# Expressiveness and Complexity of Formal Systems[*]

Giorgio Ausiello
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Via Salaria 113, Roma, Italy
ausiello@dis.uniroma1.it

Luca Cabibbo
Dipartimento di Informatica e Automazione
Università degli Studi “Roma Tre”
Via della Vasca Navale 79, Roma, Italy
cabibbo@inf.uniroma3.it

## Abstract

Formalization has a central role in computer science. Communication between man and computers as well as information processing within computer systems require that concepts and objects of the application domain are formally represented through some artificial language. In the last fifty years, the development of formal models has concerned practically all aspects of human life, from computing to management, from work to game and sex. Such development is one of the important contributions of computer science to human epistemological and intellectual development, which expands the powerful role that mathematical formalization has played through the centuries. In building a formal model of reality, though, an important issue arises, the complexity of the formal system that is adopted; in fact, the stronger is the expressive power of the system, the higher is the complexity of using the model. An “optimal” balance between expressive power and complexity is the key that explains the success of some formal systems and languages that are widely used in computer science and applications. After discussing the contrasting needs of expressiveness and computational tractability in the construction of formal models, in the paper two case studies are considered: Chomsky formal grammars and relational database languages, and the balance between expressive power and complexity for such systems is analyzed in greater detail. Finally, the concept of succinctness of a formal system is taken into consideration and it is shown that the role of succinctness in affecting the complexity of formal systems is crucial.

## 1. Representation, modeling, formalization

Formalization has a central role in computer science. Communication between man and computers as well as information processing within computer systems require that concepts and objects of the application domain are formally represented through some artificial language. For this reason computer science has emphasized the importance of formal modeling and of analyzing formal properties (“expressiveness”, “complexity” etc.) of modeling languages.

Before discussing formal modeling in computer science, let us briefly address the more familiar issue of mathematical modeling. As it will become evident, no

[*] The first author gratefully acknowledges the support of the Italian MURST National Project “Efficienza di algoritmi e progetto di strutture informative.” The second author has been partially supported by MURST and CNR.

clearcut can separate a discussion on formalization in computer science from the issue of mathematical modeling or, even more, from the issue of representation of reality throughout mankind history.

The need for men to represent and understand physical phoenomena is at the base of the use of mathematical models. We might say that mathematics itself was born to satisfy such need and, also, to provide formal tools for answering questions raised by the early engineering problems. It is well known, for example, that land measurement and astronomical observations led to the development of geometry in Egypt and in Greece, in ancient times.

Indeed, looking back in mankind history we may realize how profound the need for representing reality was for men, way before the beginning of mathematics. In fact, long time before starting to "formalize" concepts of the real world in mathematical terms, man started to feel the need to "represent" reality. Since the beginning of men's social life, representation of reality had a magic and an epistemologic role at the same time. By drawing horses and wild oxen on a cave's walls the hunters of neolithic age could "capture" the animals with their minds before chasing them with their primitive weapons.

In the same way, "capturing" concepts has been for millennia the aim of the intellectual activity by which men, through an abstraction process that from the observation and representation of reality leads to the classification of "objects" and to the formalization of their "interrelationships", have expanded their knowledge.

Clearly, mathematics has played a major role in such a process, extending the ability of men to model reality in a formal way, and this role has continued through the centuries, first with the applications to commerce and finance, in the Middle Ages, and then, in the Modern Age, with the great theoretical developments that accompanied the physical discoveries (mechanics, electromagnetism etc.) and the industrial revolution.

In the last fifty years, due to the use of computers, the need for formal modeling has expanded in all directions of human activity. Indeed, it has been in connection with the first military applications of computers in organization and logistics, that new mathematical modeling and problem solving methodologies (namely, operations research) were developed. Subsequently, new formal tools were introduced for modeling the various aspects of productive life where computers were gradually introduced, such as industrial production, information management, office automation and workflow management, etc. More recently, the large increase of computer applications in everyday life has led to the development of formal systems for representing and automating a wide variety of human activities, ranging from natural language processing to common sense reasoning, from image understanding to interactive graphics.

Beside enhancing the ability of modeling reality, the use of computers has brought the need for formalizing and understanding aspects of computing itself: semantics and efficiency of programs, computation models, concurrency, man-machine interaction, etc. Within the field of computer science a substantial body of theoretical knowledge has been created around these subjects during the last three decades. In particular, computer scientists have developed a research area which is not only of practical interest (in connection with software engineering methodologies) but also of great epistemological interest: the study of formal methods and languages for building computer applications.

In fact, the development of formal models in computer applications often proceeded in parallel with the evolution of programming methodologies. While new domains were attacked, new formal programming concepts and new information structures were needed for representing reality. While imperative programming was suitable for (and, indeed, born together) scientific numerical applications, information management eventually led to database models, declarative languages, and to structured design methodologies. Recently, more advanced interactive applications have led to the widespread use of the object-oriented approach, in which the real world is represented in terms of objects, classes, and interaction methods; then, the essence of modeling is captured by offering the programmer the paradigms of abstraction, generalization, inheritance, etc.

