Applying Formal Methods to Networking: Theory, Techniques and Applications

Article · November 2013

Source: arXiv

2 authors:

Junaid Qadir
Information Technology University of the Punjab
171 PUBLICATIONS **1,491** CITATIONS

SEE PROFILE

Osman Hasan
National University of Sciences and Technology
249 PUBLICATIONS **1,285** CITATIONS

SEE PROFILE

Applying Formal Methods to Networking: *Theory, Techniques and Applications*

Junaid Qadir and Osman Hasan
School of Electrical Engineering and Computer Science (SEECS),
National University of Sciences and Technology (NUST), Islamabad, Pakistan
{*junaid.qadir,osman.hasan*}*@seecs.edu.pk*

I. Abstract

Despite its great importance, modern network infrastructure is remarkable for the lack of rigor in its engineering. The Internet which began as a research experiment was never designed to handle the users and applications it hosts today. The lack of formalization of the Internet architecture meant limited abstractions and modularity, especially for the control and management planes, thus requiring for every new need a new protocol built from scratch. This led to an unwieldy ossified Internet architecture resistant to any attempts at formal verification, and an Internet culture where expediency and pragmatism are favored over formal correctness. Fortunately, recent work in the space of clean slate Internet design—especially, the software defined networking (SDN) paradigm—offers the Internet community another chance to develop the right kind of architecture and abstractions. This has also led to a great resurgence in interest of applying formal methods to specification, verification, and synthesis of networking protocols and applications. In this paper, we present a self-contained tutorial of the formidable amount of work that has been done in formal methods, and present a survey of its applications to networking.

II. Introduction

The networking industry in a way is a victim of its own popularity. Internet, which began as a research experiment in the late 1960s, became popular before many aspects of Internet's design could be formally contemplated and designed [1]. The overwhelming success of the Internet led to the need of rapid innovations in applications and protocols. This has helped develop a culture that values engineering judgment, heuristics, and running code[1] more than it values sound engineering and rigorous verification. Unfortunately, the expedient rapid innovations resulting from this approach has resulted in a hit-and-trial hacking based software development culture. In contrast to well-honed verification and testing tools available for other fields such as ASIC hardware design, large-scale software systems, the networking industry has a very primitive testing tool-chain. The lack of rigor in networking industry, on the other hand, can be observed by the fact that simulation based testing—which is inherently a trial-and-error process—is routinely used to 'establish' the correctness of networking protocols, software, and hardware. With exponential number of possibilities, exhaustive testing is almost always impossible and thus subtle bugs remain unchecked and undetected until they manifest themselves at invariably inopportune times where the consequences of bugs in the wild can be drastic [2] [3]. Such a lack of rigor is totally unacceptable in most other mature engineering or manufacturing fields, and the networking community is increasingly realizing the need for better tools and techniques for verification and testing. Using formal methods will allow us to not only verify the properties of protocols and systems, but also will help us deepen our conceptual understanding of large classes of protocols.

A standard technique to manage complexity in computer systems is to utilize abstractions and modularity. Apart from the lack of a developed verification tool-chain, the Internet also suffers from a paucity of useful abstractions, especially for the control plane, which has led to accumulation of a "big bag of protocols" (documented in more than 7000 RFCs!) [4]. This is in contrast with other fields of computer science: e.g., the software industry has matured to incorporate a hierarchy of abstractions designed to simplify the task of programming while ensuring correctness—e.g., in software development, the high-level end-to-end requirements are separated from the low-level machine code by various abstractions such as algorithms, programming languages, compilers, tracers and debuggers, static analysis tools, etc. The lack of abstractions has resulted in an unwieldy complex Internet architecture, with underdeveloped underlying principles and theoretical foundations, that is totally ill-suited to the kind of dependence that is expected of the modern Internet.

Formal methods—computer techniques based on mathematical logic—are poised to play a central role in future networking as the research community increasingly converges towards a firm realization that traditional informal methods are grossly inadequate for *specification*, *analysis* and *validation* of networking protocols [5]. Formal methods have been extensively applied to the *verification* of hardware design [6], communication protocols [7] [8] (e.g., routing protocols [9]), secure software systems [10], engineering systems [11], programming languages [12], network simulations [13], large software programs [14], etc.

Unfortunately, there has been an impression in the networking community that formal methods do not return benefits

[1]The ethos of the Internet research is reflected in the famous quote of David Clark: "We reject: kings, presidents and voting. We believe in: rough consensus and running code".

commensurate with the effort to use them. Vint Cerf has written that "Formal methods have not yielded results commensurate with the effort to use them. They are overblown, verbose, hard to use, hard to understand." [15]. This criticism has unfortunately resulted from the lack of appreciation of advances in formal verification and sometimes due to poor communication between the formal verification community and the networking community. It is imperative in today's world, and it will become increasingly important in the future, to move away from manual error-prone methods of verification and automate as much of the verification tasks as we can [16]. Formal methods are still useful even if they do not meet the utopian "gold standards" of complete automation and complete generality of mathematical proofs—in particular, interactive theorem proving, abstracted models, and light-weight methods are highly suited to certain niche applications [5]. Advances in modern technology has fortunately facilitated development of many automatic and semi-automatic tools that can be conveniently used by practitioners with limited specialized background knowledge of formal methods.

With the increasingly central role networks play in all aspects of our lives (business, personal, entertainment, etc.), the correct functioning of networking protocols and systems has never been more important. In recent times, there has been significant interest in the application of formal methods to networking [17], not only due to the importance of this subject, but also due to the possibilities created by recent architectural developments in the networking community. In particular, the software defined networking (SDN) architecture, which proposes splitting of the control/ data planes and the management of multiple data planes through a centralized controller to allow programming the network in a software-like fashion, makes networking accessible via formal methods. This has accentuated the networking community's interest in applying formal methods to networking [18]. With the use of formal methods in networking, the field of network verification looks set to evolve from the current set of ad-hoc verification tools and emerge as an engineering discipline.

Contributions of this work: In this paper, we provide a self-contained tutorial covering the vast amount of work that has been done in the area of formal methods with a special focus on their applications in the domain of networking. Due to the great breadth of the subject, and vast amount of works in associated fields, we cannot hope to be comprehensive in every respect—nonetheless, we provide an extensive, self-contained, description of application of formal methods to networking with an adequate background on logic, programming languages, automatic verification, etc. This work is different from existing surveys [19] [20] [21] [22] in its exclusive focus on application of formal methods to networking and incorporation of new trends that have emerged with recent network architectural developments (such as the development of the SDN networking architecture). The emergence of SDN, and other recent innovations, have spurred a surge of interest in the application of formal methods to networking [23]. Our paper is timely since, despite the recent focus and interest in our subject area, there does not exist a unified survey paper that a networking researcher can use to develop a high-level broad understanding of formal methods and techniques and learn about their applications in the context of networking. This paper attempts to fill this void, and will be valuable to networking researchers who wish to exploit the large amount of work done in the formal methods community to build reliable future networks whose correctness is formally verifiable.

The remainder of this paper is organized as follows. The necessary background on logic is provided in section III. Various tools for specification are described in section IV. Different methods for formal verification, such as model checking, theorem proving, static analysis, etc., are described in section V. The role played by ideas in programming languages is introduced in section VI. Various applications of formal methods to networking is surveyed in section VII. Various open issues and future works are identified in section VIII. Finally, this paper is concluded in section IX.

III. Logic—the Foundation of Formal Methods

Logic is the branch of knowledge that focuses on systemizing truth, reasoning, and inference. Studied by generations of philosophers (Socrates, Plato, Aristotle, Kant, etc.), logic has a rich ancient tradition in philosophy [24]. Logic was developed in ancient Greece as a device for systematizing *deduction* through which true statements, or *conclusions*, could be derived from *premises*—statements that are *assumed* to be correct. Although, utilized in mathematics at least since Euclid (2300 BC), the incorporation of logic into a mathematical framework has occurred mostly in the last two centuries [25] through the efforts of Frege, Peano, and Russell to axiomatize mathematics. In the field of computer science, logic has been referred to as the "the calculus of computer science"[2] [26] to highlight its pivotal, and indeed "unusually effective" [27], role in the fields of formal methods [28], artificial intelligence [29], and theoretical computer science [30]. Formal methods, which utilize logic for modeling and reasoning about computer systems, have been extensively for formal verification of computer systems (both hardware and software) [28].

What is logicism: As per Aristotle's definition, logic is new and necessary reasoning—*new* since we learn what we did not know, and *necessary* because the conclusions are inescapable. Leibniz dreamed of such a mechanical system of reasoning which he called *calculus ratiocitinator* to calculate new and necessary conclusions from facts described in a logical symbolic language, which Leibniz called *characteristica universalis*. Frege devised a set-based logical language for developing Mathematics on a solid footing. Frege (1848-1925) conceived of an ambitious project, called *logicism*, which aimed at deducing mathematics (more specifically, set theory, number theory, and analysis) from laws of logic [31]. This project after Frege was taken up most notably by Russell, along with Whitehead, who embarked on an ambitious project to put mathematics on firm foundations. The use of symbolic notation, an integral component of Russell's attempt to formalize mathematics, allowed rapid progress and allowed emphasis on the structure and the form of reasoning.

[2] In a metaphorical reference to the central place calculus occupies in natural sciences.

The 'failure' of logicism: It was discovered by Russell that a logical language based on naive set theory—which defined sets to be a collection of objects and allowed sets to contain sets (including possibly itself) as elements—could not be used as the foundation of all mathematics because it suffered from paradoxes. Russell showed the following simple example, known as Russell's paradox, to illustrate this: does the set S of all sets that do not contain itself contain the set S itself? This riddle exposed that naive set theory is not sufficient to act as a foundation of mathematics leading to axiomatized set theory and various typed set theory to deal with the self-referential that created the Russell's paradox. In mathematics, the standard form of axiomatic set theory is the Zermelo-Franenkel set theory with the axiom of choice (ZFC) which acts as the most common foundation of mathematics. Eventually, Fregian logic also had to be restricted—into what is now known as first-order logic—to deal with Russell's paradox, and this restricted logic was incorporated by ZF set theory. In 1931, Godel dealt a deathly blow to logicism when he proved that any axiomatic system capable of expressing the laws of arithmetic is *incomplete*—i.e., there will always be some truth of arithmetic that cannot be proved using the axioms of the system. While logicism 'failed' in its aim of deducing arithmetic from the axioms of logic, it was instrumental in establishing the limits of computation and of "formal reasoning". It helped identify the limits of computation and of axiomatized logic systems.

