

Type Consistency Problems for Queries in Object-Oriented Databases

石原 靖哲 関 浩之 伊藤 実
Yasunori ISHIHARA Hiroyuki SEKI Minoru ITO

奈良先端科学技術大学院大学 情報科学研究科
Graduate School of Information Science
Nara Institute of Science and Technology

This paper discusses the computational complexity of type consistency problems for queries in object-oriented databases (OODBs). A database instance is said to be consistent under a database schema if, for every method invocation m , the definition of m to be bound is uniquely determined. In this paper, we adopt update schemas introduced by Hull et al. as a model of OODB schemas, and show that (1) the problem of determining whether there exists an inconsistent instance under a given *recursion-free* update schema and (2) the problem of determining whether there exists an inconsistent *acyclic* instance under a given *recursion-free* update schema are both NEXPTIME-complete. It is also shown that (3) the problem of determining whether there exists an inconsistent *acyclic* instance under a given *arbitrary* update schema is undecidable.

1 Introduction

Among many features of object-oriented programming languages, method invocation (or message passing) mechanism is an essential one. It is based on method name overloading and late binding by method inheritance along the class hierarchy. For a method name m , different classes may have different definitions (codes, implementations) of m . When m is applied to an object o , one of its definitions is selected depending on the class which o belongs to, and is bound to m in run-time (late binding or dynamic binding).

This paper discusses the computational complexity of type consistency problems for queries in object-oriented databases (OODBs). A database instance is said to be consistent under a database schema if, for every method invocation m , the definition of m to be bound is uniquely determined by using the class hierarchy with inheritance. Then the type consistency problem is to determine whether there exists an inconsistent instance under a given database schema.

Abiteboul et al. [1] introduced *method schemas*, which correspond to a model of OODB schemas without updating database instances. In Ref. [1], it is shown that

1. the type consistency problem for method schemas is undecidable in general,
2. NP-complete if every method is recursion-free, and
3. solvable in polynomial time if a given method schema is monadic (i.e., every method in the schema has at most one argument).

On the other hand, Hull et al. [2] introduced *update schemas*, in which updating database instances is simply modeled as assignment of objects or basic values to attributes of objects. Every method in update schemas is monadic. In Ref. [2], it is shown that the type consistency

Table 1: Complexity of type consistency problems.

	Instance	
	Acyclic	Arbitrary
Recursion-Free update schema	NEXPTIME-complete [†]	NEXPTIME-complete [†]
Arbitrary update schema	Undecidable [†]	Undecidable

†: Results of this paper.

problem for update schemas is undecidable in general. In Ref. [3], a subclass of update schemas, called *non-branching update schemas*, is introduced. And, it is shown that the problem of determining whether there exists an inconsistent *acyclic* instance under a given non-branching update schema is solvable in polynomial time.

Update schemas have all of the basic features of OODBs such as class hierarchy, inheritance, complex objects, and so on. In this paper, we adopt update schemas as a model of OODB schemas, and show that

1. the problem of determining whether there exists an inconsistent instance under a given *recursion-free* update schema is NEXPTIME-complete,
2. the problem of determining whether there exists an inconsistent *acyclic* instance under a given *recursion-free* update schema is also NEXPTIME-complete, and
3. the problem of determining whether there exists an inconsistent *acyclic* instance under a given *arbitrary* update schema is undecidable (see Table 1).

2 Definitions

2.1 Syntax of Database Schemas

A *database schema* is a 4-tuple $\mathbf{S} = (C, \leq, Ad, Impl)$ where:

1. C is a finite set of *class names*.
2. \leq is a partial order on C representing a *class hierarchy*. If $c' \leq c$, then we say that c' is a *subclass* of c and c is a *superclass* of c' . We assume that the class hierarchy is a forest on C , that is, for all $c_1, c_2, c \in C$, the following condition is satisfied:

If $c \leq c_1$ and $c \leq c_2$, then $c_1 \leq c_2$ or $c_2 \leq c_1$.

