Automated Reasoning

Robert Constable & Christoph Kreitz

1. What is Automated Reasoning?
2. The NUPRL project
3. Applications
4. Areas for Study and Research
Why Automated Reasoning?

- **Too many errors in informal arguments**
  - 40–50% of the published results turn out to be wrong

- **We can’t afford errors in software development**
  - It affects air traffic, banking, government, utilities, schools, e-commerce, …
  - Errors are annoying (Reboot, loss of data, …)
  - Errors are expensive (Pentium bug, Ariane rocket, Mars Climate Orbiter)
  - Errors cost lives (Airbus crashes in the early 1990’s)

- **Current software development methods are unreliable**
  - Tested programs still contain errors
  - Correctness proofs are tedious and error-prone

↓

Need support for formal logical reasoning
Automated Reasoning: Leibniz’s dream fulfilled?

~1700: “Make logical reasoning precise”
- A universal & accurate scientific language
- Rules for evaluating scientific arguments

1890: Formal logics
- Formal language + Inference rules
- Laws of thought expressed by mechanical manipulation of text

1950: Computers – tools for symbolic manipulations
- Error-free application of rules
- Ability to search for solutions by exploring millions of possibilities

Simulate logical reasoning on a computer
Theoretical Limitations

- There are **no general algorithms** to decide
  - whether a given logical formula is **valid**
  - whether a given program **terminates**
  - whether a given program is **correct**
  - whether two programs have the **same functionality**

- **We can only search for positive results**
  - Infinite search tree - no answer in negative case
  - Search techniques from AI do not apply to Theorem Proving, Software Verification, or Program Synthesis

---

Intelligent proof techniques required
**Machine Support for Formal Reasoning**

- **Interactive Proof Editors**
  - User constructs proofs interactively by applying rules
  - Machine executes rules and returns unsolved subproblems
  - Basic mechanism: pattern matching + term rewriting

- **Automated Proof Procedures**
  - **Tactics**: programmed application of individual reasoning steps
  - **Decision Procedures** for restricted domains
  - **Proof Search** strategies, complete for small logics
    - Resolution, Matrix Methods, Model Checking, Computer Algebra, ...
  - **Knowledge-Based Reasoning**: search guided by domain knowledge
**Major Application Areas**

- **Prove mathematical theorems**
  - Detect and correct *errors* in proofs
  - Find new *proofs* automatically

- **Support development of reliable software**
  - Find bugs / prove *correctness*
  - Improve *performance*
  - Generate from specifications

- **Inference engine for AI-Systems**
  - General problem solver, robot planning, ...
Success Stories

70’s: **Four-color problem**
- Special-purpose software checks thousands of critical constellations

1993: **Synthesis of Transport Scheduling Algorithms**
- KIDS generates correct-by-construction algorithm within a few hours
- 2000 times faster than existing US-Army ADA program

1995: **Pentium Bug**
- Division algorithm mapped incorrectly onto hardware tables
- Model Checking finds countermodel

1996: **Robbins Algorithm Conjecture**
- General-purpose prover EQP produces readable proof in 7 days

1998–: **Network Verification, Optimization & Design**
- Verification uncovers subtle bug in total order protocol
- Fast-path optimization improves system performance by factor 3–10
- Formal design of verified adaptive network protocol
The Nuprl Project

- **Computational Formal Logics**
  - Extension of Martin-Löf’s constructive **Type Theory**
  - Class theory + meta-reasoning + reflection + . . .
- **Proof & Program Development Systems**
  - **Nuprl** Logical Programming Environment
  - Proof search techniques + inference engines
  - Natural language generation
- **Application to Networked Systems**
  - Verification of communication protocols
  - Optimization of **ENSEMBLE** protocol stacks
  - Formal design of adaptive systems
Nuprl’s Formal Logic: Type Theory

● Logic for constructive reasoning

● Open-ended, expressive data type system
  – Function, product, disjoint union, Π- & Σ-types, atoms
  – Integers, lists, inductive types
  – Propositions as types, equality type, void, top, universes
  – Subsets, subtyping, quotient types
  – (Dependent) intersection, union, records

  New types can/will be added as needed

  → programming
  → inductive definition
  → logic
  → mathematics
  → modules, program composition

● Uniform internal notation + “free syntax” → display forms

● Refinement calculus
  – Top-downsequent calculus

  → interactive proof development

● User-defined extensions possible
  – Language extensions (abstractions) + user-defined inference rules (tactics)
Nuprl’s Automated Reasoning System

