Chapter 11

Deductive Rule Prototyping

This chapter has been contributed by Rainer Manthey (University of Bonn).

During the design of an application, we have collected a number of independently designed deductive rules and integrity constraints, each responding to a well-identified applicative need; we have been primarily concerned with the correctness of each rule independent from each other. During prototyping, we are instead only concerned about the mutual logical dependencies among rules; we investigate whether all rules of an application are coherent and meaningful and whether they “fit” with the remainder of the respective application schema.

Deductive rules are associated with very specific and elaborate forms of inferences performed during query and update management:

- Whenever a query addresses a derived concept, answers and intermediate results defined via deductive rules are automatically computed. We call this kind of inference query-driven.
- Whenever an update of some stored data occurs, a number of derived data may become derivable, and a number of formerly derived data may cease to be derivable. In particular, constraint violations can be seen as data which become derivable at a given point of time, as the effect of an inconsistent update transaction. We call this form update-driven inference.

In order to guarantee controlled and meaningful execution of such inference processes, all rules contained in an application schema must be well-formed according to a number of general criteria. In particular, deductive inference is expected to always terminate and to lead to identical results if activated repeatedly in identical situations.

We are particularly concerned with two collective properties of the rule set: stratification and satisfiability.

- A stratification specifies an order of evaluation for the rules in a rule set; a rule set is called stratifiable if it admits stratification. As we will see, certain rule sets with negated literals or set terms may not be stratified. When a rule set is not stratifiable, there may be ambiguities in the inference.
- A rule set, consisting of data derivation rules and integrity constraints, is satisfiable when it is possible to create at least one database state, consistent with schema definitions, which is valid, i.e. it satisfies all integrity constraints. If instead the rule is intrinsically contradictory, when we try to create a population all transactions are rejected. As we will see, the empty database is not consist as an interesting state for proving satisfiability, and we will look for more interesting cases.

However, there are quite a few aspects of rule design which go beyond these formal properties. A particular difficulty arises from the fact that in an object-oriented model, like Chimera, some implicit
logical dependencies between concepts depend on inheritance and may be further complicated by redefinitions.

In many cases, analyzing rule interdependencies statically at design time, independently from any actual data, can be sufficient for ensuring wellformedness of rules; the next section is devoted to static analysis of deductive rules, and in particular addresses the issue of stratification. For what concerns satisfiability, static analysis alone rarely helps in detecting shortcomings of a rule design. In these cases, prototyping by means of artificially generated test data provides the necessary insight; such techniques will be the topic of the following section on dynamic analysis of deductive rules.

11.1 Static Analysis

The basic step of all static analysis techniques for deductive rule sets consists in determining the logical interdependencies between stored and derived concepts and visualizing them by means of a dependency graph. Nodes of dependency graphs correspond to classes, attributes, constraints, and views; an arc from a node X to a node Y is introduced whenever X occurs in the body of a deductive rule implementing Y. As an example, consider the following set of rules defining various ways in which persons may be relatives of one another:

\[
\begin{align*}
  & X \text{ in } Y.\text{parents} \leftarrow \\
  & \quad \text{person}(X), \text{person}(Y), \\
  & \quad Y \text{ in } X.\text{children}; \\
  & X = Y.\text{mother} \leftarrow \\
  & \quad \text{mother}(X), \text{person}(Y), X \text{ in } Y.\text{parents}; \\
  & X \text{ in } Y.\text{sisters} \leftarrow \\
  & \quad \text{person}(X), \text{person}(Y), \\
  & \quad X \text{ in } Y.\text{parents}.\text{children}, \\
  & \quad X.\text{sex} = \text{'female'}, \\
  & \quad X \neq Y; \\
  \text{parent}(X) \leftarrow \\
  & \quad \text{person}(X), X.\text{children} \neq \emptyset; \\
  \text{mother}(X) \leftarrow \\
  & \quad \text{parent}(X), X.\text{sex} = \text{'female'}
\end{align*}
\]
Here *Sex* and *Children* are assumed to be stored attributes of the extensional (non-derived) class *Person*. *Parent* and *Mother* are derived subclasses of *Person*, whereas *Sisters* and *Parents* are derived, set-valued attributes of *Person*. The term *Mother* is also used as a derived, single-valued attribute of persons. Note that definitions of *Mother* and *Parents* both as subclasses and as derived attributes is legal in Chimera, although it introduces some degree of conceptual replication.

