# Method Schemas*

Serge Abiteboul† Paris Kanellakis‡ Sridhar Ramaswamy§ Emmanuel Waller†

(submitted July 1, 1992 – revised August 31, 1994)

**Abstract**

A method schema is a simple programming formalism for object-oriented databases with features such as classes, methods, inheritance, name overloading, and late binding. An important problem is to check whether a given method schema can lead to an inconsistency in some interpretation. This consistency question is shown to be undecidable in general. Decidability is obtained for monadic and/or recursion-free method schemas. In particular, consistency of monadic method schemas is shown to be decidable in $O(nc^3)$ time, where $n$ is the size of the method definitions and $c$ is the size of the class hierarchy; also, it is logspace-complete in PTIME, even for monadic, recursion-free schemas. Method signature covariance is shown to simplify the computational complexity of key decidable cases. For example: one coded method in the context of base methods with covariant signatures can be tested for consistency in $O(n+c)$ time for the monadic case (without covariance this problem is in $O(nc^2)$ time) and in PTIME for the fixed arity polyadic case (without covariance this problem is coNP-complete). Incremental consistency checking of method schemas is a formalization of the database schema evolution problem, for which a sound, but necessarily incomplete, heuristic is proposed.

**Keywords**: object-oriented databases, program schemas, method schemas, classes, inheritance, name overloading, late binding, deciding consistency, database schema evolution.

## 1 Introduction

Object-oriented database systems are the focus of a great deal of current experimentation and research — for motivation and terminology see [3, 4, 23, 24, 35]. Although most of the work to-date has concentrated on system development (e.g., [5, 6, 12, 27, 34]) some of the

*A preliminary version of this paper appeared in the proceedings of the 9th ACM PODS, 1990.

†INRIA, 78153 Le Chesnay Cedex, France. abitebou@inria.inria.fr, waller@seti.inria.fr. Work supported by the Projet de Recherche Coordonnée BD3.

‡Dept. of Computer Science, Brown Univ., Box 1910, Providence RI 02912, USA. pck@cs.brown.edu. Research completed while visiting INRIA Altaïr; and supported by: NSF grant IRI-8617344, NSF-INRIA grant INT-8817874, ONR contract N00014-83-K-0146 ARPA Order No. 6320-1, and an Alfred P. Sloan Fellowship.

§Dept. of Computer Science, Brown University. sr@cs.brown.edu. Work supported by ONR Contract N00014-91-J-4052, ARPA Order 8225.

principles of these new systems have also been investigated. For example, [1] and [17] address the expressibility of languages in this new context, and use many of the tools of typed complex structure and semantic modeling research. Here, we do not examine such "structural" aspects, but focus instead on some of the "behavioral" aspects of the object-oriented paradigm — for some other recent work in this direction see [8, 18, 22].

We propose *method schemas* as a simple abstraction of the object-oriented programs that are being used in most of the existing database prototypes.

The critical features of method schemas are *classes with methods and inheritance*, a syntax involving *method composition, recursion and name overloading*, and an operational semantics involving *late binding*. A key computational problem is: *consistency maintenance of method schemas*. Its solution has many applications. For example, it can be used to facilitate the support of schema evolution (e.g., [7, 29]). Let us briefly outline our formalization:

**Syntax:** We have an *isa* hierarchy of *classes*, where each class denotes a set of *objects*. Each object has an associated set of *methods* and all objects in a class have the same methods. To allow *code reuse*, the methods of a class can be inherited by its descendants in the class hierarchy. To allow *code redefinition*, method names can be reused in different parts of the hierarchy (i.e., there is *name overloading*). We consider two kinds of methods: *base* methods that can be viewed as being defined extensionally (i.e., as materialized functions, or tables stored in the database, with particular signatures); and *coded* methods that consist of pieces of code (i.e., programs) built using base and coded methods. We consider very simple programs, whose syntax involves directed acyclic graphs labeled by method names.

**Semantics:** We propose an operational semantics. For a given finite interpretation of the base methods, the coded methods are defined using *graph rewriting*. Because of overloading, a given method call is rewritten based on its context (i.e., on the classes of its arguments). This models *late binding*. Rewriting is important in order to handle recursive calls. Without recursion, the semantics is just function composition. An *inconsistency* is reached if a method is called with arguments for which this method is undefined.

**Consistency:** We study the *consistency* problem. More precisely, we want to check whether a given method schema can possibly produce an inconsistency for some finite interpretation of the base methods. Our contributions are: (1) *a formalization of method schemas*, and (2) *a static analysis of method schema consistency*. This static analysis is the first important step if one is to understand incremental algorithms for problems such as schema evolution. Based on our analysis we outline (3) *a sound heuristic for method schema consistency maintenance in an object-oriented database system.*

The syntax and the semantics we use for method schemas are reasonably language independent. We provide an abstract view of flow of control based on method composition and possibly recursive method calls. We are consistent with any programming paradigm based on inheritance, overloading, and late binding. In this sense, the present work is similar to research on *recursive applicative program schemas* [10, 14, 26]. However, there are some important differences from traditional program schemas: (a) the class hierarchy with its inheritance mechanism replaces in coded methods the *interpreted if-then-else* construct of program schemas, (b) the overloading of base method names is outside the traditional framework, and (c) the signature information and finite interpretation for base methods is particular to the new setting and important for database applications.

The operational semantics that we use can be viewed as an abstraction of a Smalltalk interpreter [13], and is relatively straightforward. On the other hand, it is hard to statically (at compile-time) analyze methods. This difficulty is substantiated by the undecidability of consistency checking, even in our simple model. We prove undecidability by reduction from problems involving *program (flowchart) schemas* [26]. It turns out that the main feature in program schemas, *conditional binary transfer*, can be simulated using method overloading. The reduction is a modification of McCarthy's simulation of program (flowchart) schemas by recursive applicative program schemas [28]. Our reduction uses inheritance and name overloading in an essential fashion. Richard Hull proved undecidability of "reachability" for a different model of method schemas [16], by reducing from the Post correspondence problem. Almost the same proof can be used for showing undecidability of consistency in our model. However, we believe that our simulation of program schemas provides some additional insights into the relationship between the theory of program schemas and method schemas.

Undecidability comes from the simultaneous presence of recursion and the use of multi-argument methods to represent contexts (three arguments suffice). Practical programs are often less complex. Thus, we analyze in detail the consistency problem for *monadic* and/or *recursion-free* method schemas, which happen to be decidable. We also quantify the effect of *covariance*, which is a widely used constraint on the signature of methods.

For the various concepts used from complexity theory, see [15, 19, 20, 30] and from database theory, see [21, 31].

We briefly summarize our other results. Let $n$ be the size of method definitions in the input method schema and $c$ the size of the class hierarchy.

**In the case of monadic schemas,** the set of possible computations can be described using a context-free language. An inconsistency may be reached if this language has a non-empty intersection with a particular regular language. To handle overloading, our proof uses a modification of a technique of [2]. The decision procedure runs in $O(nc^3)$ time.

**In the case of monadic *and* recursion-free schemas,** we show that checking a whole schema for consistency is logspace-complete in PTIME. We reduce from the circuit value problem [25]. This indicates that very efficient incremental solutions for even the simplest syntactic cases may be hard to obtain. We then focus on the key recursion-free case of a single coded method in the context of base methods. Checking a single coded method for consistency is logspace-complete in NLOGSPACE and can be done more efficiently than the recursive case above using finite state automata techniques (in $O(nc^2)$ time). Inheritance and overloading introduce some nondeterminism in the automata theoretic approach to consistency checking, but covariant signatures remove it. For example, consistency of a single coded method, assuming covariance, is in DLOGSPACE and can be checked in $O(n+c)$ time. We present a first principles algorithm to show this linear time bound using radix trees. Initially, we showed this bound using two-way deterministic pushdown automata (2dpda's) [9]. We present this proof in an appendix.

**In the case of recursion-free schemas,** the consistency problem is coNP-complete for a single coded method with two arguments. Some special cases can be shown to be in PTIME, using tree automata techniques [11, 32]. Covariance does not help in the recursive case. However, in the recursion-free covariant case we show that there is a PTIME test for a fixed arity coded method. This is interesting in practice, because it motivates our heuristic for the general case.

In an object-oriented context, it is not reasonable to recompile all methods each time the schema is updated. The problem is to obtain an *incremental* consistency checking algorithm that would avoid redoing the same verifications/computations. A solution that is adopted practically (e.g., [5]) is to maintain a dependency graph of the methods and recompile only methods that may have been affected by the update. Of course, understanding what may be affected is the crucial part. This is where our analysis contributes. We briefly discuss this aspect and refer to [33] for more details.

The rest of the paper is organized as follows. In Section 2, we present the model. Schemas with recursion are considered in Section 3, and recursion-free schemas in Section 4. Section 5 deals with covariance. The heuristic for incremental consistency checking is considered in Section 6. We conclude with some open questions in Section 7.

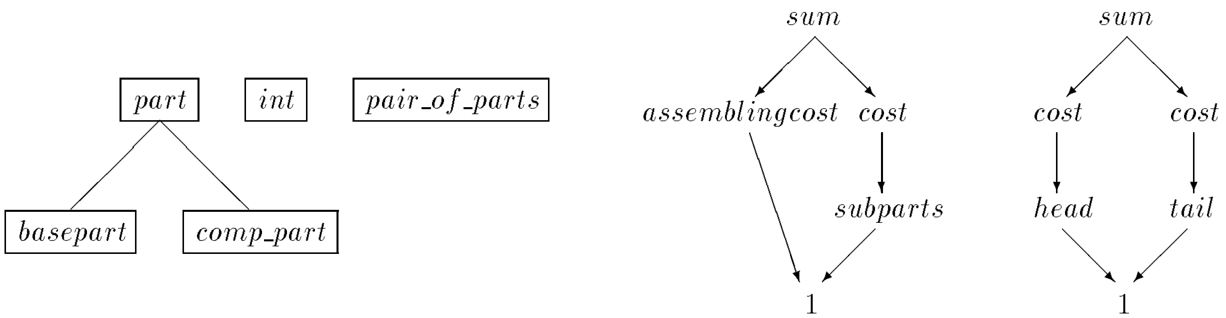
## 2 Method Schemas

We assume the existence of the following disjoint countable sets of symbols: of *classes* $(c_1, c_2, \ldots)$ and of *methods* $(m_1, m_2, \ldots)$. For each method $m$, the *arity* of $m$ is an integer greater or equal to zero ($arity(m) \geq 0$). A *signature* is an expression $c_1, \ldots, c_{k-1} \rightarrow c_k$, where each $c_i$, for $1 \leq i \leq k$, is a class. (For $k = 1$, the expresion is $\rightarrow c_1$.)

A *base definition* of $m$ at $c_1, \ldots, c_{k-1}$, for $k \geq 1$ is a pair $(m, (c_1, \ldots, c_{k-1} \rightarrow c_k))$, where $m$ is a method of arity $k-1$ and the remaining part is a signature. (For $k = 1$, the method $m$ is zero-ary.) A *coded definition* of $m$ at $c_1, \ldots, c_k$, for $k \geq 1$ is a triple $(m, (c_1, \ldots, c_k), t)$, where $m$ is a method of arity $k$, for $1 \leq i \leq k$, each $c_i$ is a class, and $t$ is a $k$-term (i.e., $k$-terms are our programming formalism).

A *$k$-term* for some $k \geq 1$ is a finite rooted ordered directed acyclic graph (dag) such that: (1) each vertex is uniquely labeled either by a method or by an integer in $\{1, \ldots, k\}$; (2) each vertex labeled by a method of arity $j$ has $j$ children which are ordered from left to right; (3) all vertices of out-degree zero are called *inputs* and each vertex labeled with an integer is an input; (4) the root is called the *output*; (5) the dag comes with a uniquely defined depth first search order going from left to right because the children of each vertex are ordered.

For example, the terms $t$ and $t'$ in Figure 1 are 1-terms. The idea is that $k$-terms represent functions of $k$ arguments, where a copy of argument $i$ is placed at each input labeled $i$. Vertices




*isa* hierarchy  term $t$  term $t'$

Figure 1: Hierarchy and Terms

can also have outdegree zero and be labeled by zero-ary methods, i.e., they are constant inputs. For example, consider the following definitions:

$$(price, (basepart \rightarrow int))$$
$$(cost, (basepart), s)$$
$$(cost, (part), t)$$

Let $s$ be a single arc with tail labeled *price* and head labeled 1, and $t$ be as in Figure 1. These definitions can be represented syntactically using an intuitive lambda notation — instead of the graphical $k$-term notation — as shown below. Note that we use the symbol colon (:) for representing base methods and the symbol equal (=) for coded methods.

$$method\ price@basepart : int$$
$$method\ cost@basepart = \lambda x_1.price(x_1)$$
$$method\ cost@part = \lambda x_1.$$
$$sum(assemblingcost(x_1), cost(subparts(x_1)))$$

**Definition 2.1** A *method schema* is a 4-tuple $(C, \leq, \Sigma_0, \Sigma_1)$ where:

1. $C$ is a finite set of classes and $\leq$ is a forest on $C$. We say that $c \leq c'$ if $c$ is a descendant of $c'$ in this forest. The partial order defined by $\leq$ is called the *isa* relation.

2. $\Sigma_0$ is a set of base definitions with classes from $C$.

3. $\Sigma_1$ a set of coded definitions with classes from $C$. Let $M_0 = \{m|(m, \ldots) \in \Sigma_0\}$ and $M_1 = \{m|(m, \ldots) \in \Sigma_1\}$. We require that $M_0 \cap M_1 = \emptyset$ and that the methods that appear in $k$-terms of $\Sigma_1$ are from $M_0 \cup M_1$.

4. Each method of arity $k$ has at most one definition at $c_1, \ldots, c_k$ for each $c_1, \ldots, c_k$. $\square$

$$
\begin{array}{lllcl}
method & sum & @int, int & : & int \\
method & head & @pair\_of\_parts & : & part \\
method & tail & @pair\_of\_parts & : & part \\
method & price & @basepart & : & int \\
method & subparts & @part & : & pair\_of\_parts \\
method & assemblingcost & @part & : & int \\
method & cost & @part & = & t \\
method & cost & @basepart & = & \lambda x_1.price(x_1) \\
method & cost & @pair\_of\_parts & = & t'
\end{array}
$$

Figure 2: Methods

**Example 2.2** Consider the class hierarchy of Figure 1 and the terms $t$ and $t'$ in that figure. Method definitions are presented in Figure 2.□

**Method Inheritance:** A method schema $(C, \leq, \Sigma_0, \Sigma_1)$ contains definitions of methods, i.e., $\Sigma_0, \Sigma_1$, which we call *explicit*. It is also used to define methods *implicitly*, through *inheritance*. This is for the convenience of code reuse. Therefore, in addition to the explicit definitions for a method, definitions are implicitly inherited along the class hierarchy. If a method $m$ is explicitly defined at classes $c'_1, \ldots, c'_k$, then its definition implicitly applies at $c_1, \ldots, c_k$ where $c_i \leq c'_i$ for $i = 1, \ldots, k$.

