CS 2800 Logic and Computation
Lecture Notes, Fall 2021

Stavros Tripakis

December 7, 2021

1 Software

Our modern societies heavily depend on software, and this dependence is likely to grow. Software is important, and it is also beautifully complex. It is complex first because of its sheer size: estimates place Google's software at about 2 billion lines of code, and Microsoft's Windows operating system at about 50 million lines of code[1]; a pacemaker has about 100 thousand lines of code, the Boeing 787 airplane has more than 10 million, and a modern high-end car has about 100 million[2]; some estimates place the size of new software produced every year to the hundreds of billions of lines of code[3].

But even very small programs can be extremely complex. The famous *Collatz conjecture* states that the following program terminates for all possible inputs:

```
n := input a natural number;
while (n > 1)
  if (n is even)
    n := n/2;
  else
    n := 3*n + 1;
```

The Collatz conjecture is an open problem in mathematics.[4] It is a *conjecture* (i.e., something we believe is true), but not a *theorem* (i.e., not something we have proven). The fact that this 6-line program defies the understanding of our best mathematicians tells us that there is something inherently complex and challenging about software. Software is the most complex artifact that humans have ever constructed. Understanding software is an important intellectual challenge for humanity.

2 Software Science

Science is knowledge that helps us make predictions. The key word is *predictions*. The stronger the science, the stronger the predictions it can make. Software science helps us make predictions about the programs that we write. Will my program terminate? Is my program correct? What does correct even mean? Will my program produce a correct output? When exactly is an output correct? Should the input satisfy any conditions in order for the output to be correct? Etc.

[1] According to https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/.

[2] According to https://www.visualcapitalist.com/millions-lines-of-code/.

[3] According to https://cybersecurityventures.com/application-security-report-2017/.

[4] If you solve it, you will become famous. See also: https://www.quantamagazine.org/can-computers-solve-the-collatz-conjecture-20200826/ (thanks to Samuel Lowe for suggesting this article).

3 This Course

Testing and proving: This course is an introduction to the science of software. You have already written programs. You have taken and will take more courses that teach you how to program. In most programming courses you will focus on checking program correctness by *testing*. Testing is very important, but as Dijkstra famously said: *"Program testing can be used to show the presence of bugs, but never to show their absence!"*

In this course we focus on *proving* program correctness. Proving is a stronger guarantee than testing. Testing checks only some inputs, whereas a proof is usually about all possible inputs. So proofs offer stronger predictions about our programs.

Logic: But in order to prove that a program is correct, we must first define what exactly do we mean by correct. For that, we will use logic. Logic is first of all a language. Contrary to natural languages (English, Greek, etc.), logic is precise and unambiguous. We can debate endlessly about the meaning of love and politics, but the meaning of a logical formula is not a matter of opinion. It is mathematically defined. This is very important because it helps avoid errors of communication. Miscommunication can be catastrophic in love and politics, but also in engineering projects.

Specification and verification: In this course we will use logic for several things.[5] We will use logic to express properties of programs. Collectively these properties define what it means for a program to be correct: they *specify* the program. This is called program **specification**. We will also use the rules of logic to prove those properties. Proving that a program satisfies its specification is called program **verification**.

In this course we will learn:

- to read functional programs with types
- to write functional programs with types
- to read formal specifications written in logic
- to write formal specifications in logic
- to read proofs
- to write proofs.

LEAN: This semester, we will use the LEAN theorem prover: `https://leanprover.github.io/`. We will write programs in LEAN's programming language, we will write specifications in LEAN's logic, and we will write proofs using LEAN's proof system.

Install LEAN on your personal computer as soon as you read this:

IMPORTANT: YOU SHOULD INSTALL LEAN 3, NOT LEAN 4!!!

We found most helpful the instructions provided here: `https://leanprover-community.github.io/`.

Other theorem provers: The goal of this course is *not* to teach you LEAN. The goal is to introduce you to the science of software, formal logic, formal specification, and formal verification. We are using LEAN as a tool and as a means to an end, rather than the end itself. LEAN is just one of many tools that could be used for this purpose. Examples of other such tools are (in alphabetical order):

- ACL2s: `http://acl2s.ccs.neu.edu/`

[5] Logic goes far beyond what we will see in this course. Logic is the foundation of mathematics. It is also the foundation of language, reason, and philosophy.