The study of modeling and formalization processes, hence, has become one of the major areas of interest in computer science. Its meta-theoretical character directly relates this field with the logical and meta-mathematical studies that, between the Thirties and the Fifties, investigated the power of computation processes and computability.

In this paper two particular aspects of formal models are considered, expressiveness and complexity, and their conflicting characters are discussed, by drawing examples from some well established fields of theoretical computer science. In the next section the concept of “life cycle” of a formal model is introduced and it is observed how the search for more expressive power is in contrast with the need for computational efficiency in the use of the model. In Section 3 and Section 4 the issue of complexity of formal models is discussed with reference to two fundamental areas of computer science: the theory of formal languages and the theory of database languages, respectively. In Section 5, the source of complexity in a formal model is analyzed and identified in the concept of succinctness of representation. Finally, Section 6 draws some conclusions.

## 2. Formal models’ life cycle

Formalization in computer science can be seen as the process of reducing a fragment of reality to a simplified system of symbols through abstraction with the aim of computing solutions of problems, proving properties, running simulations, etc.

Typically, the construction of a formal model goes through various stages. Without referring to any specific design methodology, we may roughly characterize the various stages as follows.

First, the real world domain that we wish to model is analyzed; the entities (objects) that characterize it are identified and classified through an abstraction process and so are their mutual relationships and their relationships with the outside world.

Then, a representation system (the “syntax”), consisting of formal symbols, is introduced. These symbols, that may be characters of a given alphabet, pictures, icons, are then mapped to the objects and the relationships of the real system.

Finally, the behaviour of the real system is taken into consideration and the behaviour of the symbolic system is defined in such a way to mimic it (for example, by means of state transition rules).

An interesting example, in this respect, is provided by Petri nets [15], a formal model that has been introduced in 1962 for describing systems organization and, since then, has been widely used for modeling and analyzing systems of concurrent processes and, more generally, systems in which regulated "flows" play a role, such as parts' flow in a production line, jobs' flow in a computer system, work flow in an office.

In their simplest version [16] Petri nets are used to model systems (called "condition/event systems") in which objects flowing in the net determine "conditions". In suitable cases conditions determine "events" that, in turn, determine new flow of objects in the net. A situation of this kind arises, for example, in a library. A request is submitted to the loan clerk (first condition) and, if the book is available (second condition), the loan event is determined. As a consequence, the following two conditions are created: the book is given to the customer while the request is stored in the borrowed books file. When a book is returned and the corresponding requests is in the borrowed book file, the return event is determined and the book is placed in the stack again.

From the syntactic point of view, a Petri net of this type (called Condition/Event Net, C/E Net) consists of a triple (P, T, F), where P, the set of "places", represents the conditions, T, the set of "transitions", represents the events, and F, the "flow" relationship, relates places and transitions. Graphically, places are usually drawn as circles, transitions as bars, and the flow relationship is represented by lines and arrows (see Figure 1).

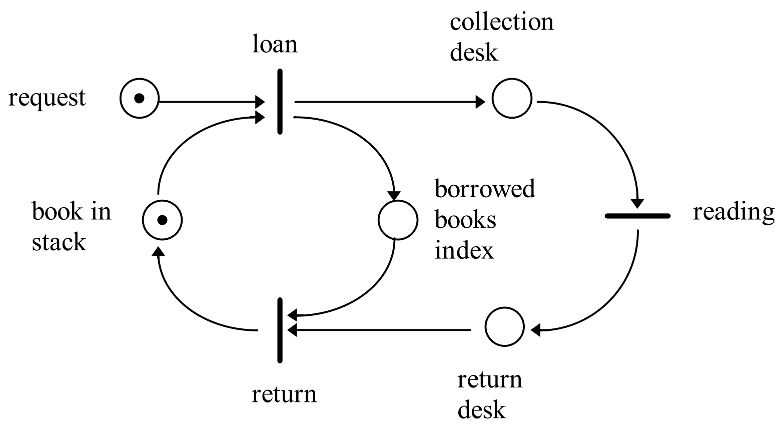

**Figure 1**. Petri net model of loans in a library.

In a C/E Net the objects flowing in the system are represented by means of "tokens". The behaviour of the net is then characterized in terms of transition rules: if all places preceding a transition contain a token the transition is enabled and, as a consequence, the tokens in such places are "consumed" while a token is "produced" in any place following the transition.

The use of a Petri net model allows to study the properties of the domain of interest, in this case a concurrent system. Typical properties that we wish to analyze are the absence of deadlock, the reachability of particular condition configurations, the fact that a given transition will eventually be enabled (liveness), etc.

Figure 1 contains a simple example of C/E Net modeling the loans in a library.

Once a formal model is established we have to check its validity by matching its features against the system that we want to represent. This is achieved by defining the semantics of the model, that is by establishing an interpretation, which maps syntactic structures of the model back to the system, and by checking whether the model satisfies two fundamental characteristics: soundness (may the properties of the model be suitably interpreted in the system?) and completeness (does the model capture all the properties of the system?).