**Name overloading:** Method inheritance implies that we can have several definitions for the same base or coded method at the same classes — although by condition 4 of the method schema definition there can be only one explicit definition. This creates overloading of names. To illustrate overloading, consider the method *cost* in the example. That method is explicitly defined on *part*, implicitly inherited by *basepart* and *comp-part*, and explicitly redefined on *basepart*; it is also explicitly defined for *pair-of-parts*.

**Resolution of name overloading:** This consists of assigning to a given method and given classes, a unique definition. The resolution of a method $m$ at some $c_1, \ldots, c_k$ is the explicit definition of $m$ for the "component-wise smallest" $c'_1, \ldots, c'_k$ at which $m$ has an explicit definition and $c_i \leq c'_i$, for $i = 1, \ldots, k$ (if such a unique component-wise minimum exists). For example, *cost* at *comp-part* is resolved to be the explicit definition from *part*, whereas *cost* at *basepart* is *basepart*'s explicit definition. If $m$ has a resolution at $c_1, \ldots, c_k$, we say that $m$ is *well defined* at $c_1, \ldots, c_k$. Otherwise, we say that $m$ is *undefined* at $c_1, \ldots, c_k$. Non-definition can come either (i) because there is no definition of $m$ above $c_1, \ldots, c_k$ or (ii) because there is ambiguity of definitions above $c_1, \ldots, c_k$.

**Example 2.3** Let $c_1, c_2$ be classes with $c_1 \leq c_2$ and consider the schema with explicit base definitions of $m$ at $c_1, c_2$ and $c_2, c_1$. Then $m$ is undefined at $c_2, c_2$ (no definition) and $m$ is undefined at $c_1, c_1$ (ambiguity of definitions). For methods of arity 1, ambiguity (as defined in part (ii) in the definition above) cannot arise because of the forest hierarchy, but we can have non-definition (as defined in part (i) in the definition above). □

We assume the existence of a countably infinite set of *objects* $(o_1, o_2, \ldots)$ disjoint from classes and methods. We use object assignments [1] to provide semantics for classes with inheritance. The semantics of base methods is defined using partial functions satisfying certain signature constraints.

**Definition 2.4** Given a method schema $(C, \leq, \Sigma_0, \Sigma_1)$, a *disjoint object assignment* $\nu$ for $C$ is a total function from $C$ to finite sets of objects such that distinct classes are mapped to disjoint sets of objects (sets with pairwise empty intersections). For each $c$, we define $\nu(c*) = \bigcup_{c_1 \leq c} \nu(c_1)$. We will refer to $\bigcup_{c \in C} \nu(c)$ as the set of objects in $\nu$. □

**Definition 2.5** An *interpretation* of a schema $S = (C, \leq, \Sigma_0, \Sigma_1)$ is a pair $I = (\nu, \mu)$ where $\nu$ is a disjoint object assignment for $C$ and $\mu$ a total function from the base methods $M_0$ in $S$ to partial functions such that:

1. If $m$ has arity $k$, $\mu(m)$ is a partial function from $k$-tuples of objects in $\nu$ to objects in $\nu$.
2. For each $c_1, \ldots, c_k$, if $m$ is undefined at $c_1, \ldots, c_k$, then $\mu(m)$ is undefined everywhere in $\nu(c_1) \times \cdots \times \nu(c_k)$. Otherwise, let the resolution of $m$ at $c_1, \ldots, c_k$ be $(m, (c'_1, \ldots, c'_k \to c))$ where $c_i \leq c'_i$ for $i = 1, \ldots, k$. Then we have that $\mu(m) \mid_{\nu(c_1) \times \cdots \times \nu(c_k)}$ is a total function into $\nu(c*)$. □

Intuitively, every class $c$ is populated by a set of objects $\nu(c*)$, some of which are in this class exclusively (those objects that are in $\nu(c)$) and the rest belong to its "subclasses" $c_1$, where $c_1 \leq c$ and $c_1 \neq c$. When a base method is defined at a class $c_1$ with signature $c_1 \to c$ (or is inherited at $c_1$ with signature $c'_1 \to c$ such that $c_1 \leq c'_1$), then its meaning at $c_1$ is a total function from $\nu(c_1)$ to $\nu(c*)$. That is, given an object exclusively in $c_1$, return an object in $c$.

The semantics of coded methods is defined using the above semantics of base methods and a rewriting technique. Without recursion, this rewriting is equivalent to function composition.

**Late binding:** This is a result of the operational definition of rewriting. Let us first give some intuition for the rewriting of coded methods. Consider a $k$-term that is a single arc. Let its one internal node be labeled *cost* and its input be replaced by an object $o$, as a "procedure call" to *cost*. Based on the class of its argument, we can replace *cost* either by some object/code if it is defined in a base/coded fashion, or by an error message if it is undefined. In general, we reduce the first method (first in the depth first ordering of the $k$-term we are processing) that has instantiated leaves as children. We continue this processing until we either obtain a result (i.e., an object), or reach an inconsistency. We might also not terminate. A partial sequence of rewritings for Example 2.2 is shown in Figure 3, where $o$ is in class *pair_of_parts* and $o', o''$ are in class *basepart*.

More formally, an *instantiated term* in interpretation $I = (\nu, \mu)$ is a $k$-term, whose integer labels have been replaced by objects. Let $c_1, \ldots, c_k$ be classes, $m$ a method of arity $k$, and for each $i$ let $o_i \in \nu(c_i)$. Then the instantiated term consisting of a single vertex labeled $m$ and $k$ inputs labeled respectively $o_1, \ldots, o_k$ is said to be a *redex*. We define *first-reducible-vertex* to be the uniquely determined first vertex in the depth first ordering of an instantiated term, which is the root of a redex.

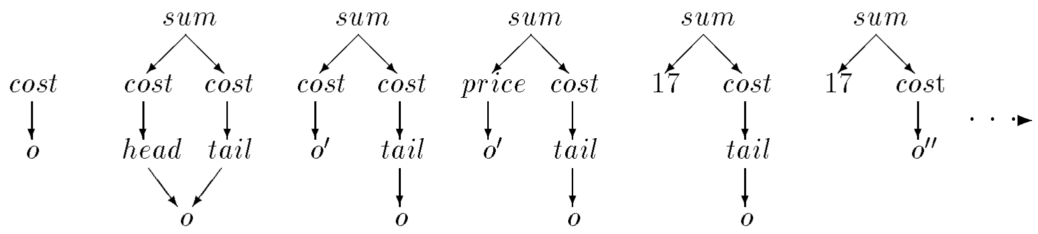



Figure 3: A (partial) sequence of rewritings

**Definition 2.6** Let $I$ be an interpretation of method schema $S$, $t$ be an instantiated term in $I$, $w$ the first-reducible-vertex of $t$, and $m$ the label of $w$. Let $o_1, \ldots, o_k$ be the labels of the children of $w$ belonging respectively to classes $c_1, \ldots, c_k$. The *reduction* $I(t)$ of $t$ in $I$ is obtained as follows:

1. If $m$ is undefined at $c_1, \ldots, c_k$ then $I(t)$ is a special symbol $\perp$.
2. If $m$ is a base method well defined at $c_1, \ldots, c_k$, let its resolution there be $(m, (c'_1, \ldots, c'_k \rightarrow c))$. (This implies that $c_i \leq c'_i$ for $i = 1, \ldots, k$.) Then $I(t)$ is the instantiated term obtained by removing from $w$ the outgoing edges and changing its label to $\mu(m)(o_1, \ldots, o_k)$. $\mu(m)$ is a total function from $\nu(c_1) \times \cdots \times \nu(c_k)$ into $\nu(c*)$. Therefore, $\mu(m)(o_1, \ldots, o_k)$ yields an object $o$ such that $o \in \nu(c*)$.
3. If $m$ is a coded method well defined at $c_1, \ldots, c_k$, then $I(t)$ is the instantiated term obtained by substituting, with the natural renaming of inputs, the vertex $w$ by the dag $s$, where $(m, \sigma, s)$ is the resolution of $m$ at $c_1, \ldots, c_k$. $\square$

When $I$ is clear from the context, we denote the reduction of $t$ to $I(t)$ as $t \rightarrow I(t)$ and a finite sequence of reductions from $t$ to $t'$ as $t \xrightarrow{*} t'$.

**Consistency:** We are interested in the rewriting sequences of a particular set of instantiated terms, i.e., these are the initial "procedure calls" that make sense according to a method schema. More formally we have the following: Let $S$ be a method schema with interpretation $I = (\nu, \mu)$, $c_1, \ldots, c_k$ be classes in $S$, $m$ a method of arity $k$ well defined at these classes, and for each $i$ let $o_i \in \nu(c_i)$. Then the instantiated term consisting of a single vertex labeled $m$ and $k$ inputs labeled respectively $o_1, \ldots, o_k$ is said to be a *start-redex*. We have an *inconsistency* in $I$ if for some start-redex $t$ we have $t \xrightarrow{*} \perp$.

**Definition 2.7** A method schema $S$ is *consistent* if for each interpretation $I$ of $S$ it is impossible to rewrite any start-redex into $\perp$ in a finite number of steps. $\square$

A variety of questions are important in this setting, e.g.: (1) Is a schema consistent? (2) Are there possible/certain diverging computations? (3) Is method $m$ (at $c$) possibly/certainly "reachable" from $m'$ (at $c'$) [18]? We concentrate here on (1) although most of the techniques that we develop can be used for studying other problems such as (2) or (3). Two properties simplify consistency checking: monadic arity and absence of recursion.

A schema is *monadic* if all its methods have arity one.

A schema is *recursion-free* if there is no directed cycle in the *method dependence graph*: where the vertices of the method dependence graph are the coded methods and there is an arc from $m$ to $n$ iff $m$ occurs in some coded definition of $n$. (Note that, our definition of recursion-free'ness is a reasonable syntactic way of avoiding recursion, but not necessarily the only one.)

A schema is *simple* if only base methods occur in its coded definitions. Note that, simple schemas are a proper subset of recursion-free schemas.

**Variations:** To conclude this section, we consider some important variations of the above definitions.

1. Constraints can be imposed on method signatures, e.g., "covariance".
2. "Virtual" classes can be considered.

In Section 5, we present definitions for these variations, study covariance, and mention briefly some unexpected aspects of virtual classes.

3. The *isa* order on classes may be more complicated. In particular, one may allow a class to have more than one direct ancestor, i.e., have "multiple inheritance".
4. In some proposals, it is required that the first argument, called the receiver, of a method must determine the resolution of name overloading. Each instance of a method is then viewed as "attached" to the class of the first argument.

We do not address multiple inheritance and attachment to first argument. Once the resolution mechanism is determined, our techniques are directly applicable to these cases also. In particular, our results hold if methods are attached to the class of the first argument (see discussion in Section 7).

# 3 Schemas with recursion

In this section, we allow recursion in method definitions. We first consider the monadic, then turn to the general case.

In the monadic case, one can use context-free language techniques to show that the problem is decidable. Here, method computations can be written down as sequences of base method applications. Possible prefixes of computations are described using a context-free language. The computations leading to inconsistencies are described using a regular language. The problem is therefore reduced to testing emptiness of the intersection of a context-free language and a regular one, i.e., to testing emptiness of a context-free language.

**Definition 3.1** The context-free grammar $G_S = (A, V_t, V_n, R)$ associated with a monadic schema $S = (C, \leq, \Sigma_0, \Sigma_1)$ is defined as follows. The start symbol is $A$. The set of terminals $V_t$ is $M_0 \cup \{error, nonterm\}$. $V_n = \{[c, m, c'] | c, c' \in C, m \in M_0 \cup M_1\} \bigcup \{A\}$ is the set of non-terminals. The set of productions $R$ is defined as follows[1]:

1. for each $m \in M_1$, and each $c, c' \in C$ such that $m$ is defined at $c$,
   $R$ contains $A \Rightarrow [c, m, c']$;

2. for each $m \in M_0 \cup M_1$, and $c, c' \in C$, such that $m$ is undefined at $c$,
   $R$ contains $[c, m, c'] \Rightarrow error$;

3. for each $m \in M_0$, and $c, c', c_1, c_2 \in C$,
   if the resolution of $m$ at $c$ is $m@c_1 : c_2$ for $c \leq c_1$ and $c' \leq c_2$, then,
   $R$ contains $[c, m, c'] \Rightarrow m$;

4. for each $m \in M_1$ and $c, c_1 \in C$,
   if the resolution of $m$ at $c$ is $m@c_1 = \lambda x.x$ for $c \leq c_1$, then,
   $R$ contains $[c, m, c] \Rightarrow \epsilon$;

5. for each $m \in M_1$, and $c, c_1 \in C$,
   if the resolution of $m$ at $c$ is $m@c_1 = \lambda x.m_n(m_{n-1}(\ldots(m_2(m_1(x)))\ldots))$ where $c \leq c_1$, then, for each $Z_1, \ldots, Z_n \in C$,
   $R$ contains $[c, m, Z_n] \Rightarrow [c, m_1, Z_1][Z_1, m_2, Z_2] \cdots [Z_{n-1}, m_n, Z_n]$;

6. for each $m \in M_0 \cup M_1$, $c, c' \in C$,
   $R$ contains $[c, m, c'] \Rightarrow nonterm$.

The language generated by $G_S$ is denoted by $L(G_S)$. $\Box$

A point about Definition 3.1. Rules generated according to part 4 correspond to "identity" methods. These codes present a problem to the normal way we think about coded methods. Recall from Definition 2.5 that if the resolution of a monadic base method $m$ at $c_1$ is $(m, (c'_1 \rightarrow c))$, where $c_1 \leq c'_1$, it means that $m$, when it is called with an argument object of class $c_1$, can return an object in $c$ or any class "below" $c$ in the inheritance hierarchy. A similar semantics applies to coded methods also. Identity codes are an exception because the object they return is the input and thus always belongs to the class of the argument they are given. This is the reason why they are handled separately in the definition above. Part 4 of the definition forces the class of the input and the output to be the same for identity codes.

**Lemma 3.2** Let $S = (C, \leq, \Sigma_0, \Sigma_1)$ be a monadic method schema. Then $S$ is inconsistent if and only if

$$X = L(G_S) \cap M_0^* error(M_0 \cup \{error, nonterm\})^* \neq \phi$$

[1] To distinguish grammatical productions from reductions in method computations we use $\Rightarrow$.

**Proof:** (Only if) First suppose that $S$ is inconsistent. We will show that $X \neq \phi$. Let $I = (\nu, \mu)$ be an interpretation of $S$, such that $m$ is well defined at the class $\widehat{c} \in C$. Let $\widehat{o} \in \nu(\widehat{c})$. Suppose that the start-redex, $m(\widehat{o})$ reduces to $\perp$. By analyzing the rewriting sequence $m(\widehat{o}) \xrightarrow{*} \perp$, we construct iteratively the derivation of a word in $X$.