- Agda: https://wiki.portal.chalmers.se/agda/
- Coq: https://coq.inria.fr/
- Idris: https://www.idris-lang.org/
- Isabelle: https://isabelle.in.tum.de/
- PVS: https://pvs.csl.sri.com/

The above list is by no means exhaustive. This is an active area of research, and new tools are being developed or new capabilities are added to existing tools all the time. Each tool has its own pros and cons, just like different programming languages and systems have their own pros and cons. Nevertheless, some basic concepts and principles are common to all these tools. It is these concepts and principles that we strive to teach you in this course, and it is these concepts and principles that you should strive to learn.

Having fun with proofs: Proving theorems with a tool like LEAN is a lot of fun. It's like playing a game. The goal of the game is to prove the theorem. This is like solving a puzzle, or finding our way out of a maze. We will learn which moves to make to help us find the exit of the maze. WARNING: this game can become addictive!

How to succeed in this course: You learn by experimenting, asking questions, and making mistakes. Making mistakes is great (as long as they are not catastrophic mistakes, like drinking and driving and car crashing). Fortunately, computer science provides you with a very safe environment for making mistakes: the worst that can happen is that your program doesn't compile, or that it doesn't return the right result. Big deal. In this course, what can go wrong? Maybe LEAN does not accept your function definition and you don't see why. Or maybe your function doesn't work as expected. Or you cannot complete a proof. Etc. Try to experiment to see what goes wrong. For LEAN specific things, consult the LEAN documentation. Ask questions when you are deadlocked. *There are no stupid questions.*

A good way to know whether you are learning what you are supposed to be learning is whether you are *able to do all the homework problems by yourself.* If you are, you will do well in the course. If you are not, you should be worried. Come to our office hours if you are worried.

4 These Lecture Notes

These lecture notes will be sparse. This is intentional. Their aim is not to be a comprehensive textbook, but rather to guide you in the course (like a map). The philosophy of the course is *learning by doing.* Is there any other way to learn really?

In particular, these lecture notes are *not* about learning LEAN. There are many many good resources on LEAN freely available online: examples, tutorials, online textbooks, and many more. References to those will be provided as the course progresses.

These lecture notes will be permanently under construction. They will be updated regularly as we advance in the course. The latest version will serve as the reference point. Please look at the date of these notes, compare it to the date in your own copy, and use the latest version.

5 Other Reading

Documentation on LEAN: There is a lot of documentation available on LEAN from the following web sites:

- https://leanprover.github.io/
- https://leanprover-community.github.io/

Unfortunately, there is no single document that matches exactly what we present in this course, so you will have to collect information from multiple sources.[6] Also, much of the LEAN documentation is under construction and/or incomplete. We recommend starting with this (although there is a lot from the link below that we will *not* cover/emphasize in this course, like type theory, dependent types, etc., for instance):

- https://leanprover.github.io/theorem_proving_in_lean

You can also consult the reference manual (unfortunately the programming part is missing):

- https://leanprover.github.io/reference/

You can also look directly at the LEAN code, libraries, etc.:

- https://github.com/leanprover/lean/tree/master/library/init

For those interested in using LEAN for formal mathematics, here's a couple of links:

- https://leanprover-community.github.io/mathematics_in_lean/index.html
- https://www.quantamagazine.org/building-the-mathematical-library-of-the-future-20201001/, https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/ (thanks to Samuel Lowe for suggesting these articles).
- *The Future of Mathematics?* talk by Kevin Buzzard: https://www.youtube.com/watch?v=Dp-mQ3HxgDE (thanks to William Schultz for suggesting this link).

Type Theory: LEAN is based on so-called *type theory* which studies *type systems*. LEAN has a type system, and many (typed) programming languages also have type systems. Type systems are fundamental in programming (languages), but also in logic and the foundation of mathematics. However, we will not study type systems nor type theory in this class, as our main focus is to learn how to do proofs by doing. Those interested in type theory can consult the references below:

- *Types and Programming Languages* by Benjamin C. Pierce.
- *Advanced Topics in Types and Programming Languages* by Benjamin C. Pierce, editor.
- A short introduction to LEAN's type system can be found here: https://leanprover.github.io/theorem_proving_in_lean/dependent_type_theory.html.

as well as relevant courses in programming languages.