As a consequence of the semantic analysis we may realize that the formal model does not capture all aspects of reality that we want to simulate and process. In other words the expressive power of the formal model is insufficient and an enrichment is necessary. If we go back to the Petri nets example, we notice that the Condition/Event model is actually rather simplistic. Suppose that books are classified (some can be borrowed for one week, some other for one month). This case cannot be modeled by a C/E Net. In fact tokens representing requests and books should be labeled and only corresponding labels can activate a loan. More complex types of Petri nets have been introduced for dealing with situations of this kind (for example, Colored Petri Nets [8]).

An immediate consequence of the increase in expressive power of the model is a corresponding increase in complexity. In particular the flow relationship, which relates places and transitions, becomes more complex. Running the model on a computer may become heavy, proving its properties may become computationally intractable or even undecidable.

When constructing a formal model we are therefore faced with two contrasting needs: on the one hand, the need to make the model more expressive, more powerful, in order to capture more and more aspects of reality; on the other hand, the need to keep the complexity of the model low, in order to preserve the tractability of its properties. Usually formal models "stabilize" in an equilibrium point that constitutes an "optimal" compromise between the two needs. This position is maintained until the advances of technology push toward the adoption of a new model which will also reach, after a while, through different stages, its optimal equilibrium point.

Thus, the practical success of various formal systems in computer science is due to the fact that for such systems the optimal compromise between expressiveness and complexity has been reached. A typical example is represented by finite state automata, introduced in the Fifties by E.F. Moore and used for specifying sequential machines, for which most properties can easily be proved and whose computational power can be characterized under various points of view: in terms of classes of input sequences (called regular events in [11]), by means of equivalence with respect to the neural net model of McCulloch and Pitts (see again [11]), in terms of closure under algebraic operators (see [7]), etc. Other examples of formal systems in which the expressive power is conveniently balanced by processing efficiency are deterministic context free languages (whose syntactic properties provide the paradigm for the structure of most programming languages and at the same time allow the efficient compiling of programs) and relational database languages (whose strong expressive power, characterized in terms of both relational algebra and calculus, is nevertheless compatible with efficient implementation of query processing in commercial database systems).

In the next two sections we will see some specific aspects of this correlation between expressive power and complexity, occurring in the case of such particular, well studied families of formal systems.

## 3. Chomsky's grammars

Chomsky grammars were introduced by Noam Chomsky in the Fifties [2] with the aim of providing a formal basis to the understanding of the structure of natural languages. Chomsky grammars are a generative formal system, based on "productions rules" (a particular kind of rewriting rules, related to earlier work of the logicians Axel Thue and Emil Post, see [6]) and allow the characterization of classes of languages, in relation with the structural properties of the corresponding production rules. Formally, a *Chomsky grammar* consists of a 4-tuple ⟨T, V, S, P⟩ where T is the alphabet of *terminal symbols*, V is the alphabet of *non-terminal* (*variable*) *symbols*, S is a particular symbol of V, called *axiom*, the seed of the generation process, and P is the set of *production rules*. A production rule is of the form:

$$u \to v$$

where *u* and *v* are strings of, possibly, terminal and non-terminal symbols with the only constraint that u contains at least one non-terminal symbol. Such a rule specifies that, if the substring *u* occurs in the string that we are in the process of generating, then we can replace *u* by *v*. The *language generated by the grammar* is the set of all strings consisting of only terminal symbols that can be generated starting from the axiom and repeatedly applying the production rules. An example is the following. Consider the grammar $\langle \{a, b\}, \{A\}, A, \mathrm{P}\rangle$, where P is the set of production rules:

$$A \to aAb$$
$$A \to ab$$

The grammar generates the language consisting of all strings of the form $a^n b^n$ for all n>0, that is, all strings consisting of a sequence of n *a*'s followed by a sequence of an equal number of *b*'s.

The example is indeed rather simple and corresponds to a particular case of grammar that is called *context free grammar*. In this type of grammar, in each production the left hand string consists of just one symbol (say, *A*), which can be replaced by the right hand string no matter the context in which it occurs. Other important particular classes of grammars are *context sensitive grammars*, where the right hand string and the left hand string are of general type but the right hand string cannot be shorter than the left hand string, and *regular grammars*, a particular case of context free grammars in which the right hand string consists of a single terminal symbol or of a terminal symbol followed by a non-terminal symbol.

Chomsky grammars (especially context free grammars) played an important role in the study and understanding of natural language concepts; however, after a while it became clear that their structure was too simple for such aim and other formal generative devices were introduced [4]. Nevertheless, despite being inadequate to formalize natural languages, Chomsky grammars had a remarkable effect on the syntactic development of artificial languages used for computer programming. A few years after the seminal work of Noam Chomsky, John W. Backus [1] and Peter Naur [14] introduced the so-called Backus Normal Form (or Backus Naur Form, BNF, for short) for the description of the programming language ALGOL, which was based on a sort of production system and indeed could be seen as a reformulation of context free grammars. In BNF, for example, the structure of an if-statement in a programming language could be expressed in the following way:

$$\langle\text{if-statement}\rangle ::= \textbf{if}\ \langle\text{condition}\rangle\ \textbf{then}\ \langle\text{statement}\rangle\ \textbf{else}\ \langle\text{statement}\rangle$$

which is clearly structured as a context free production with **if**, **then**, **else**, as terminal symbols and ⟨if-statement⟩, ⟨condition⟩ and ⟨statement⟩ as non-terminal symbols.

