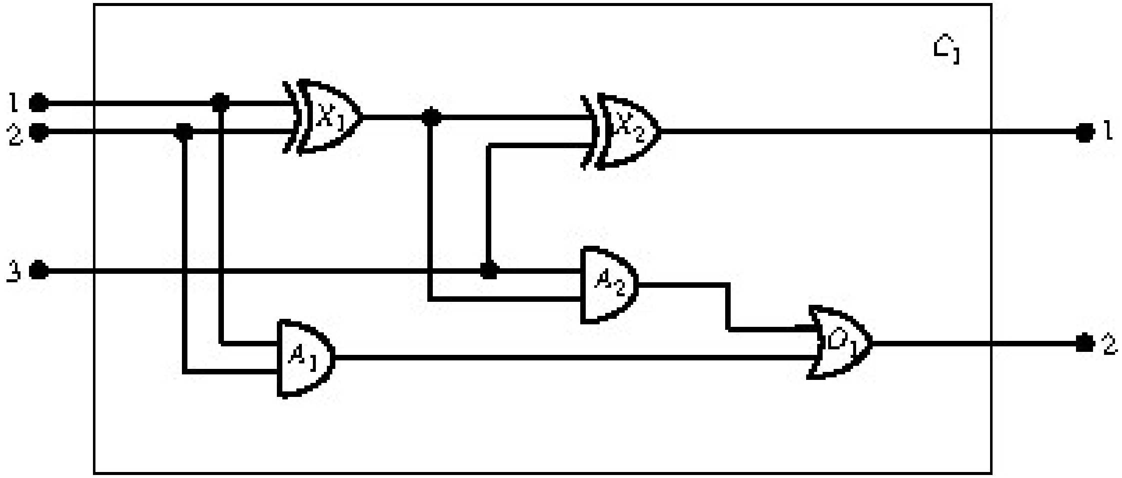
Practical Applications of FOL, Resolution Theorem Provers

- Applied to synthesis and verification of both HW and SW
  - Used in fields of HW design, programming languages, and SW engineering (in addition to AI)
- For HW:
  - Axioms describe interactions between signals and circuit elements
  - Have been used to verify entire CPUs, including timing properties
- For SW:
  - Reasoning about programs is similar to reasoning about actions
  - Formal synthesis of algorithms was an early use of theorem provers
  - SW verification is commonly done with theorem proving
    - E.g., for spacecraft control, verification of RAS public key encryption, string matching, etc.
  - Fully automated techniques for general-purpose programming are not yet feasible
    - But, some algorithms have been generally deduced using theorem proving

(1) HW Example: Verifying Circuits (Sect. 8.4.2)

- Given a circuit, we could ask:
  - Does it work properly?
  - Given certain inputs, what is the output
  - Does the circuit contain feedback loops?
  - Etc.



Digital circuit, purporting to be a 1-bit full adder.
First 2 inputs are bits to be added; 3rd bit is carry bit.
First output is sum, 2nd output is carry bit for the next adder.

(1) HW Example: Verifying Circuits (con't.)

- To design, first decide what the relevant knowledge is:
  - Circuits consist of wires and gates
  - Signals flow along wires to input terminals of gates
  - Each gate produces a signal on the output terminal that flows along another wire
  - There are 4 types of gates that transform their inputs differently: AND, OR, XOR, NOT
  - All gates have 1 output terminal

- To reason about functionality and connectivity:
  - We just need to talk about the connections between terminals
  - Don't have to bother with paths of wires, or junctions where they come together

- If we wanted to verify timing, or faulty circuits, etc., then we would add that info to our knowledge base

(1) HW Example: Verifying Circuits (con’t.)

- Next, decide on vocabulary:
  - Constants:
    - AND, OR, NOT, XOR, 1, 0, Nothing
  - Predicates:
    - Gate(x)
    - Type(x)
    - Circuit(x)
    - In(1, x) // refers to first input terminal for gate x
    - Out(1, x) // refers to first output terminal for gate x
    - Arity(c,i,j) // circuit c has i input and j output terminals
    - Connected(t_1, t_2) // says terminals t_1 and t_2 are connected
    - Signal(t) // denotes signal value (0 or 1) for terminal t

(1) HW Example: Verifying Circuits (con't.)

- Next, encode general domain knowledge (should be just a few general rules):
  - Gates, terminals, signals, gate types, and Nothing are all distinct:
    - $\forall$ g, t Gate(g) $\wedge$ Terminal(t) $\Rightarrow$ g $\neq$ t $\neq$ 1 $\neq$ 0 $\neq$ 2 $\neq$ OR $\neq$ AND $\neq$ XOR $\neq$ NOT $\neq$ Nothing
  - If 2 terminals are connected, then they have the same signal:
    - $\forall t_1, t_2$ Terminal(t_1) $\wedge$ Terminal(t_2) $\wedge$ Connected(t_1, t_2) $\Rightarrow$ Signal(t_1) = Signal(t_2)
  - The signal at every terminal is either 1 or 0:
    - $\forall$t Terminal(t) $\Rightarrow$ Signal(t) = 1 $\vee$ Signal(t) = 0
  - Connected is commutative:
    - $\forall t_1, t_2$ Connected(t_1, t_2) $\Leftrightarrow$ Connected(t_2, t_1)
  - There are 4 types of gates:
    - $\forall$ g Gate(g) $\wedge$ k = Type(g) $\Rightarrow$ k = AND $\vee$ k = OR $\vee$ k = XOR $\vee$ k = NOT

(1) HW Example: Verifying Circuits (con’t.)