The corresponding dependency graph is in Figure 11.1; attributes are represented by ovals, classes are represented by boxes with rounded corners; extensional information has a gray shade. Note that the path expression `Y.parents.children` in the deductive rule defining *Sisters* is modeled by making this derived attribute dependent on both attributes *Person*, *Parents* and *Person*, *Children*. Note also that the inequality `X != Y`, which restricts bindings otherwise associated to variables `X` and `Y`, does not correspond to a dependency; similarly, predicates comparing attribute values with constants need not be represented.

An acyclic dependency graph (corresponding to a non-recursive rule set) indicates that any inference process involving the respective rules will terminate, regardless of the actual extensional data stored in the database. In such a schema there will always be a clear hierarchical way in which derived data can be computed, layer by layer. For instance, in Figure 11.1, we derive the attribute *Parents* and subclass *Parent* (actually by means of independent derivations), then subclass *Mother*, then the derived attribute *Mother*. Attribute *Sisters* depends on the derived attribute *Parents* and on extensional data, so it can derived immediately after the attribute *Parents* is derived. Thus, the dependency graph defines a partial order between derivations, also called a *stratification* of the derivations.

Another necessary prerequisite for meaningful derivations is that all derivation paths must be *well-founded*, i.e. must be based on extensionally stored data from which to start derivation. In other words, every derived concept in the graph has to be reachable by a directed path from at least one extensionally defined concept, represented in gray. This property can be easily checked by inspecting the dependency graph; for instance, in Figure 11.1 all concepts are well-founded, although the only extensional data concern the class *Person* and its attributes *Children* and *Sex*.

### 11.1.1 Recursion

Recursion might be the source of potential problems during inference, such as non-termination and order-dependence of the individual inference steps. As an example consider a different set of rules defining other relationships between relatives (already discussed in Chapter 9).

\[
X \text{ in } Y.\text{parents} \leftarrow \\
\quad \text{person}(X), \text{person}(Y), \ Y \text{ in } X.\text{children};
\]

\[
X \text{ in } Y.\text{siblings} \leftarrow \\
\quad \text{person}(X), \text{person}(Y), \ X.\text{parents} \text{ in } Y.\text{parents}, \ X \neq Y;
\]

\[
X \text{ in } Y.\text{ancestors} \leftarrow \\
\quad \text{person}(X), \text{person}(Y), \ X \text{ in } Y.\text{parents};
\]

\[
X \text{ in } Y.\text{ancestors} \leftarrow \\
\quad \text{person}(X), \text{person}(Y), \ X \text{ in } Y.\text{ancestors}.\text{parents};
\]

\[
X \text{ in } Y.\text{cousins} \leftarrow \\
\quad \text{person}(X), \text{person}(Y), \ X \neq Y, \ X \text{ in } Y.\text{parents}.\text{siblings}.\text{children};
\]

\[
X \text{ in } Y.\text{cousins} \leftarrow \\
\quad \text{person}(X), \text{person}(Y), \ X \neq Y, \ X \text{ in } Y.\text{parents}.\text{cousins}.\text{children};
\]
The corresponding dependency graph, shown in Figure 11.2, exhibits that all derived attributes (including the recursive ones) are indeed properly founded on the class Person and its stored attribute Children. Thus, each recursively defined concept is well-founded.

Plain recursion with terms that positively depend on other terms, as the one introduced by the above formulas, introduces cycles in the dependency graph, but it cannot introduce problems due to nontermination. However, such problems may occur with negation and uncontrolled generation of terms, as introduced by aggregates, arithmetics, and set constructions. We postpone the treatment of negation until Section 11.1.3.

11.1.2 Inheritance and Overriding

Object-oriented data models are much more expressive than the relational model. This means concretely that a lot of implicit semantic assumptions are built into the model, thus relieving schema designers from explicitly stating them in each application schema. Many of these assumptions could have been made explicit by means of additional deductive rules and constraints. Consequently, such implicit meaning has to be taken into account when analyzing interdependencies between derived concepts. These “hidden” rules should in particular be taken into consideration when constructing a dependency graph for a given schema.

First of all, each rule targeted to a particular class contains at least one implicit dependency on the class to which it is targeted, which in Chimera is not mentioned in the rule itself. The following implementation definition for an object class Employee illustrates this issue:

```plaintext
define implementation for employee
attributes
    Self.incomeTax = Y <--
        real(Y), Y = 0.3 * Self.grossSalary;
end;
```

Taken out of its context, the rule must be augmented by a class formula for the variable Self, which introduces into the dependency graph a node relative to the target. By renaming Self into a generic variable X, we obtain:

```plaintext
X.incomeTax = Y <--
```

1The predicate X \( \leftarrow \) Y, present in deductive rules for Siblings and Cousins, introduces a restriction of bindings and should not be modeled in the graph or interpreted as a negation.
11.1. STATIC ANALYSIS

employee(X), real(Y), Y = 0.3 * X.grossSalary;

Furthermore, each subclass relationship introduces logical dependencies which could be made explicit by means of rules. If Engineer is a subclass of Employee, the implicit assumption that all instances of a subclass are to be considered instances of the superclass leads to the population rule:

employee(X) ← engineer(X).