Since the early Sixties, therefore, the study of syntactic aspects of programming languages as well as compiler design techniques have been based on the structure and properties of Chomsky grammars. The main problem we have to solve in this context is the following. We are given a text consisting of several thousands of lines of source code, and we want to translate it efficiently into machine code. This means that by scanning the text (possibly only once) we want to perform at the same time the following operations:

- check for syntactic correctness and, in case, report errors;
- interprete the meaning of the code;
- translate into machine language.

Moreover, we don't want to waste too much space in doing this. Clearly the cost of the said operations changes according to the syntactical structure of the text, that is, according to the type of grammar that defines the language. Table 1 reports the cost of verifying syntactic correctness with different types of grammars. More precisely, the table indicates time and space needed for solving the so called *recognition problem*, that is the problem of deciding, given a grammar of a certain type, whether a string has been derived according to the rules of such grammar. Note that, for the general class of unrestricted Chomsky grammars, such problem is undecidable. Actually, Chomsky proved that the computational power of such grammars coincides with the computational power of Turing machines [3].

| **Type of grammar** | **Time** | **Space** |
|---|---|---|
| Regular | O(n) | O(1) |
| Deterministic Context Free (LR(k)) | O(n) | O(log n) |
| Context Free | $O(n^3)$ | $O(\log^2 n)$ |
| Context Sensitive | $O(2^{n^2})$ | $O(n^2)$ |
| Unrestricted | undecidable | |

**Table 1**. Complexity of the recognition problem for various classes of formal languages.

From the table the following facts can be seen. Languages defined by regular grammars can indeed be recognized very efficiently, in linear time with a constant amount of memory; unfortunately, the syntactic structure of regular grammars is too simple for real programming languages. For example it allows to express sequences of simple commands (such as the ones that are used in an electronic mail system) but does not allow to define algebraic expressions in infix notation. For representing all types of algebraic expressions that are needed in computer programming as well as the parenthetical structure of nested programming constructs (if statements, while statements, for statements etc.) context free grammars are needed, but then, in general, the syntactic analysis costs become prohibitively high.

In order to overcome this difficulty several new classes of languages have been introduced, properly included in the general class of context free languages (actually all of them included or coinciding with the class of the so called *deterministic context free languages*) but strictly more powerful than regular languages, for which both time and space-efficient parsing algorithms could be defined. In particular, the *LR(k) languages* [12] are a class of languages that can be efficiently parsed from left to right if we allow a *k* symbols look-ahead. These languages are enough expressive to allow the definition of the syntactic constructs required by most programming languages while achieving, at the same time, very good efficiency in syntactic recognition, parsing, and translation. These languages represent a lucky example, in this context, of the optimal balancing between the needs of expressiveness and the needs of efficiency of a formal system that was discussed above.

## 4. Database query languages

A *database* is a collection of structured data, to represent some aspect of the real world for a specific purpose. Typically, databases are large, persistent, and shared by several users. Database technology offers the software tools for an efficient and effective management of databases. The theoretical foundation of current database management systems is provided by the *relational model of data*, a formal model proposed by E.F. Codd in the early Seventies. (See [10] for a survey on relational database theory.)

Intuitively, data in a relational database consist of sets of rows, each row representing a relationship among a set of values; rows with uniform structure and intended meaning are grouped into tables. More precisely, a *relational database* is a collection of *relations*. Each relation has a structural part (called the *scheme*) and an extensional part (called the *instance*). A relation scheme consists of a name (unique in the database) together with a tuple of distinct names, called the *attributes* of the relation. A relation instance is a finite set of tuples over the attributes specified in the scheme.

**Flights** (*Company, Flight-No, From-Airport, To-Airport*)
**Airports** (*Airport, Full-Name, City*)
**Trains** (*Train-No, From-City, To-City*)

**Flights**

| *Company* | *Flight-No* | *From-Airport* | *To-Airport* |
|---|---|---|---|
| Alitalia | AZ2010 | FCO | LIN |
| Alitalia | AZ2011 | LIN | FCO |
| Alitalia | AZ638 | FCO | CHI |
| KLM | KL264 | FCO | AMS |
| TWA | TW312 | JFK | CHI |
| UsAIR | US913 | AMS | JFK |

**Airports**

| *Airport* | *Full-name* | *City* |
|---|---|---|
| AMS | Schiphol | Amsterdam |
| CHI | O'Hare Intl. | Chicago |
| FCO | Leonardo da Vinci | Rome |
| JFK | J.F. Kennedy Intl. | New York |
| LIN | Linate | Milano |

**Trains**

| *Train-No* | *From-City* | *To-City* |
|---|---|---|
| E654 | Rome | Florence |
| EC699 | Paris | Frankfurt |
| IC412 | Florence | Milan |
| EC511 | Milan | Paris |