3. $Ad : C \times Attr \rightarrow C$ is a partial function representing *attribute declarations*, where $Attr$ is a finite set of *attribute names*. By $Ad(c, a) = c'$, we mean that the value of attribute a of an object of c must be an object of c' or its subclass.
4. $Impl : C \times Meth \rightarrow S$ is a partial function representing *method implementations*, where $Meth$ is a finite set of *method names* and S is a set of *well-formed sequence of sentences* defined below.

A sentence is an expression which has one of the following forms:

- | | |
|--------------------|---------------------|
| 1. $y := y'$, | 4. $y := m(y')$, |
| 2. $y := self$, | 5. $self.a := y'$, |
| 3. $y := self.a$, | 6. $return(y')$, |

where y, y' are variables, a is an attribute name, m is a method name, and $self$ is a reserved word that denotes the object on which a method is invoked (or, to which a message is sent). Let $s_1; s_2; \dots; s_n (= \alpha)$ be a sequence of sentences. We say that α is *well-formed* when the following two conditions hold:

- No undefined variable is referred to. That is, for each s_i ($1 \leq i \leq n$), if s_i is one of $y := y'$, $y := m(y')$, $self.a := y'$, and $return(y')$, then there exists a sentence s_j ($j < i$) that must be one of $y' := y''$, $y' := self$, $y' := self.a'$, and $y' := m'(y'')$ (y'' is a variable, a' is an attribute, and m' is a method).
- Only the last sentence s_n must have the form $return(y')$ for some variable y' . Thus the other sentences s_1, s_2, \dots, s_{n-1} must be one of types 1 to 5.

Without loss of generality, we often omit temporary variables for readability. For example, we write " $y := m(self.a)$ " instead of " $y' := self.a; y := m(y')$," where y' is a temporary variable.

The *method dependence graph* [1] $G = (V, E)$ of \mathbf{S} is defined as follows:

- V is the set of all the method names in \mathbf{S} ; and
- An edge from m to m' is in E if and only if there is some c such that m appears in $Impl(c, m')$.

If the method dependence graph of \mathbf{S} is acyclic, then we say that \mathbf{S} is *recursion-free*.

Lastly, we define the *description size* of \mathbf{S} , denoted $|\mathbf{S}|$, as follows:

$$\begin{aligned}
 |\mathbf{S}| = & |C| \\
 & + (\text{the number of attributes}) \\
 & + (\text{the number of attribute declarations} \\
 & \quad \text{given by } Ad) \\
 & + (\text{the number of methods}) \\
 & + (\text{the total number of sentences} \\
 & \quad \text{given by } Impl).
 \end{aligned}$$

2.2 Semantics of Database Schemas

Let $\mathbf{S} = (C, \leq, Ad, Impl)$ be a database schema. The *inherited implementation* of method m at class c , denoted $Impl^*(c, m)$, is defined as $Impl(c', m)$ such that c' is the smallest superclass of c (with respect to the partial order \leq) at which an implementation of m exists, that is, if $Impl(c'', m)$ is defined and $c \leq c''$, then it must hold that $c' \leq c''$. If such an implementation does not exist, then $Impl^*(c, m)$ is undefined. Similarly, the *inherited attribute declaration* of attribute a at class c , denoted $Ad^*(c, a)$, is defined as $Ad(c', a) = c''$ such that c' is the smallest superclass of c at which an attribute declaration of a exists. If such an attribute declaration does not exist, then $Ad^*(c, a)$ is undefined. A *database instance* of \mathbf{S} is a pair $\mathcal{I} = (\nu, \mu)$, where:

1. To each $c \in C$, ν assigns a disjoint, finite set, denoted $\nu(c)$. Each $o \in \nu(c)$ is called an *object* of class c .
2. To each object $o \in \nu(c)$ and each attribute $a \in A$ such that $Ad^*(c, a) = c'$, μ assigns an object, denoted $\mu(o, a)$, that is called the value of attribute a (or simply a -value) of o . If $Ad^*(c, a) = c'$, then $\mu(o, a)$ must belong to $\nu(c')$ for some c'' ($c'' \leq c'$).