A. Components of logical reasoning

In modern terms, every logic-based language is defined in terms of three components: syntax, semantics, and proof theory. The *syntax* of a language specifies all the components that can be part of a well-formed formulae. The purpose of standardizing a syntax is to aid in understanding, communicating, and reasoning. The *semantics* of a language, informally speaking, deals with the "meaning" of the formulae, or sentence, formatted according to the language's syntax. In logic, the semantics of a language specifies the truth of a formulae with respect to each possible world [29]. As an example, $x + y = 2$ is true when x and y are both equal to 1 but false in a world where x and y are both equal to 2. More formally, the term 'model'—which is used in the name of a technique known as "model checking" that we shall see later in section V-A—is used in logic in place of "possible world". The meaning of a statement $\mathcal{M}$ is a model of α (commonly depicted as $\mathcal{M} \models \alpha$, and read as $\mathcal{M}$ models α) is that the formulae α is true in situation represented by model $\mathcal{M}$. The concept of *logical entailment* is similar: we can denote in notation $\alpha \models \beta$, i.e., the formulae α entails the formulae β if and only if every model in which α is true, β is true as well. In other words, logical entailment $\alpha \models \beta$ implies that if α is true, β must also be true. Lastly, *proof theory* is concerned with manipulating formulae according to certain rules.

B. Propositional Logic

Propositional logic, also called propositional calculus or sentential logic, was developed into a formal logic by Chrysippus and developed further by the Stoics and eventually by Leibniz[3]. Propositional logic differs from syllogistic logic, proposed by Aristotle, in that it focuses on propositions which are declarative sentences that can only take values of *True* or *False*. Since the propositions are akin , to Boolean variables, propositional logic is also known as Boolean logic [29]. Propositional logic is important for two main reasons. Firstly, it is fundamentally important for computer systems since it is the theory behind digital circuits. It is also important since more complex logics (such as first-order logic, also called predicate logic, which is covered in section III-C) builds upon propositional logic.

TABLE I. TRUTH-TABLE OF TRUTH-FUNCTIONAL CONNECTIVES.

α	β	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \rightarrow \beta$	$\alpha \leftrightarrow \beta$	$\neg\alpha$	$\neg\beta$
T	T	T	T	T	T	F	F
T	F	F	T	F	F	F	T
F	T	F	T	T	F	T	F
F	F	F	F	T	T	T	T

In propositional logic, new propositions are generated from old through *truth-functional connectives* [32], which define the formal grammar of propositional logic, such as the *not* operator ($\neg$), the *and* operator ($\wedge$), the *or* operator ($\vee$), the *if, or implies, or the conditional* operator ($\rightarrow$), and the *iff, or equivalence, or the biconditional* operator ($\leftrightarrow$). Although, the Boolean propositional operators have intuitive analogues in natural language, they are defined formally. Sometimes, the mathematical terminology has a direct analogue with our intuition: e.g., the Boolean operator *and* is an operator that is defined to give a *true* value if and only if applied to two expressions whose values are true [25]. At other times, the mathematical terminology may extend our intuitive interpretation: e.g., mathematical usage of the implication logical connective extends the intuitive concept of implication by divorcing the concept of causality from implication [33]. Similarly, the Boolean operator *or*, when applied to two expressions, has the intuitive analogue of *inclusive or*, i.e., any one or both expressions are true. It is important to stress that these operators are formally defined through a truth table, and these operators may not exactly match our everyday understanding of these words. The truth tables of the logical connectives used in propositional logic can be seen in table I.

Propositional logic formulae: The formulae of a formal language built on propositional logic are expressions that can be recursively built from propositional variables by using connectives. There are four important concepts that apply to formulae. Two of these concepts are important *properties* of a formulae: *i)* being a tautology, *ii)* being a contradiction, while the remaining two concepts refer to *relations* between formulae: *iii)* tautological implication, and *iv)* tautological equivalence.

There are two fundamental concepts that deal with formulae of all logics: *i) satisfiability*—is this formula *ever* true? and *ii) validity*—is this formula always true? It may be noted that the satisfiability problem is very general, indeed various computer science problems can be reduced to a satisfiability formulation.

[3]Leibniz is also credited for being the developer of symbolic logic, along with his more famous contributions towards development of calculus

Determining the satisfiability of sentences in propositional logic was the first problem that was proved to be NP-complete [29]. Similarly, determining the validity of logic formulae is an extremely important problem. Another important problem that deals with propositional logic is the propositional tautology or equivalence checking.

Traditionally, propositional logic has been regarded as uninteresting due to several limitations. While propositional logic is trivially decidable in theory, the propositional satisfiability (SAT) problem is the canonical NP-complete problem which makes it intractable in practice. Fortunately, most practical propositional SAT problems can be solved efficiently in practice. There has been a remarkable upsurge of interest in propositional logic in the last decade or so since a diverse class of problems (including scheduling, planning, problems) can be expressed as propositional satisfiability problems.

C. Predicate Logic

Developed initially by Frege and Peirce, predicate logic enhances propositional logic—which only allowed propositional symbols along with operators—with predicates, functions, and quantifiable variables. Predicate logic expressions can include: *i)* propositional symbols, *ii)* predicates, *iii)* functions and constant symbols, *iv)* quantifiers, *v)* equality, and, *vi)* variables [29]. It was felt that truth-functional connectives of propositional logic (such as *not, and, or, if, iff,* etc.) alone were not rich enough to capture the much richer logical structure of natural language which often uses quantifiers, or modifiers, such as 'there exists', 'all', 'some', 'among', 'only', etc. This has motivated the desire to develop a richer, more nuanced, logic. To capture the modal quantification of every day life, predicate logic, or quantificational [32] logic [28], allows for a universal quantifier, $\forall$, meaning 'for all', and an existential quantifier, $\exists$ meaning 'for some'. Predicate logic is extremely important, especially for our subject topic of formal verification of computer systems, as it is used to formalize the semantics of programming languages, and to specify and verify programs.

Propositional logic and predicate logic are also called *propositional calculus* and *predicate calculus*, respectively, since both of these logics, like calculus, define a set of symbols and a system of rules for manipulating those symbols [26]. Propositional logic and predicate logic are calculi for reasoning about propositions and predicates, respectively. It is worth emphasizing the difference between a proposition and a predicate. A proposition is a statement that is either true or false—for example, IPv4 addresses are 32 bits long is a *true* statement. A predicate, on the other hand, is used to capture relation(s) or dependence on some input parameter(s)—a predicate evaluates to true or false depending on some input parameters. In the case of a *unary predicate*—e.g., x is a philosopher—the truth of the statement depends on the a solitary input variable. For *binary predicates*, however, the truth of a statement depends on two input variables—e.g., $x > y$ depends on the values of both x and y. In general, predicate logic may have n-ary relations between objects [29].

D. First-Order Logic

Predicate logic can be categorized into various orders depending on how the quantifiers are used in predicate logic. In first-order logic, it is assumed that the world contains objects (such as switches, routers, users, etc.), relations (faster than, happens after, etc.), functions (one more than, next hop of, etc.), and quantifiers through which facts can be expressed about some or all the objects in the universe [29]. In first-order logic, quantifiers can range over individuals, whereas in second-order logic, the quantifiers can also range over sets, or relations. Higher-order logic can also be defined, with ω-order logic being essentially the simple theory of types. First-order predicate logic is very popular amongst mathematicians and is the language of choice for most mathematicians [25]. While predicate logic subsumes first-order logic, second-order logic, or infinitary logic, etc., the unqualified use of predicate logic typically refers to first-order logic. First-order logic was delineated by Hilbert, and then Skolem who proposed building set-theory on the basis of first-order logic. It has been shown that first-order logic, along with a sufficiently powerful axiom system, has sufficient expressiveness for formulating virtually all of mathematics. First-order logic, like propositional logic, is a complete system [31] (first proved by Godel in his completeness theorem). There are various useful verification tools that are based on first-order logic including the Alloy analyzer (which we will discuss later in Section V-C).

In 1928, David Hilbert proposed the *Entscheidungsproblem*, German for the 'decision problem' [34], which asked for an algorithm which will take a statement of a first-order logic as input, and answer if the statement is universally valid—i.e., valid in every structure satisfying the axioms—with a "yes" or a "no". Hilbert's intent was to find a system for completely axiomatizing, and formalizing, all mathematical knowledge and proofs. In 1936, Alonzo Church and Alan Turing independently showed that a general solution to the Entscheidungsproblem is impossible—thus, no mechanical, or algorithmic, method can prove the validity of arbitrary predicate logic statements. The Church-Turing result for the Entscheidungsproblem also has significant implications for the use of *automatic theorem proving methods for software systems*. In particular, we cannot write a program (written in any common language such as Java, C, etc.) which will be able to always answer the decision question: given a logical formula ϕ in predicate logic, does $\models \phi$ hold, yes or no?

The unfortunate implication of this is that no automatic deductive verification tool can exist that will work with any arbitrary predicate logic formula instance as an input and always terminate while producing a correct 'yes'—corresponding to a valid input formula—or a 'no' answer corresponding to an invalid input formula [28]. This poses a fundamental, and insurmountable, problem to the automatic theorem proving approach of verification, also known as automatic deductive verification. Therefore, first-order logic, unlike propositional logic is only a semi-decidable theory—i.e., there exists an effective method for telling if any arbitrary given formula is in the theory, but it may give either a negative answer or no answer at all when the formula is not in the theory.

E. Higher-Order Logic

A *higher-order logic* (HOL) is more expressive than first-order logic as it uses some additional quantifiers along with stronger semantics. Unlike first-order logic in which variables can not denote predicates, variables in second-order logic can denote predicates allowing the logic to talk about itself more easily. There can be higher-orders beyond second-order logic. The main strength of HOL is that it is highly expressive, and can express any mathematical theory, like multi-variable calculus [35] and probability [36], in its true form. The higher expressiveness associated with higher-order logic, however, is tempered with the downside that model-theoretic properties of higher-order logic are less well-behaved than those of first-order logic. In particular, validity in higher-order logic is not even semi-decidable (or anywhere in the arithmetical hierarchy).