The sentential form corresponding to $m(\widehat{o})$ consists of a single symbol $[\widehat{c}, m, c_{arb}]$ for some arbitrary class $c_{arb}$. This word is derivable from the start symbol $A$ since $m(\widehat{o})$ is a valid start-redex.

At one step of the method rewriting, let $t = m_{i_k}(\ldots(m_{i_1}(m_i(o))\ldots)$ be the current instantiated term for some $m_i, m_{i_1}, \ldots, m_{i_k} \in M_0 \cup M_1$ and an object $o \in \nu(c)$ for some $c \in C$. Method computation (i.e., rewriting) proceeds from inside out. From the way we have written our grammatical rules, leftmost derivations correspond to method computations. Let $u$ be the word derivable from $A$ corresponding to the current instantiated term $t$. It is an easy induction on the number of steps of the rewriting that the leftmost non-terminal of $u$ is of the form $[c, m_i, Z]$. If $t \rightarrow t'$, then we derive a word $u'$ from $u$ by changing the leftmost non-terminal in $u$, say $[c, m_i, Z]$, as follows:

1. if $m_i$ is a coded method, and the resolution of $m_i$ at $c$ is $m_i@c_1 = \lambda x.(m_{j_l}(\ldots(m_{j_2}(m_{j_1}(x)))\ldots))$, where $c \leq c_1$, then we use a rule, $[c, m_i, Z] \Rightarrow [c, m_{j_1}, Z_1]\ldots[Z_{l-1}, m_{j_l}, Z]$ by Definition 3.1, part 5. The choice of each $Z_i$ is given by:
   (a) if $m_{j_p}(\ldots(m_{j_1}(o))\ldots)$ is defined then $Z_{j_p}$ is the class such that $m_{j_p}(\ldots(m_{j_1}(o))\ldots) \in \nu(Z_{j_p})$. We can determine this class because we know the method computation completely.
   (b) otherwise, the choice for $Z_{j_p}$ is arbitrary.
2. if $m_i$ is a base method and its resolution at $c$ is $m_i@c_1 : c_2$, then $Z = c'$ for some $c' \leq c_2$ by the way we make choices for the $Z_{j_p}$'s above. Then $[c, m_i, c'] \Rightarrow m_i$, by Definition 3.1, part 3.
3. if $m_i$ is a coded method and its resolution at $c$ is $m_i@c' = \lambda x.x$, then $Z = c$ by the way we make choices for the $Z_{j_p}$'s above. Then $[c, m_i, c] \Rightarrow \epsilon$, by Definition 3.1, part 4.
4. if $m_i$ is undefined at $c$ (that is, $t'$ is $\perp$), then $[c, m, Z] \Rightarrow error$ by Definition 3.1, part 2.

When $\perp$ is obtained, we have derived a word $v$. Note that $v$ has a prefix in $M_0^* error$. It now suffices to rewrite all the non-terminals of $v$ to symbols in $(M_0 \cup \{nonterm, error\})$ by Definition 3.1, rules 2, 3, and 6. (Note that the arbitrary choice of $c_{arb}$ and eventually of some $Z_i$ does not prevent us from doing so.) Clearly, the word thereby obtained is in $X$.

(If) Conversely, assume $X \neq \phi$. We will show that $S$ is inconsistent. Consider a string $u$ such that

$$u \in L(G_S) \cap M_0^* error(M_0 \cup \{error, nonterm\})^*.$$

Then, for some $n > 0$, $u$ has a prefix $m_1 \ldots m_{n-1} error$ where the $m_i$'s are in $M_0$. Consider a derivation tree of $u$. It is easy to see that the respective parents of the leaves corresponding to the prefix $m_1, \ldots, m_{n-1}, error$ are $[c_1, m_1, c_2]$,...,$[c_n, m_n, c_{n+1}]$ for some classes $c_1, \ldots, c_{n+1}$ and some method $m_n$.

We first build a "partial" interpretation $I = (\nu, \mu)$ as follows. For $i = 1, \ldots, n$, let $o_i$ be in $\nu(c_i)$. For each $i = 1, \ldots, n-1$, let $(\mu(m_i))(o_i) = o_{i+1}$. By the construction of the grammar $G_S$, Definition 3.1, this partial interpretation is compatible with $S$. Also, $\mu(m_n)$ is undefined at $o_n$. This partial interpretation can be naively completed to an interpretation of $S$.

Consider the first step in the derivation of $u$. This step is $A \Rightarrow [c_1, m, Z]$ for some coded method $m$ and $Z$. By a process similar to the one we used in the only if part of the proof, we can show that $m$ is well defined at $c_1$ and that $m(o_1)$ reduces to $\perp$. $\square$

We are finally left with one more hurdle. The grammar we have used in the proof of Lemma 3.2 has exponential size. The number of rules generated according to Definition 3.1, part 5 is exponential in the length of the coded method because we are guessing the signatures of all the methods in its definition at once. The number of rules generated by all the other parts of Definition 3.1 can be easily seen to be polynomial in the size of the input.

**Lemma 3.3** Let $S = (C, \leq, \Sigma_0, \Sigma_1)$ be a monadic method schema. Let $n$ be the size of the method definitions in $S$ and $c$ be the size of its inheritance hierarchy. Let $G_S$ be as in Definition 3.1. There exists a context-free grammar, with $O(nc^3)$ size, that is equivalent to $G_S$.

**Proof:** As we observed before, only Definition 3.1, part 5 — the part that generates rules for a coded method definition — generates an exponential number of rules. All the other parts in Definition 3.1 generate $O(nc^2)$ rules.

We use a simple rewriting trick to reduce the number of rules needed for a coded method. It is based on the idea that we do not have to guess the input/output classes of all the methods in the coded method definition all at once. By guessing them one at a time, we can achieve a polynomial bound on the number of rules.

First, we modify the set of non-terminals $V_n$ of $G_S$ as follows. For each $m \in M_1$, and $c, c', c'' \in C$, if the resolution of $m$ at $c$ is $m@c' = \lambda x.m_l(m_{l-1}(\ldots(m_2(m_1(x)))\ldots))$, such that $c \leq c'$ we add the following non-terminals to $V_n$: $[c, m_q m_{q+1} \ldots m_l, c'']$ for $q = 1, \ldots, l$.

Further, instead of writing rules according to part 5 of Definition 3.1, we do the following:

- $R$ contains the rule $[c, m, c_1] \Rightarrow [c, m_1 m_2 \ldots m_l, c_1], \forall c_1 \in C$
- For each non-terminal $[c, m_1 m_2 \ldots m_l, c_1]$ produced according to the previous rule, R contains $[c, m_1 m_2 \ldots m_l, c_1] \Rightarrow [c, m_1, c_2][c_2, m_2 \ldots m_l, c_1], \forall c_2 \in C$
- For each non-terminal $[c_2, m_2 m_3 \ldots m_l, c_1]$ produced according to the previous rule, R contains $[c_2, m_2 m_3 \ldots m_l, c_1] \Rightarrow [c_2, m_2, c_3][c_3, m_3 m_4 \ldots m_l, c_1], \forall c_3 \in C$
  - ⋅
  - ⋅
  - ⋅

- For each non-terminal $[c_{l-1}, m_{l-1}m_l, c_1]$ produced according to the previous rule, R contains $[c_{l-1}, m_{l-1}m_l, c_1] \Rightarrow [c_{l-1}, m_{l-1}, c_l][c_l, m_l, c_1], \forall c_l \in C$

By induction on the length of the coded method, the rules above generate the same strings as the rules in Definition 3.1, part 5 and vice-versa. Notice that the construction has size $O(nc^3)$. The lemma follows.□

**Theorem 3.4** Consistency of monadic schemas can be tested in $O(nc^3)$ time, where $n$ is the size of the method definitions in the schema and $c$ is the size of its class hierarchy.

**Proof:** In order to check the consistency of a monadic schema, we generate the context-free grammar corresponding to it per Lemma 3.3. We intersect this grammar with the linear-sized regular grammar in Lemma 3.2. The resulting context-free grammar has size $O(nc^3)$. Now, testing the emptiness of a context-free grammar can be reduced to depth first search [15]. The theorem follows. □

We now consider the most general case. In particular, we show that the consistency problem for arbitrary schemas is undecidable. This was first shown by Richard Hull for a different method formalism, [16]. To prove our result, we show that method schemas can simulate *program (flowchart) schemas*. The undecidability results then come from the undecidability results for these program schemas of [26]. See also [14] for *linear recursion*.

**Definition 3.5** For program (flowchart) schemas, we will adopt the formal language of [26], which contains the following symbols:

(i) $F_1^1, F_2^1, F_3^1, \ldots, F_1^2, F_2^2, F_3^2, \ldots$ (operator symbols with superscripts denoting arity),
(ii) $L_1, L_2, L_3, \ldots$ (location symbols),
(iii) $T_1, T_2, T_3, \ldots$ (transfer symbols),
(iv) $:=$ (assignment symbol),
(v) $STOP$,
(vi) (, ), and comma (auxiliary symbols), and
(vii) numerals (symbols other than $L, F, T$, and $STOP$ denote numerals).

The permissible statements (or instructions) of the language are of three types. In each case, the numeral $k$ is called the prefix or address of the instruction.

(i) *Assignment instructions*

$$k.L_i := F_j^n(L_{k_1}, L_{k_2}, \ldots, L_{k_n})$$

(ii) *Transfer instructions*

$$k.T_i(L_j)m, n$$

where $m, n$ are numerals (transfer addresses).

(iii) *Stop instruction*

$$k.STOP$$

A *Program (Flowchart) Schema* $P$, is a finite sequence of permissible instructions such that $(i)$ the prefix of each instruction is its position in the sequence, $(ii)$ all transfer addresses are prefixes of $P$, and $(iii)$ the last instruction is either a transfer or $STOP$.□

A program schema, like a method schema, represents a family of programs. Just as the meaning of a coded definition is dependent upon the interpretation attached to the base definitions and the classes that it uses, the meaning of a program schema is dependent upon the interpretation attached to operator and transfer symbols in it. The computation in a program schema begins at the first instruction and proceeds sequentially unless a transfer instruction is encountered. The location symbols have an initial set of data attached to them (which is part of the "interpretation" given to the program schema). The assignment instruction has the obvious meaning and the transfer instruction is an *if-then-else* statement. The computation terminates when a $STOP$ instruction is encountered.

**Example 3.6** Consider the following program schema (this is also from [26]):

1. $L_2 := F_1(L_2)$,
2. $T_1(L_1)5, 5$,
3. $L_2 := F_2(L_1, L_2)$,
4. $L_1 := F_3(L_1)$,
5. $T_1(L_1)6, 3$,
6. STOP.

In [26], it is shown that the program above can compute the factorial function under some interpretations and under others, over list structures, it can compute the reverse function.□

For simplicity of presentation, we omit the detailed definitions of the semantics of program schemas and refer the reader to [26].

We will now argue that method schemas can "simulate" program schemas. Intuitively, we carry the context of the program (content of variables) as arguments of the methods in a standard way. Base functions in the method schemas correspond naturally to operator symbols in program schemas. We simulate conditional transfer by overloaded methods. The computation domain of the program schema is represented by a class $c$. The booleans are represented by two classes $c_t$ and $c_f$ with $c_f \leq c_t$. Let us now give the formal details.

**Simulation of Program Schemas by Method Schemas**

Let $P$ be a program schema. We construct a method schema $S_P$ that has three classes $c, c_t$, and, $c_f$, with $c_f \leq c_t$. If there are $p$ location symbols used in $P$, $S_P$ has the following methods, where $c^p$ stands for $c, \ldots, c$ repeated $p$ times:

1. a base method $f_j^i$ with signature $c^i \rightarrow c$ for each operator $F_j^i$ occurring in $P$ (notice that $i$ has to be less than or equal to $p$ since we have only $p$ location symbols);

2. a base method $t_i$ with signature $c \to c_t$ for each transfer $T_i$ occurring in $P$;
3. a coded method $inst_k@c^p$ for each instruction $k$ occurring in $P$;
4. a coded method $ite_k@c^p, c_t$ for each transfer instruction $k$ occurring in $P$;
5. a coded method $run@c^p$;

The method $run$ is defined by:

$$method\ run@c, \ldots, c = \lambda x_1, \ldots, x_p.inst_1(x_1, \ldots, x_p).$$

The instructions of $P$ correspond to the following methods:

- assignment $k.L_i := F_j^n(L_{k_1}, \ldots, L_{k_n})$

$$method\ inst_k@c, \ldots, c = \lambda x_1, \ldots, x_p.inst_{k+1}(x_1, \ldots, x_{i-1}, f_j^n(x_{k_1}, \ldots, x_{k_n}), x_{i+1}, \ldots, x_p)$$

- transfer $k.T_i(L_j)\ m, n$

$$method\ inst_k@c, \ldots, c = \lambda x_1, \ldots, x_p.ite_k(x_1, \ldots, x_p, t_i(x_j))$$

with an implementation of if-then-else by

$$\begin{array}{l} ite_k@c, \ldots, c, c_t = \lambda x_1, \ldots, x_p, x_{p+1}.inst_m(x_1, \ldots, x_p) \\ ite_k@c, \ldots, c, c_f = \lambda x_1, \ldots, x_p, x_{p+1}.inst_n(x_1, \ldots, x_p) \end{array}$$

- stop $k.STOP$

$$method\ inst_k@c, \ldots, c = \lambda x_1, \ldots, x_p.x_1.$$

A run of program schema $P$ is simulated by a call $run(o_1, \ldots, o_p)$ where the $o_i$'s represent the initializations of the $p$ locations.

For each finite interpretation $I_P$ of the program schema $P$, the corresponding interpretation $I_{S_P} = (\nu, \mu)$ of the method schema $S_P$ is defined as follows: $\nu(c)$ is the computation domain of $P$, $\nu(c_f) = \{o_f\}$, $\nu(c_t) = \{o_t\}$ where $o_f, o_t$ are two new symbols. The base method semantics in $S_P$ are the same as the operator and transfer symbol semantics in $P$.

For each interpretation $\mathcal{I}_{S_P}$ of the method schema $S_P$, the corresponding interpretation $\mathcal{I}_P$ of the program schema $P$ is obtained as follows: the computation domain is $\nu(c)$. The operator and transfer symbol semantics in $P$ are the same as the $f$ and $t$ base method semantics in $S_P$. (Note that if $c_f$ or $c_t$ has an empty population, then this implies that the transfer functions are constant.)

By simple structural induction, it is now possible to show two simulation lemmas.

**Lemma 3.7** For each finite interpretation $I_P$ of the program schema $P$, the corresponding interpretation $I_{S_P}$ of the method schema $S_P$ is such that: each run of $P$ with initialization $o_1, \ldots, o_p$ terminates iff the call $run(o_1, \ldots, o_p)$ does, and if it terminates, the value of $run(o_1, \ldots, o_p)$ is exactly the content of location 1 after the run of $P$.