Software Foundations: https://softwarefoundations.cis.upenn.edu/. *Software Foundations* is a book series available online. It goes much further than we do in this course, but its first part (Volume 1) serves as good reading material for this course. *Software Foundations* uses a different theorem prover, called Coq. LEAN is quite similar to Coq, and you should be able to follow and re-do most of the things described in *Software Foundations* in LEAN. We often borrow exercises from *Software Foundations* and adapt them to our course. We thank the authors of *Software Foundations* for making the series freely available.

Other Courses: In addition to the *Software Foundations* online series, there is a number of courses available online which are related to our course. Here's a partial list for those interested:

- *Logic and Proof* at CMU: https://leanprover.github.io/logic_and_proof/. This course is also based on LEAN.

[6] This is also what you will have to do in your "real life" outside the university.

- *Logical Verification* at Vrije Universiteit Amsterdam: https://lean-forward.github.io/logical-verification/2020/. This course is also based on LEAN.

- *Semantics of Programming Languages* at TU Munich: http://www21.in.tum.de/teaching/semantik/WS1920/. This course is based on another theorem prover called Isabelle.

- *Formal Reasoning About Programs* at MIT: http://adam.chlipala.net/frap/. This course is based on Coq.

A formal methods course database is available here: https://fme-teaching.github.io/courses/

There are also regularly held summer schools and other seminars on logic and related formal techniques:

- See the *Speaking Logic* material by Natarajan Shankar (http://fm.csl.sri.com/SSFT21/speaklogicV10.pdf) as part of the Summer School on Formal Techniques: http://fm.csl.sri.com/SSFT21/.

- See list here: http://user.it.uu.se/~bengt/Info/summer-schools.shtml.

Textbooks: *THERE IS NO REQUIRED TEXTBOOK FOR THIS COURSE.* For those interested in learning more about logic and its use in computer science in general and specification/verification in particular, here are some textbooks:

- *Logic in Computer Science: Modelling and reasoning about systems*, by Huth and Ryan [10].

- *Handbook of Practical Logic and Automated Reasoning*, by Harrison [7].

- *The Calculus of Computation - Decision Procedures with Applications to Verification*, by Bradley and Manna [3].

For those interested in learning more about verification and formal methods:

- *Model Checking*, by Clarke, Grumberg and Peled [4].

- *Principles of Model Checking*, by Baier and Katoen [1].

- Several books on the *SPIN Model Checker* by Holzmann [8, 9].

- Books by Manna and Pnueli: *The Temporal Logic of Reactive and Concurrent Systems: Specification*, *Temporal Verification of Reactive Systems: Safety*, and *Temporal Verification of Reactive Systems: Progress* (the third is available online as an unpublished draft) [12, 13].

- *Handbook of Model Checking*, by Clarke, Henzinger, Veith, Bloem [5].

Other relevant books are the following:

- *Computer-Aided Reasoning: An Approach* by Kaufmann, Manolios and Moore [11].
  See https://www.cs.utexas.edu/users/moore/publications/acl2-books/car/index.html.

- *Coq'Art* by Bertot and Castéran [2].

- *Certified Programming with Dependent Types* by Adam Chlipala.
  Available at http://adam.chlipala.net/cpdt/.

- *Formal Reasoning About Programs* by Adam Chlipala.
  Available at http://adam.chlipala.net/frap/.

- *Isabelle/HOL – A Proof Assistant for Higher-Order Logic* by Nipkow et al [14].

- *Concrete Semantics with Isabelle/HOL* by Nipkow and Klein.
  Available at http://www.concrete-semantics.org/.

- *Functional Algorithms, Verified!* by Nipkow et al.
  Available at https://functional-algorithms-verified.org/.

The history of logic, in comics: The following is a wonderful book on the history of logic and foundations of mathematics, written by famous computer scientist Christos Papadimitriou:

- *Logicomix: An Epic Search for Truth*, by Papadimitriou, Doxiadis and Papadatos [6].

6 Course Outline

To be populated as we go along.

