On the Unusual Effectiveness of Logic in Computer Science*

Joseph Y. Halpern† Robert Harper‡ Neil Immerman§ Phokion G. Kolaitis¶
Moshe Y. Vardi|| Victor Vianu**

January 2001

1 Introduction and Overview

In 1960, E.P. Wigner, a joint winner of the 1963 Nobel Prize for Physics, published a paper titled *On the Unreasonable Effectiveness of Mathematics in the Natural Sciences* [Wig60]. This paper can be construed as an examination and affirmation of Galileo's tenet that "The book of nature is written in the language of mathematics". To this effect, Wigner presented a large number of examples that demonstrate the effectiveness of mathematics in accurately describing physical phenomena. Wigner viewed these examples as illustrations of what he called *the empirical law of epistemology*, which asserts that the mathematical formulation of the laws of nature is both appropriate and accurate, and that mathematics is actually the *correct* language for formulating the laws of nature. At the same time, Wigner pointed out that the reasons for the success of mathematics in the natural sciences are not completely understood; in fact, he went as far as asserting that " ...the enormous usefulness of mathematics in the natural sciences is something bordering on the mysterious and there is no rational explanation for it."

In 1980, R.W. Hamming, winner of the 1968 ACM Turing Award for Computer Science, published a follow-up article, titled *The Unreasonable Effectiveness of Mathematics* [Ham80]. In this article, Hamming provided further examples manifesting the effectiveness of mathematics in the natural sciences. Moreover, he attempted to answer the "implied question" in Wigner's article: "Why is mathematics so unreasonably effective?" Although Hamming offered several partial explanations, at the end he concluded that on balance this question remains "essentially unanswered".

Since the time of the publication of Wigner's article, computer science has undergone a rapid, wide-ranging, and far-reaching development. Just as in the natural sciences, mathematics has been highly effective in computer science. In particular, several areas of mathematics, including linear algebra, number theory, probability theory, graph theory, and combinatorics, have been instrumental in the development of computer science. Unlike the natural sciences, however, computer science has also benefitted from an extensive and continuous interaction with logic. As a matter of fact, logic has turned out to be significantly more effective in computer science than it has been in mathematics. This is quite remarkable, especially since much of the impetus for the development of logic during the past one hundred years came from mathematics.

*This paper summarizes a symposium, by the same title, which was held at the 1999 Meeting of the American Association for the Advancement of Science. The authors wrote the following: Section 1 and 7 – Kolaitis, Section 2 – Immerman, Section 3 – Vianu, Section 4 - Harper, Section 5 - Halpern, and Section 6 – Vardi.

†Cornell University. Work partially supported by NSF Grant IRI-96-25901.

‡Carnegie-Mellon University Work partially supported by NSF Grant CCR-9502674 and DARPA Contract F19628-95-C-0050.

§University of Massachusetts, Amherst. Work partially supported by NSF grant CCR-9877078.

¶University of California, Santa Cruz. Work partially supported by NSF Grant CCR-9610257.

||Rice University. Work partially supported by NSF Grants CCR-9700061, CCR-9988322, IIS-9978135, and CCR-9988322.

**University of California, San Diego. Work partially supported by NSF Grant IIS-9802288.

Indeed, let us recall that to a large extent mathematical logic was developed in an attempt to confront the crisis in the foundations of mathematics that emerged around the turn of the 20th Century. Between 1900 and 1930, this development was spearheaded by Hilbert's Program, whose main aim was to formalize all of mathematics and establish that mathematics is *complete* and *decidable*. Informally, completeness means that all "true" mathematical statements can be "proved", whereas decidability means that there is a mechanical rule to determine whether a given mathematical statement is "true" or "false". Hilbert firmly believed that these ambitious goals could be achieved. Nonetheless, Hilbert's Program was dealt devastating blows during the 1930s. Indeed, the standard first-order axioms of arithmetic were shown to be incomplete by Gödel in his celebrated 1931 paper [Göd31]. Furthermore, A. Turing, A. Church, and A. Tarski demonstrated the undecidability of first-order logic. Specifically, the set of all valid first-order sentences was shown to be undecidable [Chu36, Tur37], whereas the set of all first-order sentences that are true in arithmetic was shown to be highly undecidable [Tar35].

Today, mathematical logic is a mature and highly sophisticated research area with deep results and a number of applications in certain areas of mathematics. All in all, however, it is fair to say that the interaction between logic and mathematics has been rather limited. In particular, mathematical logic is not perceived as one of the mainstream area of mathematics, and the "typical" mathematician usually knows little about logic. Along these lines, R.W. Hamming's judgment [Ham80] is not uncommon, albeit perhaps severe: "...we have had an intense study of what is called the foundations of mathematics ...It is an interesting field, but the main results of mathematics are impervious to what is found there."

In contrast, logic has permeated through computer science during the past thirty years much more than it has through mathematics during the past one hundred years. Indeed, at present concepts and methods of logic occupy a central place in computer science, insomuch that logic has been called "the calculus of computer science" [MW85]. Our goal in this article is to illustrate the effectiveness of logic in computer science by focusing on just a few of the many areas of computer science on which logic has had a definite and lasting impact. Specifically, the connections between logic and computational complexity will be highlighted in Section 2, the successful use of first-order logic as a database query language will be illustrated in Section 3, the influence of type theory in programming language research will be addressed in Section 4, the deployment of epistemic logic to reason about knowledge in multi-agent systems will be covered in Section 5, and the connections between logic and automated design verification will be presented in Section 6.

2 Descriptive Complexity

A fundamental issue in theoretical computer science is the computational complexity of problems. How much time and how much memory space is needed to solve a particular problem?

Let DTIME$[t(n)]$ be the set of problems that can be solved by algorithms that perform at most $O(t(n))$ steps for inputs of size n. The complexity class Polynomial Time (P) is the set of problems that are solvable in time at most some polynomial in n. Formally, P $= \bigcup_{k=1}^{\infty}$ DTIME$[n^k]$.

Some important computational problems appear to require more than polynomial time. An interesting class of such problems is contained in nondeterministic polynomial time (NP). A nondeterministic computation is one that may make arbitrary choices as it works. If any of these choices lead to an accept state, then we say the input is accepted.

The three-colorability problem — testing whether an undirected graph can have its vertices colored with three colors so that no two adjacent vertices have the same color — as well as hundreds of other well-known combinatorial problems are NP-complete. (See [GJ79] for a survey of many of these.) This means that not only are they in NP, but they are the "hardest problems" in NP: all problems in NP are reducible (in polynomial time) to each NP-complete problem. At present, the fastest known algorithm for any of these problems is exponential. An efficient algorithm for any one of these problems would translate to an efficient

algorithm for all of them. The P =?NP question, which asks whether P and NP coincide, is an example of our inability to determine what can or cannot be computed in a certain amount of computational resource: time, space, parallel time, etc.

