreateQuery (UPDATE Cls SET $sal = sal$+100 WHERE $age \geqslant$30).executeUpdate().

The equivalent abstract syntax is $\langle\{\texttt{Cls}\},\ age \geqslant 30,\ \texttt{UPD}(\langle sal\rangle,\ \langle sal+100\rangle)\rangle$. The abstract transition semantics of m_{upd} on P is:

$$\begin{aligned}\mathscr{T}_{hql}[\![m_{upd}]\!]\texttt{P} &= \mathscr{T}_{hql}[\![\langle\{\texttt{Cls}\}, age \geqslant 30, \texttt{UPD}(\langle sal\rangle, \langle sal + 100\rangle)\rangle]\!]\texttt{P}\\ &= \mathscr{T}_{sql}[\![\langle\{\texttt{Tab}\}, tage \geqslant 30, \texttt{UPDATE}(\langle tsal\rangle, \langle tsal + 100\rangle)\rangle]\!]\texttt{P} = \{\texttt{P}'_T, \texttt{P}_F\}\ where\end{aligned}$$

$$\texttt{P}'_T = \langle\{teid \geqslant 1, -teid \geqslant -4, tsal \geqslant 900, -tsal \geqslant -2100, tage \geqslant 30, -tage \geqslant -62\}, 3\rangle.$$
$$\texttt{P}_F = \langle\{teid \geqslant 1, -teid \geqslant -4, tsal \geqslant 800, -tsal \geqslant -2000, tage \geqslant 18, -tage \geqslant -29\}, 3\rangle.$$

2. Delete: $\mathscr{T}_{sql}[\![\langle\{t\}, \phi_d, \texttt{DELETE()}\rangle]\!]\texttt{P} = \{(\texttt{P} \sqcap \neg\phi_d)\}$

Example 5. Consider the following `Session` method:

m_{del} ::=`ses.CreateQuery (DELETE from Cls WHERE` $age \geqslant$ `30).executeUpdate().`

The transition semantics of m_{del} on `P` is:

$$\begin{aligned}\mathscr{T}_{hql}[\![m_{del}]\!]\texttt{P} =& \mathscr{T}_{hql}[\![\langle\{\texttt{Cls}\}, age \geqslant 30, \texttt{DEL()}\rangle]\!]\texttt{P}\\ =& \mathscr{T}_{sql}[\![\langle\{\texttt{Tab}\}, tage \geqslant 30, \texttt{DELETE()}\rangle]\!]\texttt{P}\\ =& \{P \sqcap \neg(tage \geqslant 30)\} = \{\texttt{P}'\} \quad where\end{aligned}$$

$\texttt{P}' = \langle\{teid \geqslant 1, -teid \geqslant -4, tsal \geqslant 800, -tsal \geqslant -2000, tage \geqslant 18, -tage \geqslant -29\}, 3\rangle$

3. Insert: Let $\vec{x}$ be the fields of a class c. Given an object `obj` of c, suppose $\vec{v}$ be the values in $\vec{x}$ of the object. The transition relation is defined as

$$\mathscr{T}_{hql}[\![\langle\{c\}, false, \texttt{SAVE}(obj)\rangle]\!]\texttt{P} = \mathscr{T}_{sql}[\![\langle\{t\}, false, \texttt{INSERT}(attribute(t), \vec{v})\rangle]\!]\texttt{P} = \{\texttt{P}'\}$$

where $\texttt{P}' = \texttt{P} \sqcup \{attribute(t) = \vec{v} \mid \vec{v} \in \mathbb{R}^n\}$.