**Figure 2**. An example transportation database.

As an example, consider the transportation database (shown in Figure 2) representing information about flight and train connections. Data are split into three relations, named **Flights**, **Airports**, and **Trains**, each of them representing facts about a specific kind of information. Relation **Flights** has attributes *Company* (the name of the company offering a flight), *Flight-No* (the flight number), *From-Airport* and *To-Airport* (the codes of the departure and arrival airports). Each tuple in this relation contains a value for each attribute, establishing a relationship among the values; for instance, the tuple (*Alitalia*, *AZ638*, *FCO*, *CHI*) states that Alitalia offers a flight, coded AZ638, from airport FCO to airport CHI. The information about the cities connected by the flight can be found by means of the relation **Airports**: since FCO and CHI correspond to airports in Rome and Chicago, respectively, this means that AZ638 is a flight connecting Rome to Chicago. The fact that different information are spread across different relations is for the sake of succinctness, to avoid redundancy. However, information can be extracted by correlating tuples in multiple relations, mainly relying upon equality of values. This activity is referred to as *querying* the database, the main topic in database management together with *updating*. Conversely, an update consists of a set of additions, removals, and/or modifications of tuples within relations. (We will not consider updates anymore, since a discussion of this topic is beyond the scope of the paper.)

Formally, a *query* is a function from databases to databases, specified by means of an expression of some *query language*. (With a little abuse of terminology, the term "query" is often used for both the function, the expression, and a natural language description of the function.) Possible queries over the transportation database are the following.

**Q1** Is there any direct flight connection from Rome to New York?

**Q2** What are the cities for which there is a direct connection (either by flight or by train) from Rome?

**Q3** What are the pairs of cities for which there is a direct flight connection but not a direct train connection?

**Q4** Is there any flight connection (possibly involving intermediate stops) from Rome to New York?

**Q5** What are the pairs of cities having an airport but for which there is no flight connection?

The relational model adopts the so-called *closed world assumption*, according to which the facts stored in a database are the only ones to be true, and the ones that are not present in the database are assumed to be false. (For example, our transportation database assumes that there is no flight connection from Rome to Paris, since there is no explicit fact stating it.)

The relational model is provided with two basic query languages, stemming from different paradigms. The *relational algebra* is a “procedural” language, based on a few algebraic operators. In order to allow for composition, all the operators produce relations as results. On the other hand, the *relational calculus* is a “declarative” language, based on the first order predicate calculus. By “procedural” we mean that a query specifies the actions that must be performed to compute the answer to a query. Conversely, “declarative” means that a query is specified in a high-level manner, essentially by stating the properties that the result should satisfy; in this case, an efficient execution of the query has to be worked out by the interpreter of the language. Thus, “declarative” concerns “what” and “operational” concerns “how.”

In spite of the differences in the two languages, they are equivalent to each other, that is, they can express exactly the same queries. This equivalence is known as Codd’s Theorem, and can be specialized to suitable restrictions and extensions of the relational calculus and algebra. Codd’s Theorem has great practical significance: the translation of calculus into algebra reveals a procedural evaluation for a query defined declaratively by a calculus expression.

Instead of introducing the relational algebra or calculus, we discuss some examples referring to a declarative query language, stemming from logic programming, called *datalog*. Intuitively, queries are specified in *datalog* by means of sets of rules, called *programs*. The left-hand-side of a rule is a conjunction of literals, referring to relation names and variables: if there are values for the variables such that all the literals in the left-hand-side are known to be true in the database, then we can infer the truth of the literal in the right-hand-side of the rule. Figure 3 shows the programs implementing the queries **Q1** to **Q5** over the transportation database.

| | | | |
|---|---|---|---|
| **Q1** | Ans1 | ← | Flights(C,N,F,T), Airports(F,FN,“Rome”), Airports(T,TN,“New York”) |
| **Q2** | Ans2(X) | ← | Flights(C,N,F,T), Airports(F,FN,“Rome”), Airports(T,TN,X) |
| | Ans2(X) | ← | Trains(N,“Rome”,X) |
| **Q3** | Conn-by-train(X,Y) | ← | Trains(N,X,Y) |
| | Ans3(X,Y) | ← | Flights(C,N,F,T), Airports(F,FN,X), Airports(T,TN,Y), **not** Conn-by-train(X,Y) |
| **Q4** | Flight-conn(X,Y) | ← | Flights(C,N,X,Y) |
| | Flight-conn(X,Y) | ← | Flight-conn(X,Z), Flight-conn(Z,Y) |
| | Ans4 | ← | Flight-conn(F,T), Airports(F,FN,“Rome”), Airports(T,TN,“New York”) |
| **Q5** | Flight-conn(X,Y) | ← | Flights(C,N,X,Y) |
| | Flight-conn(X,Y) | ← | Flight-conn(X,Z), Flight-conn(Z,Y) |
| | Ans5(X,Y) | ← | Airports(SX,NX,X), Airports(SY,NY,Y), **not** Flight-conn(SX,SY) |

**Figure 3**. Examples of Datalog queries.

*Datalog* has many variants, some of which are described as follows.