Complexity theory typically considers yes/no problems. This is the examination of the difficulty of computing a particular bit of the desired output. Yes/no problems are properties of the input. The set of all inputs to which the answer is "yes" have the property in question. Rather than asking the complexity of checking if a certain input has a property T, in Descriptive Complexity we ask how hard is it to express the property T in some logic. It is plausible that properties that are harder to check might be harder to express. What is surprising is how closely logic mimics computation: descriptive complexity exactly captures the important complexity classes.

In Descriptive Complexity we view inputs as finite logical structures, e.g., a graph is a logical structure $\mathcal{A}_G = \langle \{1, 2, \ldots, n\}, E^G \rangle$ whose universe is the set of vertices and E^G is the binary edge relation.
Proviso: We will assume unless otherwise stated that a total ordering relation on the universe ($\leq$) is available.

In first-order logic we can express simple properties of our input structures. For example the following says that there are exactly two edges leaving every vertex.

$$(\forall x)(\exists yz)(\forall w)(y \neq z \wedge E(x,y) \wedge E(x,z) \wedge (E(x,w) \rightarrow w = y \vee w = z)) .$$

In second-order logic we also have variables X_i that range over relations over the universe. These variables may be quantified. A second-order existential formula (SO$\exists$) begins with second order existential quantifiers and is followed by a first-order formula. As an example, the following second-order existential sentence says that the graph in question is three-colorable. It does this by asserting that there are three unary relations, Red (R), Yellow (Y), and Blue (B), defined on the universe of vertices. It goes on to say that every vertex has some color and no two adjacent vertices have the same color.

$$(\exists R)(\exists Y)(\exists B)(\forall x)\Big[(R(x) \vee Y(x) \vee B(x)) \wedge (\forall y)\Big(E(x,y) \rightarrow \neg(R(x) \wedge R(y)) \wedge \neg(Y(x) \wedge Y(y)) \wedge \neg(B(x) \wedge B(y))\Big)\Big]$$

Descriptive Complexity began with the following theorem of R. Fagin. Observe that Fagin's Theorem characterizes the complexity class NP purely by logic, with no mention of machines or time,

Theorem 1 ([Fag74]) *A set of structures T is in* NP *iff there exists a second-order existential formula, Φ such that $T = \{\mathcal{A} \mid \mathcal{A} \models \Phi\}$. Formally,* NP = SO$\exists$.

Define CRAM$[t(n)]$ to be the set of properties checkable by concurrent-read, concurrent-write, parallel random-access machines using polynomially many processors in parallel time $O(t(n))$. FO, the set of first-order expressible properties, exactly captures the complexity class CRAM[1], i.e., constant parallel time. It is possible to increase the power of FO by allowing longer descriptions for longer inputs. Let FO$[t(n)]$ be those properties describable by a block of restricted quantifiers that may be iterated $t(n)$ times for inputs of size n.

Theorem 2 ([Imm88]) *For all constructible*[1] $t(n)$, FO$[t(n)]$ = CRAM$[t(n)]$.

[1]"Constructible" means that the function $n \mapsto t(n)$ can be computed in space $t(n)$. All but very bizarre functions are constructible. Another proviso of this theorem is that for $t(n) < \log n$, the first-order formulas may have access not only to ordering but to the addition and multiplication relations on the n-element universe.

Thus, parallel time corresponds exactly to first-order iteration, i.e., quantifier-depth. Rather than iterating blocks of quantifiers, a natural way to increase the power of first-order logic is by allowing inductive definitions. This is formalized via a *least-fixed-point* operator (LFP).

As an example, the reflexive, transitive closure $E^\star$ of the edge relation E can be defined via the following inductive definition,

$$E^\star(x,y) \;\equiv\; x = y \;\vee\; E(x,y) \;\vee\; (\exists z)(E^\star(x,z) \wedge E^\star(z,y)) \;.$$

Equivalently, this can be expressed using the least-fixed-point operator,

$$E^\star(x,y) \;\equiv\; \mathrm{LFP}_{R,x,y}(x = y \;\vee\; E(x,y) \;\vee\; (\exists z)(R(x,z) \wedge R(z,y)) \;.$$

It is exciting that the natural descriptive class FO(LFP) — first-order logic extended with the power to define new relations by induction — precisely captures polynomial time.

Theorem 3 ([Imm82, Imm86, Var82]) *A problem is in polynomial time iff it is describable in first-order logic with the addition of the least-fixed-point operator. This is equivalent to being expressible by a first-order formula iterated polynomially many times. Formally,* $\mathrm{P} = \mathrm{FO(LFP)} = \mathrm{FO}[n^{O(1)}]$.

Theorems 1 and 3 cast the P =?NP question in a different light. (In the following we are using the fact that if P were equal to NP, then NP would be closed under complementation. It would then follow that every second-order formula would be equivalent to a second-order existential one.)

Corollary 4 P *is equal to* NP *iff every second-order expressible property over finite, ordered structures is already expressible in first-order logic using inductive definitions. In symbols,* $(\mathrm{P} = \mathrm{NP}) \;\Leftrightarrow\; \mathrm{FO(LFP)} = \mathrm{SO}$.

The following theorem considers the arbitrary iteration of first-order formulas, which is the same as iterating them exponentially, and is more general than monotone iteration of first-order formulas. Such iteration defines the *partial*-fixed-point operator. The theorem shows that this allows the description of exactly all properties computable using a polynomial amount of space.

Theorem 5 ([Imm81, Imm82, Var82]) *A problem is in polynomial space iff it is describable in first logic with the addition of the partial-fixed-point operator. This is equivalent to being expressible by a first-order formula iterated exponentially. Formally,* $\mathrm{PSPACE} = \mathrm{FO(PFP)} = \mathrm{FO}[2^{n^{O(1)}}]$.

A refinement of Theorem 5 shows that the precise amount of space used can be characterized via the number of distinct variables in the relevant first-order formula, i.e., the number of descriptive variables captures space, for $k = 1, 2, \ldots,$ $\mathrm{DSPACE}[n^k] = \mathrm{VAR}[k+1]$, [Imm91].

Combinatorial games due to Ehrenfeucht and Fraïssé have been used to prove many inexpressibility results. These bounds provide useful insights but they do not separate relevant complexity classes because they are proved without the ordering relation [Ehr61, Fra54, Imm99]. No such lower bounds were known for separating the classes corresponding to P and PSPACE. Abiteboul and Vianu showed why, thus proving another fundamental relationship between logic and complexity. In the following, FO(wo$\leq$) means first-order logic without a given ordering relation.