- An AND gate’s output is 0 iff any of its inputs is 0:
  - $\forall g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{AND} \Rightarrow \text{Signal}(\text{Out}(1,g)) = 0 \Leftrightarrow \exists n\ \text{Signal}(\text{In}(n,g)) = 0$
- An OR gate’s output is 1 iff any of its inputs is 1:
  - $\forall g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{OR} \Rightarrow \text{Signal}(\text{Out}(1,g)) = 1 \Leftrightarrow \exists n\ \text{Signal}(\text{In}(n,g)) = 1$
- An XOR gate’s output is 1 iff its inputs are different:
  - $\forall g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{XOR} \Rightarrow$

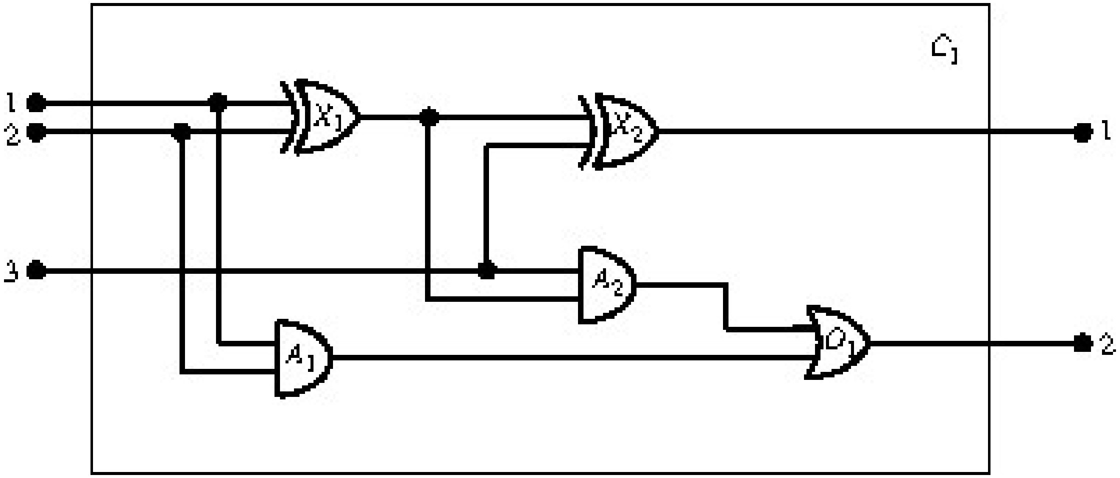
    $\text{Signal}(\text{Out}(1,g)) = 1 \Leftrightarrow \text{Signal}(\text{In}(1, g)) \neq \text{Signal}(\text{In}(2,g))$
- A NOT gate’s output is different from its input:
  - $\forall g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{NOT} \Rightarrow \text{Signal}(\text{Out}(1,g)) \neq \text{Signal}(\text{In}(1, g))$
- The gates (except for NOT) have 2 inputs and 1 output:
  - $\forall g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{NOT} \Rightarrow \text{Arity}(g,1,1)$
  - $\forall g\ \text{Gate}(g) \wedge k = \text{Type}(g) \wedge (k = \text{AND} \vee k = \text{OR} \vee k = \text{XOR}) \Rightarrow \text{Arity}(g,2,1)$
- A circuit has terminals, up to its input and output arity, and nothing beyond its arity:
  - $\forall c, i, j\ \text{Circuit}(c) \wedge \text{Arity}(c,i,j) \Rightarrow$

    $\forall n\ (n \leq i \Rightarrow \text{Terminal}(\text{In}(c,n))) \wedge (n > i \Rightarrow \text{In}(c,n) = \text{Nothing}) \wedge$

    $\forall n\ (n \leq j \Rightarrow \text{Terminal}(\text{Out}(c,n))) \wedge (n > j \Rightarrow \text{Out}(c,n) = \text{Nothing})$
- Gates are circuits:
  - $\forall g\ \text{Gate}(g) \Rightarrow \text{Circuit}(g)$

(1) HW Example: Verifying Circuits (con't.)

- Now, encode specific problem instance:



Circuit(C_1) ∧ Arity(C_1, 3, 2)
Gate(X_1) ∧ Type(X_1) = XOR
Gate(X_2) ∧ Type(X_2) = XOR
Gate(A_1) ∧ Type(A_1) = AND
Gate(A_2) ∧ Type(A_2) = AND
Gate(O_1) ∧ Type(O_1) = OR

Connected(Out(1, X_1), In(1, X_2))
Connected(Out(1, X_1), In(2, A_2))
Connected(Out(1, A_2), In(1, O_1))
Connected(Out(1, A_1), In(2, O_1))
Connected(Out(1, X_2), Out(1, C_1))
Connected(Out(1, O_1), Out(2, C_1))

Connected(In(1, C_1), In(1, X_1))
Connected(In((1, C_1), In(1, A_1))
Connected(In((2, C_1), In(2, X_1))
Connected(In((2, C_1), In(2, A_1))
Connected(In((3, C_1), In(2, X_2))
Connected(In((1, C_1), In(1, A_2))

(1) HW Example: Verifying Circuits (con't.)