Example 6. Let `obj` be an object of POJO class `Cls` in Figure 3 whose fields $\vec{x} = \langle eid, sal, age\rangle$ are set to values $\vec{v} = \langle 5, 600, 35\rangle$. Consider the following `Session` method m_{ins} ::= `ses.save(obj)`. The transition semantics is:

$$\begin{aligned}\mathscr{T}_{hql}[\![m_{ins}]\!]\texttt{P} =& \mathscr{T}_{hql}[\![\langle\{\texttt{Cls}\}, false, \texttt{SAVE(obj)}\rangle]\!]\texttt{P}\\ =& \mathscr{T}_{sql}[\![\langle\{\texttt{Tab}\}, false, \texttt{INSERT}(\langle teid, tsal, tage\rangle, \langle 5, 600, 35\rangle)\rangle]\!]\texttt{P} = \{\texttt{P}'\}\end{aligned}$$

where $\texttt{P}' = \texttt{P} \sqcup \{teid = 5, tsal = 600, tage = 35\} = \langle\{teid \geqslant 1, -teid \geqslant -5, tsal \geqslant 600, -tsal \geqslant -2000, tage \geqslant 18, -tage \geqslant -62\}, 3\rangle$

4. Select: The select operation does not modify any information in a polyhedron. Therefore, the transition relation is defined as:

$$\mathscr{T}_{sql}[\![\texttt{T}, \phi_d, \texttt{SELECT}\big(f(\vec{exp'}_d),\ r(\vec{h}(\vec{x}_d)),\ \phi'_d,\ g(\vec{exp}_d)]\!]\texttt{P} = \{\texttt{P}\}$$

5 Illustration on the Running Example

Let us illustrate our proposed approach on the running example (Figure 1) in section 2.

Syntax-Based Slicing. The backward slice of `Prog` *w.r.t.* ψ_1 is depicted in Figure 4. This is done by traversing the syntax-based dependence graph (Figure 2) in backward direction *w.r.t.* the slicing criteria $\langle 21, \{sal, age, maxsl\}\rangle$. Observe that the sliced-database on which the slice performs its computation, is a part of the original database. This has crucial role in data-provenance applications.

```
1.  class service {
2.   public static void main(String arg[]){
3.        Configuration cfg =new Configuration();
4.        cfg.Configuration("hibernate.cfg.xml");
5.        Session ses =(cfg.buildSessionFactory()).openSession();
7.        int i = (new Scanner(System.in)).nextInt();
13.       if(i==2){
14.             int id_v = getparam(...);
15.             int sal_v = getparam(...);
16.             int r1 = (ses.createQuery("UPDATE emp e SET e.sal=
                e.sal+:xsal WHERE e.eid=:xid").setParameter("xsal",sal_v)
               .setParameter("xid",id_v)).executeUpdate();}
17.       if(i==3){
18.            int id_v = getparam(...);
19.             int r2 = (ses.createQuery("DELETE from emp e WHERE
                e.eid=:xid").setParameter("xid",id_v)).executeUpdate();}
21.       ses.close(); } }
```

(a) Slice of `Prog` $w.r.t\ \psi_1$

teid	*tsal*
1	1200
2	600
3	1000
4	2500
5	1600

(b) Slice of `temp` $w.r.t\ \psi_1$

Fig. 4. Syntax-based Slice $w.r.t.\ \psi_1$.

Semantic-Based Slicing. The initial polyhedron[2] corresponding to the database `dB` (Figure 1b) is $\mathtt{P}_{dB} = \langle\{teid \geqslant 1, -teid \geqslant -5, tsal \geqslant 600, -tsal \geqslant -2500, tage \geqslant 20, -tage \geqslant -45\}, 3\rangle$. The pictorial representation is shown in Figure 5(a).