Theorem 6 ([AV91]) *The following conditions are equivalent:*

1. $\mathrm{FO(wo}\leq\mathrm{)(LFP)} = \mathrm{FO(wo}\leq\mathrm{)(PFP)}$

2. FO(LFP) = FO(PFP)

3. P = PSPACE

Descriptive complexity reveals a simple but elegant view of computation. Natural complexity classes and measures such as polynomial time, nondeterministic polynomial time, parallel time, and space have natural descriptive characterizations. Thus, logic has been an effective tool for answering some of the basic questions in complexity. [2]

3 Logic as a Database Query Language

The database area is an important area of computer science concerned with storing, querying and updating large amounts of data. Logic and databases have been intimately connected since the birth of database systems in the early 1970's. Their relationship is an unqualified success story. Indeed, first-order logic (FO) lies at the core of modern database systems, and the standard query languages such as *Structured Query Language* (SQL) and *Query-By-Example* (QBE) are syntactic variants of FO. More powerful query languages are based on extensions of FO with recursion, and are reminiscent of the well-known fixpoint queries studied in finite-model theory (see Section 2). The impact of logic on databases is one of the most striking examples of the effectiveness of logic in computer science.

This section discusses the question of why FO has turned out to be so successful as a query language. We will focus on three main reasons:

- FO has syntactic variants that are easy to use. These are used as basic building blocks in practical languages like SQL and QBE.
- FO can be efficiently implemented using *relational algebra*, which provides a set of simple operations on relations expressing all FO queries. Relational algebra as used in the context of databases was introduced by Ted Codd in [Cod70]. It is related to Tarski's Cylindric Algebras [HMT71]. The algebra turns out to yield a crucial advantage when large amounts of data are concerned. Indeed, the realization by Codd that the algebra can be used to efficiently implement FO queries gave the initial impetus to the birth of relational database systems[3].
- FO queries have the potential for "perfect scaling" to large databases. If massive parallelism is available, FO queries can *in principle* be evaluated in *constant time*, independent of the database size.

A relational database can be viewed as a finite relational structure. Its signature is much like a relational FO signature, with the minor difference that relations and their coordinates have names. The name of a coordinate is called an *attribute*, and the set of attributes of a relation R is denoted $att(R)$. For example, a "beer drinker's" database might consist of the following relations:

frequents	*drinker*	*bar*
	Joe	*King's*
	Joe	*Molly's*
	Sue	*Molly's*
	...	...

serves	*bar*	*beer*
	King's	*Bass*
	King's	*Bud*
	Molly's	*Bass*
	...	...

The main use of a database is to query its data, e.g., find the drinkers who frequent only bars serving Bass. It turns out that each query expressible in FO can be broken down into a sequence of simple subqueries.

[2] This section is based in part on the article [Imm95]. See also the books [EF95, Imm99] for much more information about descriptive complexity.

[3] Codd received the ACM Turing Award for his work leading to the development of relational systems.

Each subquery produces an intermediate result, that may be used by subsequent subqueries. A subquery is of the form:

$$R := \exists \vec{x}\, (L_1 \wedge \ldots \wedge L_k),$$

where L_i is a literal $P(\vec{y})$ or $\neg P(\vec{y})$, P is in the input or is on the left-hand side of a previous subquery in the sequence, and R is not in the input and does not occur previously in the sequence. The meaning of such a subquery is to assign to R the result of the FO query $\exists \vec{x}\, (L_1 \wedge \ldots \wedge L_k)$ on the structure resulting from the evaluation of the previous subqueries in the sequence. The subqueries provide appealing building blocks for FO queries. This is illustrated by the language QBE, in which a query is formulated as just described. For example, consider the following query on the "beer drinker's" database:

Find the drinkers who frequent some bar serving Bass.

This can be expressed by a single query of the above form:

$$(\dagger)\ \ \mathit{answer} := \exists b\, (\mathit{frequents}(d, b) \wedge \mathit{serves}(d, \mathit{Bass})).$$

In QBE, the query is formulated in a visually appealing way as follows:

answer	*drinker*
	d

$\leftarrow$

frequents	*drinker*	*bar*
	d	b

serves	*bar*	*beer*
	b	*Bass*

Similar building blocks are used in SQL, the standard query language for relational database systems.

Let us consider again the query (†). The naive implementation would have us check, for each drinker d and bar b, whether $frequents(d, b) \wedge serves(d, Bass)$ holds. The number of checks is then the product of the number of drinkers and the number of bars in the database, which can be roughly n^2 in the size of the database. This turns out to be infeasible for very large databases. A better approach, and the one used in practice, makes use of relational algebra. Before discussing how this works, we informally review the algebra's operators. There are two set operators, $\cup$ (union) and $-$ (difference). The *selection* operator, denoted $\sigma_{cond}(R)$ extracts from R the tuples satisfying a condition *cond* involving (in)equalities of attribute values and constants. For example, $\sigma_{beer=Bass}(serves)$ produces the tuples in *serves* for which the beer is *Bass*. The *projection* operator, denoted $\pi_X(R)$, projects the tuples of relation R on a subset X of its attributes. The *join* operator, denoted by $R \bowtie Q$, consists of all tuples t over $att(R) \cup att(Q)$ such that $\pi_{att(R)}(t) \in R$ and $\pi_{att(Q)}(t) \in Q$. A last unary operator allows to *rename* an attribute of a relation without changing its contents.

Expressions constructed using relational algebra operators are called relational algebra queries. The query (†) is expressed using relational algebra as follows:

$$(\ddagger)\ \ \pi_{drinker}(\sigma_{beer=Bass}(frequents \bowtie serves)).$$

A result of crucial importance is that *FO and relational algebra express precisely the same queries*.

The key to the efficient implementation of relational algebra queries is twofold. First, individual algebra operations can be efficiently implemented using data structures called *indexes*, providing fast access to data. A simple example of such a structure is a binary search tree, which allows locating the tuples with a given attribute value in time $log(n)$, where n is the number of tuples. Second, algebra queries can be simplified using a set of *rewriting rules*. The query (‡) above can be rewritten to the equivalent but more efficient form:

$$\pi_{drinker}[frequents \bowtie \pi_{bar}(\sigma_{beer=Bass}(serves))].$$

The use of indexes and rewriting rules allows to evaluate the above query at cost roughly $n\ log(n)$ in the size of the database, which is much better than n^2. Indeed, for large databases this can make the difference between infeasibility and feasibility.

The FO queries turn out to be extremely well-behaved with respect to *scaling*. Given sufficient resources, response time can *in principle* be kept constant as the database becomes larger. The key to this remarkable property is parallel processing. Admittedly, *a lot* of processors are needed to achieve this ideal behaviour : polynomial in the size of the database. This is unlikely to be feasible in practice any time soon. The key point, however, is that FO query evaluation admits *linear* scaling; the speed-up is proportional to the number of parallel processors used.

Once again, relational algebra plays a crucial role in the parallel implementation of FO. Indeed, the algebra operations are *set oriented*, and thus highlight the intrinsic parallelism in FO queries. For example, consider the projection $\pi_X(R)$. The key observation is that one can project the tuples in R independently of each other. Given one processor for each tuple in R, the projection can be computed in constant time, independent of the number of tuples. As a second example, consider the join $R \bowtie Q$. This can be computed by joining all *pairs* of tuples from R and Q, independently of each other. Thus, if one processor is available for each pair, the join can be computed in constant time, independent on the number of tuples in R and Q.