6.1 Introduction (Lectures 1 and 2)

- Course goals and logistics.
- Introductions.
- A glimpse into LEAN and other theorem provers (Coq, Isabelle, ACL2s).

6.2 Functional Programming with Types in LEAN (Lectures 3 - 6)

Programming in a functional language with types.

- Basic expressions, predefined operations and types in LEAN.
- `#eval`, `#reduce`, `#check`, `#print`
- Defining simple non-recursive functions in LEAN.
- Strong typing, type errors, and function types as input-output contracts.
- Predefined types `bool`, `nat`, `int` and `list nat`.
- Defining functions using pattern-matching.
- Recursive functions on `nat` and `list nat`, and a word about termination.
- Anonymous functions (lambda abstraction).
- Booleans and functions on booleans.
- Product types and currying.

6.3 Testing as Proving (Lectures 6 - 7)

Writing tests as "mini-theorems" using `example`.

- Introduction to proofs.
- The LEAN proof environment.
- The proof state, goals, and hypotheses.
- The `reflexivity` tactic.
- Tests = simple proofs.

6.4 Introduction to Specifications (Lecture 7)

- The type `Prop`.
- Properties and specifications.
- Informal and formal specifications.
- `example`, `lemma`, `theorem`.
- `sorry`.

6.5 Defining new types (Lectures 7 - 8)

- Defining our own types.
- Constructors.
- Enumerative types: the type `weekday`.
- Inductive data types.
- Defining the natural numbers: the type `mynat`.
- Defining recursive functions on inductive data types by pattern matching (*data-driven definitions*).
- Trees.
- Helper functions.

6.6 For-All Specifications (Lecture 9)

- Writing specifications with `forall` (∀).
- Formal specification and verification.
- Diving more into proofs.
- Proof tactics: `intro`.
- `try`.

6.7 Equational Reasoning and Introduction to Logic (Lectures 10 - 11)

- Equational reasoning.
- More forall specifications.
- Proof by cases.
- Introduction to logic: conjunction, disjunction, negation, implication.
- Proof tactics: `intro` (again, for implication), `intros`, `dunfold`, `cases`.

6.8 Review (Lectures 12 and 13)

- Where we stand.
- Definitions with overlapping cases.
- Weird LEAN behavior.
- Proof tactics: `unfold`, `rewrite`, `simp`, `repeat`.
- When a tactic "applies" to a proof state.
- Type coercions (are bad).
- Higher-order logic.
- Fermat's last theorem.
- The dangers of if-then-else.

6.9 Logic (Lectures 12 - 19)

- Review of propositional (boolean) logic: syntax, semantics, truth tables, boolean functions, satisfiability, validity, ...
- How many boolean functions of arity n are there?
- Syntax vs semantics.
- `bool` vs `Prop`.
- Proving propositional logic tautologies in LEAN vs. by truth table.
- Negation.
- If-and-only-iff (iff).
- Exclusive-OR (xor).
- Propositions as types, theorems as functions.
- Modus ponens.
- Using lemmas and theorems.
- Constructive vs. classic logic.
- The axioms `classical.em` (law of excluded middle) and `classical.by_contradiction`.
- Proof tactics: `trivial`, `assumption`, `exact`, `left`, `right`, `cases` (again, for conjunctive and disjunctive hypotheses), `split`, `have`.

6.10 Exam 1 – 28 Oct 2021

Taken online using Gradescope. Material: everything covered so far up to and including `have` (see above).

6.11 Induction and Functional Induction (Lectures 20 - 25)

- Proofs by induction. Base case. Induction step. Induction hypothesis.
- Proof by induction vs proof by cases.
- Multiple base cases.
- Multiple induction steps.
- Multiple induction hypotheses.
- Induction on `nat`s, lists, trees, and other inductive data types.
- The `induction` tactic.
- Choosing the induction variable.
- Effect of induction on hypotheses.
- Discovering, writing, and using lemmas.
- Delaying introductions.
- The `revert` tactic.
- Notation: `local notation`.
- ”Libraries”: `import`.
- Functional induction.
- The power of generalization.
- Induction schemes generated by functions.

