Relationships  Class definition allows the specification of (single) inheritance with hierarchical structures; associations (with multiplicity 1 − 1, 1 − N and M − N) to form networks (associations) of objects and aggregation relationships to represent whole/part structures [9]. Relationships between classes are established in the class definition when the structure of a database is created. Objects that are inserted subsequently into the database must obey such relationships, both in main and in secondary memory.

Methods  Methods must be defined using extended Horn clauses. They are declarative and can use any of the elements described in the query language (Section 4). Previous method definitions may be overridden by derived classes (polymorphism). In this first version of SABDOOD all methods are public.

Type Verification  Most of the variables have a specific type. The compiler infers as many types as possible. Other types are deduced at execution time. This implies consistent information handling within the database as well as in the rules that manipulate it.

3  Examples

In this section, we present several examples that illustrate the main features of our language.

3.1  Bill of Materials

The first example is Bill of Materials [4]. This example shows several properties of the language: inheritance, complex objects constructors, structured types, predefined predicates, recursive rules, and objects. In this example, the database represents the inventory of a manufacturing company. In particular, it represents the way parts are manufactured by assembling other subparts, the cost of manufacturing a part from its subparts, etc. Some database definitions in SABDOOD are as follows:

class part
  key: int;
  name: string;
  price: real;
endclass

Base parts have the set of their suppliers:

class base_part isa part  /* inheritance*/
  supplied_by: {supplier};
endclass

Composed parts have the assembly-time and the set of their components. Each component has a part number and the number of subparts needed.
class composite_part isa part
  time: real;
  components: {(part, int)}; /* set of tuples */
endclass

In accordance with [4], a simple query must be answered in a simple fashion. For instance, a rule in SABDOOD to answer the query: “find all the base parts that cost more than $100” is:

expensive(P) :-
  base_part(P),
  P.price > 100.

Rules in SABDOOD are extended Horn clauses written in a Prolog–like syntax. Variables begin with upper case letters. In addition, SABDOOD rules deal with objects (like P) and can access object attributes (like P.price). Note that in SABDOOD, as well as in others object database models, users can directly access objects attributes. For example, consider now the query: “find the total cost of a part”, where the total cost of a composed part is the sum of the cost of its components and an assembly cost. First, we define an auxiliary recursive rule to find the subparts of a (composed) part:

subpart(P1, P2) :-
  member(P1, P2.components).
subpart(P1, P2) :-
  member((P3, _), P2.components),
  subpart(P3, P1).

where member(X, Set) is a predefined predicate that checks if X is an element of Set. We require that the types of all head variables in a rule must be specified before its definition.

The transitive closure rule to find all the subparts of a part is:

components(P2, <P1>) :-
  subpart(P1, P2).

This rule uses multiset-grouping to create a multiset (<P1>) of pairs of the form (part, number), consisting of the subparts of a part. The rule to find the total cost of a (simple or composite) part is:

total_price(Part, R) :-
  components(Part, C),
  prices(C, P),
  sum(P, S),
  R = S + Part.price.

where sum(Set, S) is a predefined predicate that computes in S the sum of the elements of the multiset Set, and

prices(C, <X>) :-
  member((P, B), C),
  X = P.price * B.