Note that adding this rule in explicit form is not possible for two reasons. First, Chimera restricts the use of population rules to the specialization of subclasses via restricting conditions on their superclasses, rather than for generalizing. Thus, in Chimera we could instead explicitly state the following rule for a derived class:

engineer(X) ← employee(X), X.profession='engineer'.

Second, the implicit deduction rule induced above represents just one aspect of the population of Employee. There may be employees who do not belong to the subclass Engineer, for instance because they are extensionally defined and entered in the Employee class.

Overriding introduces another implicit set of dependencies, which are more serious as they come along with the introduction of negation and thus may have an influence on the stratifiability of the schema. In the context of deductive rules, overriding means reimplementation of a derived attribute in a subclass. Assume that income tax rates for employees are:

X.incomeTax = Y ← employee(X), real(Y), Y = 0.3 * Self.grossSalary;

If we further assume that income taxes for engineers are higher than for ordinary employees, the respective attribute would have to be redefined in the subclass:

redefine implementation for engineer
attributes
    Self.incomeTax = Y ←
    real(Y), Y = 0.4 * Self.grossSalary;
end;

In this case, the rule implementing the attribute for the superclass Employee is assumed not to be applicable for those employees that are engineers as well. This implicit assumption must be made explicit during static rule analysis, by augmenting the respective rule as follows:

X.incomeTax = Y ←
    employee(X), not engineer(X),
    real(Y), Y = 0.3 * Self.grossSalary;
X.incomeTax = Y ←
    engineer(X),
    real(Y), Y = 0.4 * Self.grossSalary;

Thus, an additional arc has to be introduced in the dependency graph connecting nodes Engineer and Employee.incomeTax and labelled by a negation sign. Such "hidden" negations due to overriding may become important during stratifiability analysis.

Finally, the fixed-format constraints contained in a schema may represent implicit dependencies as well which have to be considered when constructing the dependency graph of an object-oriented schema with deductive rules. The way how fixed-format constraints can be alternatively expressed by means of generic, targeted constraints has already been explained and illustrated in Chapter 9 and is not be repeated here.
11.1.3 Negation and Stratification

As soon as negation occurs in deductive rules, the order in which inference steps are performed may become very critical. The intuitive reason for this is that in order to conclude that some fact is not true in the database, sufficient information about those facts that indeed are true has to be at hand. If negation applies to base data, it is enough to check whether the respective fact is actually stored or not: If it is stored, the negative formula is wrong, if it is not stored, the negative formula is true. This simple operational principle of implementing negation has been called **negation as failure** in the literature: Failure to find the fact in the database leads to the conclusion of its falsity.

If negation applies to a derived concept, however, the DBMS has first to generate the positive data, against which to perform the negation-as-failure test. As an example, consider two rules defining derived subclasses of Employee:

\[
\text{engineer}(X) \leftarrow \\
\text{employee}(X), X.\text{grade} = \text{'eng'}; \\
\text{technician}(X) \leftarrow \\
\text{employee}(X), \text{not } \text{engineer}(X)
\]

Intuitively, if we want to determine who is a Technician, we first have to know all Engineers, in order to evaluate the negative formula over the full extension of the class Engineer. Thus all derivations involving the Engineer rule have to be performed before any derivation involving the Technician rule takes place. This precedence is traced on the dependency graph in figure 11.3 by labeling the arc connecting an Engineer to a Technician with a negation.

More in general, a **negative dependency** exists between two concepts A and B whenever a rule for A contains a negative subformula containing B. When organizing inference processes correctly, the DBMS has to automatically determine a proper hierarchical order in which to perform the inference, such that the derivation of concept A comes later in the ordering than the derivation for a concept B. Such ordering is called a **stratification**.