Since each algebra operation can be evaluated in constant parallel time, each algebra query can also be evaluated in constant time. The constant depends only on the query and is independent of the size of the database. Of course, more and more processors are needed as the database grows.

In practice, the massive parallelism required to achieve perfect scaling is not available. Nevertheless, there are algorithms that can take optimal advantage of a given set of processors. It is also worth noting that the processors implementing the algebra need not be powerful, as they are only required to perform very specific, simple operations on tuples. In fact, it is sufficient to have processors that can implement the basic Boolean circuit operations. This fact is formalized by a result due to Immerman [Imm87] stating that FO is included in AC_0, the class of problems solvable by circuits of constant depth and polynomial size, with unbounded fan-in.

In conclusion, logic has proven to be a spectacularly effective tool in the database area. FO provides the basis for the standard query languages, because of its ease of use and efficient implementation via relational algebra. FO can achieve linear scaling, given parallel processing resources. Thus, its full potential as a query language remains yet to be realized.

A good introduction to the database area may be found in [SKS97], while [Ull88] provides a more in-depth presentation. The first text on database theory is [Mai83], followed more recently by [AHV95]. The latter text also described database query languages beyond FO, including fixpoint logics. An excellent survey of relational database theory is provided in [Kan91]. The relationship between finite-model theory and databases is discussed in [Via].

4 Type Theory in Programming Language Research

In the 1980's and 1990's the study of programming languages was revolutionized by a remarkable confluence of ideas from mathematical and philosophical logic and theoretical computer science. Type theory emerged as a unifying conceptual framework for the design, analysis, and implementation of programming languages. Type theory helps to clarify subtle concepts such as data abstraction, polymorphism, and inheritance. It provides a foundation for developing logics of program behavior that are essential for reasoning about programs. It suggests new techniques for implementing compilers that improve the efficiency and integrity of generated code.

Type theory is the study of type systems. Reynolds defines a *type system* to be a "syntactic discipline for enforcing levels of abstraction" [Rey85]. A type system is a form of context-sensitive grammar that imposes restrictions on the formation of programs to ensure that a large class of errors, those that arise from misinterpretation of values, cannot occur. Examples of such errors are: applying a function on the integers to a boolean argument; treating an integer as a pointer to a data structure or a region of executable code;

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2 \\
\textit{Expressions} & e & ::= & x \mid n \mid e_1 \circ e_2 \mid \mathtt{true} \mid \mathtt{false} \mid e_1 \mathtt{=} e_2 \mid \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \mid \\
 & & & \mathtt{fun}\, f\, (x \mathtt{:} \tau_1)\, \mathtt{:} \tau_2\, \mathtt{is}\, e \mid e_1\, (e_2) \\
\textit{Values} & v & ::= & x \mid n \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{fun}\, f\, (x \mathtt{:} \tau_1)\, \mathtt{:} \tau_2\, \mathtt{is}\, e
\end{array}
$$

The operator $\circ$ ranges over the arithmetic operations $+$, $-$, and $\times$.

The variable n ranges over numerals for the natural numbers.

The variables f and x are bound in the expression $\mathtt{fun}\, f\, (x \mathtt{:} \tau_1)\, \mathtt{:} \tau_2\, \mathtt{is}\, e$.

Figure 1: Abstract Syntax of MinML

$$
\frac{}{\Gamma \vdash x : \Gamma(x)}
$$

$$
\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{int} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1 \circ e_2 : \mathtt{int}}
$$

$$
\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}
$$

$$
\frac{\Gamma \vdash e_1 : \mathtt{int} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1 \mathtt{=} e_2 : \mathtt{bool}} \qquad \frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 : \tau}
$$

$$
\frac{\Gamma, f \mathtt{:} \tau_1 \to \tau_2, x \mathtt{:} \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}\, f\, (x \mathtt{:} \tau_1)\, \mathtt{:} \tau_2\, \mathtt{is}\, e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, (e_2) : \tau}
$$

Figure 2: Type System of MinML

over-writing a program's memory without regard to its purpose or validity; violating the assumptions of a procedure by calling it with too few arguments or arguments of the wrong type.

A type system is typically defined by an inductive definition of a *typing judgement* of the form $\Gamma \vdash e : \tau$. Here e is an expression, τ is its type, and Γ assigns types to the global variables that may occur within e. The typing judgement is defined to be the least three-place relation closed under a given collection of *typing rules* that determine whether or not an expression is well-typed.

The abstract syntax of an illustrative fragment of the ML language is given in Figure 1. Its type system is given in Figure 2. Note that the language constructs are grouped according to their type. Each type comes with expressions to denote its values together with operations for manipulating those values in a computation.

The rules governing the function type constructor exhibit an intriguing similarity to the introduction and elimination rules for implication in Gentzen's system of natural deduction. This similarity is not accidental: according to the *propositions-as-types principle* [CF58, CHS72, How80] there is an isomorphism between propositions and types with the property that the natural deduction proofs of a proposition correspond to the elements of its associated type. This principle extends to the full range of logical connectives and quantifiers, including those of second- and higher-order logic.

An *operational semantics* defines how to execute programs. It is useful to define the operational semantics of a language as a transition relation between states of an abstract machine, with certain states

$$\frac{e_1 \mapsto e_1'}{e_1 \circ e_2 \mapsto e_1' \circ e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1 \circ e_2 \mapsto v_1 \circ e_2'} \qquad \frac{(n = n_1 \circ n_2)}{n_1 \circ n_2 \mapsto n}$$

$$\frac{e_1 \mapsto e_1'}{e_1 = e_2 \mapsto e_1' = e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1 = e_2 \mapsto v_1 = e_2'} \qquad \frac{}{n_1 = n_2 \mapsto \begin{cases} \texttt{true} & \text{if } n_1 = n_2 \\ \texttt{false} & \text{if } n_1 \neq n_2 \end{cases}}$$

$$\frac{e_1 \mapsto e_1'}{e_1\,(e_2) \mapsto e_1'\,(e_2)} \qquad \frac{e_2 \mapsto e_2'}{v_1\,(e_2) \mapsto v_1\,(e_2')} \qquad \frac{(v = \texttt{fun}\, f\,(x\texttt{:}\tau_1)\,\texttt{:}\,\tau_2\,\texttt{is}\,e)}{v\,(v_1) \mapsto [v, v_1/f, x]e}$$

The notation $[v, v_1/f, x]e$ stands for the result of substitution v for free occurrences of f and v_1 for free occurrences of x in the expression e.

Figure 3: Operational Semantics of MinML

designated as final states. For the illustrative language of Figure 2 the states of the abstract machine are closed expressions; the final states are the fully-evaluated expressions. The transition relation is given in Figure 3 using Plotkin's technique of *structured operational semantics* [Plo81]. These rules constitute an inductive definition of the call-by-value evaluation strategy, in which function arguments are evaluated prior to application, and for which function expressions are fully evaluated.