The abstract syntax of `Session` methods m_{sel}, m_{upd} and m_{del} at program points 10, 16, 19 respectively in `Prog` are:

$$m_{sel} ::= \langle \mathtt{C}, \phi, \mathtt{OP}\rangle \text{ where } \mathtt{C} = \{\mathtt{emp}, \mathtt{Job}\},\ \phi = \{eid = jobid,\ eid = id_v\},$$
$$\mathtt{OP} = \mathtt{SEL}(f(\vec{exp'}),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp}\rangle\ = \mathtt{SEL}(id, \mathtt{ALL}(id(jname)), true, id)\rangle,$$
and id denotes identity function.

$$m_{upd} ::= \langle\{\mathtt{emp}\}, \{eid = id_v\}, \mathtt{UPD}(\langle sal\rangle, \langle sal + sal_v\rangle)\rangle$$
$$m_{del} ::= \langle\{\mathtt{emp}\}, \{eid = id_v\}, \mathtt{DEL}()\rangle$$

The transition semantics on $\mathtt{P}_{dB}$ are:

$$\begin{aligned}&\mathscr{T}_{hql}[\![m_{sel}]\!]\mathtt{P}_{dB}\\ &=\mathscr{T}_{hql}[\![\langle\{\mathtt{emp}, \mathtt{Job}\}, \{eid = jobid, eid = id_v\}, \mathtt{SEL}(id, \mathtt{ALL}(id(jname)), true, id)\rangle]\!]\mathtt{P}_{dB}\\ &=\mathscr{T}_{sql}[\![\langle\{\mathtt{temp}, \mathtt{tjob}\}, \{teid = tjobid, teid = tid_v\}, \mathtt{SELECT}(id, \mathtt{ALL}(id(tjname)), true, id)\rangle]\!]\mathtt{P}_{dB}\\ &=\{\mathtt{P}_{dB}\}\end{aligned}$$

[2] For the sake of simplicity, we consider the polyhedron in the space involving only three attributes *teid*, *tsal* and *tage*.

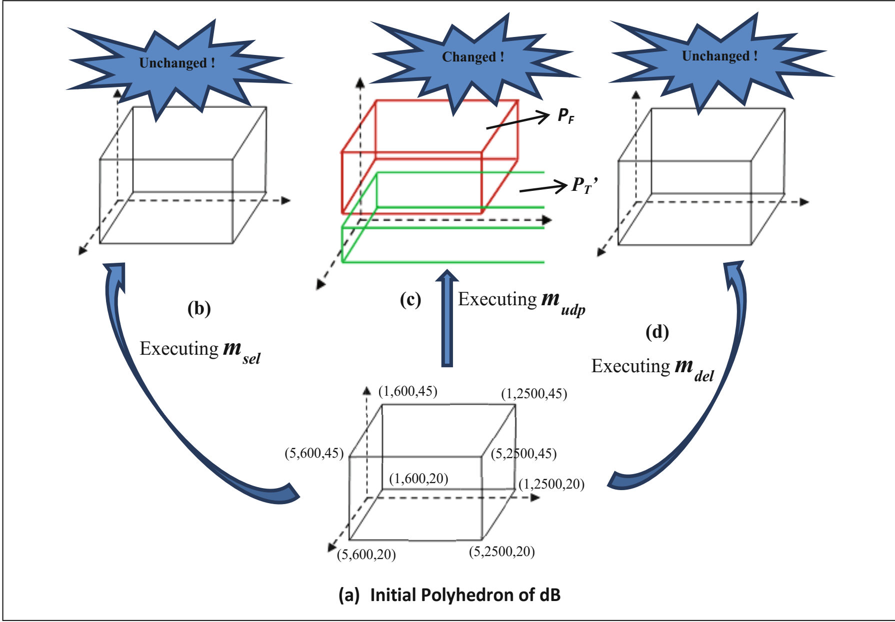


Fig. 5. Polyhedra representation of `temp` on various operation (m_{sel},m_{udp},m_{del}).