**Lemma 3.8** For each finite interpretation $\mathcal{I}_{S_P}$ of the method schema $S_P$, the corresponding interpretation $\mathcal{I}_P$ of the program schema $P$ is such that: each call $run(o_1, \ldots, o_p)$ terminates iff the run of $P$ with initialization $o_1, \ldots, o_p$ terminates and if it terminates, the value in location 1 after the run of $P$ is exactly the value returned by $run(o_1, \ldots, o_p)$.

The above lemmas allow us to derive a variety of undecidability results for method schemas.

**Lemma 3.9** The following properties of a method schema $S$ with a method $m$ are not recursively enumerable (r.e.), even if $S$ is a consistent method schema:

1. All calls to $m$ terminate in each interpretation.
2. All calls to $m$ do not terminate in each interpretation.

**Proof:** We observe that the method schema that we construct in the simulation of program schemas by method schemas is always consistent. We then use the simulation of program schemas by method schemas, and Theorem 4.1 (respectively d,b) in [26].□

**Lemma 3.10** The following properties of a method schema $S$ with a method $m$ are r.e.:

1. Some call to $m$ terminates in some interpretation.
2. Some call to $m$ does not terminate in some interpretation.

**Proof:** Suppose that an interpretation exists such that some call to a method $m$ terminates. Since we consider only finite interpretations and finite computations, we can guess the interpretation as well as the call that will terminate. We only need to verify that this call does indeed terminate in a finite number of steps.

The second part of the lemma is equally easy to prove. Instead of a call terminating, we are interested in some call looping in some interpretation. As before, we guess the (finite) interpretation and the call. We start our computation and keep a directed graph whose nodes consist of the coded method calls — methods names as well as arguments — that we issue. Note that this set is finite. If a call to a method $m_0$ with arguments $(o_1, \ldots, o_k)$ results in a call to $m'_0$ with arguments $(o'_1, \ldots, o'_k)$, we draw a directed arc from $(m_0, o_1, \ldots, o_k)$ to $(m'_0, o'_1, \ldots, o'_k)$. After each call, we check to see if the graph is acyclic. If the graph has a cycle, we have found a call that loops. □

**Theorem 3.11** (1) Inconsistency checking of method schemas is r.e.

(2) Consistency checking of method schemas is not r.e.

**Proof:** To prove (1), we have to enumerate the possible computations and check whether they yield inconsistencies. This is possible because of the finiteness of interpretations.

To prove (2), we claim that:

> Given a consistent method schema $S$ and a method $m$, we can construct a method schema $S_m$ such that all calls to method $m$ do not terminate in each interpretation of $S$ iff $S_m$ is consistent.

In order to prove this, we first define a new method $m'$ on the same domain as $m$:

$$method\ m'@\_ \ldots \_ = \lambda x_1, \ldots, x_j.f(m(x_1, \ldots, x_j))$$

where $f$ is a new method. There is no definition for $f$, i.e., this new method is undefined everywhere. $S_m$ will be the original schema $S$ augmented with this single coded method definition.

Since the method schema $S$ that we started out with is consistent, the only way to obtain an inconsistency in $S_m$ is that in a call to $m'$, $m$ terminates. Thus, the schema that we obtain is consistent iff all calls to $m$ do not terminate in each interpretation of $S$.

Now, (1) follows from the claim and Lemmas 3.7–3.9.□

**Remark**: We consider in this paper only finite interpretations of method schemas. Undecidability results hold for all (finite and infinite) interpretations by the same reduction and [26].

# 4 Recursion-Free Schemas

In this section, we study recursion-free schemas. We consider the monadic case first, and then turn to the general case. As we know from the previous section, consistency of monadic schemas can be tested in $O(nc^3)$ time. For simple, monadic schemas, we can check consistency in $O(nc^2)$ time. We will give an example which will illustrate the technique.

**Example 4.1** We want to check the consistency of the following schema. The classes in the schema are:

$$\begin{array}{lll} class & c & \\ class & c' & isa\ c \end{array}$$

The base methods are:

$$\begin{array}{llcll} method & m_1 & @c & : & c' \\ method & m_2 & @c & : & c \\ method & m_2 & @c' & : & c' \\ method & m_3 & @c' & : & c \end{array}$$

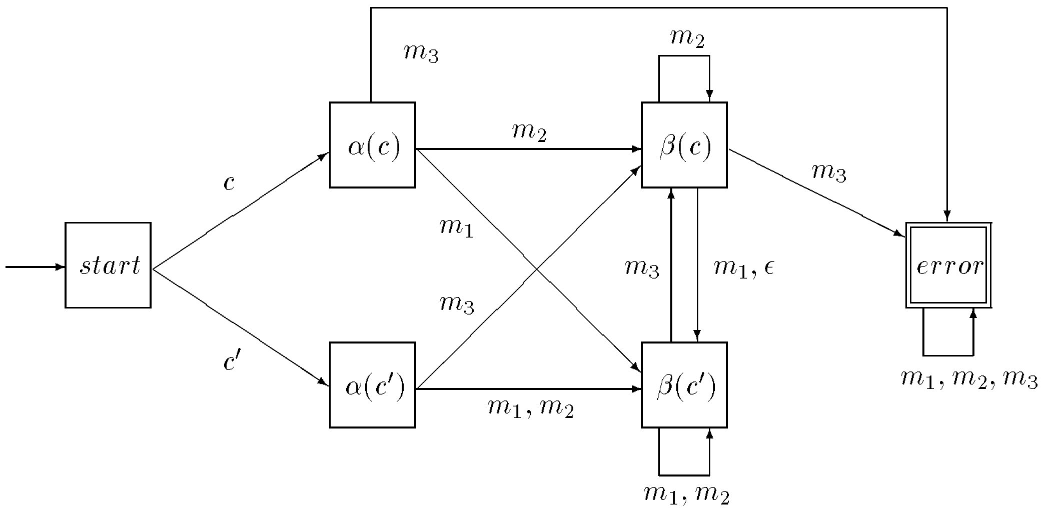



Figure 4: The FSA

The coded method definitions are:

$$\begin{array}{lccl} method & m & @c & = & \lambda x_1.m_1(m_2(m_1(x_1))) \\ method & m' & @c & = & \lambda x_1.m_3(m_3(m_1(x_1))) \end{array}$$

First, we will make use of a straightforward correspondence between monadic coded method definitions and strings of class and method names. We will represent the definition for $m$ from the example above as $cm_1m_2m_1$. More generally, we will say that $m_x$ has a definition $cm_{n_1}m_{n_2}\ldots m_{n_k}$ to mean that $m_x$ is defined as $m_x@c = \lambda x_1.m_{n_k}(m_{n_{k-1}}(\ldots(m_{n_1}(x_1))\ldots))$. We will make use of this alternate string representation of monadic coded methods frequently for notational convenience. Also, we will assume that all the coded method definitions are syntactically valid. That is, they use only valid method names. Obviously, this is very easy to check. This means that our string notation will only have strings of the form $cm_{n_1}m_{n_2}\ldots m_{n_k}$ (i.e., the first letter is a class name and the letters that follow this are all method names).

To check the consistency of the example schema, we transform the method schema into a nondeterministic FSA such that the strings not accepted by the FSA correspond to valid method definitions. (See Figure 4.) In the string representation, the first letter is always a class name and class names never occur again. This is the reason that the *start* state of the FSA has transitions labeled with each class in the hierarchy. That is, *start* has outgoing transitions labeled with $c$ and $c'$ leading to $\alpha(c)$ and $\alpha(c')$. After scanning for a class name, the FSA looks for method names. Outgoing transitions labeled with method names point to the class of the result. Therefore, the states $\beta(c)$ and $\beta(c')$ correspond to the classes. $m_1$ gives a result of class $c'$ when given an input of class $c$. Hence, there is a transition out of state $\alpha(c)$ to $\beta(c')$ labeled with $m_1$. The nonempty transitions to all but the final (i.e., *error*) state represent all possible method resolutions, that is, all possible cases of well defined base methods at classes. The nonempty transitions to the *final* state represent where base methods are undefined.

The reason for the empty transitions ($\epsilon$) is as follows. Consistency checking is essentially nondeterministic because of the semantics of signatures. A definition $(m_3, c' \to c)$ indicates that given an object in $c'$, $m_3$ returns an object in $c$ or in a subclass of $c$ (here, $c'$). Recall from Definition 2.5 that the semantics of $m_3$ at $c'$ is a total mapping from $c'$ into $\nu(c*)$, i.e., the output can be an object in $c$ *or* $c'$. The empty transitions allow descending the hierarchy from $c$ to $c'$.

Of the coded definitions, the first is consistent and the second is not. To verify that, it suffices to check that the words $cm_1m_2m_1$ and $c'm_1m_2m_1$ are rejected by the FSA of Figure 4 and that the word $cm_1m_3m_3$ is accepted. □

This example illustrates how we can test the consistency of simple, monadic schemas. Testing a single method is linear in the size of the method to be checked, but quadratic in the size of the class hierarchy; this is because we make explicit, in the automaton, a quadratic number of implicit resolutions. We generalize this technique to show that consistency of simple, monadic schemas can be tested in NLOGSPACE and $O(nc^2)$ time.

We first prove a lemma dealing with monadic method schemas with a single recursion-free (and thus simple) coded method definition. This is done by exhibiting a bridge between consistency checking and FSA computations, as follows:

**Definition 4.2** Let $S = (C, \leq, \Sigma_0, \{\theta\})$ be a monadic schema with $\theta$ a single coded method, and $M_0 = \{m | (m@c : c') \in \Sigma_0\}$. The *FSA associated with* $S$ is given by:

1. states: $C_\alpha \cup C_\beta \cup \{start, error\}$ where *start* is the initial state, *error* is the final state, and

$$C_\alpha = \{\alpha(c) \mid c \in C\}, \quad \text{and} \quad C_\beta = \{\beta(c) \mid c \in C\}$$

are two disjoint sets of new symbols,

2. alphabet: $M_0 \cup C$,

3. transitions[2]:

   - $start \overset{c}{\hookrightarrow} \alpha(c), \forall c \in C$,
   - $\beta(c) \overset{\epsilon}{\hookrightarrow} \beta(c'), \forall c' \leq c$ in $C$,
   - $\alpha(c) \overset{m}{\hookrightarrow} \beta(c')$, and $\beta(c) \overset{m}{\hookrightarrow} \beta(c')$ if the resolution of base method $m$ at $c$ is $m@c_1 : c'$,
   - $\alpha(c) \overset{m}{\hookrightarrow} error$, and $\beta(c) \overset{m}{\hookrightarrow} error$ if base method $m$ is undefined at $c$,
   - $error \overset{x}{\hookrightarrow} error, \forall x \in M_0$.

The language accepted by $A_S$ is denoted by $L(A_S)$. □

[2]To distinguish FSA transitions from reductions in method computations and grammatical productions, we use here $\hookrightarrow$.

**Lemma 4.3** Let $S = (C, \leq, \Sigma_0, \{m@c_0 = \lambda x.m_n(m_{n-1}(\ldots(m_2(m_1(x)))\ldots))\})$, $n \geq 1$, be a simple, monadic schema, and $A_S$ the FSA associated with $S$. Let $c$ be a class where $m$ is well defined. Then

1. $cm_1 \ldots m_n \in L(A_S)$ iff for some interpretation $I = (\nu, \mu)$ of $S$ and for some $o$ in $\nu(c)$, $m(o)$ is undefined.
2. $start \overset{cm_1 \ldots m_n}{\hookrightarrow} \beta(c')$ iff in some interpretation $I = (\nu, \mu)$ of $S$ and for some $o$ in $\nu(c)$, $m(o)$ reduces to some object $o'$ in $\nu(c')$.

**Proof:** To prove (1), suppose that $cm_1 m_2 \ldots m_n \in L(A_S)$. Let

$$start \overset{c}{\hookrightarrow} \alpha(c) \overset{m_1}{\hookrightarrow} \beta(c_1) \cdots \overset{m_q}{\hookrightarrow} \beta(c_q) \overset{m_{q+1}}{\hookrightarrow} error \cdots \overset{m_n}{\hookrightarrow} error$$

be an accepting computation. (Empty transitions are not shown.) Each step after the first involves reading a symbol, then possibly following $\epsilon$ transitions. Let the error state be first reached after reading $m_{q+1}$. (Notice that an error state cannot be reached using an $\epsilon$ transition.) An interpretation $I = (\nu, \mu)$ of $S$ is built as follows. Let $o$ be an object in $\nu(c)$, and for $j = 1, \ldots, q$, let $o_j$ be an object in $\nu(c_j)$. Let $(\mu(m_1))(o) = o_1$, and for $j = 2, \ldots, q$, let $(\mu(m_j))(o_{j-1}) = o_j$. (Remember that for $j = 1, \ldots, q$, $\mu(m_j)$ is a function.) This can be naively completed to be an interpretation of $S$.

The call $m(o)$ is legal because $m$ is well defined at $c$ and $o \in \nu(c)$. Clearly, $m(o) \overset{*}{\to} m_n(\ldots(m_{q+1}(o_q))\ldots)$. Since $\beta(c_q) \overset{m_{q+1}}{\hookrightarrow} error$, $m_{q+1}$ is undefined at $c_q$, $m_n(\ldots(m_{q+1}(o_q))\ldots) \to \perp$. Hence, $m(o)$ yields an inconsistency.

Suppose now that for some interpretation $I = (\nu, \mu)$ of $S$ and for some $o$ in $\nu(c)$, $m(o)$ is undefined, i.e., $m_n(\ldots(m_1(o))\ldots)$ reduces to $\perp$. Consider the classes $c_1, \ldots, c_j$ of the intermediate results before $\perp$ is obtained. Induction on $j$ provides the proof. As a basis, consider the class of the result of $m_1(o)$. We have two cases. First, $m_1$ has a base definition $m_1@c : c_1$. In this case, we have that $start \overset{c}{\hookrightarrow} \alpha(c) \overset{m_1}{\hookrightarrow} \beta(c_1)$. That is, $start \overset{cm_1}{\hookrightarrow} \beta(c_1)$. The other possibility is that $m_1$ has a base definition $m_1@c : c'$ and $c_1 \leq c'$ (the result is in a subclass of $c'$). In this case, $start \overset{cm_1}{\hookrightarrow} \beta(c')$ as before. We now can move from state $\beta(c')$ to state $\beta(c_1)$ following $\epsilon$ transitions. These transitions are guaranteed to be present as per Definition 4.2. Thus the basis is established. The induction step is equally easy and follows from the construction of the FSA. Therefore, we have that:

$$start \overset{c}{\hookrightarrow} \alpha(c) \overset{m_1}{\hookrightarrow} \beta(c_1) \cdots \overset{m_j}{\hookrightarrow} \beta(c_j) \overset{m_{j+1}}{\hookrightarrow} error \cdots \overset{m_n}{\hookrightarrow} error.$$

Thus $cm_1 \ldots m_n$ is accepted.