- Finally, we can pose queries to inference procedure:
  - What combinations of inputs would cause the first output of C_1 (the sum bit) to be 0 and the second output of C2 (the carry bit) to be 1?

    $\exists\, i_1, i_2, i_3\ \ \text{Signal}(\text{In}(1, C_1)) = i_1 \ \wedge\ \text{Signal}(\text{In}(2, C_1)) = i_2 \ \wedge\ \text{Signal}(\text{In}(3, C_1)) = i_3$
    $\wedge\ \text{Signal}(\text{Out}(1, C_1)) = 0 \ \wedge\ \text{Signal}(\text{Out}(2, C_1)) = 1$

    - The answers are substitutions to variables such that the resulting sentence is entailed by the knowledge base:
      - Answers are $\{i_1/1, i_2/1, i_3/0\}$, $\{i_1/1, i_2/0, i_3/1\}$, $\{i_1/0, i_2/1, i_3/1\}$
  - What are the possible sets of values of all the terminals for the adder circuit?

    $\exists\, i_1, i_2, i_3, o_1, o_2\ \ \text{Signal}(\text{In}(1, C_1)) = i_1 \ \wedge\ \text{Signal}(\text{In}(2, C_1)) = i_2 \ \wedge\ \text{Signal}(\text{In}(3, C_1)) = i_3$
    $\wedge\ \text{Signal}(\text{Out}(1, C_1)) = o_1 \ \wedge\ \text{Signal}(\text{Out}(2, C_1)) = o_2$

    - The answers give a complete I/O table for the device, which can be used to confirm that it properly adds its inputs.

Practical Applications of FOL, Resolution Theorem Provers

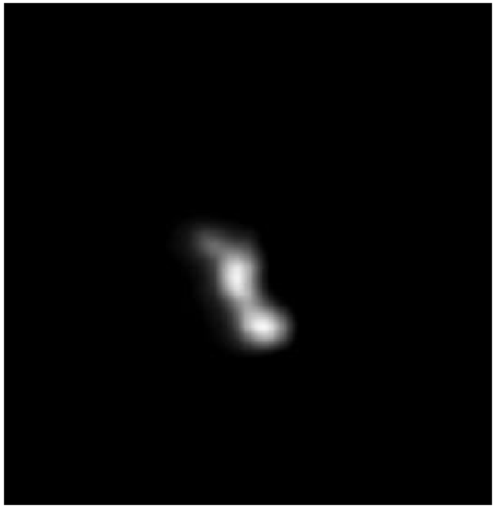
- Applied to synthesis and verification of both HW and SW
  - Used in fields of HW design, programming languages, and SW engineering (in addition to AI)
- For HW:
  - Axioms describe interactions between signals and circuit elements
  - Have been used to verify entire CPUs, including timing properties
- For SW:
  - Reasoning about programs is similar to reasoning about actions
  - Formal synthesis of algorithms was an early use of theorem provers
  - SW verification is commonly done with theorem proving
    - E.g., for spacecraft control, verification of RAS public key encryption, string matching, etc.
  - Fully automated techniques for general-purpose programming are not yet feasible
    - But, some algorithms have been generally deduced using theorem proving

(2) SW Example: Verifying Spacecraft Control

- Havelund, et al (2000), NASA Ames Research Center
  - Used formal methods to verify deep space autonomy flight software
  - Approach found several concurrency errors
    - Developers believe these errors would *not* have been found through "usual" testing

- Remote Agent (RA) autonomous spacecraft controller, successfully demonstrated in flight on Deep Space 1 (1999)
  - RA is complex, concurrent SW system employing several automated reasoning engines using AI
  - Formal verification is critical to SW acceptance by science mission managers

Deep Space 1 – conducted fly-by of asteroid 9969 Braille



Asteroid 9969 Braille, as imaged by Deep Space 1

(2) SW Example: Verifying Spacecraft Control (con't.)

- During development (1997), a *subset* of the RA executive was modeled and verified, discovering several concurrency errors
- But, during flight, another concurrency error occurred:
  - Activation of error depended on a priori unlikely scheduling conditions between concurrent tasks
  - Error had not appeared in over 300 hours of system-level testing on JPL's flight system testbed
  - Flight conditions under which error occurred were not anticipated during testing
  - Problem was solved by engineers
  - However, lesson learned was that full code verification is needed, along with easy-to-use tools to do so

(2) SW Example: Verifying Spacecraft Control (con't.)