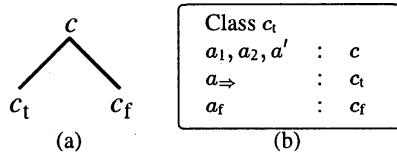
Hereafter, we denote $\mu(o, a)$ by $o.a$. If every object o in \mathcal{I} satisfies

$$o.a_1.a_2 \dots a_n \neq o$$

for any sequence of attributes a_1, a_2, \dots, a_n , then \mathcal{I} is said to be *acyclic*.

The *operational semantics* of a database schema \mathbf{S} under a given database instance \mathcal{I} is formally defined by using a *method execution tree* [2]. Here, we do not repeat the formal definition. Instead, we briefly explain its intuitive meaning. As stated before, $self$ represents the object on which a method is invoked; it is called a *self object*.

1. The meaning of a sentence $y := y'$ is obvious.
2. $y := self$ means that the self object is assigned to variable y .
3. $y := self.a$ means that the a -value of the self object is assigned to y .
4. If the control reaches a sentence $y := m(y')$, then method m is invoked on the object assigned to y' (or, message m is sent to the object assigned to y') and the "returned value" is assigned to y . Assume that an object o of a class c is assigned to y' . If $Impl^*(c, m) = \alpha$, then o is bound to "self" in α , α is executed, and the returned value is assigned to y . If $Impl^*(c, m)$ is undefined, then a run-time type error occurs.

Fig. 1: \leq and Ad used in Sect. 3.

5. Consider a sentence $\text{self}.a := y'$, and let o be the object assigned to y' when the control reaches this sentence. Assume that $Ad^*(c, a) = c' \in C$. If o is an object of a class c'' and $c'' \leq c'$, then the value of attribute a of the self object becomes o . Otherwise, a run-time type error occurs.

2.3 Consistency of Database Schemas

Let S be a database schema, and \mathcal{I} be a database instance of S . We say that \mathcal{I} is *consistent* under S when the following condition holds:

Let m be an arbitrary method of S and $o \in \nu(c)$ be an arbitrary object in \mathcal{I} . If $Impl^*(c, m)$ is defined, then no type errors occur during the execution of m on o .

If \mathcal{I} is not consistent under S , then we say that \mathcal{I} is *inconsistent* under S .

3 Basic Techniques

In this section, we present some basic techniques which are used in the following sections. Throughout this section, C , \leq , and Ad are defined as follows:

- $C = \{c, c_t, c_f\}$;
- \leq is the reflexive closure of $\{(c_t, c), (c_f, c)\}$ (i.e., c is a superclass of both c_t and c_f , see Fig. 1(a)); and
- Ad is shown in Fig. 1(b).

Let o be an object of class c_t . Each attribute $a \in \{a_1, a_2, a', a_f\}$ of o represents a Boolean value: a represents true if $o.a = o$, and false otherwise. Note that $o.a_f$ always represents false because of the declaration $Ad(c_t, a_f) = c_f$.

First, we define a method $\text{nor}[a_1, a_2]$ as shown in Fig. 2, which calculates NOR of $o.a_1$ and $o.a_2$. Since any Boolean operator can be represented by NORs, we can construct a method which calculates any given Boolean formula by using $\text{nor}[a_1, a_2]$. Formally, we have the following lemma:

Lemma 1: Let o be an object of class c_t . Let o' denote the object returned by the execution of method $\text{nor}[a_1, a_2]$ on o . Then, the following equation holds:

$$o' = \begin{cases} o & (\text{if } o.a_1 \neq o \text{ and } o.a_2 \neq o), \\ o.a_f & (\text{otherwise}). \end{cases}$$

Proof: Consider how $o.a'$ changes during the execution of $\text{nor}[a_1, a_2]$. First, $o.a'$ is set to o . By the second line of $(c, \text{nor}[a_1, a_2])$, $o.a'$ is set to $o.a_f$ if $o.a_1 = o$, and unchanged otherwise. Similarly, by the third line, $o.a'$ is set