One role of a type system is to preclude execution errors arising from misinterpretation of values.

Theorem 1: *[Type Soundness] If $\vdash e : \tau$, then either e is fully evaluated or there exists e' such that $\vdash e' : \tau$ and $e \mapsto e'$.*

A *type error* is an expression e such that e is not a value, yet there is no e' such that $e \mapsto e'$. In practice type errors correspond to illegal instructions or memory faults; the type soundness theorem ensures that well-typed programs never incur such errors.

The structure of more realistic programming languages can be described using very similar techniques. According to the type-theoretic viewpoint programming language "features" correspond to types. The following chart summarizes some of the main correspondences:

Concept	*Type*	*Values*	*Operations*
booleans	`bool`	true, false	conditional
integers	`int`	integer numerals	integer arithmetic
floating point	`float`	f.p. numerals	f.p. arithmetic
tuples	$\tau_1 \times \tau_2$	ordered pairs	component projection
disjoint union	$\tau_1 + \tau_2$	tagged values	case analysis
procedures	$\tau_1 \to \tau_2$	procedure definition	procedure call
recursive types	$\mu t.\tau$	heap pointers	traversal
polymorphism	$\forall t.\tau$	templates, generics	instantiation
data abstraction	$\exists t.\tau$	packages, modules	opening a package
mutable storage	t `ref`	storage cells	update, retrieve
tagging	`any`	tagged values	dynamic dispatch

Organizing programming languages by their type structure has a number of benefits. We mention a few salient ones here. First, language concepts are presented modularly, avoiding confusion or conflation

of distinct concepts. Traditional concepts such as "call-by-reference" parameter passing emerge instead as functions that take values of reference type. Second, type structure can be exploited to reason about program behavior. For example, the technique of *logical relations*, which interprets types as relations between values, may be used to characterize interchangeability of program fragments. Third, it becomes possible (as outlined above) to give a precise statement and proof of safety properties of a programming language. Moreover, the type annotations on programs form a certificate of safety that can be checked prior to execution of the program. Fourth, types may be exploited by a compiler to improve run-time efficiency and to provide a "self-check" on the integrity of the compiler itself [TMC+96].

5 Reasoning about Knowledge

The formal study of epistemic logic was initiated in the 1950s and led to Hintikka's seminal book *Knowledge and Belief* [Hin62]. The 1960s saw a flourishing of interest in the area in the philosophy community. More recently, reasoning about knowledge has been shown to play a key role in such diverse fields as distributed computing, game theory, and AI. The key concern is the connection between knowledge and action. What does a robot need to know in order to open a safe? What do processes need to know about other processes in order to coordinate an action? What do agents need to know about other agents to carry on a conversation? The formal model used by the philosophers provide the basis for an appropriate analysis of these questions.

We briefly review the model here, just to show how it can be used. The syntax is straightforward. We start with a set Φ of *primitive propositions*, where a primitive proposition $p \in \Phi$ represents a basic fact of interest like "it is raining in Spain" and a set $\{1, \ldots, n\}$ of agents. We then close off under conjunction and negation, as in propositional logic, and modal operators $K_1, \ldots, K_n$, E, and C, where $K_i\varphi$ is read "agent i knows φ", $E\varphi$ is read "everyone knows φ" and $C\varphi$ is read "φ is common knowledge. Thus, a statement such as $K_1K_2p \wedge \neg K_2K_1K_2p$ says "agent 1 knows agent 2 knows p, but agent 2 does not know that 1 knows that 2 knows p". More colloquially: "I know that you know it, but you don't know that I know that you know it."

The semantics for this logic, like that of other modal logics, is based on *possible worlds*. The idea is that, given her current information, an agent may not be able to tell which of a number of possible worlds describes the actual state of affairs. We say that the agent *knows* a fact φ if φ is true in all the worlds she considers possible. We formalize this intuition using *Kripke structures*. A *Kripke structure*[4] M for n agents is a tuple $(W, \mathcal{K}_1, \ldots, \mathcal{K}_n, \pi)$, where W is a set of possible worlds, $\mathcal{K}_i$ is a binary relation on W—that is, a set of pairs $(w, w') \in W \times W$, and π associates with each world a truth assignment to the primitive propositions (that is, $\pi(w)(p) \in \{\mathbf{true}, \mathbf{false}\}$ for each primitive proposition $p \in \Phi$ and world $w \in W$). Intuitively, $(v, w) \in \mathcal{K}_i$ if, in world v, agent i considers world w possible.

We can define $(M, w) \models \varphi$, read "φ is true in world w in structure M", by induction on the structure of formulas:

$(M, w) \models p$ (for a primitive proposition $p \in \Phi$) iff $\pi(w)(p) = \mathbf{true}$

$(M, w) \models \varphi \wedge \varphi'$ iff $(M, w) \models \varphi$ and $(M, w) \models \psi'$

$(M, w) \models \neg\varphi$ iff $(M, w) \not\models \varphi$

$(M, w) \models K_i\varphi$ iff $(M, w') \models \varphi$ for all $(w, w') \in \mathcal{K}_i$

$(M, w) \models E\varphi$ iff $(M, w) \models K_i\varphi$ for $i = 1, \ldots, n$

[4] Kripke structures are named after Saul Kripke, who introduced them in their current form in [Kri63], although the idea of possible worlds was in the air in the philosophy community in the 1950s.

$(M, w) \models C\varphi$ iff $(M, w) \models E^k\varphi$ for $k = 1, 2, 3, \ldots$, where E^k is defined inductively by taking $E^1\varphi := E\varphi$ and $E^{k+1}\varphi := EE^k\varphi$.

Note how the semantics of $K_i\varphi$ captures the intuition that agent i knows φ exactly if φ is true at all the worlds he considers possible. Clearly $E\varphi$ is true iff $K_i\varphi$ is true for each agent i. Finally, $C\varphi$ is true iff everyone knows φ, everyone knows that everyone knows, and so on.

What is the appropriate structure for analyzing a complicated multi-agent system? It turns out that a natural model for multi-agent systems can be viewed as a Kripke structure. (The phrase "system" is intended to be interpreted rather loosely here. Players in a poker game, agents conducting a bargaining session, robots interacting to clean a house, and processes in a computing system can all be viewed as multi-agent systems.) Assume that, at all times, each of the agents in the system can be viewed as being in some *local* state. Intuitively, the local state encapsulates all the relevant information to which the agent has access. In addition, there is an *environment*, whose state encodes relevant aspects of the system that are not part of the agents' local states. For example, if we are modeling a robot that navigates in some office building, we might encode the robot's sensor input as part of the robot's local state. If the robot is uncertain about its position, we would encode this position in the environment state. A *global state* of a system with n agents is an $(n + 1)$-tuple of the form $(s_e, s_1, \ldots, s_n)$, where s_e is the state of the environment and s_i is the local state of agent i.