6.12 Termination (Lectures 26 - 31)

- The hardness of proving theorems.
- The hardness of checking termination.
- Alan Turing.
- Undecidability.
- Proving termination.
- Measure functions.
- Persuading LEAN that a function terminates.
- Termination vs computational complexity.
- The Hydra game.
- Why program termination is important.
- Why non-termination is also useful.

- (Not really about termination) Dealing with tail-recursive functions.

6.13 Exam 2 – 30 Nov 2021 (Lecture 32: Review)

Taken online using Gradescope. Material: everything covered so far.

6.14 Reasoning about imperative code with Hoare triples (Lectures 32 - 36)

- A simple imperative programming language.
- Hoare triples.
- Deduction rules for Hoare triples.
- Examples: if-then-else absolute value, trivial counter, sum, binary search.

7 Summary of Proof Tactics

Here's a summary of the proof tactics that we have learned so far in this course:

1. `reflexivity`, abbreviated `refl`: applies when the goal is of the form $A = A$, or can be easily simplified/reduced to $A = A$.

   Intuition/justification: $\forall A, A = A$ ("*for all A, A is equal to A*") is an axiom of logic, called *reflexivity of equality.*

   In practice `refl` applies also to goals that are not strictly of the form $A = A$, but can be reduced to that form after performing computations (reductions) to the left and/or right hand sides of the equation.

2. `intro`: applies when the goal is of the form $\forall x : T, ...$; eliminates $\forall$-quantified variable x from goal and introduces $x : T$ into the hypotheses.

   Intuition/justification: If I have to prove something like $\forall x : T, P$, it suffices to prove P assuming x is an arbitrary element of type T.

3. `intros`: repeatedly applies `intro`.

4. `repeat { ... }` : repeats the sequence of tactics within `{ ... }` as many times as it can.

5. `unfold` and `dunfold`: simplify/reduce function applications of the form $(f\ e)$ for given f. If we add `at` H at the end, then the tactic applies to hypothesis H, instead of the goal.

   Intuition/justification: If I have to prove P, and $(f\ e)$ appears somewhere in P, and $(f\ e) = g$ for some g, then it suffices to prove P where $(f\ e)$ is replaced by g.[7]

6. `simp [H]`: simplifies the goal according to H. H is optional: you can also just issue `simp`.

   Similar to `unfold`, but can simplify more. For instance, `simp` can simplify if-then-else statements, e.g., it simplifies `ite (tt = tt) A B` to `A`.

   H could be a function, a hypothesis, or a previously proven lemma/theorem. We can also add multiple simplification rules, like: `simp [H1,H2,...]`.

   If we add `at` A at the end, then the tactic applies to hypothesis A, instead of the goal.

   Intuition/justification: Similar to that of `unfold/dunfold`.

[7] The phrase "replaced by g" is a bit simplistic, as the rules of substitution are not as trivial as they might seem at first glance. Luckily, we don't have to worry about defining precisely what the rules of substitution are, going over all its subtleties (free vs. bound variables, etc), in this course. The reason is that LEAN is watching over us and performs substitutions correctly on our behalf.

7. `rewrite [<-] H`: rewrites the goal based on the equality or equivalence H. H could be a function, a hypothesis, or a previously proven lemma/theorem.

   By default rewrites from left to write. If `<-` is added, rewrites from right to left. Abbreviated `rw`.

   If we add `at` A at the end, then the tactic applies to hypothesis A, instead of the goal.

   Intuition/justification: If I rewrite based on a proven equality $A = B$, then in order to prove goal G it suffices to prove G' which is obtained from G by substituting any occurence of A with B (or vice versa, of B with A, for `rewrite <-`). If I rewrite based on a proven equivalence $G \iff G'$ then I can replace goal G with G'. If I rewrite function f and by definition of f I know that $(fe) = g$, then similar to the intuition/justification of `unfold/dunfold`.

8. `cases` x:

   - if x is an element of a certain data type such as `bool` or `nat`, splits a proof/goal into several subproofs/subgoals depending on the type of x;

     Intuition/justification: If I have to prove P assuming that x is of some inductive data type T, then it suffices to prove P for each of the possible objects that x could be, based on the constructors of T.

   - if x is a hypothesis of the form $P \vee Q$, splits a proof/goal into two subproofs/subgoals, one where P is assumed, and another where Q is assumed;

     Intuition/justification: If I have to prove G assuming that $P \vee Q$ holds, then it suffices to prove G in each of the two cases: Case (1): P holds, and Case (2): Q holds.