The proof of (2) is similar. The only difference is that we reach a state $\beta(c_k)$ instead of reaching the *error* state. The basis for the induction is when the coded definition consists of a single base method; the induction step follows from the way we have constructed the FSA. □

**Theorem 4.4** Consistency of simple, monadic schemas can be tested in $O(nc^2)$ time, where $n$ is the size of the method definitions in the schema and $c$ the size of its class hierarchy, and is logspace-complete in NLOGSPACE.

**Proof:** We first show that we can the check consistency of simple, monadic schemas in $O(nc^2)$ time. We consider the coded definitions one at a time. We build the automaton corresponding to the schema according to Definition 4.2. Building the automaton takes no more than $O(nc)$ time, as is clear from the definition. We now use Lemma 4.3. For each coded method $m$ such that the resolution of $m$ at $c$ is $m@c_1 = \lambda x.(m_l(\ldots(m_2(m_1(x)))\ldots))$, for $c \leq c_1$, we check to see that the string $cm_1m_2\ldots m_l$ is not accepted by the FSA. This consistency checking can be done by using the following simple algorithm:

1. $S := \{\alpha(c) \mid \text{resolution of } m \text{ at } c \text{ is } m@c_1 = \lambda x.(m_l(\ldots(m_2(m_1(x)))\ldots))\ \}$
2. for $i := 1$ to $l$ do
   (a) $Snew := \phi$
   (b) for each $x \in S$
       $Snew := Snew \bigcup \{\beta(c') \mid \beta(c')$ is reachable from $x$ using $m_i$ (and possibly some $\epsilon$ transitions) $\}$
   (c) $S := Snew$

The inner *for* loop in the algorithm above can take as much as $O(c^2)$ time, because the cardinality of $S$ can be as much as $c$ and $c$ states may be reachable from each of the states in $S$. Therefore, it is easy to see that this naive algorithm runs in $O(lc^2)$ time, where $l$ is the length of the coded method. Since we are considering simple schemas, we can check the consistency of the whole schema in $O(nc^2)$ time.

To show that the consistency problem is in NLOGSPACE, we use the algorithm above. We start by trying to prove that inconsistency is in NLOGSPACE. We will guess a particular path that the FSA will take to the *error* state, given $cm_1m_2\ldots m_l$ as input. We do not have to write down the complete FSA all at once. Given the base method definitions, we can compute parts of the transition function of the FSA as we need it. Thus, inconsistency is in NLOGSPACE. We now use the fact that NLOGSPACE is closed under complement [19, 30], and conclude that consistency is in NLOGSPACE.

Finally, we show that consistency of simple, monadic schemas is hard using a reduction of the reachability problem for graphs of out-degree 2 which is known to be logspace-complete in NLOGSPACE [20].

Let $G = (V, E)$ be a graph where each vertex has out-degree exactly 2; and for each $w$ in $V$, let $f(w)$ and $g(w)$ be the two successors of $w$. Let $v, v'$ be in $V$. The problem is to decide whether there is a path from $v$ to $v'$. We exhibit a schema $S = (C, \leq, \Sigma_0, \Sigma_1)$, such that there is a path from $v$ to $v'$ in $G$ if and only if $S$ is inconsistent.

The schema $S$ uses two base methods *choose*, *branch* and a coded method *test*, and is defined as follows:

1. for each $w \in V$, $C$ contains three classes $c_w, c'_w, c''_w$, with $c''_w \leq c'_w$,

2. for each $w$ except for $v'$, $choose@c_w : c'_w$ is in $\Sigma_0$,

3. for each $w$, $branch@c'_w : c_{f(w)}$ and $branch@c''_w : c_{g(w)}$ are in $\Sigma_0$,

4. Let $x(choose, branch)^n$ denote:
$$branch(choose(\ldots \text{ repeated } n \text{ times } \ldots(branch(choose(x)))\ldots)).$$
Let $\Sigma_1 = \{test@c_v = \lambda x.(x(choose, branch)^{n+1})\}$ where $n$ is the number of vertices in $V$.

First suppose that there is a path from $v$ to $v'$ in $G$. We may assume without loss of generality that this path is acyclic. Let $k$ be its length (so $k \leq n$). We next define an interpretation $I = (\nu, \mu)$ of $S$ that exhibits the inconsistency of $S$. For each $w$ on the path, let $o_w, o'_w, o''_w$ be objects, respectively in $\nu(c_w), \nu(c'_w), \nu(c''_w)$. The semantics of *branch* and *choose* are defined as follows. For each edge $(w, x)$ in the path,

1. if $x = f(w)$, $(\mu(choose))(o_w) = o'_w$; otherwise, $(\mu(choose))(o_w) = o''_w$,

2. $(\mu(branch))(o'_w) = o_{f(w)}$ and $(\mu(branch))(o''_w) = o_{g(w)}$.

This doesn't define the interpretation $I$ completely, but in any "completion" of the above, the call $test(o_v)$ (note that $test$ is well defined at $c_v$) reduces to $\perp$, because $(choose; branch)^k(o_v) \in \nu(c_{v'})$, where *choose* is undefined.

Conversely, if $S$ is inconsistent, it is straightforward to show that there is a path from $v$ to $v'$ in $G$.$\square$

**Remark:** The FSA technique can also be extended to check the consistency of monadic, recursion-free schemas in $O(nc^2)$ time. We do this by ordering the method definitions in the schema in a sequence $d_1, d_2, \ldots, d_q$ such that $d_i$ does not refer to the method defined in $d_j$ is $i < j$. This sequencing is possible because we are considering recursion-free schemas. We then solve for the consistency of the method schema incrementally, replacing coded methods with base methods having equivalent "signatures". One technical difficulty is that we need an extension of the notion of signature in order to capture the behaviour of methods that may return objects from different incomparable classes in the hierarchy. Another technical difficulty involves the special handling of identity methods. The development is fairly intuitive, but notationally cumbersome.

Next, we show that consistency of recursion-free monadic schemas is complete for PTIME under logspace reductions. This indicates that efficient parallel and/or dynamic algorithms for this problem may be hard to obtain.

**Theorem 4.5** Consistency of recursion-free, monadic schemas is logspace-complete in PTIME.

**Proof**: To show that the consistency of monadic schemas is logspace-complete in PTIME (P-complete), we reduce from the *circuit value problem* (CVP).

We define a *boolean circuit* to be a sequence $\overline{B} = (B_1, \ldots, B_n)$ where each $B_i$ is either a TRUE or FALSE or an expression $\text{NAND}(B_{i_1}, B_{i_2})$ where $i_1, i_2 < i$ and NAND stands for the usual boolean operator. Let $value(B_i) = B_i$ if $B_i$ is a truth value TRUE or FALSE, and if $B_i = \text{NAND}(B_{i_1}, B_{i_2})$, then let $value(B_i) = \text{NAND}(value(B_{i_1}), value(B_{i_2}))$. Let $value(\overline{B}) = value(B_n)$. The *circuit value problem* is: Given a boolean circuit $\overline{B}$, decide if $value(\overline{B}) = \text{TRUE}$. This problem is known to be P-complete [25].

Given an instance of the CVP, we define a monadic, recursion-free method schema $M$ that is consistent iff the boolean circuit evaluates to TRUE. $M$ has three classes, $c_t$, $c_f$, and $c_{nf}$ which are not related to each other in the inheritance hierarchy. $M$ has the following base method definitions.

$$\begin{array}{lll} nand@c_f : c_t & & nand@c_t : c_f \\ nand'@c_f : c_{nf} & & nand'@c_t : c_t \\ true@c : c_t & c \in \{c_t, c_f\} & true@c_{nf} : c_f \\ false@c : c_f & c \in \{c_t, c_f, c_{nf}\} & \end{array}$$

For each $B_i$ in the CVP, we have the following coded method definitions depending upon whether $B_i$ is TRUE, FALSE, or a term $\text{NAND}(B_{i_1}, B_{i_2})$.

- $B_i = \text{TRUE}$

$$\begin{aligned} b_i@c_t &= \lambda x.true(x) \\ b_i@c_f &= \lambda x.true(x) \\ b_i@c_{nf} &= \lambda x.false(x) \end{aligned}$$

- $B_i = \text{FALSE}$

$$\begin{aligned} b_i@c_t &= \lambda x.false(x) \\ b_i@c_f &= \lambda x.false(x) \\ b_i@c_{nf} &= \lambda x.false(x) \end{aligned}$$

- $B_i = \text{NAND}(B_{i_1}, B_{i_2})$

$$\begin{aligned} b_i@c_t &= \lambda x.nand(b_{i_1}(nand'(b_{i_2}(x)))) \\ b_i@c_f &= \lambda x.nand(b_{i_1}(nand'(b_{i_2}(x)))) \\ b_i@c_{nf} &= \lambda x.false(x) \end{aligned}$$

We shall now prove the theorem by making a series of claims about the schema that we have defined. Consider the class of the result that $b_i$ returns when called with an object of class $c_t$ or $c_f$. We claim that, for all $i$, $b_i$ returns objects of just one class ($c_t$ or $c_f$) regardless of whether the class of the input is $c_t$ or $c_f$.

This is proved by structural induction on the instance of the CVP. As a basis, observe that whenever $B_i$ is TRUE or FALSE, the corresponding $b_i$ returns an object from only one class — one of $c_t$ or $c_f$. When $B_i = nand(B_{i_1}, B_{i_2})$, we use induction and do a case analysis of the different cases possible, as shown below, to substantiate the claim. (In the case analysis, columns correspond to the classes of the results of method applications.)

| $b_{i_1}$ | $b_{i_2}$ | $nand'(b_{i_2})$ | $b_{i_1}(nand'(b_{i_2}))$ | $nand(b_{i_1}(nand'(b_{i_2})))$ |
|---|---|---|---|---|
| $c_f$ | $c_f$ | $c_{nf}$ | $c_f$ | $c_t$ |
| $c_f$ | $c_t$ | $c_t$ | $c_f$ | $c_t$ |
| $c_t$ | $c_f$ | $c_{nf}$ | $c_f$ | $c_t$ |
| $c_t$ | $c_t$ | $c_t$ | $c_t$ | $c_f$ |

The claim, combined with the fact that all $b_i$'s return objects of class $c_f$ at $c_{nf}$ implies, among other things, that *nand* and *nand'* will never be called with an object of class $c_{nf}$. Since all the other methods in the schema are defined at all the classes, the schema that we have defined so far is consistent.

We now claim that the class of the object returned by $b_i$, when called with an argument of type $c_t$ or $c_f$, corresponds to the logical value for $B_i$. The proof of this is a similar structural induction. Once again, observe that the claim holds directly when $B_i$ is TRUE of FALSE. We prove the induction hypothesis using the case analysis shown above.

We now introduce $M$ as the coded method that produces the inconsistency if the boolean circuit evaluates to FALSE. We define:

$$M@c_t = \lambda x.check(b_n(x))$$

where *check* is defined as a base function at $c_t$ and does not have a definition at $c_f$. It is easy to see that $M$ is inconsistent iff the CVP evaluates to FALSE.

Finally, notice that the reduction we use from the CVP to monadic, recursion-free schemas can be done in logarithmic space. The size of the resultant schema is linear in terms of the size of the CVP. Since the CVP is P-complete, it follows that consistency of monadic, recursion-free schemas is also P-complete. □

The situation is more complicated if methods with multiple arguments are considered. We will first consider simple schemas — schemas in which coded definitions do not reference other coded definitions — and recursion-free schemas of bounded arity. We will then briefly discuss recursion-free schemas of unbounded arity.

**Theorem 4.6** Consistency of recursion-free schemas with arities bounded by a constant is in coNP. Deciding whether a simple schema with a single coded method definition is consistent is coNP-complete, even if all arities are bounded by 2.

**Proof:** We first show that consistency of simple schemas is coNP-complete and then argue that the problem remains in coNP when we consider recursion-free schemas with arities bounded by a constant.

Inconsistency of simple schemas is clearly in NP. We guess an execution of a method call — this involves guessing the method, the classes of its arguments and the classs of intermediate results — and check that it corresponds to a possible execution and leads to an inconsistency. Thus consistency is in coNP.

To show the lower bound, we use a reduction of 3SAT [20] to the schema inconsistency problem. Consider a 3SAT formula $\Phi$ over variables $\{X_1, \ldots, X_k\}$. The corresponding schema $(C, \leq, \Sigma_0, \Sigma_1)$ is defined as follows:

1. $C = \{c_t, c_f, c_u\}$ (for true, false and unknown);

2. $c_t \leq c_u, c_f \leq c_u$;

3. $\Sigma_0$ has the following definitions:

$$\begin{array}{lll} not@c_t : c_f & not@c_f : c_t & not@c_u : c_u \\ and@c_u, c_u : c_u & or@c_u, c_u : c_u & \\ and@c_t, c_t : c_t & or@c_f, c_f : c_f & \\ and@c_f, c : c_f & or@c_t, c : c_t & c \in \{c_f, c_t, c_u\} \\ and@c, c_f : c_f & or@c, c_t : c_t & c \in \{c_f, c_t, c_u\} \\ pre@c_t : c_f & pre@c_f : c_t & pre@c_u : c_t \\ check@c_t : c_t & & \\ chooseX_i@c_u : c_u & 1 \leq i \leq k & \end{array}$$

   Note that *check* is the only method that is not total on its domain.

4. Let $\overline{\Phi}$ be a formula equivalent to $\Phi$ using only binary $\wedge$ and $\vee$. Then $\Sigma_1$ consists of a single method definition:

$$method\ M@c_u = \lambda x.\ check\ (pre(t_\Phi(x)))$$

   where $t_\Phi$ is the 1-term obtained from $\overline{\Phi}$ by replacing:

   (a) each $\wedge$, $\vee$, $\neg$, by *and*, *or* and *not* respectively.

   (b) each $X_i$ by the result of base method $chooseX_i$ applied to the single input argument. Note that $chooseX_i$ occurs only once in $t_\Phi$. Multiple occurrences of $X_i$ in $\Phi$ are all replaced by the result of the single occurrence of $chooseX_i$. (This is possible because of the *dag* structure of the 1-terms.)

We will first explain the intuition behind this reduction. A brute-force approach to solving the 3SAT problem would involve going through all possible assignments to the variables and checking to see if any of them satisfy the boolean formula in hand. The schema we have constructed provides a shorthand for doing this. Notice that we have adopted a straightforward translation of the boolean formula except for the presence of the $chooseX_i$'s. Using the *isa* hierarchy, the $chooseX_i$'s provide choice for each variable. In particular, $chooseX_i$ can return a result of class $c_u$, $c_t$, or $c_f$ (because $c_t \leq c_u$ and $c_f \leq c_u$), depending on the interpretation attached to it. Since consistency is quantified over all possible interpretations to the schema, we will go through all possible assignments to the variables.

We now claim that $\Phi$ is satisfiable iff the above schema is inconsistent.