Note that, select operation does not change the database, hence the polyhedron remains unchanged (see Figure 5(b)).

$$\begin{aligned}\mathscr{T}_{hql}[\![m_{upd}]\!] =& \mathscr{T}_{hql}[\![\langle\{\mathtt{emp}\}, \{eid \geqslant id_v\}, \mathtt{UPD}(\langle sal\rangle, \langle sal + sal_v\rangle)\rangle]\!]\mathtt{P}_{dB} \\ =& \mathscr{T}_{sql}[\![\langle\{\mathtt{temp}\}, \{teid \geqslant id_v\}, \mathtt{UPDATE}(\langle tsal\rangle, \langle tsal + sal_v\rangle)\rangle]\!]\mathtt{P}_{dB} \\ =& \{\mathtt{P}'_T, \mathtt{P}_F\} \quad where\end{aligned}$$

$$\begin{aligned}\mathtt{P}_T &= \mathtt{P}_{dB} \sqcap \{teid \geqslant id_v\} \\ \mathtt{P}'_T &= \mathscr{T}_{sql}[\![tsal := tsal + sal_v]\!](\mathtt{P}_T) \\ &= \langle\{teid \geqslant 1,\ -teid \geqslant -5,\ tage \geqslant 20,\ -tage \geqslant -45,\ tsal \geqslant 600\},\ 3\rangle. \\ \mathtt{P}_F &= \{\mathtt{P}_{dB} \sqcap \neg(teid \geqslant id_v)\} = \{\mathtt{P}_{dB}\}\end{aligned}$$

Note that, since the updation of $tsal$ depends on run-time input, we project-out the upper bound of the attribute from the set of restraints in $\mathtt{P}'_T$ in order to guarantee the soundness. The polyhedron representation of $\mathtt{P}'_T$ is shown in Figure 5(c). Also note that, P_F is sound-approximated by disregarding the effect of "$teid = id_v$" as it depends on run-time input.

$$\begin{aligned}\mathscr{T}_{hql}[\![m_{del}]\!] =& \mathscr{T}_{hql}[\![\langle\{\mathtt{emp}\}, \{eid = id_v\}, \mathtt{DEL}()]\!]\mathtt{P}_{dB} \\ =& \mathscr{T}_{sql}[\![\langle\{\mathtt{temp}\}, \{teid = id_v\}, \mathtt{DELETE}()]\!]\mathtt{P}_{dB} = \{\mathtt{P}_{dB}\}\end{aligned}$$

Note that, the semantics of m_{del} is sound-approximated by disregarding the effect of "$teid = id_v$" as it depends on run-time input (see Figure 5(d)).

```
 1. class service {
 2.  public static void main(String arg[]){
 3.        Configuration cfg =new Configuration();
 4.        cfg.Configuration("hibernate.cfg.xml");
 5.        Session ses =(cfg.buildSessionFactory()).openSession();
 7.        int i = (new Scanner(System.in)).nextInt();
13.        if(i==2){
14.              int id_v = getparam(...);
15.              int sal_v = getparam(...);
16.              int r1 = (ses.createQuery("UPDATE emp e SET e.sal=e.sal+:xsal WHERE
    e.eid=:xid")
             .setParameter("xsal",sal_v).setParameter("xid",id_v)).executeUpdate();}
21.        ses.close(); } }
```

Fig. 6. Semantics-based slice of `Prog` $w.r.t\ \psi_1$

Observe that the initial polyhedron of dB is covered by the polyhedron representing ψ_1. This can verified by performing the operations, e.g. interaction, emptiness checking, etc. on the polyhedra domain. Therefore, initially ψ_1 is satisfied by dB. The abstract semantics-based analysis proves that m_{del} does not change the initial polyhedron $\mathtt{P}_{dB}$ (see Figure 5). Therefore it is semantically irrelevant *w.r.t.* ψ_1. The semantics-based slice is shown in Figure 6 which disregards the semantically irrelevant statements *w.r.t.* ψ_1, yielding to a slice more accurate than the syntax-based one (Figure 4).

6 Conclusion

In this paper, we introduce a policy-based slicing of Hibernate Query Language where the policies are defined on persistent objects, rather than transient objects. We extend the Class Dependence Graph (ClDG) [16] of object-oriented languages to the case of HQL, and propose a refinement of dependence graph for removing false dependences by analyzing programs in relational or non-relational abstract domains by following the Abstract Interpretation framework.