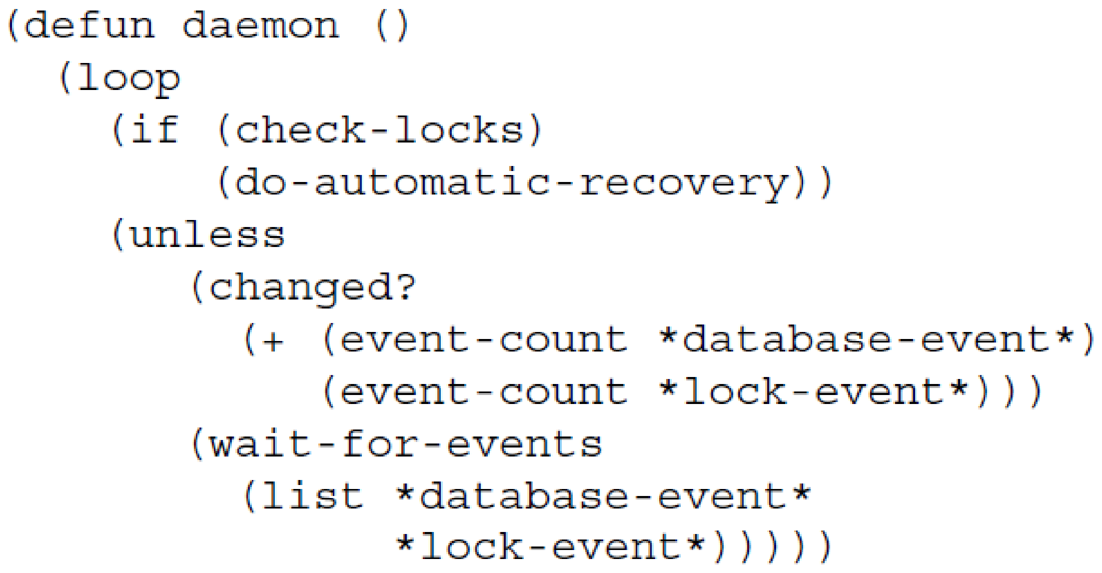
- Remote Agent (RA) controller:
  - Planner and Scheduler: Given a mission goal, it produces sequences of tasks for achieving the goal using available system resources.
  - Smart Executive: Receives plan from planner/scheduler, and then commands spacecraft to take necessary actions to achieve and maintain specified spacecraft states
  - Mode Identification and Recovery: Monitors state of spacecraft, detects and diagnoses failures, and suggests recovery actions to Executive
- Verification work: focused on Smart Executive
  - Includes multi-threaded operating systems
  - Prolog-like AI languages based on sub-goals
  - Written in multi-threaded LISP

(2) SW Example: Verifying Spacecraft Control (con't.)

- RA Executive:
  - Supports execution of tasks, which often require specific properties to hold during its execution
  - When task is started, it tries to achieve properties on which it depends; then it begins
  - Several tasks may try to achieve conflicting properties
    - E.g., one task might turn on a camera; another task might turn it off
  - To prevent conflicts, a task has to lock (in a lock table) any property it wants to achieve
    - Once a property is locked, it can be achieved by the task locking the property
  - Problem: property by be unexpectedly broken during execution
    - Thus, during execution, a database is maintained of all properties that are actually true at any time
    - Inconsistency can be detected by comparing database with lock table
    - Tasks depending on broken property must be interrupted
  - A daemon monitors this consistency
    - This daemon contained the concurrency errors

(2) SW Example: Verifying Spacecraft Control (con’t.)

- Daemon code:



```
(defun daemon ()
  (loop
    (if (check-locks)
        (do-automatic-recovery))
    (unless
       (changed?
         (+ (event-count *database-event*)
            (event-count *lock-event*)))
       (wait-for-events
         (list *database-event*
               *lock-event*)))))
```

- Code checked for two properties:
  - Release property: A task releases all of its locks before it terminates
  - Abort property: If an inconsistency occurs between the database and an entry in the lock table, then all tasks that rely on the lock will be terminated, either by themselves or by the daemon

(2) SW Example: Verifying Spacecraft Control (con't.)

- Verification of the two properties led to direct discovery of 5 programming errors:
  - One breaking the release property
  - Three breaking the abort property
  - One being a non-serious efficiency problem where code was executed twice instead of once
- Example of error:
  - Daemon is prompted to perform check of lock table
  - Finds everything consistent and checks the event counters to see if there have been any new events
  - This isn't the case, and the daemon decides to wait for events
  - At this point, an inconsistency is introduced, and a signal is sent by the environment, causing event counter for the database event to be increased
  - Change in counter is not detected by daemon, since it has already decided to wait
    - A solution would be to enclose test and wait in same critical section
    - But, how to detect these sorts of errors when not coded properly to begin with?

(2) SW Example: Verifying Spacecraft Control (con’t.)