F. Hoare Logic

Hoare logic (also known as *Floyd-Hoare logic* or *program logic*) is a formalism that defines logical rules—i.e., axioms and inference rules—to provide an axiomatic basis for verifying computer programming [37]. The central construct used in Hoare logic is the partial correctness specification in the form of a *Hoare triple*[4]: $\{P\}\ C\ \{Q\}$ where P is the pre-condition, Q is the post-condition, and C is the command. Hoare logic builds upon other conventional logic, e.g., first-order logic, for specifying the pre- and post-conditions.

Hoare Logic is a deductive proof system for Hoare triples $\{P\}\ C\ \{Q\}$. The partial correctness specification $\{P\}\ C\ \{Q\}$ means that whenever C is executed in a state satisfying P, and if the the execution of C terminates, then the terminating state after C's will satisfy Q. Hoare logic deals with verification of partial correctness of a command, and termination of a program has to be separately proved to show total correctness. The generality of Hoare's approach is based on its characterization of programming constructs as transformations of states which can universally apply to any imperative programming language construct. The underlying semantics of a program can be viewed a set of transformations from an initial state to a final state. Since a sequential program can also be envisioned as a transformational system, Hoare logic is particularly suited to analysis and verification of sequential computer programs. Hoare logic is a sound system (every provable formula is true) but not a complete system (i.e., not all true statements are provable). More details about Hoare logic can be found in [38].

Hoare logic, and the use of Hoare-style pre-conditions and post-conditions, is commonly used in many settings. As an example, the Java Modeling Language (JML) defines a specification language for Java programs, following the design by contract paradigm, which uses Hoare style pre-conditions and post-conditions and invariants for extended static checking. The same style is inherited by ESC/ Java.

[4]The Hoare triple is also known as partial correctness assertion or PCA and is partially based on Floyd's intermediate assertion method

G. Modal Logic

Modal logic is an expressive form of logic that uses additional quantifiers. Modal logic was originally developed by philosophers to study different 'modes of truth'—e.g., an assertion P may be false in the present world, however, the assertion 'possibly P' will be true if the assertion P is true in some alternate world [39]. Temporal logics essentially have two kinds of operators: logical operators (loaned from traditional the logic framework in which temporal logic is used) and modal operators. The modal operators capture in modal logic the intuitive notions of *necessarily, always*, *possibly, sometimes*, etc. The symbols N, F, G, A, E represent ***N**ext*, ***F**uture*, ***G**lobally*, ***A**ll* and ***E**xists*, respectively. In typical notational terms, the box symbol is used to represent *necessity*, while the diamond symbol is used to represent *possibility*. For example, Gp would mean always p; Fp will mean sometimes p; $\diamond p$ means possibly p; $\Box p$ means necessarily p.

Modal claims can be understood semantically in a theory of "possible worlds"—an idea commonly attributed to Leibniz which was advanced by Saul Kripke in the late 1950s. Kripke advanced Leibniz's conception of the actual world being one "possible world" amongst other, by proposed a mathematical theory of models (now known as Kripke models) for possible worlds. A statement is "possible" in modal logic if it is true in at least one possible world; a statement is "necessary" if it is true in all possible worlds. We will see later that "model checking" (covered in section V-A) depends fundamentally on the concept of possible worlds and utilizes Kripke models.

H. Temporal Logic

The use of temporal logic, a special type of modal logic, for formal specification and verification of computer systems was proposed by Amir Pnueli in a highly influential paper [40] in 1977. In this paper, Pnueli argued that temporal logic—a formalism for dealing with how truth values of assertions change over time—is especially appropriate for describing reactive systems such as operating systems and network communication protocols. In a reactive system, which contrast with sequential terminating programs that essentially transform the input to the output and then terminate, the normal behavior is to engage in a nonterminating computation that continuously interacts with the environment. Examples of reactive systems include operating systems and network communication protocols. Temporal logic is especially invaluable in the field of model checking finite-state *concurrent* programs [41]: Leslie Lamport, in his highly cited paper "what good is temporal logic?", has highlighted that the main utility of temporal logic is in modeling concurrent systems [42].

Temporal logic formulae differ from ordinary Boolean formula in that the temporal formulae have new modal operators—which allow qualitative description of temporal events by implicitly incorporating temporal ordering of events—in addition to the traditional Boolean operators—"and", "or", "not", and "implies" [43] [44]. The usage of temporal logic has been widely adopted for use with finite-state programs with algorithmic methods available that can verify the temporal-logic properties of finite-state systems. While the

capacity to only include finite states may appear too limiting, it turns out that a wide range of systems, especially, hardware systems and communication protocols, can be modeled as finite-state programs. Some (linear) temporal logic operators include G (*Globally*), F (*Eventually, Finally*), X (*Next*), and U (*Until*). For example, we may want to reason about the temporal properties of a protocol in the following way: a message is not received unless one is sent, a message that is sent is eventually received, etc.

Temporal logic has been extensively applied to computer systems, and is a key component of the popular model checking approach (discussed in section V-A), because it can capture two keys notions of computer performance. Firstly, temporal logic can capture "*liveness*" property that some good thing will happen in the future—i.e., the form Fp, which indicates that some proposition will be true in the *future* in the course of the computation. Secondly, the "*safety*" property of the form Gp can capture the desire that globally p is ensured which incorporates the proposition that undesirable states are never obtained. In addition, the "*fairness*" property is also defined which states given certain conditions, an event will occur, or will fail to occur, infinitely often. The fairness property is often expressed with Gp (infinitely often) and Fp (eventually always). Efficient methods exist that can work with temporal logics. While validity in first-order logic is semi-decidable (i.e., it is possible that complete proof procedures will run forever on invalid formulas), validity/satisfiability in many temporal logics is decidable.

There are two important subtypes of temporal logic: linear temporal logic (e.g., LTL)—where each moment in time has a unique future trajectory or possible future—and branching temporal logic [45] in which each moment can be split into many different possible futures. *Linear temporal logic (LTL)* is a susbset of the more complex CTL that additionally allows branching time and quantifiers. LTL is also sometimes called propositional temporal logic, abbreviated PTL. LTL can use both propositional and first-order forms. LTL is popularly used, in both these forms, in the specification and verification of programs [39]. The SPIN model checker [46] is based on LTL and has been extensively used for communication protocol verification [47]. *Computation tree logic* (CTL) is an example of branching temporal logic that has additional path quantifiers such as A (for all paths $\forall$) and A (there exists a path) that denote universal and existential quantification over paths starting in a certain state. CTL is used mostly for applications in hardware verification, while LTL is used mostly for applications in software verification. While CTL and LTL do have overlapping expressiveness, each logic can express properties outside the domain of the other—e.g., LTL can express fairness properties which CTL cannot, but CTL can express the so-called reset property which LTL cannot. The NuSMV model checking tool is based on CTL. CTL is extensively used in the formal verification of reactive networked systems. As an example, it is used in the recent work of Reitblatt et al. [48] which also uses model checking with the NuSMV [49] tool for verification. Other works that incorporate CTL include the ConfigChecker tool [50], the Splendid Isolation project [51], etc. Lastly, we will mention that the *computation tree logic star* (CTL$*$) logic, not as commonly used as LTL and CTL, has been proposed as a generalization of both LTL and CTL.

I. Other Logics

Relational Logic: The logic used in the Alloy analyzer [52] is a relational logic that combines the quantifiers of first-order logic with the operators of the relational calculus. Relational logic extends first-order logic by incorporating transitive closure allowing greater expressiveness Since first-order logic is undecidable, the focus of the Alloy analyzer is in *model finding* rather that exhaustive model checking—in particular, not finding a model does not preclude a model in a larger scope. Most tools for relational notation, other than Alloy analyzer, e.g., PVS etc., focus instead on theorem proving and are thus not fully automated. Kodkod [53] is an example tool that is based on the relational logic of Alloy. The inclusion of "transitive closure" enables expressiveness (beyond that offered by first-order logic) that can be used to encode common reachability constraints. Since the relational logic of Alloy uses multi-arity relations instead of functions over sets, it is first-order and thus amenable to automatic analysis due to its simplicity.

Router Logic : Feamster et al. [54] proposed *routing logic* to define a set of rules that can be used to determine if a routing protocol satisfies various properties. Feamster et al. also utilized this logic for analyzing the behavior of BGP protocol under various conditions. Importantly, Feamster et al. suggested that in addition to analysis of existing configurations, router logic can be used to synthesize network-wide router configurations from a high-level description.

J. Satisfiability of logic formulae: the SAT problem

A fundamental concept that applies to all logic formulae is the concept of *satisfiability*: is this formula ever true? The Boolean satisfiability (abbreviated as SAT) problem is an important problem in theoretic computer science having wide-range applications. The SAT problem can be defined as: Given any arbitrary formula, find a satisfying assignment or prove that no satisfying assignment is possible. Such an assignment may not always exist—in which case, we will say that the problem is over-constrained, and the solver will report that satisfying the formula is not possible. The Boolean satisfiability problem is also alternatively known as propositional satisfiability or simply as the satisfiability problem. The SAT problem was the first problem shown to be NP-complete[5], and many practical problems can be reduced to a SAT formulation [55] and solved through off-the-shelf SAT solvers.

The SAT problem has applications in scheduling, automated theorem proving, planning, model checking, software verification, synthesizing consistent network configurations, etc.

[5]Any instance of NP-complete problem can be transformed into an instance of another NP-complete problem quite easily. As an example, both graph coloring and SAT problems are NP-complete, and an instance of the former problem (i.e., graph coloring) can easily be transformed into an instance of the latter (i.e., SAT).

SAT solvers are thus very versatile tools useful for solving constraint satisfaction problem in a variety of settings. The SAT problem is at the very heart of the problems of design, specification and verification of computer systems [28] for diverse logics. The problem of formal verification fundamentally deals with the satisfiability relation expressed as $\mathcal{M} \models \phi$ where $\mathcal{M}$ is a *model* of a system and ϕ is a specification expressing what should be true in situation $\mathcal{M}$.