During static analysis the designer has to make sure that a proper ordering of derivations is indeed possible. The only case in which there is no way of correctly organizing inference processes is if some derived concept at the same time recursively and negatively depends on itself. Such situations logically correspond to paradox definitions of the form:

\[
\text{engineer}(X) \leftarrow \text{not } \text{engineer}(X)
\]

Of course, no designer would write a rule like this! But recursive dependencies may be indirect ones and the corresponding cycles may be rather long. Negative dependencies in such cycles might be "hidden" ones as sketched above, so that it might happen "by accident" that negative cycles
are introduced, simply because the logical structure of part of a schema becomes too complex and designers lose control. A more realistic example of definition which, although still unacceptable, could be inadvertently produced is the following:

\[
\text{engineer}(X) \leftarrow \\
\text{employee}(X), \text{not technician}(X)
\]

\[
\text{technician}(X) \leftarrow \\
\text{employee}(X), \text{not engineer}(X)
\]

The dependency graph for the above two rules is shown in Figure 11.4; note that \textit{Engineer} and \textit{Technician} recursively and negatively depend on each other.

Stratifiability checking amounts to partitioning the nodes in the dependency graph of the respective application schema into layers (strata) in such a way that the evaluation order required for proper treatment of negation is respected:

1. Nodes in a stratum may not depend on any nodes in higher strata; in case of recursion, they may depend on nodes of the same stratum.

2. Negative dependencies must always lead from a lower to a higher stratum.

Such a way of layering a dependency graph can always be reached, unless the graph contains a \textit{cycle involving at least one negative dependency}. Thus, testing for stratification reduces in practice to identifying such undesired situation; the testing is very easy, once that all dependencies are explicitly shown by a dependency graph.

In Figure 11.3, a stratification simply consists in evaluating \textit{Engineer} first, and \textit{Technician} second; a stratification for Figure 11.4 cannot be found.

Checking for stratifiability could be done fully automatically by means of a suitable tool. However, if the tool turns out to be unable to stratify a given application, it is necessary that the designer breaks any illegal cycle of recursion involving negation or set-terms by reformulating certain rules. Unfortunately, there is no general method for “repairing” stratification violations. What has to be done depends heavily on the context of the application and on the specific semantics of the concepts.
in such a forbidden cycle. Carefully analyzing the reason for the paradox represented by such a cycle will normally lead to a clue about a way how to break it. Fortunately, it is rather unlikely that a larger part of an application schema will ever be linked together by means of dependencies. Normally, one should expect that just comparatively small, local areas within an overall schema graph will be interrelated by explicit or implicit rules. Displaying and checking such logically connected components should be feasible in most cases.

There are cases of meaningful rule sets which are not stratifiable. Such cases are rare and often exotic, but a considerable amount of research has already been devoted to the investigation of classes of such rules which can still be handled by advanced evaluation methods. Such methods, however, may suffer of severe efficiency problems and are therefore not supported by Chimera and by most deductive DBMS today.

11.2 Dynamic Analysis

When speaking about dynamic analysis of deductive rules, we have in mind the process of testing such rules on test data. Such data don't have to reflect reality, but are purely aimed at representing prototypical examples which may arise in an analogous manner and on larger scale once the database has been filled with "real" data. Although it might appear not very meaningful to play around with small example databases containing just invented objects rather than real ones, we indeed recommend to make use of this strategy during rule prototyping.

The most important problem which can be handled by means of a systematic approach to generate consistent test databases is the satisfiability problem, addressed in more detail in the next two subsections. Satisfiability is a well-known property of arbitrary systems of "axioms" (such as rules and constraints) which has been formally investigated by logicians for many decades. But there may be other properties of schemas which are neither of general nature, nor correspond to any abstract concept of logic, but purely depend on the application at hand. Deductively defined concepts are inherently difficult to understand as they are formulated on an abstract, general level. It is therefore useful to get a concrete feeling for the implications of such high-level definitions. Analysis by example is likely to be a good candidate for gaining experience with the "behaviour" of rules and constraints in concrete, though exemplary situations.

When designing the static part of a Chimera application containing deductive rules and constraints, it is therefore crucial to make sure that the "axioms" designed are satisfiable before running into trouble afterwards. As satisfiability checking may be a rather tedious job (the problem as such is even undecidable, as we will see), it is best performed during passive rule prototyping. In this section, we will provide a short introduction to the satisfiability problem, which has been quite seriously neglected in database research in the past, but becomes relevant as soon as more powerful data models and languages like Chimera are available. We will also briefly outline how satisfiability checking can be done, either by an automated tool for rule prototyping or by hand.