A system is not a static entity; it changes over time. A *run* is a complete description of what happens over time in one possible execution of the system. For definiteness, we take time to range over the natural numbers. Thus, formally, a run is a function from the natural numbers to global states. Given a run r, $r(0)$ describes the initial global state of the system in r, $r(1)$ describes the next global state, and so on. We refer to a pair (r, m) consisting of a run r and time m as a *point*. If $r(m) = (s_e, s_1, \ldots, s_n)$, we define $r_i(m) = s_i, i = 1, \ldots, n$; thus, $r_i(m)$ is agent i's local state at the point (r, m).

The points in a system can be viewed as the states in a Kripke structure. Moreover, we can define a natural relation $\mathcal{K}_i$: agent i thinks (r', m') is possible at (r, m) if $r_i(m) = r'_i(m')$; that is, the agent has the same local state at both points. Intuitively, the local state encodes whatever the agent remembers about the run. An *interpreted system* $\mathcal{I}$ consists of a pair $(\mathcal{R}, \pi)$, where $\mathcal{R}$ is a system and π associates with each point in $\mathcal{R}$ a truth assignment to the primitive propositions in some appropriately chosen set Φ of primitive propositions. We can now define truth of epistemic formulas at a point in an interpreted system just as we did for a Kripke structure. That is, an interpreted system $\mathcal{I}$ can be viewed as a set of possible worlds, with the points acting as the worlds.[5] In particular, we have

$$(\mathcal{I}, r, m) \models K_i\varphi \text{ if } (\mathcal{I}, r', m') \models \varphi \text{ for all } (r', m') \text{ such that } r_i(m) = r'_i(m').$$

As an example of how this framework can be used in analyzing distributed protocols, consider the *coordinated attack problem*, from the distributed systems folklore [Gra78]. It abstracts a problem of data recovery management that arises when using standard protocols in database management called *commit protocols*. The following presentation is taken from [HM90]:

> Two divisions of an army are camped on two hilltops overlooking a common valley. In the valley awaits the enemy. It is clear that if both divisions attack the enemy simultaneously they will win the battle, whereas if only one division attacks it will be defeated. The generals do not initially have plans for launching an attack on the enemy, and the commanding general of the first division wishes to coordinate a simultaneous attack (at some time the next day). Neither general will decide to attack unless he is sure that the other will attack with him. The generals

[5] In an interpreted system we can also deal with temporal formulas, that talk about what happens at some point in the future, although that is unnecessary for the issues discussed in this section. See Section 6 for more discussion of temporal logic.

can only communicate by means of a messenger. Normally, it takes the messenger one hour to get from one encampment to the other. However, it is possible that he will get lost in the dark or, worse yet, be captured by the enemy. Fortunately, on this particular night, everything goes smoothly. How long will it take them to coordinate an attack?

Suppose the messenger sent by General A makes it to General B with a message saying "Let's attack at dawn". Will General B attack? Of course not, since General A does not know that B got the message, and thus may not attack. So General B sends the messenger back with an acknowledgment. Suppose the messenger makes it. Will General A attack? No, because now General B does not know that General A got the message, so General B thinks General A may think that B didn't get the original message, and thus not attack. So A sends the messenger back with an acknowledgment. But of course, this is not enough either.

In terms of knowledge, each time the messenger makes a transit, the *depth* of the generals' knowledge increases by one. Suppose we let the primitive proposition m stand for "A message saying 'Attack at dawn' was sent by General A." When General B gets the message, $K_B m$ holds. When A gets B's acknowledgment, $K_A K_B m$ holds. The next acknowledgment brings us to $K_B K_A K_B m$. Although more acknowledgments keep increasing the depth of knowledge, it is not hard to show that by following this protocol, the generals never attain common knowledge that the attack is to be held at dawn.

What happens if the generals use a different protocol? That does not help either. As long as there is a possibility that the messenger may get captured or lost, then common knowledge is not attained, even if the messenger in fact does deliver his messages. It would take us too far afield here to completely formalize these results (see [HM90] for details), but we can give a rough description. We say a *system $\mathcal{R}$ displays unbounded message delays* if, roughly speaking, whenever there is a run $r \in \mathcal{R}$ such that process i receives a message at time m in r, then for all $m' > m$, there is another run r' that is identical to r up to time m except that process i receives no messages at time m, and no process receives a message between times m and m'.

Theorem 2: [HM90] *In any run of a system that displays unbounded message delays, it can never be common knowledge that a message has been delivered.*

This says that no matter how many messages arrive, we cannot attain common knowledge of message delivery. But what does this have to do with coordinated attack? The fact that the generals have no initial plans for attack means that in the absence of message delivery, they will not attack. Since it can never become common knowledge that a message has been delivered, and message delivery is a prerequisite for attack, it is not hard to show that it can never become common knowledge among the generals that they are attacking. More precisely, let *attack* be a primitive proposition that is true precisely at points where both generals attack.

Corollary 3: *In any run of a system that displays unbounded message delays, it can never be common knowledge among the generals that they are attacking; i.e., $C(attack)$ never holds.*

We still do not seem to have dealt with our original problem. What is the connection between common knowledge of an attack and coordinated attack? As the following theorem shows, it is quite deep. Common knowledge is a prerequisite for coordination in any *system for coordinated attack*, that is, in any system that is the set of runs of a protocol for coordinated attack.

Theorem 4: [HM90] *In any system for coordinated attack, when the generals attack, it is common knowledge among the generals that they are attacking. Thus, if $\mathcal{I}$ is an interpreted system for coordinated attack, then at every point (r, m) of $\mathcal{I}$, we have*

$$(\mathcal{I}, r, m) \models attack \Rightarrow C(attack).$$

Putting together Corollary 3 and Theorem 4, we get the following corollary.

Corollary 5: *In any system for coordinated attack that displays unbounded message delays, the generals never attack.*

This negative result shows the power of the approach as a means of understanding the essence of coordination. There are positive results showing how this approach can be used to verify, analyze, and reason about distributed protocols. Of course, this brief discussion has only scratched the surface of the topic. For more details and further references, the interested reader should consult Fagin, Halpern, Moses, and Vardi's book [FHMV95].

6 Automated Verification of Semiconductor Designs

The recent growth in computer power and connectivity has changed the face of science and engineering, and is changing the way business is being conducted. This revolution is driven by the unrelenting advances in semiconductor manufacturing technology. Nevertheless, the U.S. semiconductor community faces a serious challenge: chip designers are finding it increasingly difficult to keep up with the advances in semiconductor manufacturing. As a result, they are unable to exploit the enormous capacity that this technology provides. The International Technology Roadmap for Semiconductors[6] suggests that the semiconductor industry will require productivity gains greater than the historical 20% per-year to keep up with the increasing complexity of semiconductor designs. This is referred to as the "design productivity crisis". As designs grow more complex, it becomes easier to introduce flaws into the design. Thus, designers use various validation techniques to verify the correctness of the design. Unfortunately, these techniques themselves grow more expensive and difficult with design complexity. As the validation process has begun to consume more than half the project design resources, the semiconductor industry has begun to refer to this problem as the "validation crisis".