What is satisfiability mathematically? A logic language is composed of logical symbols with fixed interpretation (e.g., in propositional logic, the logical connective such as $\wedge$, $\vee$, etc. are logical symbols) and other non-logical ones (such as propositional variables p, q, etc.) whose interpretations may vary. These symbols can be combined together to form *well-formed logical formulae*. A formula is *satisfiable* if it has an interpretation that makes it logically true. In this case, we say the interpretation is a *model*; a formula is *unsatisfiable* if it does not have any model. A logical formula is valid if it is logically true in any interpretation. Conversely, a propositional formula is valid *if and only if* its negation is unsatisfiable. As an example, consider a Boolean variable p. The formula $p \wedge \neg p$ is unsatisfiable since it is not true in any interpretation—in other words, it does not have any model. The formulas p and $\neg p$ are, on the other hand, satisfiable but not valid since they are true in some, but not all, interpretation(s). Finally, the formula $p \vee \neg p$ is valid since it is true in all interpretations. In the SAT problem, we seek a *satisfying assignment* for a given propositional formula on a set of Boolean variables which assigns values to the variables such that the formula evaluates to *True*.

1) Variations of SAT: While we are mostly interested in propositional satisfiability due to its tractability, the concept of satisfiability can be generalized to other Boolean logics—in particular, the quantified Boolean formulas (QBF) problem generalizes the SAT problem[6] and refers to the problem of deciding the satisfiability of quantified Boolean formulae, or QBF, in which the variables can be either universally or existentially quantified. The ability to utilize universal and existential quantifiers in arbitrary ways makes QBF considerably expressive than SAT. It must be noted that SAT is NP-complete which means any NP problem can be encoded in SAT. Similarly, QBF is PSPACE-complete, i.e., any PSPACE problem can be encoded in QBF. Unfortunately, current QBF solvers do not scale, and therefore, our primary focus will be on the SAT problem and solvers.

The SAT problem has many interesting variations. For example, the MaxSAT problem is the application of SAT problem to optimization theory, the AllSAT problem aims to determine all satisfying assignments, etc. Motivated by the success of SAT solvers, researchers have recently given significant attention to Satisfiability Modulo Theories (SMT). In the SAT problem, the logical operatives were restricted to the conjunctive normal form (CNF) and qualifiers such as "for all such things", or "there is one such thing" were not allowed. The SMT problem is considered more difficult than the SAT problem [56]. While SAT solvers determine the satisfiability of propositional formulas, SMT solvers can, on the other hand, check the satisfiability of formulas in some decidable first-order theory (e.g., linear arithmetic, array theory, uninterpreted functions, bit-vectors, etc.) [57]. SMT is seeing rapid progress and initial commercial use in software verification [58].

2) SAT/ SMT solvers: Since the SAT problem is NP-complete, the general problem is theoretically intractable. All currently known SAT solutions thus perform poorly in the worst-case—i.e., with exponentially increasing computation cost as the instance size increases. Fortunately, the intractability of the general SAT problem does not practically rule out efficient solutions of special cases. There has been great advances recently in the field of formal verification based on the discovery that SAT solvers can solve a wide variety of practical SAT problems quite efficiently [56]. Modern tools can solve practical industrial SAT problems having millions of variables and constraints in mere seconds. In practice, such approaches can help avoid the daunting proposition of redeveloping algorithmic solutions for solving new problems, thus enabling a wide variety of application areas to benefit.

Broadly speaking, there are *two ways to use a SAT solver*. The first, and simplest way, is the *eager approach* for the application to generate a Boolean formula for the SAT solver so that it may determine that the satisfiability of the formula. Alternatively, the application can use the *lazy approach* to reduce a problem to a series of inter-related SAT queries, in which the SAT solver incrementally solves subsequent queries dynamically generated based on the results of previous queries [59]. Much of the improvement in SAT solver performance in recent years has been driven by several improvements to the basic DPLL algorithm such as *i)* non-chronological backjumping and learning conflict clauses; *ii)* optimization of constraint propagation rules; *iii)* heuristics for picking split variables (even restarting with a different split sequence); *iv)* Highly efficient data structures. A detailed account of various algorithms for solving the SAT problem is presented in [60], whereas recent advances in SAT-based formal verification can be viewed at [61] [62]. A comparison of propositional satisfiability and the related field of constraint programming can be seen in [59].

Various SAT/ SMT tools have been proposed with rapid progress in this field being sustained by Moore's law and consistent advances in algorithms, data structures, and decision heuristics [63]. Example SAT/ SMT solver tools include MiniSAT [64], Chaff [65], and the Z3 tool from Microsoft [66]. Due to the great generality of SAT/ SMT solvers, it is remarkable that various contemporary verification tools that differ in terms of source language, methodology, and degree of automation, eventually fall back on these solvers for the core task of checking validity and satisfiability. With their impressive generality, scalability, and maturity, SAT/ SMT solvers look set to play a significant role in future formal verification technology.

3) Applications of SAT/ SMT solvers to Networking:: Recent advances in SAT/ SMT solvers have significantly advanced the state of the art in formal verification, and SAT/ SMT tools are routinely used in network verification projects. We present

[6] In the SAT problem, all the variables are implicitly existentially quantified.

a few works as examples. Zhang et al. have presented an approach for verifying and synthesis of firewalls using SAT and QBF [67]. FLOVER, a model checking system, implemented using the Yices SMT solver [68], verifies that the networks security policy is not violated by the aggregate of flow policies instantiated within an OpenFlow network [69]. Recently, there has been work in verifying the data plane through SAT solvers. Anteater [70] verifies the data plane by translating connectivity invariants into SAT problems that are checked against the data plane by a general SAT solver to return a counter example in case of violation of invariants. NetSAT is another data plane verification project that is SAT based [71]. Some more examples of the use of SAT/ SMT technology in the context of networking can be seen in table II.

K. Algebra and Logic

An algebra is a structure that consists of sets and operations that act on those sets. Using the tools of algebra, logical statements can incorporate unknowns, symbols, and formulas. This symbolic calculus enables correct reasoning with economy of mental effort and has led to rapid development in mathematical knowledge. To paraphrase Alfred Whitehead, symbolism facilitates understanding, and tracking of transitions in, reasoning almost mechanically by the eye without undue taxing of the brain. Mathematical logic, or symbolic logic, improved upon the logic of Aristotle by exploiting symbolic manipulations—or, essentially the methods of algebra.

Boolean Algebra: The algebra of logic was founded by George Boole (1815 to 1864), and perfected by later logicians, to formalize the "laws of thought". Boolean algebra is essentially the 'algebraization' of classical propositional logic and the bridge between logic and algebra.

Relational Algebra: The field of databases extensively utilizes ideas from relational algebra. Relational algebra, an offshoot hybrid of first-order logic and of algebra of sets, essentially deals with manipulations of relations. The formalism of relational algebra, proposed by E.F. Codd in the 1970s, can be used as a query language for relations and serves as a theoretical foundation of databases.

Kleene Algebra: The study of semantics and logics of programs utilizes Kleene algebra which defines algebraic structures with operators +, ., *, 0, and 1 satisfying certain axioms. Kleene algebras arise in many diverse contexts: relational algebra, semantics and logics of programs, etc. Kleene algebra was extended to incorporate tests to produce *Kleene algebra with tests* (KAT) [72]. KAT has recently been used in the NetKat [73] project to provide consistent reasoning principles about network applications in the setting of SDN.

Algebraic path-finding: In networking context, algebra can be viewed as a concise language useful for describing combinatorial problems. Researchers have applied algebraic ideas to network routing through algebraic path finding methods that exploit the fact that numerous practical network problems are in fact instances of the same abstract "algebraic path problem" (e.g., a classical example of an abstract algebraic path problem is shortest path routing) [74]. Routing algebra meta-language (RAML), which builds upon Sobrinho's *Routing algebra* [75], was proposed by Griffin in the "metarouting" project [76]. Metarouting aims at equipping network operators with the ability to define their own routing protocols in a high-level declarative manner using a domain-specific language customized for specification, verification, and implementation of routing path metrics. Sobrinho's Routing algebra [75], which can be understood as generalization of shortest path routing, is expressive enough to adequately model complex policy-based routing typified by ubiquitous the Border Gateway Protocol (BGP) routing protocol. A key feature of the metalanguage proposed for metarouting, which is especially relevant to our subject topic, is that algebraic properties required for guaranteeing correctness can be automatically derived.

IV. Tools for Specification and Modeling

There are three important components of a verification framework. Firstly, since it is often cumbersome and unwieldy to work with real systems, there has to be a *i) framework for modelling systems*: this typically employs a *description language* of some sort—especially, when considering hardware systems. Secondly, a *specification language*—typically a logic-based language—is needed for specifying the desired properties that are to be verified. Lastly, a *verification method* is needed to establish if the system model satisfies the specification.

In this section, we will study techniques for modeling systems in section IV-A and for specifying properties in section IV-B. We will cover verification methods later in section V.

A. Modeling Systems

Systems can be divided into two broad classes. *Transformational systems* may be modeled as black boxes that take certain input and produce a final result as output and terminate. Such systems can modeled in terms of their input/ output relations. Formal methods developed for such transformational systems include the Floyd-Hoare logic (section III-F), which allow reasoning about such systems through pre- and post-conditions, and specification languages like Z (which we will cover in section IV-B). *Reactive systems*, on the other hand, maintain an ongoing interaction with their environment, and thus such systems must be specified and verified in terms of their ongoing behaviors. Formal methods proposed for such reactive systems have to use more sophisticated techniques than those provided by the pre- and post-conditions in notations such as Z. In particular, label transition systems (called Kripke structure) based on the concept of finite state machines (FSMs) and temporal logic have been proposed for modeling reactive systems.

In the following subsections, we will discuss various approaches for modeling systems. We will cover FSMs, Kripke structures, binary decision diagrams (BDDs), and model extraction from code in sections IV-A1, IV-A2, IV-A3, and IV-A4, respectively.