Our approach is comparable with conditioned slicing and specification-based slicing (several variants exist, *e.g.* precondition−, postcondition−, contract−, assertion−based, etc.) [3]. The method defined for finding a conditioned slice is to use symbolic execution and reject infeasible paths based on the constraints defined by the first order logic equations. Specification-based slicing approach is proposed based on the axiomatic semantics (weakest precondition or strongest postcondition computations) in program verification. All these approaches use SMT solver which has exponential complexity [15]. Our proposal is based on the semantic analysis in an abstract domain following the Abstract Interpretation framework. The framework allows to perform analysis in a relational or non-relational abstract domain based on the properties of interest. Analysis in the domain of polyhedra uses Linear Problem Solver [1]. Although in some cases the worst-case computational complexity in polyhedron domain is exponential [14, 18], but there is an opportunity to chose another less costlier weakly relational domains such as the domain of octagons [19].

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: The ppl: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Tech. rep., Dipartimento di Matematica, Universit'a di Parma, Italy (2006)
2. Bauer, C., King, G.: Hibernate in Action. Manning Publications Co. (2004)
3. Canfora, G., Cimitile, A., Lucia, A.D.: Conditioned program slicing. Information and Software Technology **40**(11–12), 595–607 (1998)
4. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 3–18. Springer, Heidelberg (2008)
5. 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)
6. 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 1977, pp. 238–252 (1977)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the POPL 1978, pp. 84–96 (1978)
8. Glass, R.L.: It failure rates-70% or 10–15%? IEEE Software **22**(3), 112–111 (2005)
9. Gyimóthy, T., Paakki, J.: Static slicing of logic programs. In: Automated and Algorithmic Debugging, pp. 87–103 (1995)
10. Halder, R., Cortesi, A.: Abstract program slicing of database query languages. In: Proc. of the 28th Annual ACM SAC, pp. 838–845 (2013)
11. Halder, R., Cortesi, A.: Abstract interpretation of database query languages. Computer Languages, Systems & Structures **38**, 123–157 (2012)
12. Halder, R., Cortesi, A.: Abstract program slicing on dependence condition graphs. Sci. Comput. Program. **78**(9), 1240–1263 (2013)
13. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on PLS **12**(1), 26–60 (1990)
14. Kelner, J.A., Spielman, D.A.: A randomized polynomial-time simplex algorithm for linear programming. In: Proc. of the Theory of Computing, pp. 51–60 (2006)
15. King, T., Barrett, C., Tinelli, C.: Leveraging linear and mixed integer programming for smt. In: Proc. of the 14th ICFMCAD lausanne, Switzerland (to appear) (2014)
16. Larsen, L., Harrold, M.J.: Slicing object-oriented software. In: Proc. of the 18th ICSE, pp. 495–505 (1996)
17. Mastroeni, I., Zanardini, D.: Data dependencies and program slicing: from syntax to abstract semantics. In: Proc. of the ACM symposium on PEPM, pp. 125–134 (2008)
18. Megiddo, N.: Linear programming in linear time when the dimension is fixed. Journal of the ACM **31**(1), 114–127 (1984)
19. Minè, A.: The octagon abstract domain. Higher Order Symbol. Comput. **19**(1), 31–100 (2006). http://www.astree.ens.fr/
20. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. ACM SIGPLAN Notices **19**(5), 177–184 (1984)
21. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Transactions on Software Engineering **16**(9), 965–979 (1990)

22. Rodrigues, N.F., Barbosa, L.S.: Slicing functional programs by calculation. In: Beyond Program Slicing, November 06–11, 2005 (2005)
23. Tip, F.: A survey of program slicing techniques. Tech. rep. (1994)
24. Weiser, M.: Program slicing. IEEE Trans on SE **SE–10**(4), 352–357 (1984)
25. Willmor, D., Embury, S.M., Shao, J.: Program slicing in the presence of a database state. In: Proc. of the 20th Int. Conf. on Software Maintenance, pp. 448–452 (2004)