- Tools used for model checking:
  - PROMELA verification modeling language
    - Used to model the software
  - SPIN model checker
    - General tool for verifying correctness of distributed SW
    - Verifies properties stated using Linear Temporal Logic

(3) Algorithm Example: Verifying RSA Encryption

- Boyer and Moore, 1984, used Proof Checking to verify the RSA encryption algorithm
- Statement of problem:
  - CRYPT(M, e, n) is encryption of message M with key (e,n).
  - CRYPT has 3 important properties:
    1) It is easy to compute CRYPT(M, e, n) = M^e mod n
    2) CRYPT is invertible

       i.e., if M is encrypted with key (e, n) and then decrypted with key (d, n), the result is M; precisely: CRYPT(CRYPT(M, e, n),d,n) = M
    3) Publicly revealing CRYPT and (e, n) does not reveal an easy way to compute (d, n).
  - Rivest, Shamir, and Adleman (1978) proved first 2 properties, but not 3rd. (Instead, they stated informally that, since there is no known algorithm for efficiently factoring large composites, the security property of CRYPT is obtained by constructing n as the product of two very large primes)
- Work of Boyer and Moyer was to show a mechanical proof of properties 1 and 2

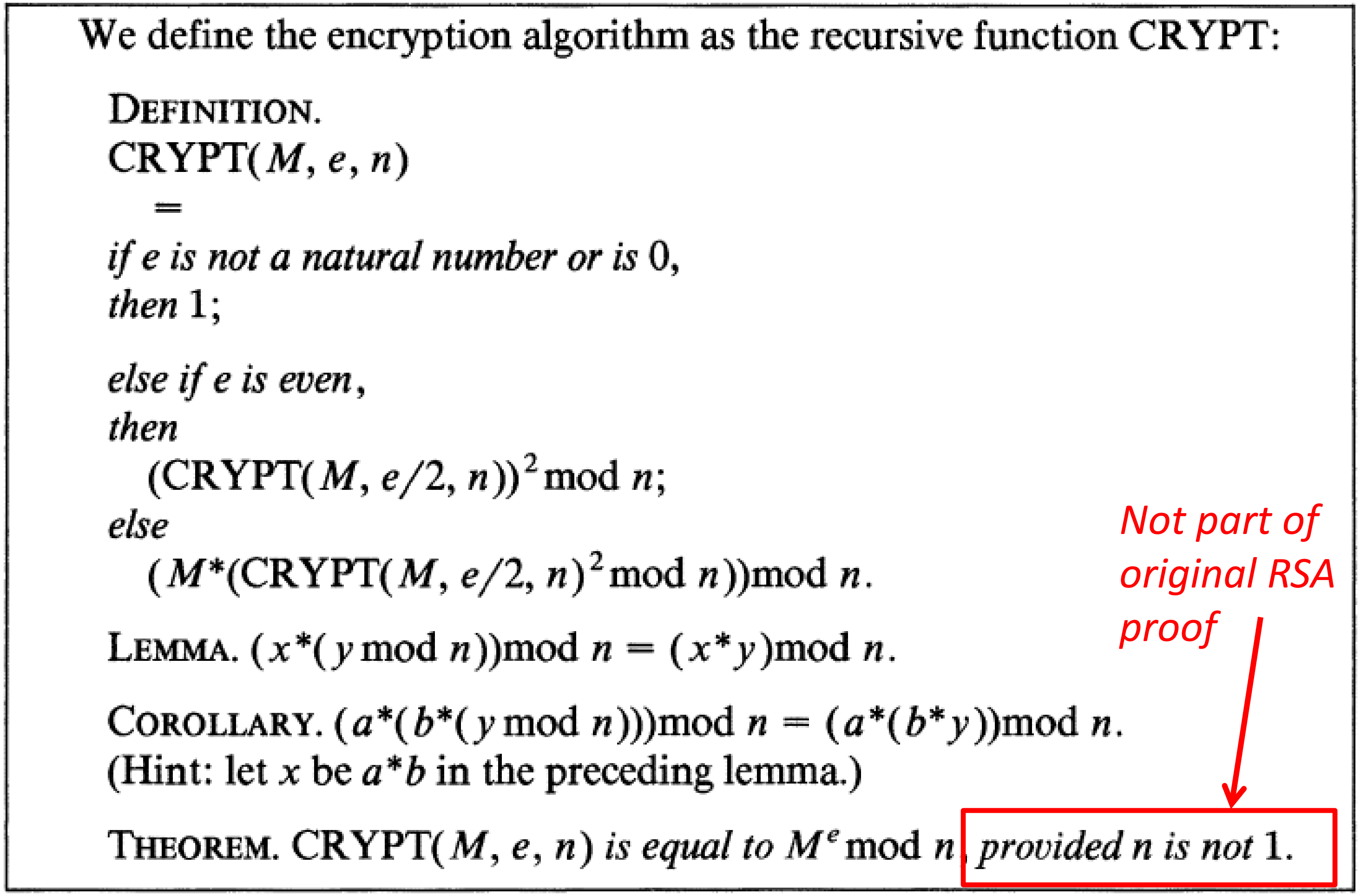
(3) Algorithm Example: Verifying RSA Encryption (con't.)

- Theorem-prover used:
  - Quantifier-free first order logic:
    - With equality, recursively defined functions, mathematical induction, and inductively constructed objects such as natural numbers and finite sequences

- Main proof techniques:
  - Simplification – use rewrite rules to simplify expressions
    - Example: $\text{prime}(p) \rightarrow [p \mid a*b \leftrightarrow (p|a \vee p|b)]$
  - Elimination of undesirable function symbols
    - Example: For natural number i and positive integer j, there exist natural numbers $r < j$ and q such that $i = r + qj$. Thus, can replace (i mod j) with r and i/j with q
  - Strengthening the conjecture to be proved
  - Induction

(3) Algorithm Example: Verifying RSA Encryption (con't.)