1) Finite State Machines: The mathematical formalism of finite state machine (FSM), or finite state automaton, is commonly used in the study of the design of computer programs and sequential circuits [77]. An FSM can be conceived as an abstract machine having finite states in which the machine can be in only one state at any given time. The FSM can make a transition—i.e., change its state from the *current state* to another state when triggered by some event or condition. A given FSM is defined by its set of states, and the triggering conditions for each transition. The "state transition model" of FSM has been extensively used in formal verification and serves as the basis of system modeling in "model checking". The state transition model is amenable to mechanical automated verification, but suffers from the "state space explosion" problem, which describes the case when the number states of the system model becomes so large that it becomes infeasible to exhaustively explore the state-space using the available computational resources.

Broadly speaking, there are two kinds of FSMs: *i)* the more general *Mealy machines*, in which the output depends not only on the system state but also on the system input, and *ii) Moore machines*, which are special cases of Mealy machines, in which the output is determined by only the system state. A FSM is deterministic if the next state and the output are uniquely determined by the current state and input, otherwise, the FSM is non-deterministic if a given state and input can non-deterministically lead to one of many possible next states and outputs. Non-deterministic FSM (NFSM) can be viewed as a generalization of deterministic FSM.

A protocol specification can be translated into a FSM model, with each asynchronous process coded as a separate FSM, extended by message queues and variables if necessary. The system remains finite and amenable to exhaustive search if the queue size and the range of variables is bounded. The system is non-deterministic in general since in each system state a number of transitions may be simultaneously executable. There are two important structural properties of FSMs when used to represent protocols [78]. Firstly, the state space is sparse, i.e., the set of effectively reachable state is much less than the number of potentially reachable states with a ratio of 1 in 10^9 being typical. Secondly, the state space is tightly connected, i.e., the states are usually reachable by mildly different paths that differ only in the order in which the execution of the asynchronous protocol is interleaved.

There is a well-developed theory for verification of FSMs: e.g., reachable states, and equivalence, etc., that can readily exploited for network verification tasks. In particular, reachability of states is very relevant in a networking context. The FSM formalism has been extensively used in formal verification works for networking [48] [50] [79] [80]—in these works, the packet is considered as an FSM. Many network verification projects model the network as a large state machine (see description in [22] and [21]). Unfortunately, the FSM verification problem is PSPACE-complete, and therefore is computationally very complex. The problem, however, reduces to be NP-complete if the FSM can be formulated as a combinational logic network.

Automata Theory is a field of theoretical computer science that has been used in the study of computability and languages [81]. Finite automata constitute an important formalism in theoretical computer science. It is useful for modeling a wide variety of systems that have finite number of states (e.g., communication protocols, for lexical analysis as used in compilers, for scanning text, for expression pattern matching, etc.). An automaton can be envisioned as a special case of Moore machines in which only two outputs—ACCEPT and REJECT—are defined. Variations on the general theme of automata, with varying degrees of expressiveness, have been proposed [77]: e.g., timed-automata [82], Petri nets [83], etc. These formalisms have been adopted in the field of formal verification: e.g., Petri nets have been commonly used for representing concurrent network protocols [84] while timed-automatons have been used for verifying timing properties of network protocols and real-time systems in time-automata based model checking tools (to be discussed later in section V-A) such as UPPAAL [85].

2) Kripke Structure: Kripke structure is a labeled state transition graph that can adequately capture the temporal behavior of reactive systems. From a practical point of view, the Kripke structure is nothing but a *labeled* FSM extended to incorporate a labeling function that maps states to sets of atomic propositions making it possible to specify simple propositional properties on the FSM. When used in conjunction with some temporal operators, these propositional properties can be used to specify properties like "from a state labeled *REQ*, the state labeled *ACK* will eventually be reached" [86]. Kripke structure can easily model diverse kinds of systems that are described using formulae of first-order logic.

Kripke structures are often used to model reactive systems that interact with the environment in a continuous fashion without terminating [87]. Since such systems do not terminate, input-output transformation characterization is not sufficient. Instead, it is important to capture the *state* of the system, and how the system state changes as a result of some action. One way of doing this is by identifying the transition of the system—which describes the system state before an action occurs and after it occurs, respectively.

More formally, Kripke structures consist of a set of states, set of transitions between states, and labels for each states defining properties that are true in that state. A Kripke structure $\mathcal{M}$ over AP, representing a set of atomic propositions, is a 4-tuple $\mathcal{M} = (S, S_0, R, L)$ where *i)* S is the *finite* set of states, *ii)* S_0 is the set of initial states, *iii)* R is the transition relation, and *iv)* L is a labeling function that labels every state with the set of atomic propositions that are true in that state.

3) Binary decision diagrams (BDDs): The concept of "binary decision diagrams" (BDDs) is quite old but was popularized by Bryant in 1992 [88] as an efficient method for representing state transition systems [89] [90]. It has been pointed out earlier that techniques like model checking suffer from the problem of state explosion which is quite likely to occur if the system under study is composed of components that can perform transitions in parallel. This can cause the system states to grow exponentially leading many experts to be skeptic about the ability of model checking

to scale to large systems. Model checking owes most of its success to the development of the data structure of BDDs which allows efficient verification of large transition systems. Computer science luminary Don Knuth cites BDDs as one of the most fundamental data structure development in the last 25 years which allows solutions to problems previously imagined as intractable [90]. The BDD data structure allows concise representation of large transition systems and easy manipulation, and is therefore an important component of many logic synthesis and formal verification systems [90] [91].

Bryant also observed that reduced ordered BDDs (OBDDs) are a canonical representation of Boolean functions. The use of reduction and ordering is common in BDDs, and in fact, the term BDD is commonly understood to refer to reduced ordered BDDs [90]. BDDs are able to reduce the space required for storing state transition systems by identifying redundancies through the following three rules: *i)* merge equivalent leaves, *ii)* merge isomorphic nodes, and lastly *iii)* to eliminate redundant tests.

It was noted in [92] that the rulesets that network administrators typically write lead to small BDDs. BDD is a very popular data structure that can be used, along with efficient graph algorithms for BDDs, to significantly improve the computing time and space efficiency of algorithms [93] [50].

4) Model extraction from code: One of the hindrances in the popularization of formal verification is the tediousness of the task of creating system models. A possible solution to this problem for the specific case of *software systems* is to apply verification methods not to models of code, but to implementation code directly through some automated model extraction technique. Some example efforts in this domain include extension of the SPIN model checker for support of embedded software in abstract models [94], formal verification of device driver code at Microsoft [95] through automatic predicate abstraction of C programs [96], and CMC tool at Stanford that works directly with C code [97].

B. Formal Specification

In networking protocols, it is important that protocols are defined unambiguously. Traditionally, the specification process adopted by Internet Engineering Task Force (IETF) is based largely on specification through informal English prose, with implementations also serving as an informal specification surrogate. Although in the early 1980s, various IETF standards have been formally specified by various academics (including an Estelle [98] description of Transport Control Protocol, TCP), the IETF has not embraced the use of formal description techniques and continues to specify protocols informally relying primarily on the implementation as the specification. The tendency to use the implementation as the specification has the drawback of not cleanly separating what is part of the protocol and must be conformed to and what is system and implementation dependent. The lack of the emphasis on formal specifications for Internet protocols has created a problem where it is considered acceptable to create software without fully understanding the implications leading to an ad-hoc hit-and-trial based software development culture [99]. Experience with Internet protocols has shown that simple informal English prose is insufficient for specifying and communicating protocols [5]. Many of the problems that arise due to informal specifications can be redressed through formal methods for specification which aid not only in verification and communication, but also in analysis [1]. In particular, analytical tools can analyze the formal description to ensure that absence of protocol deadlock, data loss, races, hazards, and other pathological behaviors.

Formal specification can be used by the formal verification process to verify that the desired properties are held by the system model. For the purpose of formal verification, *equivalence checking* can be used to match an implementation against a full specification of what a program must do. However, due to the significant overhead involved in writing a full specification, formal verification is often done with partial specification that describes only some desired behavior of the program. This endeavor which contrasts with equivalence checking is known as *property checking*. Most property checking tools use either logical deductive interference or model checking, and report a counterexample when a property violation is seen. It is worth emphasizing that correctness is not an unqualified concept since correctness measures the relation between two entities: a specification and an implementation, or a property and a design [100]. Thus verification is only as good as the specification, making specification an extremely important part of verification.

Broadly speaking, *formal specification techniques* can be categorized into three types based on the underlying formalism. Firstly, in the *mathematical or language-based techniques*, commonly a predicate calculus based approach is taken to represent protocols. Secondly, in the *FSM-based techniques*, an existing programming language may be extended to incorporate the representation of a state machine and associated rules. Techniques like extended FSMs, Petri nets, abstract state machines fall under this category. Lastly, in the *temporal logic techniques*, which are especially useful for reactive systems, in which the protocol is described in terms of statements that implicitly incorporate the relative ordering of events and their actions. IEEE's "property specification language" (PSL) (IEEE 1850 standard) is an example specification language rooted in temporal logic that is commonly used in hardware design where it is a common practice to augment design with assertions serving to specify correct behavior.

There are many standard *formal description languages* for protocols [19]. The Estelle language [98] and the SDL language [101], specified by CCITT/ ITU, are based on a extended state model. The LOTOS language [102], on the other hand, is based on a temporal logic model. The Z (pronounced Zed) language [99] is a popular formal specification language useful for describing *transformational* systems such as sequential programs in Hoare style using pre- and post-conditions. PROMELA is a specification language used for specifying LTL formulas that can be used for validation of *reactive* systems with the SPIN model checker. The interested reader is referred to a tutorial article [19] for more details about formal description and specification techniques such as SDL, Estelle, PROMELA, LOTOS, etc.

V. Techniques for Formal Verification

Various approaches have been proposed for formal verification which include both automated and interactive techniques. We discuss model checking as an example automated method in section V-A. We will discuss theorem proving—a technique that can automated for decidable logics such as propositional or first-order logic, but which works in concert with a human expert for dealing with the undecidable higher-order logic as a proof-assistant—in section V-B. In the later part of this section, we will discuss light-weight formal methods, static analysis, and symbolic execution & simulation in sections V-C, V-D, and V-E, respectively.