11.2.1 Satisfiability of Integrity Constraints

At the end of each transaction a deductive DBMS checks whether the resulting state of the database will be consistent in the sense that all stored and derived data satisfy each of the integrity constraints in the underlying schema. Consistency is a well-defined notion in formal logic, where it means that a set of logical axioms is free of contradictions, or equivalently, that the axioms have at least one model. If looking at databases from the perspective of logic, the axioms are the constraints and the deductive rules contained in a schema. Each database state - composed of explicitly created as well as derived objects and attribute values - can be regarded as a potential model of these axioms. Consistent databases - in the sense of integrity checking - are models of rules and constraints. Thus, integrity checking has a firm logical basis, and using the term consistency is well-founded in terms of predicate logic.

During conceptual design, however, it may rather easily happen that, due to typical design errors
(such as forgotten exceptions or border cases), the rules and constraints designed turn out to be logically inconsistent. As there are no data at hand during schema design, there is no immediate way of detecting such inherent inconsistencies. Only when all attempts to populate the database fail due to integrity violations the designer will become aware of potential errors. A schema which does not admit any valid database state is also called unsatisfiable, as there is no way of satisfying the given constraints. In formal logic, satisfiability is used as a synonym for consistency.

Logical notions and properties, like inconsistency and unsatisfiability, are regarded with quite a degree of suspicion by many practitioners. On the one hand, such concepts are hard to understand, mainly because of the unfamiliar theoretical framework required for defining them. On the other hand, the relevance of such ideas for practical life are doubted. In the case of unsatisfiability of rules and constraints in a Chimera schema, there will probably be no doubt that unsatisfiable schemas are a problem, but the likeliness of their appearance in real life will be questioned by many. Therefore we are going to illustrate the unsatisfiability problem by means of a very simple example. Of course, this example is a constructed one, but it is not difficult at all to abstract from the particularities of this example and to understand that similar problems may arise quite easily in other scenarios and in particular in much more complex real-life schemas.

For the purpose of this brief introduction to the satisfiability problem consider the following Chimera schema, modeling part of an enterprise:

```plaintext
define object class employee
attributes
  name: string,
  dept: department, notnull,
  boss: employee, derived
constraints
  self-controlling
end;
define object class department
attributes
  name: string,
  leader: employee, notnull
end;
```

The implementation part of this schema defines the derived attribute and the generic targeted constraint of Employee, as well as an additional generic untargeted constraint relating departments and their leaders.

```plaintext
define implementation for employee
attributes
  Self.boss= X  <-- employee(X), Self.dept.leader=X
end;
define hard constraint self_controlling for class employee
as
  self-controlling(Self)  <--  Self = Self.boss
end;
define hard constraint foreign_leader(mgr: X, dpt:Y)
as
  foreign_leader(X,Y)  <--
    department(Y), Y.leader=X, not X.dept=Y
end;
```

Paraphrased in natural language, these constraints (including the fixed-format ones) and rules in this schema correspond to the following conditions:

- Every employee is a member of at least one department. (This is the meaning of the notnull constraint on Dept values in the definition of class Employee).
- Every department has a leader, who is an employee. (This is the notnull constraint in the definition of class Department).
• The boss of an employee is the leader of the employee's department. (This is the meaning of the rule defining the Boss attribute for employees.)

• Nobody is his own boss. (The targeted constraint for Employee.)

• The leader of a department always has to be a member of the department he leads. (The untargeted constraint relating classes Employee and Department.)

Each of these conditions taken as such sounds perfectly ok and seems to correspond to meaningful and sound business rules in an imagined company. Translated into first-order logic axioms, these conditions would probably look as follows:

1. \( \forall X, Y : leader(X, Y) \Rightarrow member(X, Y) \)
2. \( \forall X, Y, Z : member(X, Z) \land leader(Y, Z) \Rightarrow boss(X, Y) \)
3. \( \forall X : employee(X) \Rightarrow (\exists Y : department(Y) \land member(X, Y)) \)
4. \( \forall X : department(X) \Rightarrow (\exists Y : employee(Y) \land leader(Y, X)) \)
5. \( \neg \exists X : boss(X, X) \)

If we gave these logical formulas to an automated theorem prover (or, if such a device is unavailable, to an expert in predicate logic able to check formulas by hand), such a prover would also accept these formulas as perfectly ok and satisfiable.