- Property 1: Rivest, Shamir, and Adelman proved that M^e mod n is easy to compute by exhibiting an algorithm for computing it in order lg(e) steps.
- Boyer and Moore used rules of math (in logic form) to verify the algorithm



We define the encryption algorithm as the recursive function CRYPT:

DEFINITION.
CRYPT(M, e, n)
=
if e is not a natural number or is 0,
then 1;

else if e is even,
then
(CRYPT(M, $e/2$, n))^2 mod n;
else
(M*(CRYPT(M, $e/2$, n)^2 mod n))mod n.

LEMMA. (x*(y mod n))mod n = (x*y)mod n.

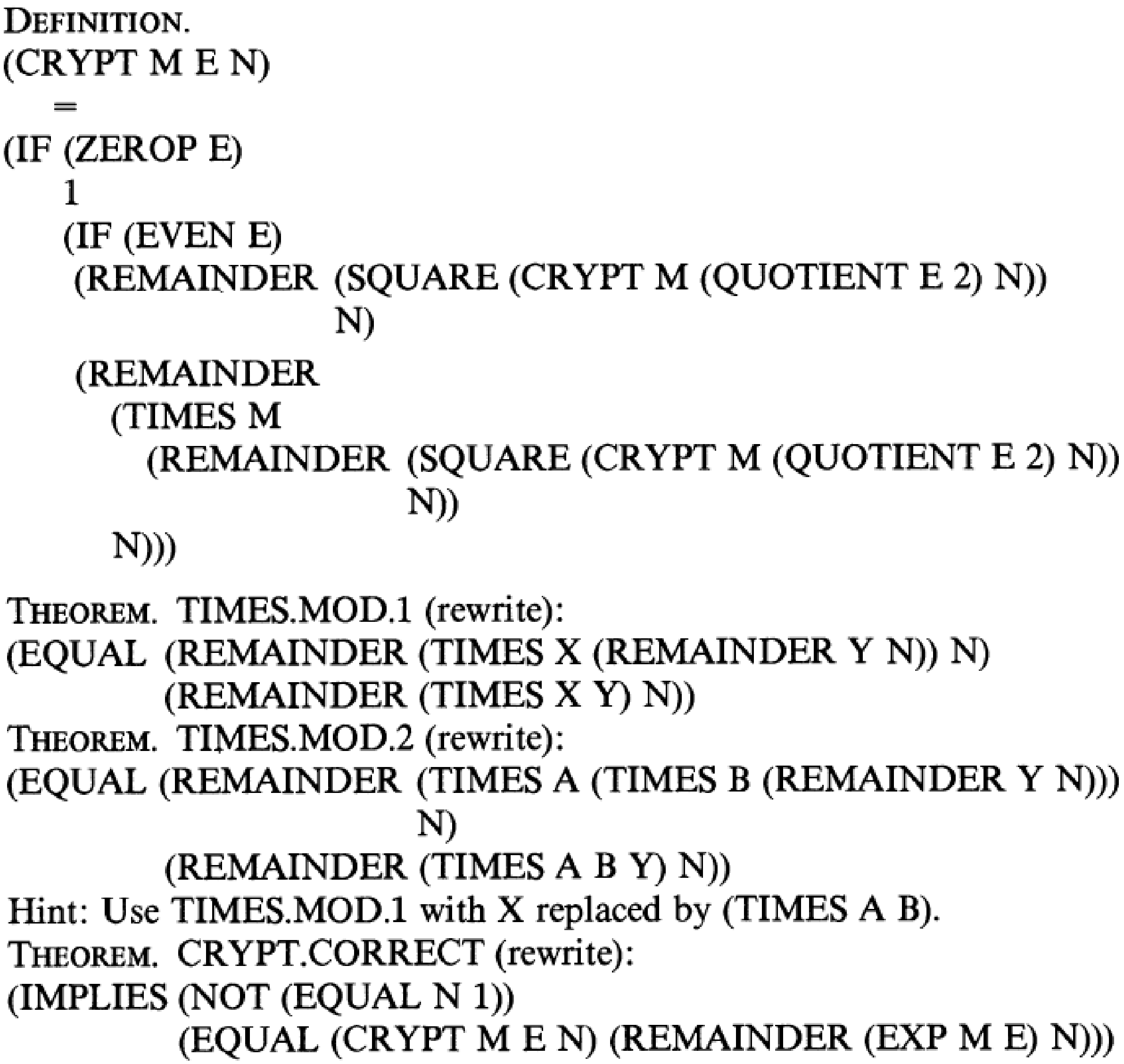
COROLLARY. (a*(b*(y mod n)))mod n = (a*(b*y))mod n.
(Hint: let x be a*b in the preceding lemma.)

THEOREM. CRYPT(M, e, n) *is equal to* M^e mod n *provided n is not* 1.

Not part of original RSA proof

(3) Algorithm Example: Verifying RSA Encryption (con’t.)