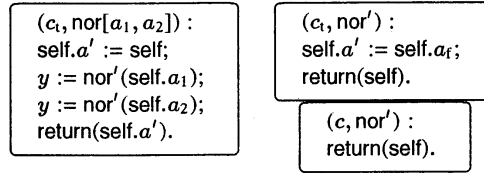
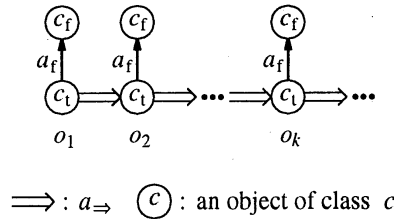
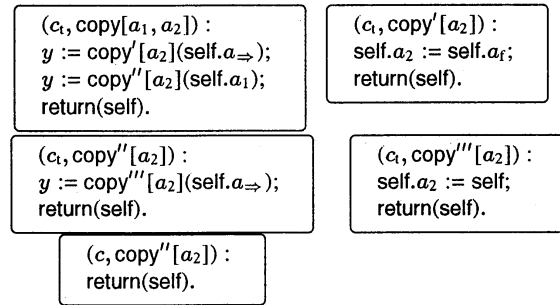
Fig. 2: Methods which calculate NOR of $o.a_1$ and $o.a_2$.

Fig. 3: A database instance.

Fig. 4: Methods which copy $o_j.a_1$ to $o_{j+1}.a_2$.

to $o.a_f$ if $o.a_2 = o$, and unchanged otherwise. Therefore, $o.a'$ is set to $o.a_f$ if $o.a_1 = o$ or $o.a_2 = o$, and unchanged (i.e., $o.a' = o$) otherwise. \square

Next, consider a database instance of this schema shown in Fig. 3. By invoking a method $\text{copy}[a_1, a_2]$ (Fig. 4) on an object o_j in the " a_{\Rightarrow} -chain," the Boolean value represented by $o_j.a_1$ is copied to $o_{j+1}.a_2$ ($= o_j.a_{\Rightarrow}.a_2$). Formally, we have the following lemma:

Lemma 2: Let o_j be an object of class c_t . After the execution of method $\text{copy}[a_1, a_2]$ on o_j , the following equation holds:

$$o_{j+1}.a_2 = \begin{cases} o_{j+1} & (\text{if } o_j.a_1 = o_j), \\ o_{j+1}.a_f & (\text{otherwise}). \end{cases}$$

Proof: An easy observation proves this lemma. \square

Lastly, see again Fig. 3. Suppose that a method m_0 returns $o.a_{\Rightarrow}$ when m_0 is invoked on o . Define method m_i ($1 \leq i \leq n$) as shown in Fig. 5. It is easy to see that m_n sequentially invokes m_0 on 2^n objects in the " a_{\Rightarrow} -chain" (see Fig. 6). Note that m_i ($1 \leq i \leq n$) can be constructed in polynomial time of n .

4 Recursion-Free Schemas

Definition 1: Problem RF/AC is to determine whether there exists an inconsistent acyclic instance under a given

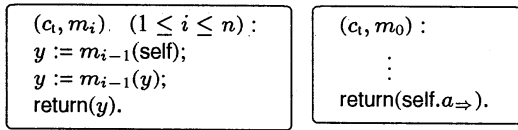


Fig. 5: Method which sequentially invokes method m_0 on 2^n objects.

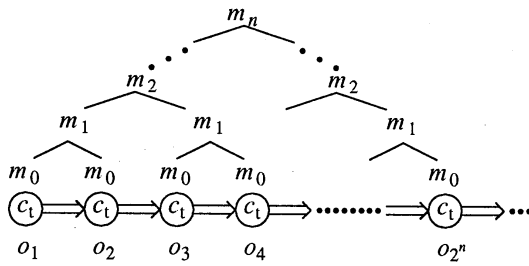


Fig. 6: Invocation of m_n on o_1 .

recursion-free schema S . □

We show that RF/AC is NEXPTIME-complete.

Lemma 3: RF/AC is in NEXPTIME.

Proof: Since S is recursion-free, execution of any method in S always terminates and the number of objects traversed during the execution is bounded by $|S|^{|S|}$. Therefore, to solve RF/AC , nondeterministically guess an instance of size $|S|^{|S|} = 2^{|S| \log |S|}$ which causes a type error. □

To show that RF/AC is NEXPTIME-hard, we reduce any language in NEXPTIME to RF/AC . To do this, for a given input string x of a fixed $2^{p(n)}$ -time bounded nondeterministic Turing machine M , we construct, in polynomial time of $|x|$, a schema $S_{M,x}$ such that there is an acyclic instance that is inconsistent under $S_{M,x}$ if and only if M accepts x . First, we define a nondeterministic Turing machine and an instantaneous description.