A. Model Checking

Developed independently in 1980's by Clarke and Emerson[7] [41], and by Queille and Sifakis [105], model checking can be envisioned as an automated debugging, or exhaustive simulation and testing, technique useful for checking any property violations (i.e., bugs or errors) [106]. While formal verification has traditionally been associated with logic-based axiomatic or deductive techniques for establishing proofs of correctness, model checking has been the first step towards engineerization of this field [106] [87].

The main insight of model checking is that proof construction—a tedious and non-trivial task requiring good deal of ingenuity and guidance from the user—is not necessary for the case of finite state concurrent systems. In proof-based verification, we are interested in showing $\Gamma \vdash \phi$ where Γ is a *set of formulas* representing the system description in a suitable logic, and ϕ is another formula representing the specification. We are interested in a deductive proof $\Gamma \vdash \phi$. Given a logical proof system that is sound and complete, $\Gamma \vdash \phi$ holds iff $\Gamma \models \phi$ (semantic entailment). Semantic entailment is undecidable for first-order logic while model checking is decidable. In model checking, we are interested in showing that $\mathcal{M} \models \phi$ where $\mathcal{M}$ represents a Kripke structure[8], or a labeled transition graph, as a model of system description while the specification is still a formula (typically written in propositional temporal logic). More specifically, the model checking problem is (from [107]): "Let $\mathcal{M}$ be a Kripke structure (i.e., a state transition graph), ϕ be a formula of temporal logic (i.e., the specification). Find all states s of $\mathcal{M}$, such that $\mathcal{M}, s \models \phi$ (i.e., $\mathcal{M}$ has property ϕ at that state s)". As discussed earlier in section IV-A2, Kripke structures are *labeled* FSM with the states labeled with a sets of atomic propositions that are true in this case; all other unlabeled propositions are assumed false according per the "closed-world" assumption. This model checking can be performed for finite state systems algorithmically, unlike proof systems, in a push-button fashion. In model checking, the verification procedure intelligently searches through the entire state space of the design in an exhaustive fashion [41], and thus this technique is applicable for finite state systems[9]. Although this looks limiting, many interesting systems (e.g., hardware devices, communication protocols, etc.) can be modeled as FSMs in practice.

It is important to ensure that the term "model" in "model checking" is not confused with its everyday usage of being an abstraction of the actual system under study. In the case of "model checking", the inventors of this method were interested in the model-theoretic interpretation [108] [109] of the term 'model'—i.e., determining that $\mathcal{M}$, representing the system interpreted as an automaton, is a (Kripke) *model* for the temporal logic formula ϕ representing the desired property [107]. It should be noted that when we say that $\mathcal{M}$ is a model for the formula ϕ, we really are paraphrasing our intention of saying 'ϕ, when interpreted as in $\mathcal{M}$, is true'. Noting the distinction between the various interpretations of models can alleviate any unnecessary confusions. To summarize, model checkers are named such because they check whether a system, interpreted as an automaton, is a (Kripke) model of a property expressed as a temporal logic formula.

Model checking has many benefits over deductive proof techniques which makes it preferable wherever it is applicable. Some compelling benefits of model checking [107] include: *i)* it is fast compared to other rigorous methods, *ii)* it provides diagnostic counterexamples, *iii)* it can work well with partial specifications/ properties, *iv)* logics can easily express various concurrency properties, and finally, *v)* it does involve any human-guided proofs.

Buchi automata has been used in model checking as a bridge between automata theory and temporal logic. In particular, Buchi automata can provide an automata-theoretic formalization of a linear temporal logic, or LTL, formula. It was shown in the mid 1980s that there exists for every temporal logic formula a Buchi automaton that accepts precisely those runs that satisfy the formula. There are algorithms that can mechanically convert any temporal logic formulae into the equivalent Buchi automaton. Typically, the property invariants are expressed as LTL formulas, and a negated version is converted to Buchi automata to be used in the model checking algorithm to detect violation of the desired property.

Scalability of model checking: The state explosion problem limits the application of model checking to large scale problems. Various approaches have been proposed for coping with this issue including symbolic model checking, bounded model checking, and statistical model checking. These approaches are covered next.

Symbolic Model Checking:

The main insight of symbolic model checking is that it is more efficient to consider large number of states simultaneously at a single step instead of traversing enumerated reachable states one at a time. Symbolic model checking facilitates such a state space traversal by allowing representations of states set and transition relations as Boolean encoded formulas, BDDs, or related data structures. This allows handling of much larger designs containing hundreds of state variables [110]

[7] For a historical account of the development of model written, the interested reader is referred to [103] and [104] (written by Emerson from his personal perspective)

[8] A Kripke structure, proposed by Saul Kripke, is a nondeterministic automaton representing a system's behavior. Kripke structures are commonly used in model checking for interpreting temporal logics.

[9] Infinite state can only be analysed with abstraction [86] and induction.

[111] [112]. Symbolic algorithms can thus work with the FSM represented implicitly as a formula in quantified propositional logic without the need of explicitly building a FSM graph. In summary, a symbolic model checking method is a model checking method that represents state sets symbolically, typically using OBDDs, as opposed to an explicit enumeration of states. Symbolic model checking is the most commonly used variant of model checking used by most industrial scale model checking tools. The first symbolic model checking tool, SMV, was developed by McMillan in 1992 and used BDDs to combat the state explosion problem [113]. More recently, SMV has been extended and reimplemented as NuSMV and NuSMV2 [49].

Bounded Model Checking:

Symbol model checking can also be performed through SAT procedures [114]. SAT procedures can operate on Boolean expressions without requiring canonical forms and without the potential space explosion of BDDs. Various efficient implementations are available for solving SAT problems. Bounded model checking (BMC) uses a SAT procedure instead of BDDs [115]. A Boolean formula is constructed that is satisfiable *iff* there is a counterexample of length k. By incrementing the bound k, longer counterexamples can be searched. If after some number of iterations, we may conclude that no counterexample exists and the specification holds. The state explosion problem is thus handled by focusing on falsification rather than exploring all reachable states. Incorporation of the falsification approach into a SAT based framework in a BMC allows scaling to much larger number of states. BMC techniques using the falsification approach are very useful since in many practical scenarios, we are more interested in finding bugs as early as possible in the design rather than in formally proving the correctness of the design. SAT-based BMC for falsification is a very popular model checking technique in the industry. As an example, safety property may be verified by increasing the number of iterations to the bound defined by the diameter of the FSM. The advantage of the bounded model checking approach is that it can quickly find counterexamples due to the depth first nature of SAT search procedures. Secondly, since the bound is increased incrementally, the approach finds the counterexample of minimum length which leads to better diagnostics. Finally, it also uses lesser space as compared to BDD-based approaches. The NuSMV2 tool [49] incorporates both BDD-based and SAT-based model checking. BMC can also be performed using SMT tools [116]. BMC tools include a CBMC [117] which is a bounded model checker for ANSI-C and C++ programs.

Statistical Model Checking:

Statistical model checking is a proposal that can allow model checking to scale to large systems by relaxing the requirements of formal correctness. The key insight is to use hypothesis testing with a simulation based approach to deduce from some sample executions if the system under test satisfies the specification [118].

Probabilistic Model Checking:

Various approaches have been proposed for building probabilistic model checking tools [119] [120] [121]. PRISM is an example probabilistic model checking tool that can be used for reachability analysis [122] and protocol verification [123]. While traditionally, establishing performance evaluation and correctness have been orthogonal tasks, a promising new direction in formal methods research is to develop probabilistic methods that can allow joint analysis of both correctness and performance [124].

Model checking for Software:

Model checking is not inherently well suited for verifying software due to the asynchronous and unstructured nature of software. While, the early successes of model checking were mainly in hardware verification, recent progress has made model checking viable for software verification [125]. Popular model checking softwares include Java Pathfinder [126], Microsoft's Slam Toolkit [95], UC Berkeley's BLAST [127]. The interested reader is referred to a detailed survey on model checking for software for more details [125].

Applications of Model Checking to Networking:

There are a great number of *model checking tools* that have been devised with some popular model checkers being SPIN [46], NuSMV [49] and Alloy [52]. SPIN, developed in early 1980s by Holzmann for assuring dependability in complex telephone switching systems, is a popular award-winning[10] *explicit-state* model checking tool. SPIN was the first model checker developed, with its initial focus being on telecommunication systems and protocol verification. SPIN is now used for diverse applications from hardware verification to distributed control software used in nuclear power plants and spacecrafts. The IEEE Futurebus cache coherence protocol is the first IEEE protocol whose specification was debugged successfully through model checking. NuSMV, in contrast to SPIN, is a *symbolic* model checking tool that also incorporates features of *bounded* model checking. NuSMV was the first implementation of symbolic model checking and was developed by McMillan in 1992 [113]. NuSMV can utilize both BDD-based and SAT-based techniques. Alloy is also a symbolic model checker that translates constraints into Boolean formulas which are then solved through an external SAT-solver. SPIN and NuSMV support temporal logic for property specifications with SPIN supporting propositional LTL and NuSMV supporting CTL. For model specification, SPIN uses the PROMELA language (which is inspired by C) while NuSMV uses the SMV description language to specify finite state machines. Alloy uses first-order logic for both model specification and property specification. A detailed comparison of SPIN, NuSMV, and Alloy, and some other model checking tools, is presented in [128]. Popular model checking tools are listed in table III, along with other popular formal verification tools, for quick reference. Apart from SPIN, NuSMV, and Alloy, it is worthwhile to mention two other popular types of model checking tools. The PRISM tool [122]

[10]The SPIN model checker has been awarded the ACM Software System Award http://www.acm.org/announcements/ss_2001.html.

is a probabilistic model checking tool, while the UPPAAL tool [85] is a model checking tool based on timed-automata which can be used for verification of real-time systems.

Model checking techniques and tools have been extensively applied in the context of networking, and we will present a representative sample. Zave et al. have used model checking to understand SIP [129]. Al-Shaer et al. have used model checking for configuration analysis for general networks [50] and for SDN networks having federated OpenFlow infrastructures [80]. In [80], network routing tables are represented as BDDs and reachability predicates are computed using model checking. In other works for OpenFlow networks, Canini et al. present the model checking based NICE platform for verification [130], and Son et al. present a model checking based security invariant property checker [69]. Most of the model checking work has focused on verifying safety property since verifying liveness property entails computing an infinite long trace of states in which the desired property is never reached with heuristics-based MaceMC being a notable exception [131]. A summary of various applications of model checking techniques in the context of networking is presented in table II.