Despite such "formal approval" by a logic checker, databases corresponding to this schema cannot be populated without integrity violations. The design mistake — which even escapes a theorem prover — can be quite easily discovered, if we try to create a minimal, meaningful example database corresponding to the schema, i.e., containing data for all classes mentioned and in addition satisfying all constraints. In such an example database there were at least one department and at least one employee. An initial Chimera transaction for creating these two objects could look as follows:

begin-transaction

create(department, (name: fun, leader: Y), X),
create(employee, (name: adam, dept: X), Y);

commit

Assume OIDs generated for the two objects created are @1 and @2. The attempt of creating an initial database will fail due to a violation of the integrity constraint targeted to Employee: we are able to derive

Self-controlling(@2),

which means that Adam is his own boss. This happens because the rule for deriving Boss values assigns every employee to the leader of the department which occurs as his Dept value. There is no way out of this situation! We cannot assign Adam (alias @2) to a different department, because of the untargeted constraint requiring department leaders to be members of the department(s) they lead. Even introducing a third employee - relieving Adam from the burden of leading the fun department - doesn't help, as now the same trouble arises for the new department leader: he leads @1, therefore - due to the second constraint - has to be member of @1, therefore - due to the rule deriving Boss values - has to be his own boss, which violates the first constraint. The example schema is unsatisfiable, unless the database remains completely empty.

Indeed, the empty state is the only consistent one in this case. This is the reason why the theorem prover didn't "complain": in terms of formal logic empty models are acceptable (unless the set of axioms explicitly contains an existence requirement precluding empty models). By adding, for example, the axiom that there must be at least one department in every valid state of the database
6) \( \exists X : \text{department}(X) \)

the inconsistency of the example axioms would even be detected by the theorem prover. However, a schema designer would probably never add such constraints in practice!

Intuitively, the reason why the schema “doesn’t work” is that department leaders have to be treated in a different way than ordinary employees as far as the subordination rules in our company are concerned. We have to introduce an exception into the constraint prohibiting “self-control”, which requires an additional subclass Manager to be introduced. The following modified schema will turn out to be satisfiable:

```plaintext
define object class employee
attribute    name: string,
             dept: department, notnull,
             boss: manager, derived
end;

define object class manager
superclasses  employee
attributes    depts: set-of(department)
end;

define object class department
attributes    name: string,
             leader: manager, notnull
end

define implementation for employee
attributes    Self.boss= X \leftarrow \rightarrow manager(X), Self.dept.leader=X
end;

define hard constraint self_controlling for class employee
as    self_controlling(Self) \leftarrow \rightarrow not manager(Self), Self = Self.boss
end;

define hard constraint foreign_leader(mgr: X, dpt: Y)
as    foreign_leader(X,Y) \leftarrow \rightarrow department(Y), Y.leader=X, not X.dept=Y
end;
```

Now the above transaction would succeed, leading to a minimal consistent database state consisting of one department and one employee, who leads the department and is its only member. In terms of logic, such a database state is a minimal, non-empty model of the “axioms” represented by the deductive rule in the example schema.

An approach to satisfiability checking based on model generation appears to be the most appropriate technique for detecting unsatisfiable database schemas; it was prototyped in a system, called SATCHMO, which is described in the annotated bibliography. Without such a supporting tool, model generation has to be attempted by hand.

In many cases the deductive rules in a schema do not mutually depend on each other too intimately. Thus, with respect to satisfiability checking, it will be sufficient to check this property separately for each rule module. In case there are no integrity constraints related to a particular cluster of rule-defined concepts, satisfiability obviously cannot be a problem.

In the following paragraph, another general aspect of the problem of constraint satisfiability will be discussed, which involves abstract notions from formal logic again, but at the same time
corresponds to concrete design flaws that can be observed in practice. Understanding these problems is a necessary prerequisite for being able to prototype a deductive application.

11.2.2 Finite Satisfiability

Theorem proving, and thus satisfiability checking, is known to be an undecidable problem. Undecidability in this context means that it is impossible to ever build an algorithm which would be able to tell for arbitrary sets of axioms whether they are satisfiable or not. This sounds like rather bad news! Many people believe, that any attempt of checking an undecidable property is hopeless and should be avoided. However, such a fatalistic attitude is completely unjustified. If a class of problems is undecidable, this simply means that there is no algorithm which is able to decide the problem for all instances of this class; but, of course, there will be problems that can be decided, and algorithms able to decide for many problem instances. So it is reasonable to ask ourselves: What causes satisfiability to be undecidable? What are the cases whose test for satisfiability will run forever? There are in fact three different situations that may arise:

1. First, a set of rules and constraints may be unsatisfiable. If this is the case, every “proper” theorem prover will be able to detect this circumstance in finite time - but it may be quite a long time.