Suppose that $\Phi$ is satisfiable. Let $v$ be a satisfying truth assignment for $\Phi$, i.e., a valuation of the variables such that $v\Phi$ holds. We construct an interpretation $I = (\nu, \mu)$ that will produce an inconsistency. First, $\nu(c_f) = o_f$, $\nu(c_t) = o_t$, and $\nu(c_u) = \phi$ where $o_f, o_t$ are two distinct objects. The semantics of *and, or, not, check* and *pre* are given by:

$$\begin{array}{ll} not(o_t) = o_f & not(o_f) = o_t \\ and(o_t, o_t) = o_t & or(o_f, o_f) = o_f \\ and(o_f, o_t) = o_f & or(o_t, o_f) = o_t \\ and(o_t, o_f) = o_f & or(o_f, o_t) = o_t \\ and(o_f, o_f) = o_f & or(o_t, o_t) = o_t \\ pre(o_t) = o_f & pre(o_f) = o_t \\ check(o_t) = o_t & \end{array}$$

Further, for $i = 1, \ldots, k$, we define $chooseX_i(o_t)$ to be $o_t$ if $vX_i$ is true. Otherwise it is $o_f$. Consider the call $M(o_t)$ ($M$ is well-defined at $c_t$). Since $v\Phi$ holds, it is easy to check that $I(t_\Phi(o_t))$ is $o_t$. Since $I(check(pre(o_t))) = I(check(o_f))$, $I(M(o_t))$ is $\perp$.

Conversely, suppose that the schema is inconsistent. One can show that inconsistencies can be obtained by having intermediate results in $\{c_t, c_f\}$ and the class $c_t$ for the term $t_\Phi$. This yields a satisfying truth assignment for $\Phi$.

Note that the schema that we constructed is simple, it has a single coded method definition and all arities are bounded by 2. Since the construction of the schema can clearly be done in PTIME, the hardness is proven.

We will now sketch the proof for the first part of the theorem. We are considering recursion-free schemas with arities bounded by a constant. Let us describe two calls to a method $m$ as being *c-different* if the classes of the arguments are different. The key observation in this case is that the number of c-different calls to a method is polynomial in the size of the input. It is exponential in the maximum arity in the schema, but we are considering arities bounded by a constant.

First, let us order the coded method definitions into a sequence $d_1, d_2, \ldots, d_n$ such that $d_j$ will not reference the method defined in $d_i$ if $j \leq i$. This sequencing is possible because we are considering recursion-free schemas. We guess the first definition in the sequence that will lead to an inconsistency. We guess an execution of this definition that will lead to the inconsistency. The definition we are considering might make calls to other coded methods that are defined before it. We have to guess executions for these coded methods verifying the classes of inputs/outputs that we have guessed for them. And so on. We can make all these guesses because we are guaranteed that the size of our guess is polynomial. (This follows from our prior observation that we cannot make too many c-different calls to any method.) It follows that inconsistency is in NP in this case. Therefore, consistency of recursion-free schemas with arities bounded by a constant is in coNP. □

**Remark:** Consistency of recursion-free schemas seems to be a much harder problem. There can be an exponential number of input/output possibilities for each coded method. A brute force approach can solve this problem in EXPSPACE, but finding a tight lower bound is an open problem.

**Remark:** If the $k$-terms of the methods to be checked are trees (not dag's), where each input $i$, for $1 \leq i \leq k$ appears at most once, consistency checking is in PTIME. This bound is obtained by generalizing the automaton technique to tree automata [11, 32].

# 5 Covariance

In this section, we consider covariance and show that it simplifies the consistency problem. We first consider simple schemas.

For simple schemas, covariance is the following constraint on the signatures of base methods. A simple schema is *covariant* if for each $m$ and for each pair

$$(m, (c_1, \ldots, c_k \rightarrow c)),$$

$$(m, (c'_1, \ldots, c'_k \rightarrow c'))$$

of definitions of a base method $m$, we have that $(\forall i \; c'_i \leq c_i) \Rightarrow c' \leq c$.

First consider the schema of Example 4.1 which is covariant. Suppose that we tested the prefix of a coded method and reached $\boxed{\beta(c)}$. We can either read a new method name or follow the $\epsilon$ edge to move to $\boxed{\beta(c')}$. Suppose that a word $\omega$ moves the automaton from $\boxed{\beta(c')}$ to $\boxed{error}$. Because of the covariance, one can show that the same word moves from $\boxed{\beta(c)}$ to $\boxed{error}$. Thus there is no need to follow the $\epsilon$ transition to detect the error. Therefore, in the covariant case, it is possible to remove the $\epsilon$ transitions of the automaton and still have the property that a definition is consistent iff the associated word is rejected by the automaton.

We can see from the example that the covariance condition on the base methods simplifies consistency checking. Indeed, the following result should be contrasted with Theorem 4.4 which says that consistency of simple monadic schemas without covariance is in NLOGSPACE.

**Theorem 5.1** Consistency of covariant, simple monadic schemas is in DLOGSPACE.

**Proof**: (sketch) We claim that we can use the following naive algorithm to check consistency of covariant, simple, monadic schemas.

For each coded definition $m@c_0 = \lambda x.(m_l(\ldots(m_2(m_1(x)))\ldots))$, we use the following algorithm to check consistency:

1. $c := c_0$
2. for $i := 1$ to $l$ do

   if the resolution of $m_i$ at $c$ is $m_i@c' : c''$, then $c := c''$

   else output *inconsistent*

Compare the algorithm above with the algorithm in the proof for Theorem 4.4. We use induction on the length of the coded method to prove that we don't have to descend into any subclass of $c$ while checking the consistency of covariant schemas. The basis with a coded definition of length 1 is immediate. Assume that the class of the result of $m_k(m_{k-1}(\ldots(m_2(m_1(c)))\ldots))$ is $c_k$. Suppose, in order to prove inconsistency, that we must descend into some class $c'_k$ such that $c'_k \leq c_k$. If $m_{k+1}$ has definitions at $c'_k$ and $c_k$, say $m@c'_k : c'_{k+1}$ and $m@c_k : c_{k+1}$, respectively, it follows from the covariance condition that $c'_{k+1} \leq c_{k+1}$ (and we could descend from $c_{k+1}$ to $c'_{k+1}$ instead of from $c_k$ to $c'_k$). This proves that we don't have to descend into any subclass when considering covariant schemas.

Since the algorithm outlined above can be executed in DLOGSPACE, the theorem follows. □

It turns out that under covariance, consistency of simple monadic schemas can be checked in linear time. This can be proved by showing that it can be checked using a deterministic two-way pushdown automaton (see Appendix A). We present a first principles argument here. Note that naive use of the *isa* can be quadratic.

**Algorithm 5.2**:

**Input:** Let the input be a covariant, simple, monadic schema where $n$ is the size of the method definitions and $c$ the size of the class hierarchy (represented as a forest). Let the schema have $p$ class names and $q$ method names. We can assume without loss of generality that each of the $p$ class names takes $\lg p$ bits to represent and each of the $q$ method names takes $\lg q$ bits to represent.

**Data Structure and Description:** We use a data structure that is two-tiered. Building this structure takes several steps. We will describe the easy steps verbally and use pseudo-code to describe the more difficult steps. In the first tier of the data structure, we have an array of the $p$ class names that occur in the input. We will refer to this first level array as the *class array*. Each element in the array has three slots. The first slot points to the *class' parent in the inheritance hierarchy*. The second slot points to the *next class in the breadth-first ordering of the inheritance hierarchy* (for this assume that the class hierarchy has been preprocessed in breadth-first order from parent to children). The third slot contains a pointer to a *radix tree of base method names*, which we shall refer to as the *method tree* for that class.

Each leaf in this method tree contains a pointer to the class of the result of that method name at that class. For example, given a base method definition of the form $m@c : c'$, we insert $m$ into $c$'s method tree. We then create a pointer to $c'$ from the leaf corresponding to $m$.

At the end of this insertion procedure, each class in the input hierarchy has a method tree associated with it. This method tree contains those base method names that have an explicit definition at that class (i.e., it does not include definitions that are inherited). Further, it is easy to see that this preprocessing step takes only linear time because insertion into a radix tree takes only linear time. It follows that the size of the data structure that we create is linear in the size of the input. The processing that we do following this creation step will not affect the size of the data structure since we only change some pointers.

We can assume, without loss of generality, that we do not have any redundant information, i.e., our hierarchy is a forest. For example, if we have that $c_1 \leq c_2$ and $c_2 \leq c_3$, it follows that $c_1 \leq c_3$ is redundant. We process these relationships and fill the parent and breadth-first-neighbour slots in the class array.

We now process the inheritance definitions. This part is the key component of the algorithm. We process the classes in the breadth-first order (bfs order) and for each class $c$ that has a parent $c'$ we issue a call *propagate(root_of_c'_method_tree, root_of_c_method_tree)*. (See Figure 6.) Each one of these propagate calls to the roots can call itself recursively on the nodes of the method trees of classes $c$ and $c'$. This processing is explained in the example and the lemmas which follow. □

**Example 5.3** Let us consider a schema with three classes $c_1, c_2$, and $c_3$. Let $c_1 \leq c_2$, and $c_2 \leq c_3$. Let the schema have the following base method definitions:

$$m_1@c_1 : c_2$$
$$m_2@c_2 : c_3$$
$$m_3@c_3 : c_1$$

We will represent $c_1, c_2$, and $c_3$ by array positions $0, 1$, and 2. and $m_1, m_2$, and $m_3$ by codes $00, 01$, and 10. The top half of Figure 5 shows how the data structure would look after the first preprocessing steps (before issuing the calls to *propagate*). The bottom half shows the data structure after the calls to *propagate* are completed. □

**Lemma 5.4** Let $S$ be a simple, monadic, covariant schema. Let the data structure corresponding to it be created as outlined in the algorithm above. Each class in the inheritance hierarchy inherits correctly — in its method tree — method definitions from all its ancestors.

**Proof:** We call procedure *propagate* for every parent-child pair in the inheritance hierarchy. We not only call the procedure *propagate* for every parent-child pair in the input inheritance hierarchy, but also make these calls in the breadth-first search order so that *propagate* is called with parent-child pairs sorted according to their depth in the inheritance hierarchy.

Now, the claim that a class inherits method definitions from all its ancestors can be shown by induction on the depth of the inheritance hierarchy. The basis consists of showing that a class at unit depth in the hierarchy inherits method definitions correctly from its parent, a root. If a child is to inherit any method definition, either the conditional in line (7) or the conditional in line (16) will have will have to evaluate to true. Lines (10) and (19) will perform (respectively) the addition of definitions into the child tree. Note that these statements add, in constant time, pointers to entire subtrees of defintions. Observe that *propagate* terminates when the parent subtree or the child subtree is not present. Therefore, we will visit every node in the child tree, whenever the corresponding parent node is not NIL. (If the parent node is NIL, there is nothing to inherit from that subtree of the method tree.) The induction hypothesis follows easily by a similar argument. A child at depth $d$ inherits definitions from all its ancestors correctly. Its child at depth $d+1$ inherits from all its ancestors including its parent by exactly the argument above. □

**Lemma 5.5** The procedure outlined in Algorithm 5.2 runs in linear time.

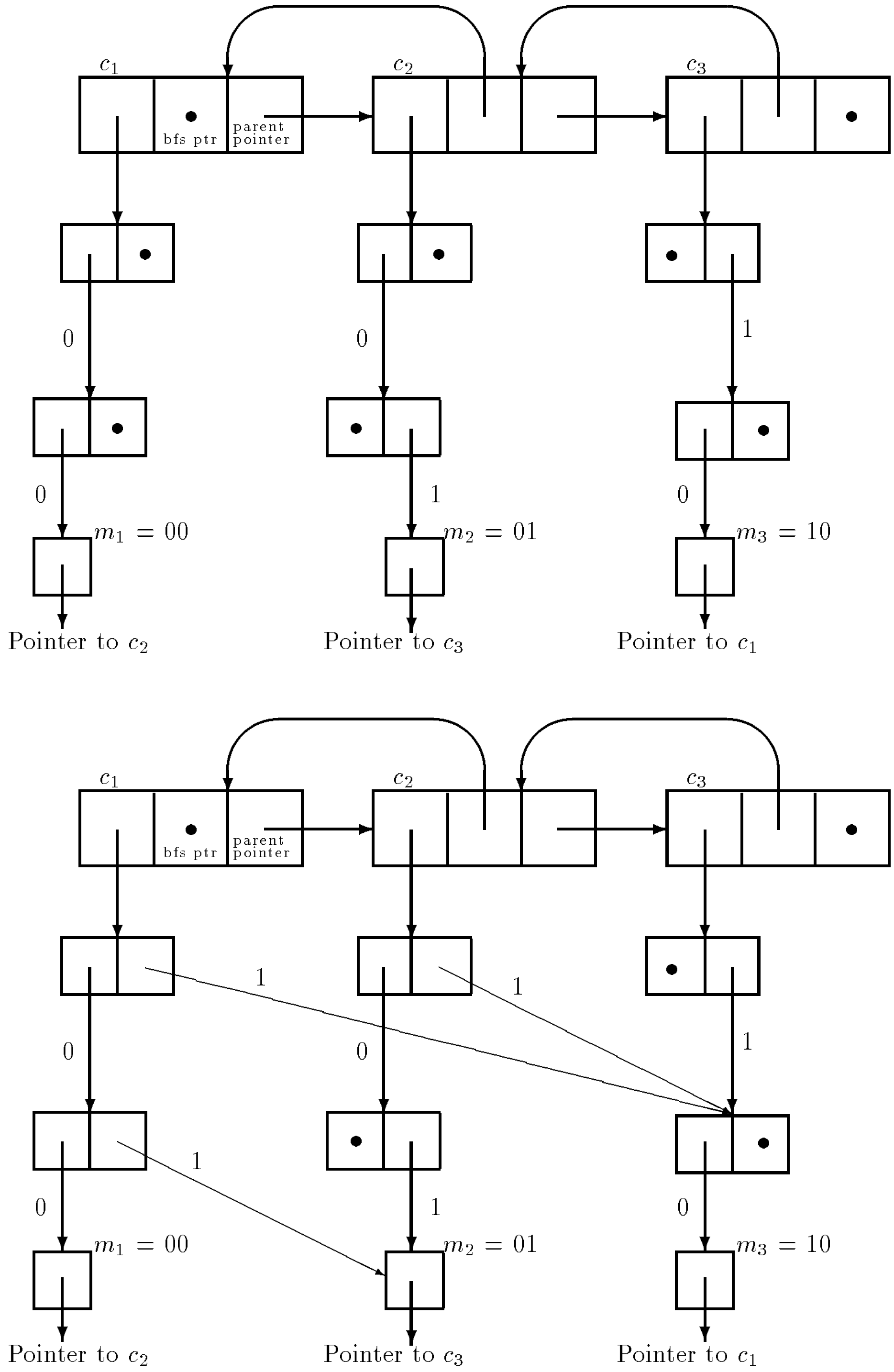



Figure 5: Example for the linear time algorithm (NIL pointer is $\bullet$).

```
procedure propagate (parent, child)
    if (child→leftChild ≠ NIL) and (parent→leftChild ≠ NIL)
    then
        /* Have definitions here. Check with child */
        propagate ( parent→leftChild, child→leftChild );
    else
        if (child→leftChild = NIL) and (parent→leftChild ≠ NIL)
        then
            /* Inherit definitions from parent */
            child→leftChild := parent→leftChild;
    if (child→rightChild ≠ NIL) and (parent→rightChild ≠ NIL)
    then
        /* Have definitions here. Check with child */
        propagate ( parent→rightChild, child→rightChild );
    else
        if (child→rightChild = NIL) and (parent→rightChild ≠ NIL)
        then
            /* Inherit definitions from parent */
            child→rightChild := parent →rightChild;
```

Figure 6: Preprocessing to inherit definitions.