Definition 2: A nondeterministic Turing machine M is a triple (Q, Σ, δ) , where

- Q is a finite set of states. Q has three special states: the initial state q_0 , the accepting state q_{yes} , and the rejecting state q_{no} ;
- Σ is a finite set of symbols. Σ has two special symbols: the blank symbol B and the first symbol \triangleright . The first symbol is always placed at the leftmost cell of the tape; and
- δ is a function which maps $(Q - \{q_0, q_{yes}, q_{no}\}) \times \Sigma$ to the power set of $Q \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. δ must satisfy the following conditions:
 - For each pair $(q, \sigma) \in (Q - \{q_0, q_{yes}, q_{no}\}) \times \Sigma$, $|\delta(q, \sigma)|$ (the number of possible nondeterministic choices) is at most two. Assume that the elements of each $|\delta(q, \sigma)|$ are identified by 0 and 1; and
 - If $(q', \sigma, d) \in \delta(q, \triangleright)$, then $\sigma = \triangleright$ and $d = \rightarrow$. Therefore, the tape head never falls off the left end of the tape. □

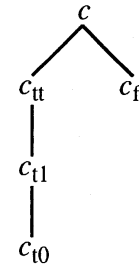


Fig. 7: \leq of $S_{M,x}$.

Class c_{tt} $a_1, \dots, a_K : c$ $a'_1, \dots, a'_K : c$ $a''_1, \dots, a''_K : c$ $a'''_1, \dots, a'''_K : c$ $a_{ch} : c$ $a_f : c_f$	Class c_{t1} $a_{\Rightarrow} : c_{tt}$ $a_1, \dots, a_K : c$ $a'_1, \dots, a'_K : c$ $a''_1, \dots, a''_K : c$ $a'''_1, \dots, a'''_K : c$ $a_{ch} : c$ $a_f : c_f$
---	---

Fig. 8: Ad of $S_{M,x}$.

Definition 3: An instantaneous description (ID) I of M is a finite sequence $\langle q_1, \sigma_1 \rangle, \dots, \langle q_k, \sigma_k \rangle$, where $q_i \in Q \cup \{\perp\}$ and $\sigma_i \in \Sigma$. It is required that exactly one q_i is in Q (i denotes the head position). The i -th pair $\langle q_i, \sigma_i \rangle$ of an ID I is denoted by $I[i]$. The transition relation \vdash_M over the set of IDs are defined as usual. Let I_j denote an ID after j -step transition of M . □

Let $M = \langle Q, \Sigma, \delta \rangle$ be a $2^{p(n)}$ -time bounded nondeterministic Turing machine. Let $x \in (\Sigma - \{B, \triangleright\})^*$ be an input string for M . Let $n = |x|$ and $N = 2^{p(n)}$. Let $K = \lceil \log(|Q| + 1) \rceil + \lceil \log |\Sigma| \rceil$. For M and x , define C, \leq , and Ad of $S_{M,x}$ as follows:

- $C = \{c, c_{t0}, c_{t1}, c_{tt}, c_f\}$;
- \leq is the smallest partial order such that
 1. $c_{t0} \leq c_{t1} \leq c_{tt} \leq c$, and
 2. $c_f \leq c$ (see Fig. 7); and
- Ad is shown in Fig. 8. Strictly speaking, some more temporary attributes, as necessary to store intermediate result of calculation.

An example of an acyclic database instance of $S_{M,x}$ is shown in Fig. 9. Note that any " a_{\Rightarrow} -chain" in any acyclic database instance terminates in an object of class c_{tt} . Objects o_1, \dots, o_{2N} in Fig. 9 are used as working space for simulating M : $I_j[i]$ is "stored" in object o_{i+j} (see Fig. 10). The class which each object o_j belongs to represents the nondeterministic choice at j -th step of M : Class c_{t0} represents choice 0 and c_{t1} does choice 1.