- *Conjunctive* programs are made of a single rule, not involving negated literals.
- *Positive existential* programs are made of multiple rules, with no negated literals and no recursive definitions.
- *First order* programs are made of multiple rules, with no recursive definitions.
- *Datalog* programs are made of multiple rules, with no negated literals.
- *Stratified* programs are made of multiple rules; a technical condition is imposed to avoid that recursive definition involves negated literal.
- *Datalog with 1-sets* are stratified programs allowing for the management of sets of values.
- *Datalog with k-nested sets* are stratified programs with sets, allowing for *k* levels of nesting.
- *Datalog with sets* are stratified programs with sets, allowing for unbounded levels of nesting.

The *expressiveness* of a query language is related to the class of queries that the language can express. Specifically, we say that a language *L* expresses a query *q* if there is an expression *E* of *L* whose semantics coincides with the function *q*. For example, the query **Q1** can be expressed by means of all the above cited languages, whereas query **Q3** can be expressed by a first order program, but not by a conjunctive program (since negation is required). We say that a language *L* is *more expressive than* another language *L'* if any query expressible in *L'* is expressible in *L* as well. For example, it turns out that *first-order datalog* is more expressive than *conjunctive datalog*. Note the the relation “more expressive than” is a partial order and not a total one, since some languages are not related according to it. For example, *datalog* and *first order datalog* are unrelated, since there are queries that are expressed only by one of the two languages.

It should be noted that *first-order datalog* is another language having the same expressiveness of the relational algebra and calculus, and that *conjunctive datalog* and *positive existential datalog* have equivalent counterparts in suitable restrictions of both the relational algebra and calculus. Thus, we can consider classes of languages having the same expressiveness. The relative expressiveness of the various classes is summarized in Figure 4. (The arrows denote greater expressive power, in the strict sense.)

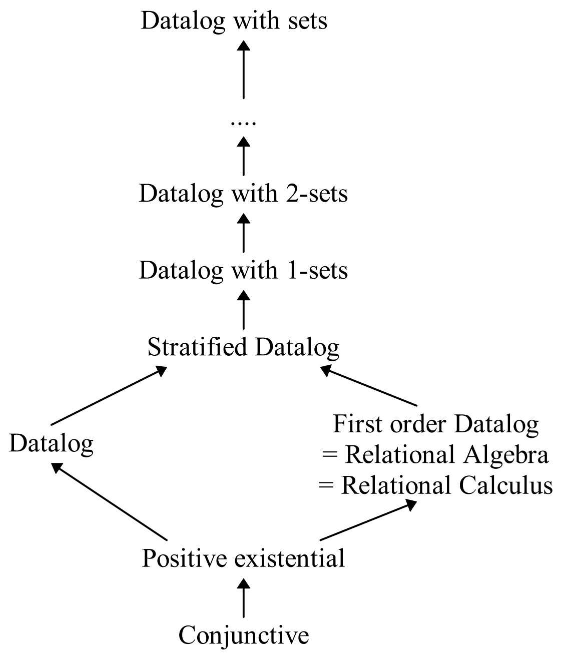

**Figure 4**. Hierarchy of query languages based on their expressiveness.

Database theoreticians consider different complexity measures, differing in the parameters with respect to which the complexity is measured. Intuitively, the complexity of evaluating a query against a database is related to both the size of the query expression and that of the database. In practice, the size of the database tipically dominates, by many orders of magnitude, the size of the query, and is therefore the parameter of interest. The *data complexity* of a query is defined as the computational complexity of testing whether a given tuple belongs to the result of the query against a database *d*.

It turns out that the relational algebra and calculus express only queries whose data complexity is in LOGSPACE. Since the class LOGSPACE is contained in the class NC[1], this implies that the two languages have a lot of potential parallelism. The other side of the coin is that the two languages cannot express queries whose data complexity is higher than LOGSPACE. Of course, the trade-off is between expressibility of a language and possibility of efficient execution.

There is another point to ponder, namely, query optimization. Usually, queries are written in a high-level language (such as the relational calculus), and automatically

[1] NC (Nick's Class, from the name of Nick Pippenger, who defined this class) is the class of problems that can be efficiently parallelized. (See [9] for more details.)

translated into an equivalent expression of a procedural language (such as the relational algebra), using Codd's Theorem. Then, the database management system tries to optimize the expression, by rewriting it into another equivalent expression that is better under some measure of complexity. The rationale for optimization consists in trying to minimize the computational cost of evaluating the query, which is mainly related to the size of intermediate results. An optimizer has a module for estimating the evaluation cost of an expression. The goal is to find an equivalent rewriting of the query with minimal esteemed cost. To do so, the optimizer generates a set of rewritings, by means of suitable heuristics, and verifies the equivalence with the initial expression. Of course, this is convenient only if the optimization process does not consume more resources than the evaluation of the initial expression. Specifically, the computational cost of optimization is mainly related with the complexity of testing for the equivalence of the rewritten expression with the initial one. Unfortunately, testing for equivalence of relational algebra expressions is undecidable, but decidable for more restricted languages. Again, there is a trade-off between expressibility of a language and possibility of efficient optimization.