**Proof:** Inserting explicit base method definitions into the data structure takes linear time, as we observed before. Also, preprocessing the hierarchy to create the parent and bfs neighbour pointers for each class name takes linear time. The only nontrivial part is proving that the code in Figure 6 takes only linear time.

As we observed before, we call the procedure *propagate* for every parent-child pair. If we can bound the amount of work done in each of these calls, we are done.

We claim that each of the above calls to *propagate* takes no more time than there are nodes in the method tree corresponding to the child. This can be seen by looking at the *if* statements in the code fragment, and observing that the procedure terminates as soon as we reach a leaf in the child tree or the parent tree. Since the number of nodes in the child tree is bounded by the number of methods that are defined explicitly in the class corresponding to the tree, we can charge each call that we make to *propagate* to the child tree. Therefore, the total time taken by these calls is bound by the size of the input (times some constant factor). $\Box$

**Theorem 5.6** Let $S$ be a simple, monadic, covariant method schemas where $n$ is the size of method definitions and $c$ the size of the class hierarchy. Let us build the data structure corresponding to $S$ according to Algorithm 5.2. With this data structure, we can check the consistency of $S$ in $O(n+c)$ time.

**Proof:** We have shown that every method tree in the data structure has its definitions as well as the definitions that it inherits. If we are given a coded method of the form

$$m@c = \lambda x.m_k(m_{k-1}(\ldots(m_2(m_1(x)))\ldots))$$

we start by checking that $m_1$ is defined at $c$. We do this checking by searching for $m_1$ in $c$'s method tree. We follow the pointer from $m_1$ and this takes us to the result. Because of covariance, we need not descend into any sub-class now (cf. the argument in Theorem 5.1). We then check $m_2$, go to the class of the result and so on. Since searching in a radix tree takes only linear time, this whole consistency checking can be done in linear time. If at any point, we discover that a method we are searching for doesn't exist in the corresponding method tree, we terminate because of the inconsistency. $\Box$

**Remark:** It is still open whether the previous result holds for monadic, covariant, recursion-free schemas. The previous algorithm cannot be extended naively. When we determine the class of a result of a method application, we have to keep track of all the subclasses of the result. This multiplies the cost of each step by a linear factor. There seems to be no simple way to avoid this problem.

Let us now consider simple polyadic schemas. We assume that arities are polyadic, but bounded by a constant (unbounded arities are unreasonable from a practical standpoint). In that case, a problem may arise if: (1) a method is called with arguments for which it is undefined, and (2) an ambiguity is encountered. These two kinds of problems can be easily detected. Indeed, we have a result that should be contrasted with Theorem 4.6.

**Theorem 5.7** Under the covariance restriction and with arities bounded by a constant, consistency of simple schemas can be tested in PTIME.

$$
\begin{array}{lcl}
isa(c,c') & \leftarrow & \text{(for each } c, c', \ c \leq c'\text{)} \\
def(m, c_1, \ldots, c_k) & \leftarrow & \text{(there is a definition of } m \text{ at } c_1, \ldots, c_k\text{)} \\
 & & \\
isa(c,c) & \leftarrow & \\
isa(c,c'') & \leftarrow & isa(c,c'), isa(c',c'') \\
less(\vec{x},\vec{y}) & \leftarrow & isa(x_1,y_1), \ldots, isa(x_k,y_k) \\
neq(\vec{x},\vec{y}) & \leftarrow & x_1 \neq y_1 \\
 & \ldots & \\
neq(\vec{x},\vec{y}) & \leftarrow & x_k \neq y_k \\
lessstrict(\vec{x},\vec{y}) & \leftarrow & less(\vec{x},\vec{y}), neq(\vec{x},\vec{y}) \\
possresolution(m,\vec{x},\vec{y}) & \leftarrow & less(\vec{x},\vec{y}), def(m,\vec{y}) \\
overwritten(m,\vec{x},\vec{y}) & \leftarrow & possresolution(m,\vec{x},\vec{z}), lessstrict(\vec{z},\vec{y}) \\
resolution(m,\vec{x},\vec{y}) & \leftarrow & possresolution(m,\vec{x},\vec{y}), \neg overwritten(m,\vec{x},\vec{y}) \\
ambiguous(m,\vec{x}) & \leftarrow & resolution(m,\vec{x},\vec{y}), resolution(m,\vec{x},\vec{z}), neq(\vec{y},\vec{z}).
\end{array}
$$

Figure 7: Datalog$^{\neg}$ program for covariant, fixed-arity schemas

**Proof:** We have to check that: (i) no ambiguity can arise, and (ii) no illegal call can arise.

(i) Checking for ambiguity. This has to be done for the base *and* coded methods. To simplify the presentation, we assume (without loss of generality) that all methods have arity $k$. Consider the stratified Datalog$^{\neg}$ program in Figure 7 (where $\vec{x} = x_1, \ldots, x_k$ and $\vec{y} = y_1, \ldots, y_k$).

One can show that $ambiguous(m, a_1, \ldots, a_k)$ is derived for some $m, a_1, \ldots, a_k$ iff a call to $m$ with arguments in $a_1, \ldots, a_k$ is ambiguous. The check for ambiguity can thus clearly be performed in PTIME.

(ii) Checking for illegal calls. Since the schema is simple and we are interested in PTIME computations, we can assume without loss of generality that there is a single coded method definition, say $(m, (b_1, \ldots, b_k), \alpha)$, where $\alpha$ is a $k$-term. Consistency is checked as follows. We first mark the leaves of $\alpha$ by the classes of the arguments. In a bottom-up traversal of the tree, we then mark all the internal leaves as follows:

> Let $v$ be a vertex labeled $m'$ with children respectively marked by $c_1, \ldots, c_k$. The marking of $v$ is given by:
>
> 1. If $resolution(m', c_1, \ldots, c_k, d_1, \ldots, d_k)$ has been derived by the program of Figure 7 for some $d_1, \ldots, d_k$, then the vector $d_1, \ldots, d_k$ is unique, and $(m', (d_1, \ldots d_k \rightarrow d))$ is in the schema for some unique $d$. Mark the vertex by $d$.
> 2. otherwise output *error*.

If the algorithm outputs *error*, it is easy to construct an interpretation that would lead to an inconsistency. Suppose now that the algorithm terminates without outputting *error*. Consider an arbitrary interpretation $(\nu, \mu)$ of the schema and a call $m(o_1, \ldots, o_k)$. Then $o_i$ is in $\nu(b_i*)$ for each $i$. Using covariance, one can show by induction than whenever a vertex of $k$-term $\alpha$ is reduced, it is reduced to an object in $\nu(d*)$ where $d$ is the marking of this vertex in (ii). Thus the call terminates successfully. $\square$

**Remark:** *A virtual class $c$* is a class that is required to always have an empty interpretation, i.e., $\nu(c) = \emptyset$. Virtual classes are useful for expressing the exact union of two classes. If virtual classes are considered, consistency of simple, monadic schemas is still in PTIME. However, with virtual classes, consistency of covariant, simple schemas can be shown to be coNP-hard. This indicates a subtlety of virtual classes.

**Remark:** To conclude this section, we consider schemas that are not simple. Here, the definition of covariance is obvious only for the base methods. Covariance does not help in the general (recursive) case, because the method schemas constructed to simulate program schemas in Section 3 are covariant with respect to the base methods. Another difficulty in this context is that, although covariance of the base methods can be tested easily, testing the "covariance" of the "inferred signatures" of coded methods is in general undecidable. This is because, the "inferred signatures" of the coded methods must be computed first. To see this, we point out that the method schema obtained in the simulation of program schemas is covariant. Thus the undecidability results also hold for covariant method schemas.

## 6 A Heuristic for Method Schema Consistency

In this section, we briefly consider practical issues. First, we consider a heuristic solution to the consistency problem in the general case. Then we consider the problem of checking incrementally for consistency.

In practice, it is common to provide both signature and code information for each method definition. In fact, while doing the consistency checking, we implicitly (if not explicitly) produce "inferred signatures" for the coded methods. By "inferred signatures" of coded methods, we mean a description of their input and output classes (similar to the ones that are provided for base methods). Let us assume now that extra signature information on coded methods is provided. This can be viewed as transforming the type inference problem that we considered so far into a type checking problem; and leads to the notion of a constrained schema.

**Constrained method schemas:** are pairs $(S,\ \tau)$, where $S = (C, \leq, \Sigma_0, \Sigma_1)$ is a method schema and $\tau$ a mapping from $\Sigma_1$ to classes. Intuitively, for each coded method definition, $\tau$ indicates the class of the return value. *Consistency* of constrained method schemas also requires the satisfaction of the constraints $\tau$.

We next provide a *sound but incomplete* consistency heuristic for constrained method schemas. Let $(S, \tau)$ be a constrained method schema. We construct a simple constrained method schema $S_\tau$ from $S$.

The schema $S_\tau$ is on the same classes as $S$ with the same hierarchy. Furthermore, each base method in $S$ is also a base method in $S_\tau$ with the same base definitions. For each coded method $m$ in $S$ that occurs in the code of another method, let $m_\tau$ be a new base method in $S_\tau$ and we construct its signatures as follows: for each $\alpha = (m, (c_1, ..., c_k), t)$ in $\Sigma_1$, $(m_\tau, (c_1, ..., c_k \rightarrow \tau(\alpha)))$

is a base definition of $S_\tau$. Finally, for each coded definition $(m, \sigma, t)$ in $S$, $S_\tau$ contains a coded definition $(m, \sigma, t')$ where $t'$ is obtained from $t$ by replacing each coded method name $m$ of $S$ by $m_\tau$. Clearly, $S_\tau$ is a simple schema. Now we have:

**Proposition 6.1** For each constrained method schema $(S, \tau)$, if $(S_\tau, \tau)$ is consistent, then $(S, \tau)$ is consistent. $\Box$

This leads to a heuristic for testing consistency of $S$ assuming certain constraints $\tau$. First construct the associated simple constrained schema and test it for consistency. To check whether $(S_\tau, \tau)$ is consistent, one would have also to consider the "inferred signatures" of coded methods and verify that they do not violate $\tau$. This can be done with the techniques outlined in the previous sections. Accept $(S, \tau)$ if $(S_\tau, \tau)$ is accepted.

The algorithm is sound; and obviously not complete. Furthermore, even in simple cases, it depends heavily on the "tightness" of the constraints specified by $\tau$ as illustrated by the following example.

**Example 6.2** Let $c$ be a class, and $c'$ be a subclass of $c$. Consider the schema $S$ and $S_\tau$ defined on $c' \leq c$ as follows:

$$
\begin{array}{llllll}
(S) & & & & & \\
method & m' & @c & : & c & \\
method & m' & @c' & : & c' & \\
method & m_0 & @c' & : & c' & \\
method & m & @c' & = & m' & : \; c' \\
method & m & @c & = & \lambda x_1.m(m'(x_1)) & : \; c \\
method & m_1 & @c & = & \lambda x_1.m_0(m(x_1)) & : \; c'.
\end{array}
$$

$$
\begin{array}{llllll}
(S_\tau) & & & & & \\
method & m' & @c & : & c & \\
method & m' & @c' & : & c' & \\
method & m_0 & @c' & : & c' & \\
method & m_\tau & @c' & : & c' & \\
method & m_\tau & @c & : & c & \\
method & m & @c' & = & m' & : \; c' \\
method & m & @c & = & \lambda x_1.m_\tau(m'(x_1)) & : \; c \\
method & m_1 & @c & = & \lambda x_1.m_0(m_\tau(x_1)) & : \; c'.
\end{array}
$$

The above algorithm will detect an inconsistency in the definition of $m_1$ in $S_\tau$. (An object in $\nu(c)$ may be mapped by $m_\tau$ to an object in $\nu(c)$ and $m_0$ is undefined on $\nu(c)$.) However, a more careful analysis of $S$ shows that the output of $m$ has to be an object in $\nu(c')$, otherwise $m$ loops forever. Therefore, no inconsistency can be reached with models of $S$. Indeed, the above algorithm would succeed if the second definition of $m$ is replaced by:

$method\ m@c = \lambda x_1.m(m'(x_1)) : c'$. $\Box$

**Unconstrained method schemas:** Suppose now that the schema is not constrained, or that only partial constraint information is available. Some pre-analysis of $S$ can suggest the choice of candidate constraints for the various methods. Furthermore, if the algorithm fails, one may attempt to refine the constraints based on the information gathered by this first phase. In this context, covariance may be useful to reduce the search space of candidate constraints.

Therefore, the proposed *heuristic for consistency* consists of two phases: (1) a signature pre-analysis phase for the unconstrained method schema and (2) a constrained method schema consistency test.

**Method Updates:** It is unreasonable, when the schema is updated, to recompile all the methods of the schema. Therefore there is a need for incremental algorithms. We first consider the insertion of a method. We present an example of an update that may yield problems, discuss it and mention some other related problems.

**Example 6.3** Let $c, c'$ be two classes with $c' \leq c$. Let the definitions be: $m'@c : c'$, $m''@c' : c'$, $m@c = \lambda x_1.m''(m'(x_1))$. The schema is consistent. Let us now insert a new base definition $m'@c' : c$. The schema is no longer consistent because of $m$, although we have apparently not modified $m$, not even have we touched the execution of $m$ on objects strictly in $c$. However, by inheritance, $m$ should also apply to $c'$-objects and this may now lead to an inconsistency. □

The problem is caused here by the compacted form of the information in a method schema (inheritance). It is clearly possible to make explicit such information, which would facilitate the detection of update effects. Such a technique obviously defeats space efficiency and code reusability.

One may think that the above problem would disappear if covariance is assumed. Indeed, under covariance, if a coded method definition is inserted, it is not necessary to recompile the schema. The code of the new method only has to be checked. However we can identify the following problems:

The "inferred signatures" of the previously inserted methods may be refined by this insertion. As a consequence, covariance may have been lost. So covariance would only be good for one insertion, and the consistency checking algorithm may have to recompile at least part of the schema if completeness is to be achieved in future insertions. (For detailed examples and some techniques for special cases see [33].)

Covariance does not help with deletions. To see this consider the last example.