2. If the set of “axioms” is satisfiable, there are two subcases that may arise:

   (a) The rules and constraints have finite models, which means in practical terms, there is at least one finite database state which satisfies all integrity constraints. Logicians say that in this case the deductive database is finitely satisfiable, and proper theorem provers are always able to check this property after finite, but maybe rather long time of work. If such a state exists, it is rather likely that it is small and may be found rather quickly, simply because it is very difficult to design conditions which exclude small databases from being consistent.

   (b) However, there may be rules and constraints which taken together do not admit any finite, consistent database, but for which a hypothetical infinitely large database state can be imagined which satisfies all constraints. Such a schema is not strictly speaking unsatisfiable, although in practice it is as useless as an unsatisfiable one. Such sets of formulae for which all models are infinite are called “axioms of infinity” in logic. The existence of these consistent but problematic sets of logical axioms make satisfiability undecidable!

Again the practitioner may ask how likely it is that he falls into the trap of ever designing an “axiom of infinity” in his life. Unfortunately, as in the case of unsatisfiability, tiny design mistakes, overlooking a border case or forgetting an exception during rule or constraint design, may result in a situation where no attempt to create example databases satisfying all constraints will ever terminate. The following is an example of such an “axiom of infinity” in Chimera syntax, which seems to be quite ok on first glance, but can be easily shown to admit only infinite database states. The example is again a variation of the employees schema already used for illustrating unsatisfiability:

```plaintext
define object class employee
attributes    name: string,
             boss: employee, notnull
end;

define object class manager
superclasses  employee
attributes   subordinates: set-of(employee), derived
constraints  cyclicHierarchy

define implementation for class manager
```
11.2. DYNAMIC ANALYSIS

attributes  
X in Self.subordinates <--- X.boss=Self,
X in Self.subordinates <---
    manager(Y), Y.boss=Self,
X in Y.subordinates

constraints  
cyclicHierarchy <---
    Self in Self.subordinates

end;

Every employee has to have a boss (the notnull constraint). The derived attribute \textit{Subordinates} defines that a manager supervises all those employees of which he is the boss, as well as those which are recursively supervised by any of his subordinates. The constraint \textit{Cyclic-hierarchy} enforces that no manager directly or indirectly supervises himself, which seems quite reasonable.

Any attempt to populate a consistent test database with respect to this schema exhibits the problem. We need at least one employee, say \#1. He needs a boss according to the notnull constraint. He can't be his own boss due to the \textit{Cyclic-hierarchy} constraint. So there has to be a second employee, who also is a manager and supervises \#1. Call this manager \#2. He is an employee himself due to the superclasses clause in the definition of class \textit{manager} (which corresponds to an implicit deductive rule). Thus he is in need of a boss as well, as every employee does. This can't be himself and it can't be \#1 either, because in both cases \#2 would be contained in his own set of subordinates. So there has to be a third employee \#3, a manager again, who is the boss of \#2. Now, \#3 needs a boss as well ... a neverending story! Attempts of generating a model (or, a consistent example database) for this schema will inevitably lead to the infinite creation of employees, or in other terms, any finite transaction trying to populate this database will be rejected. This time not because of unsatisfiability, but because of unsatisfiability in finite models!

Repairing the malicious schema is again very easy: there has to be at least one top manager, a "boss of all bosses", who is not forced to have a boss himself. This can be achieved by introducing another subclass \textit{Top-manager of Manager}, by giving up the \textit{nonnull} constraint for the \textit{Boss} attribute of employees and by replacing it by a new targeted constraint \textit{Uncontrolled}, which enables only \textit{TopManagers} to remain without a \textit{Boss}:

```plaintext
define object class employee
attributes  
    name: string,
    boss: employee
constraint  
    uncontrolled;
end;

define object class topManager
superclasses  
    employee
attributes  
    subordinates: set-of(employee), derived
end;

define hard constraint uncontrolled for class employee
as  
    uncontrolled(Self) <---
      not topManager(Self), Self.boss=null
end;
```

For the repaired schema, there is a consistent minimal example database, which consists of two employees, one being the top-manager and supervising the other, who is an ordinary employee, but no manager. Again, though the example has been a theoretical one, it illustrates the likelihood of designing finitely unsatisfiable schemas simply by forgetting something, maybe during a first attempt of modeling an application.