Table 2 summarizes, for each class of queries, the data complexity and the complexity of testing expression equivalence. (See [9] for a definition of the complexity classes mentioned in the table.)

| **Class of queries** | **Data complexity** | **Complexity of the equivalence problem** |
|---|---|---|
| conjunctive | LOGSPACE | NP-complete |
| positive existential | LOGSPACE | $\Pi^P_2$-complete |
| first order | LOGSPACE | undecidable |
| datalog | PTIME | undecidable |
| datalog with stratified negation | PTIME | undecidable |
| datalog with 1-sets | EXPTIME | undecidable |
| datalog with k-sets | k-EXPTIME | undecidable |
| datalog with sets | unbounded | undecidable |

**Table 2**. Data complexity and complexity of the equivalence problem for various query languages.

We now briefly discuss the compromise reached between the needs of expressiveness and complexity in the case of relational query languages. In practice, that is, in real database management systems, the query language often used is SQL (for Standard Query Language). SQL is a declarative language based on a variant of the relational calculus. With respect to the expressive power, it is essentially equivalent to both the relational calculus and algebra, and thus its queries can be evaluated rather efficiently. On the other side, Table 2 shows that finding the optimum rewriting of an SQL query is, in general, an unsolvable problem. In principle, the optimization could be still carried on for some SQL queries (the ones that syntactically are conjunctive or positive existential). In practice, however, the systems apply some set of predefined heuristics, usually finding "good" rewritings, but without a guarantee to find the best ones.

## 5. Expressiveness, succinctness and complexity

In the preceding sections, by means of various examples, we have seen how an increase in the expressive power of a formal model entails a corresponding increase in the cost of the computations that we want to perform on the model (such as testing for membership in formal languages or answering queries in databases).

In this section we will see that a major role in the expressiveness-complexity trade-off is essentially played by the succinctness of the encodings that the formal system allows. In particular, in strongly expressive formal systems we may encode computational processes (such as for example Turing machine computations), and what makes the properties of certain classes of formulae intrinsically hard to decide, is the fact that "short" formulae may describe "long" computations.

The first evidence of this property is provided by Cook's Theorem [5]. This result, proven by Steve Cook in 1971 and, independently, by L. Levin in 1973 [13], establishes the intrinsic complexity of deciding satisfiability of propositional formulae by showing how such formulae can encode Turing machine computations. More precisely, the result shows that, given a nondeterministic Turing machine M operating in time bounded by a polynomial p, and given a string *x*, of length n, we can build a propositional formula *w* (of length at most $p^4(n)$) that is satisfiable if and only if *x* is accepted by M in time p(n). As a consequence, the problem of deciding satisfiability of propositional formulae is proven to be at the highest level of complexity in the class NP (the class of all problems that can be solved in polynomial time by means of nondeterministic Turing machines) or, as it is customary to say, "NP-complete". It is well known that still, 25 years after Cook's result, we do not know any polynomial-time algorithm for solving neither the satisfiability problem nor any of the thousands of NP-complete problems that have been discovered since then. At the same time, thanks to Cook's result, we know that if any polynomial-time algorithm would be discovered for the satisfiability problem (an unlikely possibility, though), all other NP-complete problems could be solved in polynomial time and the class NP would be proven to coincide with the class P (the class of problems solvable by means of deterministic Turing machines in polynomial time), giving positive answer to the well known question "P = NP?".

The formula built in Cook's Theorem's proof, somehow, encodes the rules that all Turing machine computations have to satisfy (e.g., at any given instant t the head of the machine can read exactly one tape cell), encodes the initial and final configurations (at time 0 tape cells 1 through n contain string x, all other tape cells are blank; at time p(n) the machine state should be the final state), and, finally, encodes the relationship between instantaneous configurations of the tape that are implied by the transition function of the machine M.

What is particularly important is that the construction is logarithmically succinct because a nondeterministic computation of polynomial depth p(n) is indeed a tree of, possibly, $2^{p(n)}$ instantaneous configurations while in the proof the same information is compacted in a polynomially long formula. This gives the proof of Cook's Theorem all its remarkable power.

The fundamental idea to represent "long" computations with "short" formulae is at the base of practically all proofs of complexity hardness of problems (see [9]) and it is interesting to observe that the shorter are the formulae, the more complex are the properties of the formalism. Since in general, in this context, computations are modeled in terms of Turing machines, we may conclude that the intrinsic complexity of a formalism can eventually be related to the power of the formalism in describing Turing machine computations. In order to clarify this claim, let us consider the formalism of regular expressions.

Regular expressions have been introduced by S.C. Kleene (in connection with the definition of the already mentioned regular events [11]) and are used for describing sets of words on a given alphabet (sets of input sequences to an automaton, paths on a labeled graph, etc.).

In their basic formulation, given an alphabet $\{a,b\}$, regular expressions are inductively defined as follows:

- $a,b$ are regular expressions;
- if $e_1$ and $e_2$ are regular expressions, then $e_1\,e_2$, $e_1+e_2$, and $e_1^*$ are regular expressions.

The meaning of a regular expression is a language defined as follows:

- $a$ and $b$ represent the languages $L_a=\{a\}$ and $L_b=\{b\}$, respectively;
- if $L_{e_1}$ and $L_{e_2}$ are the languages represented by the regular expressions $e_1$ and $e_2$, then:
  - $e_1\,e_2$ represents the concatenation of languages, $L_{e_1} \circ L_{e_2}$;
  - $e_1 + e_2$ represents the union of languages, $L_{e_1} \cup L_{e_2}$;
  - $e_1^*$ represents the iteration (Kleene's star operator) of a language, $(L_{e_1})^*$.