**Example 6.4** Let $c, c', c'', int$ be four classes with both $c', c'' \leq c$. Let the definitions be: $m@c : c$, $m@c' : c''$, $m''@c'' : int$, $m'@c' = \lambda x_1.m''(m(x_1))$. The schema is consistent and covariance holds. Let us now delete the base definition $m@c' : c''$. The schema is no longer consistent even if $m$ is still well defined in $c$ and its subclasses.□

Finally, we would like to point out that some of the techniques of method schemas have been extended and applied to the $O_2$ object-oriented database system [36].

## 7 Conclusions and Open Problems

We have provided a basic programming model for object-oriented database systems, which captures classes, methods, inheritance, name overloading and late binding. We have investigated the consistency problem of method schemas as a form of "type inference" and have highlighted a number of efficiently decidable cases.

Our results do not depend on the exact mechanism of resolution of name overloading. In particular, our results hold with attachment to first argument. (That is, the first argument, called the receiver, of a method determines the resolution of name overloading.) This is the practice commonly adopted in many OODB formalisms. All our results for monadic schemas hold immediately in the new setting since there is no difference between the two ways of name resolution for monadic schemas. The undecidability result of Section 3 for general schemas also hold in the new setting, using essentially the same proof. The coNP-completeness result of Theorem 4.6 and the PTIME result of Theorem 5.7 also hold in the new setting, using similar proofs.

There are a variety of algorithmic questions that remain open. We do not know whether our radix-tree data structure can be extended to solve the consistency of monadic, covariant, recursion-free schemas. We also do not know whether there are efficient incremental algorithms for any of the consistency problems. Since we know that the problem of consistency checking for monadic, recursion-free schemas is P-complete, it would be interesting to know if there are good incremental algorithms for the simple, monadic or the covariant, monadic cases.

In the simulation of program schemas by method schemas, we use methods of arity at most $p + 1$ where $p$ is the number of locations used by the programs. It turns out that the results of [26] that we used also hold for program schemas with two locations only. Thus our undecidability results hold for schemas with methods of arity less than or equal to three. The complexity of method schema consistency with methods of arity two is not known.

Finally, in our formalization of method schemas, all base methods are uninterpreted. Let us suppose that we have equality, with its normal interpretation, as a base method. This method will simply tell if the two arguments that it is supplied with are the same. Consistency checking becomes undecidable with monadic coded methods and polyadic base methods when augmented with equality. With equality, we can simulate the Lisp *car* and *cdr* functions so that we can "glue" and "unglue" arguments. However, the problem of checking consistency with monadic coded methods and polyadic base methods without equality is open.

**Acknowledgements:** We thank Richard Hull for discussions on the topic and the reviewers for their comments.

## References

[1] S. Abiteboul, P. Kanellakis. Object Identity as a Query Language Primitive. *Proc. ACM SIGMOD*, 159–173, 1989. Also INRIA Tech. Rep. 1022, April 1989.

[2] E. Ashcroft, Z. Manna, A. Pnueli. Decidable Properties of Monadic Functional Schemas. *JACM*, 20:3:489–499, 1973.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik. The Object-Oriented Database System Manifesto. *Proc. First International Conference on Deductive and Object-Oriented Databases*, Japan, 1989.

[4] F. Bancilhon. Object-Oriented Database Systems. *Proc. 7th ACM PODS*, 152–162, 1988.

[5] F. Bancilhon, C. Delobel, P. Kanellakis (editors) *Building an Object-Oriented Database System: The Story of $O_2$.* Morgan Kaufman, 1992.

[6] J. Banerjee, et al., Data Model Issues for Object-Oriented Applications. *ACM TOIS*, 5:1:3–26, 1987.

[7] J. Banerjee, W. Kim, H-J. Kim, H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *Proc. ACM SIGMOD*, 311-322, 1987.

[8] A. Borgida. Type Systems for Querying Class Hierarchies with Non-strict Inheritance, *Proc. ACM PODS*, 394–400, 1989.

[9] S.A. Cook. Linear-time Simulation of Deterministic Two-way Pushdown Automata. *Proc. IFIP Congress*, 172–179, 1971.

[10] B. Courcelle. Recursive Applicative Program Schemes. *Handbook of Theoretical Computer Science*, Vol. B, Chapter 9, (J. van Leeuwen, A.R. Meyer, N. Nivat, M.S. Paterson, D. Perrin editors), North-Holland, 1990.

[11] J. E. Doner. Tree Acceptors and some of their Applications. *JCSS*, 4:406–451, 1971.

[12] D. Fishman et al. Iris: an Object-Oriented Database Management System. *ACM TOIS*, 5:1:46–69, 1987.

[13] A. Goldberg, D. Robson. *Smalltalk80: The Language and its Implementation.* Addison Wesley, 1983.

[14] S. A. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification.* LNCS vol. 36, Springer Verlag 1975.

[15] J. E. Hopcroft, J. D. Ullman *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[16] R. Hull, private communication, June 1989.

[17] R. Hull, J. Su. On Accessing Object-Oriented Databases: Expressive Power, Complexity, and Restrictions. *Proc. ACM SIGMOD*, 147–158, 1989.

[18] R. Hull, K. Tanaka, M. Yoshikawa. Behavior Analysis of Object-Oriented Databases: Method Structure, Execution Trees and Reachability. *Proc. 3rd Intl. Conf. on Foundations of Data Organization and Algorithms*, Paris, June 1989.

[19] N. Immerman. Nondeterministic Space is Closed Under Complement. *SIAM Journal of Computing*, 17 (1988) 935-938.

[20] D.S. Johnson. A Catalog of Complexity Classes. *Handbook of Theoretical Computer Science*, Vol. A, Chapter 2, (J. van Leeuwen, A.R. Meyer, N. Nivat, M.S. Paterson, D. Perrin editors), North-Holland, 1990.

[21] P.C. Kanellakis. Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. B, Chapter 17, (J. van Leeuwen, A.R. Meyer, N. Nivat, M.S. Paterson, D. Perrin editors), North-Holland, 1990.

[22] M. Kifer, W. Kim, Y. Sagiv. Querying Object-Oriented Databases. *Proc. ACM SIGMOD*, 393–402, 1992.

[23] W. Kim. Research Directions in Object-Oriented Database Systems. *Proc. 9th ACM PODS*, 1–15, 1990.

[24] W. Kim, F. Lochovsky. *Object-Oriented Concepts, Applications, and Databases.* Addison-Wesley, 1989.

[25] R. E. Ladner. The Circuit Value Problem is Logspace Complete for P. *SIGACT News 7*, p18-20.

[26] D. C. Luckham, D. M. R. Park, M. S. Paterson. On Formalized Computer Programs. *JCSS*, 4:220–249, 1970.

[27] D. Maier, A. Otis, A. Purdy. Development of an Object-Oriented DBMS. *Bulletin of IEEE on Database Engineering*, 1985.

[28] J. McCarthy. Towards a Mathematical Science of Computation. *Information Processing, Proceedings of IFIP Congress 1962*, 21–28, Ed. C. M. Popplewell, North-Holland Publishing Company, Amsterdam.

[29] A. Skarra, S. Zdonik. Type Evolution in an Object-Oriented Database. *Research Directions in Object-Oriented Programming*, 393–416, Eds. B. Shriver and P. Wegner, MIT Press, 1987.

[30] R. Szelepcsényi. The Method of Forcing for Nondeterministic Automata. *Bull. EATCS* **33**, 96–100, 1987.

[31] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, $2^{nd}$ Ed., 1982.

[32] M. Y. Vardi. Automata Theory for Database Theoreticians. *Proc. ACM PODS*, 83–91, 1989.

[33] E. Waller. Schema Updates and Consistency. *Proc. 2nd International Conference on Deductive and Object-oriented Databases*, Munich, 1991.

[34] S. Zdonik. Object Management Systems for Design Environments. *Bulletin of IEEE on Database Engineering*, 1985.

[35] S. Zdonik, D. Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1990.

[36] R. Zicari. A Framework for Schema Updates in an Object-Oriented Database System. *Building an Object-Oriented System: The Story of $O_2$.* Chapter 7, (F. Bancilhon, C. Delobel, P. Kanellakis editors), Morgan-Kaufmann, 1992.

# Appendix A Monadic, Simple, Covariant Schemas

We now give an alternate proof of Theorem 5.6. Let $S$ be a simple, monadic, covariant schema. We construct a two-way deterministic pushdown automaton that will accept $S$. We will sketch the construction of the automaton $2A_S$.

The input is a description of the schema. We assume that the classes are numbered $(c_1, \ldots, c_p)$ and, so are the methods $(m_1, \ldots, m_q)$. The input is

$$u_S = \$\$u_c\$\$u_b\$\$u_m\$\$$$

where $u_c, u_b, u_m$ represent respectively the class hierarchy, the base definitions and the code definitions as follows:

$$u_c = \alpha_1\$\ldots\alpha_l\$\ldots \qquad u_b = \beta_1\$\ldots\beta_l\$\ldots \qquad u_m = \delta_1\$\ldots\delta_l\$\ldots$$

where

1. $\alpha_l$ is of the form $i : i'$ (for $c_i \leq c_{i'}$);
2. $\beta_l$ is of the form $j@i : i'$ (for $m_j@c_i : c_{i'}$); and
3. $\delta_l$ is of the form $j@i : j_1; j_2; \ldots$ (for $m_j@c_i = \lambda x.m_{j_k}(m_{j_{k-1}}(\ldots(m_{j_2}(m_{j_1}(x)))\ldots)))$.

The coding of the schema is thus realized with the alphabet $\{\$, :, ;, @, 0, 1\}$.

The automaton works as follows. The $u_m$ part is first copied on the stack omitting the names of the coded methods. The stack then looks as follows:

$$i_1 : j_{1,1}; j_{1,2}; \ldots \$ i_2 : j_{2,1}; j_{2,2}; \ldots \$ \cdots \$\$$$

with $i_1$ on top. The automaton then tries to validate each code in turn until the stack is empty. A failure in the validation of one code yields a detection of inconsistency and the stopping of the automaton in a reject state. The completion of the validation brings the automaton to stop in an accepting state with an empty stack.

To validate the code "$i_1 : j_{1,1}; j_{1,2}; \ldots$", the automaton proceeds as follows. It first scans the tape to find a definition for $m_{j_{1,1}}$ at class $c_{i_1}$.

It does this by moving the input pointer to the beginning of the base method definitions. Let the input corresponding to the first definition be $j@i : i'$. It then moves the input pointer to the beginning of $i$ and tries to match the $i_1$ on top of the stack with the $i$. It does this by matching and popping symbol by symbol. If there is a mismatch, it backtracks and copies $i_1$ back onto the stack. If the classes match, it tries to match the method names by a similar procedure. Note that if the classes match and the method names don't, the automaton has to copy part of the method name as well as the class name back onto the stack. Proceeding in this way, the automaton can look through all the base method definitions in the search for $m_{j_{1,1}}@c_{i_1}$. Note that the automaton will use this kind of "match and on failure, re-copy" technique in other searches also.

Two cases can occur:

1. A base definition is found. Then $i_1 : j_{1,1}$ is replaced by the code of the class of the result followed by ":". The automaton doesn't have to consider any of the subclasses of the result because of covariance.
2. No definition is found. The automaton looks for the direct superclass of $i_1$. Two cases occur:
   (a) A class is found. Then $i_1$ is replaced by the code of the superclass.
   (b) None is found. The automaton stops in error.

When this is done, if no error is found, three cases may arise:

**completion:** the stacks looks like $i$ : \$\$, i.e., the checking is finished and successful. The automaton empties the stack and stops in an accepting state.

**partial completion:** the stacks looks like $i : \$i_2 : \ldots$, i.e., the current code has been checked successfully but there are more codes to check. Then $i$ : \$ is poped off the stack and the checking is resumed.

**otherwise:** the stacks looks like $i : j \ldots$ and the checking continues.

The language accepted by the above automaton is denoted $L(2A_S)$.

Now we have:

**Theorem A.1** Let $S = (C, \leq, \Sigma_0, \Sigma_1)$ be a covariant, simple, monadic schema, and $u_S$ its associated word according to the previous definition. Then $S$ is consistent if and only if $u_S \in L(2A_S)$. Thus, consistency of covariant, simple, monadic schemas can be tested in linear time.

**Proof:** (sketch) The proof relies on the same mechanism as the simple, monadic schemas technique with the FSA. The notable difference is that, because of covariance, one need not follow $\epsilon$-transitions.

First we prove that if $u_S \notin L(2A_S)$, then $S$ is inconsistent. Consider the coded definition $m@c = \lambda x.(m_e(\ldots(m_2(m_1(x)))\ldots))$ that causes the automaton to reject. Using the computation of the automaton with this definition as input, we can construct an interpretation in which a legal call to $m$ on some object in $c$ reduces to $\perp$. Let $c_1, c_2, \ldots, c_k$ be the classes of the intermediate results before the automaton rejects the input.

An interpretation $I = (\nu, \mu)$ of $S$ is built as follows. Let $o$ be an object in $\nu(c)$, and for $j = 1, \ldots, k$, let $o_j$ be an object in $\nu(c_j)$. Let $(\mu(m_1))(o) = o_1$, and for $j = 2, \ldots, k$, let $(\mu(m_j))(o_{j-1}) = o_j$. (Remember that for $j = 1, \ldots, k$, $\mu(m_j)$ is a function.) This can be naively completed to be an interpretation of $S$.

The call $m(o)$ is legal because $m$ is well defined at $c$ and $o \in \nu(c)$. Clearly, $m(o) \xrightarrow{*} m_e(\ldots(m_{k+1}(o_k))\ldots)$. Since the automaton rejects the input at this point, $m_{k+1}$ is undefined at $c_k$, and $m_e(\ldots(m_{k+1}(o_k))\ldots) \rightarrow \perp$. Hence, $m(o)$ yields an inconsistency.

Conversely, suppose that $u_S \in L(2A_S)$. Let $I = (\nu, \mu)$ be an intepretation for $S$ where $m$, a coded method name, is well-defined at $c$. Let $o \in \nu(c)$. Let the resolution of $m$ at $c$ be $m@c' = \lambda x.m_s(m_{s-1}(\ldots(m_2(m_1(x)))\ldots))$, where $c \leq c'$. Consider the computation for $m(o)$. For each $i$, let $o_i = m_i(\ldots(m_1(o))\ldots)$.

Now consider the validation of the string corresponding to the definition for $m$ at $c$ by the automaton. Let the automaton use classes $c, c_1, \ldots, c_s$ and definitions $m_1@c', \ldots m_s@c'_{s-1}$ where $c \leq c', c_1 \leq c'_1$, etc. in that validation. It easy to see that $m_1(o) \in \nu(c_1*)$. Using induction and the covariance condition, it is easy to show that for $i = 1, \ldots, s$, $o_i \in \nu(c_i*)$. Therefore $m(o)$ is defined and $S$ is consistent. $\square$