B. Theorem Provers

In the theorem proving paradigm of formal verification, the relationship (*implication* or *equivalence*) between the specification and the implementation is considered as a proof goal, which is verified using a computer-based tool called a theorem prover. The dream of having automated theorem provers is a long-standing dream of many an ambitious scientists starting from Leibniz, to Peano and Hilbert [132]. Herbrand in 1930 provided a mechanical method for proving theorems but due to lack of appropriate computing facilities the method was difficult to apply. In 1936 Church and Turing showed that it is impossible to devise a generic method of verifying the validity of first-order logic. First-order logic is said to be semi-decidable in that methods exist for verifying validity of a formula if it is indeed valid, however, such methods will never terminate in general for invalid formulae. This has defined the limits of automatic theorem proving. In the 1960s, Herbrand's method was implemented on a digital computer, followed by an even more efficient Davis-Putnam-Logemann-Loveland (DPLL) algorithm [133]. The resolution principle, proposed by Robinson in 1965, has been a major step forward. The DPLL algorithm is important for many applications including automated theorem proving and satisfiability modulo theories (SMT). The DPLL algorithm is used to solve the CNF-SAT problem—i.e., determine the satisfiability of propositional logic formulae in conjunctive normal form (CNF).

Despite the theoretic complexity of automated reasoning in expressive logics, in practice, *interactive theorem provers*—also known as proof assistants—which solve the proof verification problem are useful in many settings. Interactive theorem provers differ from automatic theorem proving in that it requires human assistance. A proof assistant is a program that takes a formalized mathematical statement and a plausible proof, and checks whether the proof is valid. There are three key ingredients of a proof assistant. Firstly, it needs to incorporate an expressive formal language and logic—which is typically, but not always, a variant of higher-order logic. Secondly, it needs to have support for checking proofs and in aiding proof construction. Lastly, it needs to have a programming language—typically a functional programming language—that allows extending the system with new proof procedures (e.g., decision procedures). There are various interactive theorem provers that have been proposed including tools that are based on first-order logic (e.g., ACL2 [134], Microsoft's Z3 tool [66]—which uses many-sorted first-order logic, etc.) and others that are based on higher-order logic (e.g., Isabelle [135], HOL [136], PVS [137], Coq [138], etc.).

Applications to networking: Theorem provers have many applications in networking research. As specific examples, we will discuss three theorem proving tools that are popularly used in networking research. The Coq tool, which incorporates higher-order logic along with richly-typed functional programming language, defines a system for manipulating and mechanical verification of formal mathematical proofs by machines [138]. Coq also supports extracting certified programs to popular functional languages like OCaml, Haskell, etc. The Coq tool has been used for verifying the network controller in SDN environments [139] and for ensuring per-packet and per-flow consistency of network updates [48]. The Z3 tool from Microsoft, which uses a portfolio of solvers, is another popular theorem prover used in many software testing, analysis and verification projects [66]. Finally, the Isabelle/HOL theorem prover has been used for network verification and the BGP policy verification [135] is a notable example in this regard.

Besides the functional verification, theorem provers have also been used for the formal performance analysis of network applications based on the higher-order-logic formalizations of probability theory [36] and Markov Chains [140]. Some notable examples in this regard include the performance analysis of the Stop-and-Wait protocol [141], scheduling algorithms of Wireless sensor networks [142], the memory contention problem in multiprocessor systems [143] and the quantitative analysis of information flow in a network [144].

C. Light-weight Formal Methods

"Full-blown" formal methods, such as model checkers and proof systems, have some limitations due to which there is interest in alternative "light-weight" formal methods [145]. Proof systems, like theorem provers, have the deficiency that they cannot be fully automated due to fundamental limits of computation. Model checkers, on the other hand, are not inherently suited to software systems due to the state explosion problem, and since they cannot deal with indirection which is a fundamental concept of software [52]. To avoid the overhead of full-blown formal methods, fully automated analysis methods based on lightweight formal methods [146] have been proposed that exploit advances in technologies such as SAT solvers.

The *Alloy analyzer* works by translating constraints to be solved from Alloy into Boolean constraints which are then fed to an off-the-shelf SAT solver. Alloy is also known as a *model-finding* tool since it aims to find an instance of a

counterexample, known as a model in logic theory, quickly rather than for completeness. Alloy defines both a language for describing structures and also a tool for exploring them—in particular, it specifies a new high-level language, inspired from Z [99], for specifying the structure an behavior of software; secondly, it uses an automated SAT-solver based analyzer to work through all the possible scenarios. The software design modeled with a high-level notation is then analyzed over billions of possible executions to catch any pathological conditions. The important consequence is that subtle design errors are caught even before the design is coded. The design, once thoroughly tested, can then be constructed with much more confidence. Alloy true to its style of being a light-weight formal method works on a analyze-first-then-prove principle. Alloy represents a new generation of software analysis engines similar in principle to tools traditionally used for verification of hardware designs [147].

Applications to networking: Light-weight formal methods in general, and particularly the Alloy tool, have been widely deployed to solve a wide variety of problems ranging from security analysis [148] to the design of telephone switching networks [5] [149].

D. Static Analysis

Static analysis is a class of techniques concerned with extracting information about the run-time behavior of a program, or a configuration file, without actually executing the source file. Static analysis which means analysis without execution (e.g., SLAM [95]) is to be contrasted with dynamic analysis which involves executing the program (e.g., Verisoft [12]). Static analysis can discover bugs in configuration files, or software systems, before they are activated or executed thus obviating the reflexive debugging that results from discovery of bugs after deployment. Due to the fundamental limits imposed by the theory of computation (cf. Turing's halting problem which is notorious for being undecidable), static analyzers *cannot* extract run-time behavior of all programs perfectly. Static analyzer attempt to defy the undecidability of the halting problem by not focusing on *completeness* or *soundness* but instead on quick and efficient debugging. The key insight used by static analysis is to utilize an approximate interpretation, or an abstract interpretation, of the program. In many cases, this approximate interpretation is finite, and thus amenable to analysis.

The term soundness has a background in mathematical logic where a system is said to be sound if it can only prove valid arguments with respect to a semantics . In the context of debugging, soundness means the ability to detect *all* possible errors of a certain class, or not miss a bug if one exists—in other words, a sound debugger will give no *false negatives*. Completeness, in contrast, implies that there will be no *false positives* which requires exhaustive analysis of every possible scenario. An effective static analyzer, thus, has to balance three desirable, but often competing, costs: *i)* the cost of false negatives due to being unsound, *ii)* the computational cost of analysis, and *iii)* the usability of the tool (which can be measured in total time investment of the user) [150]. In particular, it turns out that soundness and completeness have a tradeoff. For *assurance* based projects where soundness is needed (i.e., if told that there are no errors, we should be sure that there are none), we are limited to accepting incompleteness, or to accept false alarms or false positives. The presence of false alarms is usually irritating for customers of debugging tools—who aim incidentally to reduce the number of bugs and not necessarily eliminate all of the bugs—who often give up on soundness to reduce the number of false alarms. Most commercial debugging tools (such as Coverity, etc.) are neither sound nor complete, but perform well in practice catching many errors with lesser number of false alarms.

It is instructive to compare static analysis with model checking directly [151]. In general, model checking has some benefits that are hard for static analysis to match: e.g., *i)* it can check the implication of code, and not just surface-visible properties, *ii)* it gives stronger correctness results, etc. A major drawback of model checking approach is the need to create a correct working environment model—this restrictions makes model checking infeasible for many networking verification tasks [152] and adds significant overhead even when feasible especially for large scale systems. Also, no model is as good as the implementation itself, and the abstraction in the modeling process is a potential for producing false positives or missing critical errors. Static analysis is more useful than model checking in some aspects: e.g., *i)* it is quicker, *ii)* it can easily check millions of lines of code, *iii)* it can find thousands of errors. Some of these comparisons are direct outcomes of the fact that static analysis does not run any code, while model checking does [151]. Static analysis is a widely used technique used in many software testing tools (e.g. Coverity, FindBugs, etc.) that can analyze extremely large code-bases [153].

Static analysis is familiar to all programmers in its most basic form of typechecking in compilers (e.g., a Java compiler will catch errors such as adding a number to a Boolean, etc.) This kind of static analysis focuses on simple checking with no false alarms and thus only scratches at the surface of what can be achieved with static analysis. More extensive static analysis requires more computation but can check a wider range of properties—e.g., runtime exceptions due to division by zero, array bounds violation, etc. can be detected. Since such analysis is difficult to do precisely, such extensive static analysis can involve false positives (non-errors reported as errors) and false negatives (non-reported errors).

Extended static checking defines a powerful paradigm for program checkers in which verification conditions—i.e., a logical formula that is valid iff the program is free of the classes of error under consideration—are defined, and then counterexamples to the verification condition are searched mechanically [154]. Extended static checking for Java (ESC/Java) [155] is a compile time program checker that performs formal verification of properties of Java source code through theorem proving. ESC/Java provides an annotation language, which is effectively a subset of Java Modeling Language (JML), which a programmer can use to add Hoare-style preconditions and postconditions and loop invariants into the program with special comments in the source code.

Applications to networking: Static analysis has also been used in networking context most notably for reachability analysis in IP networks [156], firewall analysis (e.g., Margrave [157], etc.), BGP configuration fault detection [152] [158], etc. It can also be used for debugging of networking software using techniques described above.

E. Symbolic Simulation and Execution

Symbolic execution [159], also called symbolic evaluation, is a 'abstract interpretation' method for analyzing a program assuming symbolic values for inputs rather that actual inputs that would arise through the normal execution of the program. Symbol execution is essentially a technique for generating an optimized test suite that satisfies a customizable coverage criteria using which deep errors in software applications may be identified. Although the idea of symbolic execution is quite old (proposed by King in 1976 [160]), symbolic execution has emerged as an effective tool recently with advanced in constraint satisfaction tools. Symbolic execution proceeds by exploring as many program paths as it can in a given time budget, thereby creating a logical formula encoding the explored paths. A constraint solver is then used to calculate feasible execution paths. Symbolic execution are much more powerful than dynamic execution techniques, such as those incorporated in popular debugging tools like Valgrind [161], since it can find a bug if there exists *any* buggy input on a path without depending on a concrete input that triggers the bug. Symbolic simulation [162] is an extension of the idea of symbolic execution to hardware systems. Simulation is a time-test tool for formal verification. Simulation can be generalized in two different ways: *i)* ternary simulation [163]—where we have a "don't care" value X in addition to 0 and 1; *ii)* symbolic simulation—where Boolean variables can act as input parameters and outputs are functions of these parameters. Ternary symbolic simulation unfortunately suffers from the problem of large growth in the state space leading researchers to look for alternative techniques in recent times [22].