In what follows, we show that there is an inconsistent acyclic instance under $S_{M,x}$ if M accepts x . Let \mathcal{I} be an acyclic instance with an a_{\Rightarrow} -chain whose length k is greater than $2N$ (e.g., instance shown in Fig. 9). Let o_i ($1 \leq i \leq k$) be the i -th object in the a_{\Rightarrow} -chain.

Define method TM as shown in Fig. 11. Suppose that TM is invoked on o_1 . The behavior of TM is as follows:

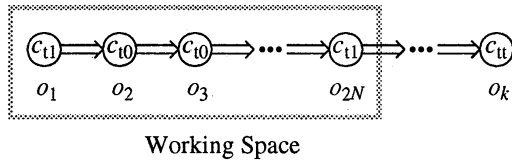
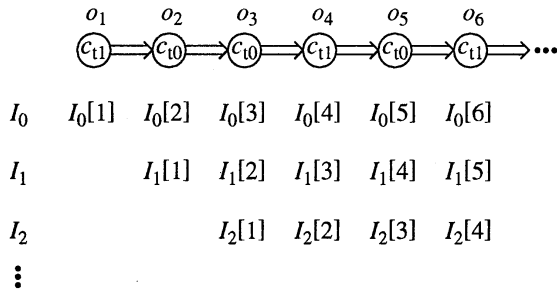
Fig. 9: An acyclic instance of $S_{M,x}$.

Fig. 10: Rewriting IDs.

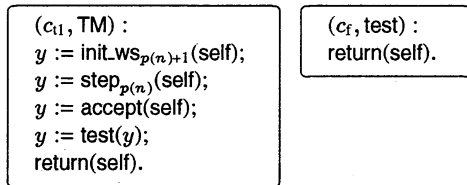
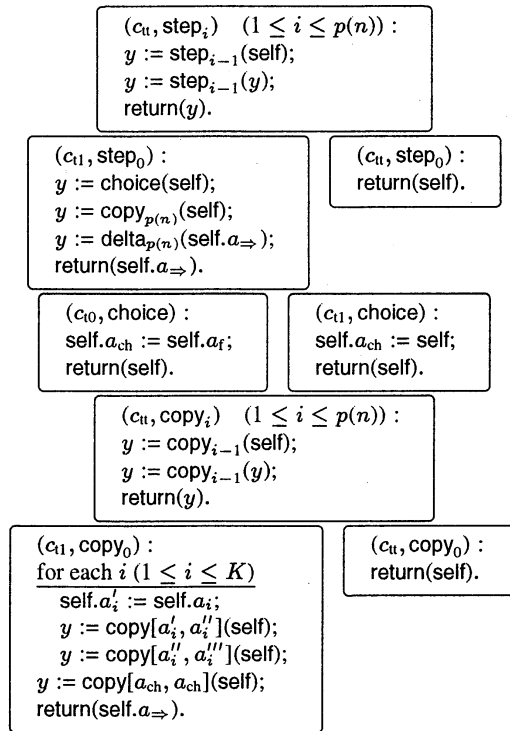
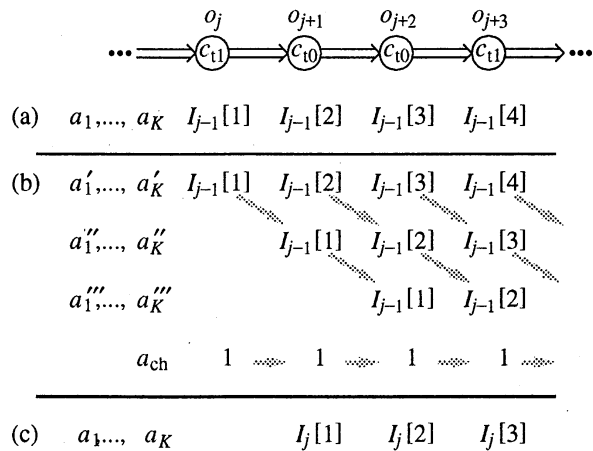


Fig. 11: Methods TM and test.