Formal verification is a process in which mathematical techniques are used to guarantee the correctness of a design with respect to some specified behavior. Algorithmic formal-verification tools, based on *model-checking technology* [CES86, LP85, QS81, VW86] have enjoyed a substantial and growing use over the last few years, showing an ability to discover subtle flaws that result from extremely improbable events. While until recently these tools were viewed as of academic interest only, they are now routinely used in industrial applications, resulting in decreased time to market and increased product integrity [Kur97].

The first step in formal verification is to come up with a *formal specification* of the design, consisting of a description of the desired behavior. One of the more widely used specification languages for designs is *temporal logic* [Pnu77]. In *linear* temporal logics, time is treated as if each moment in time has a unique possible future. Thus, linear temporal formulas are interpreted over linear sequences, and we regard them as describing the behavior of a single computation of a system.

In the linear temporal logic LTL, formulas are constructed from a set $Prop$ of atomic propositions using the usual Boolean connectives as well as the unary temporal connective X ("next"), F ("eventually"), G ("always"), and the binary temporal connective U ("until"). For example, the LTL formula $G(\mathit{request} \rightarrow F\ \mathit{grant})$, which refers to the atomic propositions *request* and *grant*, is true in a computation precisely when every state in the computation in which *request* holds is followed by some state in the future in which *grant* holds. The LTL formula $G(\mathit{request} \rightarrow (\mathit{request}\ U\ \mathit{grant}))$ is true in a computation precisely if, whenever *request* holds in a state of the computation, it holds until a state in which *grant* holds is reached.

LTL is interpreted over *computations*, which can be viewed as infinite sequences of truth assignments to the atomic propositions; i.e., a computation is a function $\pi : N \rightarrow 2^{Prop}$ that assigns truth values to the elements of $Prop$ at each time instant (natural number). For a computation π and a point $i \in N$, the notation $\pi, i \models \varphi$ indicates that a formula φ holds at the point i of the computation π. For example, $\pi, i \models X\varphi$ iff

[6] `http://public.itrs.net/files/1999_SIA_Roadmap/Home.htm`

$\pi, i+1 \models \varphi$, and and $\pi, i \models \varphi U \psi$ iff for some $j \geq i$, we have $\pi, j \models \psi$ and for all k, $i \leq k < j$, we have $\pi, k \models \varphi$. We say that π *satisfies* a formula φ, denoted $\pi \models \varphi$, iff $\pi, 0 \models \varphi$. The connectives F and G can be defined in terms of the connective U: $F\varphi$ is defined as $\mathbf{true}\, U \varphi$, and $G\varphi$ is defined as $\neg F \neg \varphi$.

Designs can be described in a variety of formal description formalisms. Regardless of the formalism used, a *finite-state design* can be abstractly viewed as a *labeled transition system*, i.e., as a structure of the form $M = (W, W_0, R, V)$, where W is the finite set of states that the system can be in, $W_0 \subseteq W$ is the set of initial states of the system, $R \subseteq W^2$ is a transition relation that indicates the allowable state transitions of the system, and $V : W \rightarrow 2^{Prop}$ assigns truth values to the atomic propositions in each state of the system. (A labeled transition system is essentially a Kripke structure.) A *path* in M that *starts at* u is a possible infinite behavior of the system starting at u, i.e., it is an infinite sequence $u_0, u_1 \ldots$ of states in W such that $u_0 = u$, and $u_i \, R \, u_{i+1}$ for all $i \geq 0$.The sequence $V(u_0), V(u_1) \ldots$ is a *computation* of M that *starts at* u. It is the sequence of truth assignments visited by the path, The *language* of M, denoted $L(M)$ consists of all computations of M that start at a state in W_0. Note that $L(M)$ can be viewed as a language of infinite words over the alphabet 2^{Prop}. $L(M)$ can be viewed as an abstract description of a system, describing all possible "traces". We say that M *satisfies* an LTL formula φ if all computations in $L(M)$ satisfy φ, that is, if $L(M) \subseteq \text{models}(\varphi)$.

One of the major approaches to automated verification is the *automata-theoretic approach*, which underlies model checkers such as SPIN [Hol97] and Cadence SMV[7]. The key idea underlying the automata-theoretic approach is that, given an LTL formula φ, it is possible to construct a finite-state automaton A_φ on infinite words that accepts precisely all computations that satisfy φ [VW94]. The type of finite automata on infinite words we consider is the one defined by Büchi [Büc62]. A *Büchi automaton* is a tuple $A = (\Sigma, S, S_0, \rho, F)$, where Σ is a finite alphabet, S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function, and $F \subseteq S$ is a set of accepting states. A *run* of A over an infinite word $w = a_1 a_2 \cdots$, is a sequence $s_0 s_1 \cdots$, where $s_0 \in S_0$ and $s_i \in \rho(s_{i-1}, a_i)$ for all $i \geq 1$. A run $s_0, s_1, \ldots$ is *accepting* if there is some accepting state that repeats infinitely often, i.e., for some $s \in F$ there are infinitely many i's such that $s_i = s$. The infinite word w is *accepted* by A if there is an accepting run of A over w. The *language* of infinite words accepted by A is denoted $L(A)$. The following fact establishes the correspondence between LTL and Büchi automata: Given an LTL formula φ, one can build a Büchi automaton $A_\varphi = (\Sigma, S, S_0, \rho, , F)$, where $\Sigma = 2^{Prop}$ and $|S| \leq 2^{O(|\varphi|)}$, such that $L(A_\varphi)$ is exactly the set of computations satisfying the formula φ [VW94].

This correspondence reduces the verification problem to an automata-theoretic problem as follows [VW86]. Suppose that we are given a system M and an LTL formula φ. We check whether $L(M) \subseteq \text{models}(\varphi)$ as follows: (1) construct the automaton $A_{\neg\varphi}$ that corresponds to the *negation* of the formula φ, (2) take the *cross product* of the system M and the automaton $A_{\neg\varphi}$ to obtain an automaton $A_{M,\varphi}$, such that $L(A_{M,\varphi}) = L(M) \cap L(A_{\neg\varphi})$, and (3) check whether the language $L(A_{M,\varphi})$ is nonempty, i.e., whether $A_{M,\varphi}$ accepts *some* input. If it does not, then the design is correct. If it does, then the design is incorrect and the accepted input is an incorrect computation. The incorrect computation is presented to the user as a finite trace, possibly followed by a cycle. Thus, once the automaton $A_{\neg\varphi}$ is constructed, the verification task is reduced to automata-theoretic problems, namely, intersecting automata and testing emptiness of automata, which have highly efficient solutions [Var96]. Furthermore, using data structures that enable compact representation of very large state space makes it possible to verify designs of significant complexity [BCM+92].