Though there is no universal algorithm for detecting such cases - due to undecidability - there is a very simple and pragmatic way how "axioms of infinity" can be discovered in practice. Let an automatic tool try to generate a finite model, or do it yourself by hand. After each reasonable interval of time, e.g., after each ten objects created, check the intermediate state constructed thus
far and compare it to your expectations about a meaningful minimal database state. If the number of objects generated - due to the need to satisfy constraints - exceeds your expectations, become suspicious of an infinite loop and start tracing each individual step of the model generation process. In simple cases like the one in our example you will immediately see where the bug is, in more complex cases logically debugging your schema may be more difficult. But at least there is a brute force way out of the dangers of undecidability!

11.3 Running Example

The running case study is a "gentle provider" of problems both for what concerns the static and dynamic analysis of its deductive rules.

![Dependency Graph of Derived Attributes](image)

Figure 11.5: Dependency Graph of Derived Attributes

11.3.1 Static Analysis

Figure 11.5 shows the dependency graph of derived attributes, drawn according to the guidelines of Section 11.1. The graph contains only the dependencies caused by the deductive rules for data derivation; generic constraints define predicates which are not used to derive further information and therefore can be safely omitted for the purpose of stratification analysis. Also there are no cases of redefinition of derived attributes, hence we do not need to introduce hidden rules along the discussion of Section 11.1.2. For simplicity, nodes relative to class formulas are omitted, and we concentrate on derived attributes (the running case study has no views).
The dependency graph is decomposed into two independent components. The simplest component contains only one derived attribute, which can obviously be computed in one step from the extensional attributes. The other component contains only one recursively defined node, corresponding to the derived attribute \textit{Supplying}, and no negated arcs. Thus, the dependency graph admits an obvious stratification, consisting in computing first attributes \textit{BranchIn}, \textit{DirectlySupplying}, and \textit{BranchOut}; next, attribute \textit{Supplying}; next, attribute \textit{powerOut}; next, attribute \textit{PowerIn}; and finally, attribute \textit{PassingPower}. These derivations can be traced easily by looking to the code of the corresponding deductive rules.

### 11.3.2 Dynamic Analysis

For what concerns dynamic analysis, we concentrate on satisfiability of integrity constraints. We proceed by recognizing clusters of constraints and building test databases for them, until all constraints admit a small test database (and, consequently, their satisfiability is proved).

- Constraints \textit{NoSource/Sink}, \textit{NotnullSite}, \textit{NotnullBranches}, \textit{InvalidSwitch}, and \textit{InvalidBranch} deal with the topology of the database; their satisfiability is checked by creating a simple database consisting of two nodes (respectively producing and consuming power), by connecting them via a branch, and by adding to them the appropriate switches; the corresponding test database shows that this cluster of constraints is satisfiable.

- Next, \textit{InvDynConnection} and \textit{InvalidState} are concerned with the network’s dynamic state; again, in order to generate a test database it is sufficient to give appropriate values to the switches so that the above single branch is active, and populate the corresponding database.

- Finally, the remaining constraints deal with the balance of \textit{Power} in the network. Satisfiability of constraints \textit{ExclPassingPower}, \textit{NotEnoughPower}, and \textit{notnull} constraints relative to \textit{MaxPower} and \textit{PassingPower} is proved just by giving two suitable values of power to the \textit{Station} and to the \textit{Node} in this small sample database; constraint \textit{ExclIncrease} is dynamic and as such cannot be violated by just looking at a database state.

Populating at hand a database of two nodes, two switches, and one branch is very instructive, as it shows the high amount of information that can be inferred by applying deductive rules to this simple database. Indeed, our running example, both in its deductive and active component, demonstrates that a huge amount of knowledge can be controlled by a very small amount of data, and that we can reason about the network’s management in a very sophisticated manner.

### 11.4 Summary

In this chapter, we have introduced two abstract properties for deductive rules: stratification and satisfiability; we have shown static and dynamic analysis techniques that can be used for proving these properties, and discussed the desiderability of these properties for practical applications. This discussion has lead us to identify stratification as an easy property to check, that can be inspected on a dependency graph syntactically drawn from deductive rules. Satisfiability is an apparently hard problem (even undecidable), however manageable in many practical cases by suitable generation of test cases.

### 11.5 Annotated Bibliography

Static analysis and stratification techniques of deductive databases are defined in classical books, such as [164] and [48]. Dynamic analysis based on test data for generic integrity constraints in relational databases is discussed in [121]; previous efforts concerned the generation of Armstrong relations for testing multi-valued dependencies [147], and tableau chasing techniques [164]. The theorem prover
SATCHEM, developed at ECRC, is a tool designed for testing satisfiability of rules and constraints in a relational schema [29], [30], [111]. Although SATCHEM was developed for relational databases, it could rather easily be adapted to an object-oriented model, like Chimera.