1. Initialize the first $2N = 2^{p(n)+1}$ objects (line 1 of (c_{t1}, TM) of Fig. 11). More precisely, for each i ($1 \leq i \leq 2N$), $I_0[i]$ is stored in $o_i.a_1, \dots, o_i.a_K$ by binary encoding.
2. Rewrite the ID stored in the working space $N (= 2^{p(n)})$ times (line 2). This phase is explained in detail below.
3. Check whether the accepting state q_{yes} is in the last ID, i.e., in objects o_{N+1}, \dots, o_{2N} (line 3). The returned value of `accept` is an object of class c_{tt} (or its subclass) if q_{yes} is in the last ID. Otherwise, an object of class c_f is returned. Method `accept` can be easily constructed by using methods `nor[a_1, a_2]`, `copy[a_1, a_2]` and m_n stated in Sect. 3.
4. Invoke `test` on the returned value of `accept` (line 4). Since method `test` is defined only for class c_f , that will cause a type error if q_{yes} is in the last ID. That is, \mathcal{I} is inconsistent under $S_{M,x}$ if M accepts x .

Now define methods $step_0, \dots, step_{p(n)}$ as shown in Fig. 12. Consider the j -th step of M . Each $I_{j-1}[i]$ ($1 \leq i \leq N$) is stored in object o_{i+j-1} (see Fig. 13(a)), and method $step_0$ is invoked on o_j . By method `choice` (Fig. 12), the nondeterministic choice $ch_j \in \{0, 1\}$ at the j -th step, which is given by the class which o_j belongs to, is stored in $o_j.a_{ch}$. Then, method `copy_0` (Fig. 12) is invoked on each o_{i+j-1} ($1 \leq i \leq N$). The underlined part (the first line of $(c_{t1}, copy_0)$) is macro notation. All of them can be expanded when M and x are reduced to

Fig. 12: Methods $step_i$, `choice`, and $copy_i$.Fig. 13: Behavior of method `step`.

$S_{M,x}$. After the invocations of `copy_0`, each object o_{i+j} ($1 \leq i \leq N$) has $I_{j-1}[i-1]$, $I_{j-1}[i]$, $I_{j-1}[i+1]$, and ch_j (see Fig. 13(b)). Lastly, method `delta_0` is invoked on each object o_{i+j} ($1 \leq i \leq N$) to obtain $I_j[i]$, which is to be stored in a_1, \dots, a_K of o_{i+j} (see Fig. 13(c)). Method `delta_0` can be constructed in constant time with respect to n by using `nor[a_1, a_2]` stated in Sect. 3.

Conversely, we show that there is an inconsistent acyclic instance under $S_{M,x}$ only if M accepts x . The whole of $S_{M,x}$ can be easily constructed so that a type error can occur only at the fourth line of (c_{tt}, TM) . And it is also easy to see that q_{yes} is stored in some of the objects

o_{N+1}, \dots, o_{2N} only if M accepts x .

Now we have the following theorem:

Theorem 1: RF/AC is NEXPTIME-complete. \square

Let RF be the problem of determining whether there exists an inconsistent instance under a given recursion-free schema S . By slightly modifying the construction of $S_{M,x}$, we have the following theorem:

Theorem 2: RF is NEXPTIME-complete. \square

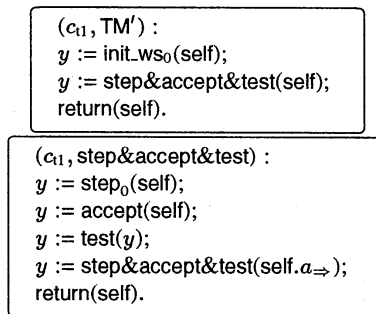
5 Arbitrary Schemas, Acyclic Instances

Definition 4: Problem AC is to determine whether there exists an inconsistent acyclic instance under a given schema S . \square

We prove that AC is undecidable by showing a reduction from any recursively enumerable language to AC . To do this, for a given input string x of a fixed Turing machine M , we construct a schema $S'_{M,x}$ such that an acyclic instance is inconsistent under $S'_{M,x}$ if and only if M accepts x .