For example, the regular expression $ab^*(aa+bb)^*$ represents the language consisting of strings that begin with $a$, and are followed by a (possibly empty) sequence of $b$'s, and by a (possibly empty) sequence of strings of the type $aa$ or $bb$.

The formulation of regular expressions may be enriched with the complement operator (that is, $e_1^C$, whose meaning is $\{x \mid x \text{ is not a string in } L_{e_1}\}$) and with the squaring operator (that is, $e_1^2 = e_1\,e_1$, whose meaning is $\{x \mid x = uv,\ u,v \in L_{e_1}\}$), without changing the expressive power. In other words, if we write a regular expression $e_1$ with the complement and/or the squaring, we know that there is another regular expression $e_2$ which does not make use of neither the complement nor the squaring and such that $L_{e_1}$ equals $L_{e_2}$.

It is important to notice that the complement and squaring operators allow to express in a more succinct form the same languages as we can represent with expressions that do not make use of such operators. Table 3 provides the complexity of the problem of deciding whether a regular expression is not equivalent to $\{a,b\}^*$ for various classes of regular expressions.

| **Regular expressions with operators** | **Complexity** | **Time complexity** |
|---|---|---|
| °, + | NP-complete | $O(n^2)$ |
| °, *, + | PSPACE-complete | $O(2^{n^k})$ |
| °, *, +, squaring | EXPSPACE-complete | $O(2^{2^n})$ |
| °, *, +, complement | non elementary | $O(2^{2^{\cdots^{2}}}\}$n times) |

**Table 3**. Complexity of the equivalence problem for various classes of regular expressions.

In all cases, except the first one, the expressions represent the same class of languages (regular languages), but the increase of problem complexity that accompanies them derives from the succinctness of the formulae. For example, it is easy to see that the squaring operator allows a logarithmic compression of strings: in fact $(\ldots((a^2)^2)^2\cdots)^2\}$n times, corresponds to a sequence of the type $a\ldots a$ of length $2^n$. Even stronger is the compression that we may achieve by making use of the complement operator and, as a consequence, the complexity of deciding whether a regular expression with complement is equivalent to $\{a,b\}^*$ becomes *non elementary*. In other words, its complexity cannot be expressed by any elementary function obtained by composition of sums, products, and exponential functions, because it grows as $2^{2^{\cdots^{2}}}\}$n times.

## 6. Conclusions

The complexity of deciding properties of a formal model is strictly related to the expressive power of the model. By referring to the well-known models of Chomsky formal grammars and of relational databases, we have seen that the success of a formal model somehow derives from the fact that in such model an optimal balance is established between two factors: (i) the need of achieving a strong expressive power in order to represent and study the fragment of reality of interest (for example, in those cases, the syntactic structure of programming languages or the structure of queries allowed on a database); and (ii) the need of preserving the efficiency in the use of the model (syntax analysis and query processing in the mentioned examples). Finally, by examining the growing difficulty of deciding equivalence in various families of regular expressions, we pointed out that, when formalisms with the same expressive power are considered, the computational complexity of the properties of the formal expressions used in a formalism strictly depend on their succinctness and in particular on the fact that very short formulae may be used for representing long computations.

## 7. References

[1] Backus, J.W. The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. *Proc. Intl. Conf. on Information Processing*, UNESCO, 125-132, 1959.

[2] Chomsky, N. Three models for the description of language. *IRE Trans. on Information Theory*, 2:3, 113-124, 1956.

[3] Chomsky, N. On certain formal properties of grammars. *Information and Control*, 2:2, 137-167, 1959.

[4] Chomsky, N. *Knowledge of Language. Its Nature, Origin and Use*. New York, Praeger, 1986.

[5] Cook, S.A. The complexity of theorem-proving procedures. *Proc. of the Third Annual Symp. on the Theory of Computing*, 151-158, 1991.

[6] Davis, M. (ed.) *Solvability, Provability, Definability: The Collected Works of Emil L. Post*. Birkhäuser, Boston, 1994.

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

[8] Jensen, K. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volume 1. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[9] Johnson, D.S. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 67-161. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[10] Kanellakis, P.C. Elements of relational database theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 1073-1156. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[11] Kleene, S.C. Representation of events in nerve nets and finite automata. In C.E. Shannon, J. McCarthy, editors, *Automata Studies*, pages 3-42. Princeton University Press, 1956.

[12] Knuth, D.E. On the translation of languages from left to right. *Information and Control*, 8:6, 607-639, 1965.

[13] Levin, L.A. Universal sorting problems. *Problemy Peredaci Informacii*, 9, 115-116, 1973 (in Russian); English translation in: *Problems of Information Transmission*, 9, 265-266, 1973.

[14] Naur, P. *et al.* Report on the algorithmic language ALGOL 60. *Comm. ACM*, 3:5, 299-314, 1960.

[15] Petri, C.A. *Kommunikation mit Automaten*. Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[16] Reisig, W. *Petri Nets. An Introduction*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.