   - if x is a hypothesis of the form $P \wedge Q$, replaces x with two hypotheses, one stating that P holds, the other stating that Q holds.

     Intuition/justification: If I have to prove G assuming that $P \wedge Q$ holds, then it suffices to prove G assuming that both P holds and Q holds.

   If we add `with ...` at the end, then we can rename the variables or labels in the various cases. Otherwise, LEAN picks the names for us.

9. `trivial`: discharges the goal when either the goal is `true` (or "obviously true"), or one of the hypotheses is `false` (or "obviously false").

   Intuition/justification: $H \to$ `true` trivially holds for any H, and `false` $\to G$ trivially holds for any G.

10. `assumption`: discharges the goal when one of the hypotheses is identical to the goal.

    Intuition/justification: $G \to G$ trivially holds for any G.

11. `exact` H: discharges the goal when hypothesis H is identical to the goal.

    Intuition/justification: $G \to G$ trivially holds for any G.

12. `left`: when the goal is $P \vee Q$, transforms the goal into P.

    Intuition/justification: to prove $P \vee Q$ it suffices to prove P.

13. `right`: when the goal is $P \vee Q$, transforms the goal into Q.

    Intuition/justification: to prove $P \vee Q$ it suffices to prove Q.

14. `split`: when the goal is $P \wedge Q$, splits the proof/goal into two subproofs/subgoals, one for P and one for Q.

    Intuition/justification: to prove $P \wedge Q$ it suffices to prove P and Q separately.

15. `have H : P := ...`: creates the new hypothesis H that P holds. We must then prove P, by filling in the `...` with a proof.

    Intuition/justification: to prove G from hypotheses $H_1, H_2, ...$, it suffices to (1) prove a new goal P from hypotheses $H_1, H_2, ...$, and then (2) prove G using the existing hypotheses $H_1, H_2, ...$ plus the newly proved result H that P holds.

16. `induction x`: if `x` is an element of a certain inductive data type `T`, perform induction on `x`. Generates several proof obligations and the corresponding induction hypotheses depending on the constructors of `T`.

    Intuition/justification: If I have to prove P assuming that `x : T`, then it suffices to prove P for each of the possible objects that `x` could be, based on the constructors of `T`. In doing so, I can assume that P holds for all previously/already constructed objects of type `T`, in order to prove P for a newly constructed object of type `T`. See also comments in lecture code `20-code.lean`.

17. `revert x`: if `x : T` is a variable of some data type `T` like `nat`, `bool`, etc, puts `x` back into the goal as $\forall$`x...` ; if `x : P` is a hypothesis of some proposition `P`, puts `P` back into the goal as `P -> ...`.

8 Allowed LEAN Library Axioms/Theorems

In your proofs, you are allowed to appeal to the following from the LEAN library:[8]

```
#check and_comm
#check or_comm
#check or_false
#check false_or
#check or_true
#check true_or
#check and_true
#check true_and
#check and_false
#check false_and


#check band_ff
#check band_tt
#check bor_ff
#check bor_tt
#check ff_band
#check ff_bor
#check tt_band
#check tt_bor
```

In addition to the above results from the LEAN library, you are also allowed to use *any* result previously proven in class, including in lectures, labs, homeworks, etc. For instance, you are allowed to use anything in given `ourlibrary24.lean` and all lecture and homework files uploaded on canvas. You are also allowed to copy your own solutions from past homeworks, define your own helper functions, define and prove your own theorems and lemmas, etc.

[8] If there is a result missing from the list that you think is reasonably basic and should be included, please let Stavros know.

References

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[2] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

[3] A. R. Bradley and Z. Manna. *The calculus of computation - decision procedures with applications to verification.* Springer, 2007.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[5] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

[6] A. Doxiadis, C.H. Papadimitriou, A. Papadatos, and A. Di Donna. *Logicomix: An Epic Search for Truth.* Bloomsbury USA, 2009.

[7] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

[8] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[9] G. Holzmann. *The Spin Model Checker.* Addison-Wesley, 2003.

[10] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004.

[11] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[13] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.