The linear-time framework is not limited to using LTL as a specification language. There are those who prefer to use automata on infinite words as a specification formalism [VW94]; in fact, this is the approach of COSPAN [Kur94]. In this approach, we are given a design represented as a finite transition system M and a property represented by a Büchi (or a related variant) automaton P. The design is correct if

[7] `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`

all computations in $L(M)$ are accepted by P, i.e., $L(M) \subseteq L(P)$. This approach is called the *language-containment* approach. To verify M with respect to P, we: (1) construct the automaton P^c that *complements* P, (2) take the product of the system M and the automaton P^c to obtain an automaton $A_{M,P}$, and (3) check that the automaton $A_{M,P}$ is nonempty. As before, the design is correct iff $A_{M,P}$ is empty. Thus, the verification task is again reduced to automata-theoretic problems, namely intersecting and complementing automata and testing emptiness of automata.

Over the last few years, automated formal verification tools, such as model checkers, have shown their ability to provide a thorough analysis of reasonably complex designs [Goe97]. Companies such as AT&T, Cadence, Fujitsu, HP, IBM, Intel, Motorola, NEC, SGI, Siemens, and Sun are using model checkers increasingly on their own designs to reduce time to market and ensure product quality.

7 Concluding Remarks

It should be made clear that we are not the first ones to single out the effectiveness of logic in computer science. In fact, already back in 1988 M. Davis wrote an eloquent essay on the *Influences of Logic in Computer Science* [Dav88], which begins by stating that "When I was a student, even the topologists regarded mathematical logicians as living in outer space. Today the connections between logic and computers are a matter of engineering practice at every level of computer organization." Davis proceeds then to examine how certain fundamental concepts from logic have found crucial uses in computer science. In particular, Davis adresses the connections between Boolean logic and digital circuits, discusses the influence of logic on the design of programming languages, and comments on the relationship between logic programming and automated theorem-proving. More recently, Davis wrote a book titled *The Universal Computer* [Dav00] in which he presents the fundamental connection between logic and computation by tracing the lives and contributions of Leibniz, Boole, Frege, Cantor, Hilbert, Gödel, and Turing.

The effectiveness of logic in computer science is not by any means limited to the areas mentioned in here. As a matter of fact, it spans a wide spectrum of areas, from artificial intelligence to software engineering. Overall, logic provides computer science with both a unifying foundational framework and a powerful tool for modeling and reasoning about aspects of computation. Computer science is concerned with phenomena that are usually described as "synthetic", because for the most part they are a human creation, unlike the phenomena studied in the natural sciences. This difference between computer science and the natural sciences can provide an explanation as to why the use of logic in computer science is both appropriate and successful. Thus, the effectiveness of logic in computer science is perhaps not mysterious or unreasonable, but still quite remarkable and unusual.

Acknowledgments. We are grateful to Susan Landau for suggesting to us the topic of this article.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading-Massachusetts, 1995.

[AV91] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 209–219, 1991.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[CHS72] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume 2*. North-Holland, 1972.

[Chu36] A. Church. A note on the Entscheidungsproblem. *J. of Symbolic Logic*, 1:40–44, 1936.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Dav88] M. Davis. Influences of mathematical logic on computer science. In R. Herken, editor, *The Universal Turing Machine: A Half-Century Survey*, pages 315–326. Oxford University Press, 1988.

[Dav00] M. Davis. *The Universal Computer*. Norton, 2000.

[EF95] H.D. Ebbinghaus and J. Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer-Verlag, 1995.

[Ehr61] A. Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fund. Math.*, 49:129–141, 1961.

[Fag74] R. Fagin. Generalized first–order spectra and polynomial–time recognizable sets. In R. M. Karp, editor, *Complexity of Computation, SIAM-AMS Proceedings, Vol. 7*, pages 43–73, 1974.

[FHMV95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.

[Fra54] R. Fraïssé. Sur quelques classifications des systèmes de relations. *Publ. Sci. Univ. Alger. Sér. A*, 1:35–182, 1954.

[GJ79] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. Freeman and Co., San Francisco, 1979.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1989.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[Goe97] R. Goering. Model checking expands verification's scope. *Electronic Engineering Today*, February 1997.

[Gra78] J. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, Vol. 66. Springer-Verlag, Berlin/New York, 1978. Also appears as IBM Research Report RJ 2188, 1978.

[Ham80] R.W. Hamming. The unreasonable effectiveness of mathematics. *American Mathematical Monthly*, 87:81–90, 1980.

[Hin62] J. Hintikka. *Knowledge and Belief*. Cornell University Press, Ithaca, N.Y., 1962.

[HM90] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[HMT71] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras Part I*. North Holland, 1971. Part II, North Holland, 1985.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[How80] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[Imm81] N. Immerman. Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences*, 22(3):384–406, 1981.

[Imm82] N. Immerman. Upper and lower bounds for first-order expressibility. *Journal of Computer and System Sciences*, 25:76–98, 1982.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Imm87] N. Immerman. Languages which capture complexity classes. *SIAM J. on Computing*, 16(4):760–778, 1987.

[Imm88] N. Immerman. Nondeterministic space is closed under complement. *SIAM Journal on Computing*, 17:935–938, 1988.

[Imm91] N. Immerman. Dspace[n^k] = var[$k+1$]. In *Proc. 6th IEEE Symp. on Structure in Complexity Theory*, pages 334–340, 1991.

[Imm95] N. Immerman. Descriptive complexity: a logician's approach to computation. *Notices of the American Mathematical Society*, 42(10):1127–1133, 1995.

[Imm99] N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1999.

[Jac99] B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, 1999.

[Kan91] P. C. Kanellakis. Elements of relational database theory. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1074–1156. Elsevier, 1991.

[Kri63] S. Kripke. A semantical analysis of modal logic I: normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963. Announced in *Journal of Symbolic Logic*, **24**, 1959, p. 323.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[Kur97] R.P. Kurshan. Formal verification in a commercial setting. *The Verification Times*, 1997.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specifications. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 97–107, 1985.

[Mai83] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[Mit96] J.C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. MIT Press, 1996.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, Naples, Italy, 1984.

[MW85] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming*. Addison-Wesley, 1985.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI–FN–19, Computer Science Department, Aarhus University, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rey85] John C. Reynolds. Three approaches to type structure. In *TAPSOFT*. Springer-Verlag, 1985.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[SKS97] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 1997.

[Tar35] A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1935.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[Tur37] A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Math. Soc. Ser. 3*, 42:230–265, 1936/37.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1988.

[Var82] M.Y. Vardi. The complexity of relational query languages. In *Proc. 14th ACM Symp. on Theory of Computing*, pages 137–146, San Francisco, 1982.

[Var96] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, Berlin, 1996.

[Via] V. Vianu. Databases and finite-model theory. In *Descriptive Complexity and Finite Models*.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[Wig60] E.P. Wigner. The unreasonable effectiveness of mathematics in the natural sciences. *Comm. on Pure and Applied Math.*, 13:1–14, 1960.