- Sample input to theorem prover:

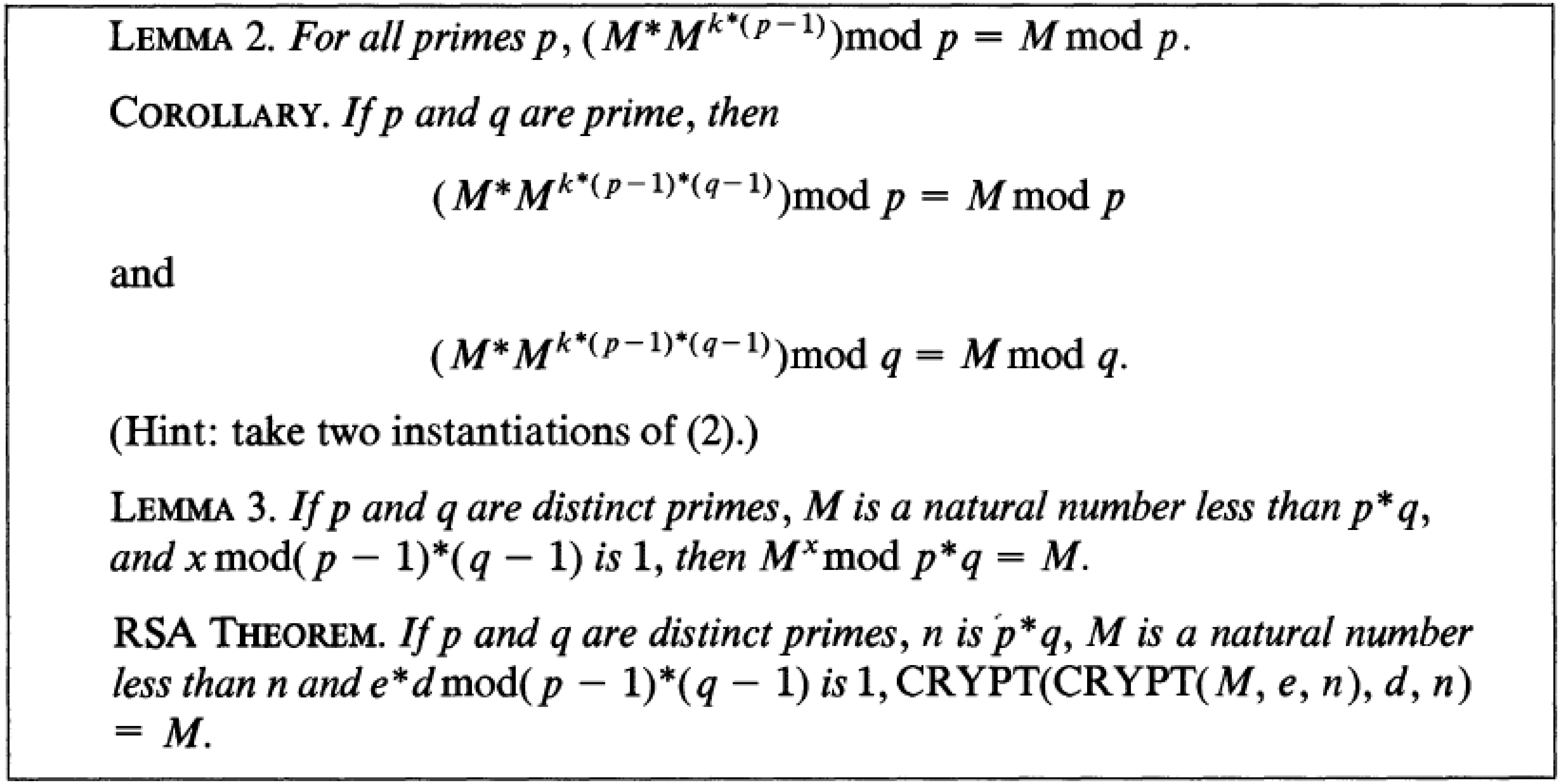


```
DEFINITION.
(CRYPT M E N)
    =
(IF (ZEROP E)
    1
    (IF (EVEN E)
     (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N))
                N)
     (REMAINDER
       (TIMES M
         (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N))
                    N))
       N)))

THEOREM. TIMES.MOD.1 (rewrite):
(EQUAL (REMAINDER (TIMES X (REMAINDER Y N)) N)
       (REMAINDER (TIMES X Y) N))
THEOREM. TIMES.MOD.2 (rewrite):
(EQUAL (REMAINDER (TIMES A (TIMES B (REMAINDER Y N)))
                  N)
       (REMAINDER (TIMES A B Y) N))
Hint: Use TIMES.MOD.1 with X replaced by (TIMES A B).
THEOREM. CRYPT.CORRECT (rewrite):
(IMPLIES (NOT (EQUAL N 1))
         (EQUAL (CRYPT M E N) (REMAINDER (EXP M E) N)))
```

(3) Algorithm Example: Verifying RSA Encryption (con’t.)

- Property 2: Boyer and Moore used rules of math (in logic form) to verify the invertibility of CRYPT



Lemma 2. *For all primes* p, $(M^*M^{k^*(p-1)}) \bmod p = M \bmod p$.