Let $M = \langle Q, \Sigma, \delta \rangle$ be a deterministic Turing machine, i.e., for each pair $(q, \sigma) \in (Q - \{q_0, q_{yes}, q_{no}\}) \times \Sigma$, $|\delta(q, \sigma)|$ is at most one. Let $x \in (\Sigma - \{B, \triangleright\})^*$ be an input string for M . Let $K = \lceil \log(|Q| + 1) \rceil + \lceil \log |\Sigma| \rceil$. For M and x , define C, \leq , and Ad to be the same as $S_{M,x}$ in Sect. 4.

Let us construct TM' which simulates M on x . Since recursion is allowed now, we modify $init_ws_0$, $step_0$, etc. in Sect. 4 so that they recursively traverse the a_{\Rightarrow} -chain until it reaches object o_k of class c_{tt} . Therefore, $init_ws_{p(n)+1}$, $step_{p(n)}$, etc. are not necessary. Moreover, it is unknown when M stops in advance. A tentative solution would be as follows:



However, this does not work since $step&accept&test$ may invoke on an object in an a_{\Rightarrow} -chain which is not initialized. Hence, it is possible that there is an acyclic instance that is inconsistent under $S'_{M,x}$ even if M does not accept x .

Instead, we define TM' and $step'$ as shown in Fig. 14. Classes c_{t0} and c_{t1} represent the choice whether rewriting the ID is continued or not. To explain this more precisely, consider the situation that TM' is invoked on o_1 in Fig. 9. All the objects in the a_{\Rightarrow} -chain are initialized by method $init_ws_0$. Then $step'$ is invoked on o_1 . If o_1 is of class c_{t0} , then the ID stored in the a_{\Rightarrow} -chain is rewritten and $step'$ is recursively invoked on o_2 . This recursive invocation is

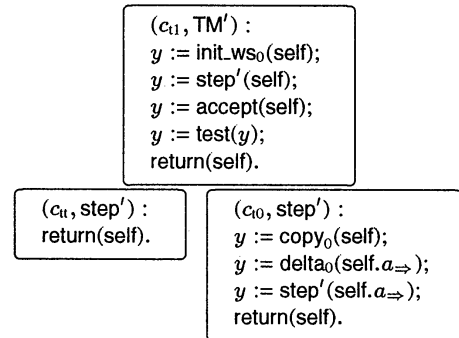


Fig. 14: Methods TM' and $step'$.

repeated until some object o_j of class c_{tt} or c_{t1} is encountered. That is, if $o_j \in \nu(c_{t0})$ for each j ($1 \leq j \leq k'$) and $o_{k'} \in \nu(c_{tt}) \cup \nu(c_{t1})$, then TM' simulates M up to k' steps.

By an easy observation, we have the following theorem:

Theorem 3: AC is undecidable. \square

Consider executing TM' on a cyclic database instance. Since all the attributes except a_{\Rightarrow} are initialized by $init_ws_0$, we can focus on the case that a_{\Rightarrow} forms a cycle. In this case, TM' does not terminate. More precisely, $init_ws_0$ is invoked infinitely many times without type error. Therefore, we have the following known result [2]:

Theorem 4: The problem of determining whether there exists an inconsistent instance under a given schema S is undecidable. \square

6 Conclusions

Theorems 1 and 2 mean that there are no algorithms to solve RF/AC or RF better than the obvious algorithm stated in Lemma 3. On the other hand, as stated in Sect. 1, these problems for method schemas are solvable in polynomial time (recall that method schemas do not update database instances). It is interesting to find a subclass of update schemas for which type consistency problems are solvable more efficiently than NEXPTIME.

References

- [1] S. Abiteboul, P. Kanellakis, S. Ramaswamy and E. Waller: "Method Schemas," J. Computer & System Sciences, Vol. 51, No. 3, pp. 433–455, Dec. 1995.
- [2] R. Hull, K. Tanaka and M. Yoshikawa: "Behavior Analysis of Object-Oriented Databases: Method Structure, Execution Trees, and Reachability," Proc. 3rd Int'l Conf. on Foundations of Data Organization and Algorithms, pp. 372–388, June 1989.
- [3] H. Seki, Y. Ishihara and M. Ito: "Authorization Analysis of Queries in Object-Oriented Databases," Proc. 4th Int'l Conf. on Deductive and Object-Oriented Databases, Lecture Notes in Computer Science 1013, pp. 521–538, Dec. 1995.