- **Interactive proof development**
  - Supports constructive proofs and program extraction
  - Some automation by tactics and two decision procedures
  - Flexible definition mechanism with customizable term display
  - Library of definitions, theorems, methods, theories, 

- **Open architecture supports cooperation**
  - Collection of cooperating processes
  - Centered around a common knowledge base
  - Connection to external systems possible (MetaPRL, JProver)
**JProver:** first-order theorem prover
- Complete for first-order intuitionistic logic
- Proof strategy produces “matrix proof”
- Implementation separate from Nuprl

**Integration methodology**
- Code module for communicating with Nuprl
  - Logic module for interpreting Nuprl formulas
  - Conversions: sequents ↔ formulas, matrix proof ↔ sequent proof
Build reliable, high-performance networks by linking the Ensemble group communication system to the Nuprl Logical Programming Environment.
Modular group communication system

- Developed by Cornell’s System Group (Ken Birman)
- Used commercially (BBN, JPL, Segasoft, Alier, Nortel Networks)

Architecture: stack of micro-protocols

- Select from more than 60 micro-protocols for specific tasks
- Modules can be stacked arbitrarily
- Modeled as state/event machines

Implementation in Objective Caml (INRIA)

- Easy maintenance (small code, good data structures)
- Mathematical semantics, strict data type concepts
- Efficient compilers and type checkers
- Assign formal semantics to system software
- Verify protocol components and system configurations
- Optimize performance of configured systems
- Formally design and verify new protocols
Nuprl Assigns Precise Meaning to System Software

- **Describe type-theoretical semantics of OCAML**
  - “Implement” OCAML semantics (abstractions) + syntax (display forms)

- **Describe programming logic for OCAML**
  - Derived inference rules for reasoning about OCAML code
  - Type theory becomes “invisible”

- **Tools for importing and exporting system code**
  - Reasoning about the actual ENSEMBLE code in Nuprl
NUPRL Provides Assurance for Reliability

- **Formalize system specification and code on 4 levels**
  
  e.g. – “Messages are received in the same order in which they were sent”
  - “Messages may be appended to global event queue and removed from its beginning”
  - “Messages whose sequence number is too big will be buffered”
  - **ENSEMBLE** module `Pt2pt.ml`: 250 lines of OCAML code

- **Verification methodology**
  
  - Verify **components** w.r.t. benign networks (subtle bug discovered!)
  - Verify protocol stacks by composition of verifications
  - Weave **aspects** into code to add fault tolerance etc.

Verification process can be reversed into network synthesis
Stacking creates performance loss
  – redundancy, internal communication, large message headers

Formal optimizations

- **Fast-path** for common execution sequences
  - Identify Common Case as Predicate
  - Analyze path of events through stack
  - Isolate code for fast-path and generate bypass
  - Insert CCP as runtime switch

- **Header** compression

- **Methodology**: compose optimization theorems
  - Fast, error-free, independent of programming language
  \( \sim \) speedup factor 3-10
• Design systems that adapt to run-time dynamics
  – On-line upgrading, security, performance, …
  – Difficult to do correctly  

• Approach: generic switching protocol
  – Normal mode: interact with one protocol
  – Switching mode: deliver old messages, buffer new ones

• Correctness issues
  – What kind of protocols are switchable at all?
  – What code invariant preserves switchable properties?

• Verification answers both questions
  – Build formal model of communication traces
  – Characterize switchable properties by meta-properties
  – Characterize switch invariant for protocol code

\[\mapsto\] Correct implementation and use of switching protocol
Areas for Study & Research

- **Courses**
  - CS 486, CS 671 (Spring), PRL Seminar (Mon 12:20–1:15, Upson 207) . . .

- **Formal Logics & Type Theory**
  - Classes & inheritance, recursive & partial objects, concurrency, real-time
  - Meta-reasoning, reflection, relating different logics, . . .

- **Theorem Proving Environments**
  - Proof presentation, WWW-embedding,
  - Logical accounting, theory modules & dependencies, evaluators, . . .

- **Automated Proof Search Procedures**
  - Matrix methods, inductive theorem proving, rewriting, proof planning
  - Decision procedures, extended type inference, cooperating provers
  - Proof reuse, analogy, distributed proof procedures, . . .

- **Applications**
  - Formal CS knowledge: graph theory, automata, trees, arrays, . . .
  - Strategies for program synthesis, verification, and optimization
  - Modelling programming languages (OCAML, JAVA, ..)
  - Designing, verifying and optimizing networked embedded systems