Corollary. *If* p *and* q *are prime, then*

$$(M^*M^{k^*(p-1)^*(q-1)}) \bmod p = M \bmod p$$

and

$$(M^*M^{k^*(p-1)^*(q-1)}) \bmod q = M \bmod q.$$

(Hint: take two instantiations of (2).)

Lemma 3. *If* p *and* q *are distinct primes,* M *is a natural number less than* p^*q, *and* $x \bmod (p-1)^*(q-1)$ *is* 1, *then* $M^x \bmod p^*q = M$.

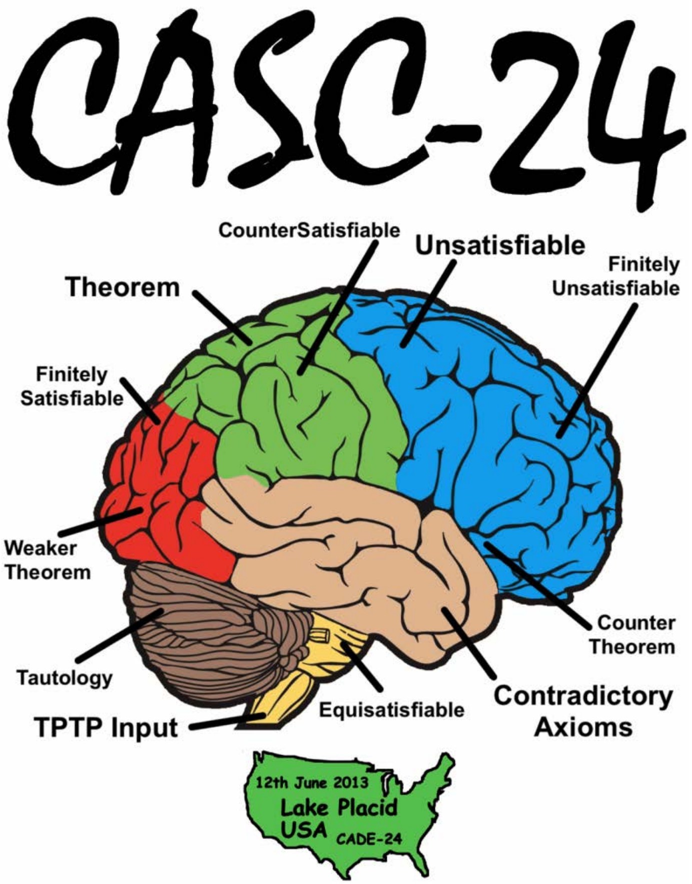
RSA Theorem. *If* p *and* q *are distinct primes,* n *is* p^*q, M *is a natural number less than* n *and* $e^*d \bmod (p-1)^*(q-1)$ *is* 1, $\mathrm{CRYPT}(\mathrm{CRYPT}(M, e, n), d, n) = M$.

(3) Algorithm Example: Verifying RSA Encryption (con’t.)

- Main point of Boyer and Moore:
  - Can use automated techniques to verify proofs and software

More on Automated Theorem Proving

- CADE Conference (Conference on Automated Deduction) holds an annual World Championship for Automated Theorem Proving (http://www.cs.miami.edu/~tptp/CASC/24/)
- Derives problems from the TPTP library (Thousands of Problems for Theorem Provers, http://www.cs.miami.edu/~tptp/)
  - Domains include:
    - Logic
    - Mathematics (e.g., set theory, graph theory, number theory, geometry, etc.)
    - Computer science (e.g., computing theory, NLP, planning, commonsense reasoning, software verification, etc.)
    - Science and engineering (e.g., HW verification, medicine)
    - Social sciences (e.g., social choice theory, management, geography, etc.)



International Joint Conference on Automated Reasoning (held bi-annually)

Topics include:

- **Logics:** propositional, first-order, classical, equational, higher-order, non-classical, constructive, modal, temporal, many-valued, substructural, description, metalogics, type theory, set theory

- **Methods:** tableaux, sequent calculi, resolution, model-elimination, connection method, inverse method, paramodulation, term rewriting, induction, unification, constraint solving, decision procedures, model generation, model checking, semantic guidance, interactive theorem proving, logical frameworks, AI-related methods for deductive systems, proof presentation, efficient data structures and indexing, integration of computer algebra systems and automated theorem provers, and combination of logics or decision procedures.

- **Applications:** of interest include: verification, formal methods, program analysis and synthesis, computer mathematics, declarative programming, deductive databases, knowledge representation, natural language processing, linguistics, robotics, and planning.

Journal of Automated Reasoning

- The spectrum of coverage ranges from the presentation of a new inference rule with proof of its logical properties to a detailed account of a computer program designed to solve industrial problems
- Topics include:
  - automated theorem proving
  - logic programming
  - expert systems
  - program synthesis and validation
  - artificial intelligence
  - computational logic
  - robotics
  - various industrial applications.
- The contents focus on several aspects of automated reasoning, a field whose objective is the design and implementation of a computer program that serves as an assistant in solving problems and in answering questions that require reasoning.