Application to networking: Header Space Analysis (HSA) [164] is an example ternary symbolic simulation implementation proposed recently for verifying various properties such as reachability, loop detection, etc. for SDNs. Canini et al. have proposed a symbolic execution and model checking based NICE framework for catching bugs which works by exploring symbolically all possible code paths [130]. In another work, Bishop et al. [165] have proposed symbolic evaluation testing of TCP implementation against a HOL specification.

VI. Programming Languages and Verification

There are three ways to establish the meaning, or *semantics*, of computer programs [166]. In *operational semantics*, the program is modeled by execution on an abstract machine—this interpretation is useful for implementing compilers and interpreters. In *axiomatic semantics*, pioneered by Hoare and Floyd, the program is modeled by the logical formulas it obeys—this interpretation is useful for proving program correctness. In *denotational semantics*, the program is modeled by mathematical objects—this interpretation is useful for developing theoretical foundations of programming.

In the remainder of this section, we will discuss the grammar of languages, declarative programming, logic programming, and functional programming in sections VI-A, VI-B, VI-B1, and VI-B2, respectively.

A. Grammar of Languages

The most common type of grammar used for specifying languages is known as the *context-free grammar*. Context-free grammars are expressive enough to capture the recursive syntactic structure of most languages of our interest. The core component of a context-free grammar is a set of rules where a rule typically defines a name and an expansion for that name. The Backus-Naur form (BNF) is a formal notation used for encoding the grammar of a language in a form amenable to human consumption. The BNF notation is used by many programming languages, protocols or formats in their specification. A rule of the BNF notation has the following structure: "$name ::= expansion$" where the symbol $::=$ means 'expands to' or 'may be replaced with'. Every name in BNF is enclosed in angle brackets, $<>$. Choice is indicated by a vertical bar, $|$. For more details about the BNF format, the interested reader is referred to [28]. The BNF format is used to specify network programming languages in FlowExp (short for Flow Expression) [79], NetCoreLib for Frenetic [167] [168], etc.

B. Declarative Programming

Declarative programming is a programming style in which we specify what the program must do without specifying how to do it. The imperative programming style adopted by imperative languages such as C, Java, etc., in contradistinction, focuses on specifying algorithmically how the computer must do its job. It may be highlighted that the imperative programming style harmonizes with the imperative procedural (how to) approach typically adopted in computer science while the declarative programming style dovetails with a mathematical or logic-based approach which emphasized declarative (what is) knowledge [169]. Imperative programming style involves the use of mutable state variables which makes reasoning and verification a difficult task. Declarative programming style, in contrast, eschews maintenance of state variables and avoids invisible side-effects and relies instead of mathematical logic and evaluation of mathematical functions and logic formulae. Declarative programming is intimately tied to mathematical logic—programs in a declarative frameworks can be thought of as theories of formal logic, and computations as deductions in that logic space. Examples of declarative languages include SQL, frameworks such as: functional programming languages, logic programming languages, constraint logic programming, etc. In recent times, there has been a lot of interest in declarative languages, and in their use in networking especially cloud networking [170], since declarative languages are well-

suited to parallel programming[11] [172]. Adoption of declarative programming languages is a manifestation of a contemporary trend in networking, brought on by the need to fix an ailing inflexible network architecture—and by software defined networking in particular, in which advanced programming techniques and database techniques are increasingly being applied to networking [173]. We present two examples of SDN declarative languages: *i)* the flow management language (FML) is a declarative language for SDN [174] designed for network operators so that static network policies may be written and maintained more efficiently; *ii)* the NetCore language [168] is a high-level declarative language proposed for SDN that allows programmer to describe what behavior is desired and not necessarily describe how to realize the implementation of that behavior. We will discuss SDN programming languages in more detail in section VII-F2.

1) Logic Programming: Logic programming provides many advantages including programmability at a very high level and natural support for formal semantics. Logic languages, such as Prolog, Lisp, have been very popular in the AI community for knowledge representation and automated reasoning. Prolog, an example declarative logic programming language designed primarily for AI based systems, works by stating and querying the logical relations between entities. Prolog like languages are also useful in formal verification for automated theorem proving. Logic programming languages are also popular in the databases community of computer science due to their support for declarative querying and symbolic manipulation. Datalog, an example database-based logic programming language, facilitates declarative definition of properties and relationships between objects with the language framework providing support for computing with these objects (including querying about objects declaratively).

The declarative programming style is superior to the procedural style in some significant ways—especially, in the context of formal verification [175]. The declarative style emphasizes the intent of a program and the static description of relationships and properties that hold in a program regardless of the computing context, thus easing understanding a computer program and reasoning about it. Unlike procedural languages, the effect of logic programming statements is not dependent on the context (i.e., the state of the computer when the preceding statements were executed).

There has been a lot of interest in using declarative logic programming languages for simplifying the implementation of Internet protocols. They have been previously used for writing parsers (like the 'yet another compiler-compiler' (yacc) parser tool [176]) for application layer protocols [177], declarative routing [178], and declarative networking [173]. The basic insight behind declarative networking is the realization that recursive query languages are a natural fit for network protocols which essentially deal with computing and maintaining distributed state (such as information about routes, sessions, etc.) across the network. Network Datalog (NDlog) is a data and query model that has been proposed for declarative networking. NDlog, which implements a network specific subset of Datalog and supports distributed programming, exposes the partitioning of data across nodes and the link graph of the network. This makes the implementation much more amenable to static analysis and verifiable using other formal verification techniques such as general-purpose theorem provers. It has been shown that declarative implementations of popular protocols can be done much more concisely and efficiently while also allowing extensibility and safety [173]. Logic programming languages have recently been proposed for SDNs [179], FlowLog: [180], with researchers also exploring declarative network verification [181]. In another work, Kazemian et al. have implemented a FML-like language in a Prolog frontend to enable network administrators to specify high-level policies [79].

2) Functional Programming: The functional programming paradigm considers computation to be the evaluation of mathematical functions—that are not dependent on state and will always provide the same output for the same input. The main reason for the importance of functional programs is due to their direct correspondence with mathematical objects, which makes it easier to reason about them [182]. The functional paradigm avoids variables, or more technically—mutable state (i.e., variables whose values can be changed), and encourages a function-based programming worldview instead. By avoiding mutable state, the source of numerous subtle bugs in imperative-style programming languages, verification of programs become more simple. In functional programming, execution of a program means evaluation of the expression represented by the functional program. The functional programming style makes no use of variables. Instead of loops, the functional program makes use of recursive functions (i.e., functions that are defined in terms of themselves).

The main downfall of imperative programming is in race conditions when concurrency is supported. Race conditions are much harder to detect and fix since they may arise of nondeterministic interleavings of concurrent threads (which may interleave in a myriad different ways). Imperative programming is always vulnerable to race conditions since it relies on mutable state. Functional programming puts up much better with such race conditions since a pure functional language has no mutable state. Since the future of programming is in concurrency and parallelism, functional programming is increasingly migrating from fringes of the programming world to the mainstream [171]. In summary, mutations allowed in the imperative programming paradigm severely limit any opportunities for automatic parallel execution, while the lack of dependencies in the (purely) functional paradigm presents great opportunities for automatic parallel execution.

The functional programming is based on the theoretical underpinnings of Alonzo Church's *lambda calculus* [183], proposed in the 1930's, defines rules about using unnamed functions for representing and evaluating expressions. Lambda calculus, although originally intended as a formal logical system for mathematics is in fact a completely general pro-

[11] Almost every successful large-scale application of parallelism, e.g., SQL server, LINQ, MapReduce, etc., has been declarative and value-oriented [171]. This trend bodes well for the use of declarative programming, especially functional programming, in parallel computing.

gramming language and defines a family of prototype programming languages. Many modern programming languages C++, Python, JavaScript, Ruby, Java 8, etc. borrow from the programming style of lambda calculus, following the lead of the Lisp programming language—which was the first mainstream language to include anonymous functions known as lambda functions. Two important features of lambda calculus is that it is functional—i.e., it is based on the concept of a mathematical function and include notation for function application and abstraction—and that is higher-order—i.e., it provides a systematic formalism and notation to deal with operators whose input and output may be other operators. The lambda calculus model significantly differs from the Turing model of a store with evolving state [184]. Interestingly, the Turing model and lambda calculus were invented in the same year, 1936. Turing showed in 1937 that both these models were equivalent and in fact defined the same class of computable functions. In any case, computer programs and mathematical proofs are directly related as the system of formal logic and computational calculi are analogous—the famous "Curry-Howard correspondence" expresses the isomorphism between proof structures and functional spaces [185].

Popular functional programming languages include Lisp, invented by the AI pioneer John McCarthy, Haskell [186], Caml, OCaml, Scala, etc. Historically, most successful languages have been written for specific purposes—e.g., Lisp was created for artificial intelligence, Fortran for numerical computation, and Prolog for natural language processing. The *raison d'etre* for ML has been the need of an efficient language for theorem proving [187]. ML originated as the metalanguage (thus its name ML) of the famous theorem proving system called Edinburgh LCF [188] for writing theorem proving algorithms in formal deductive calculus. ML was designed to have the full power of higher-order functional programming so that it could represent necessary inference rules and proof strategies. Since early time, functional languages and theorem proving (and formal verification in general) have been intimately intertwined. (Edinburgh) ML has spawned a wide range of ML-based descendant languages including Standard ML (SML) and OCaml. OCaml is a programming language specifically designed for writing theorem provers, with numerous major systems being written in it (e.g., SLAM verification system from Microsoft, HOL Light theorem prover, etc.). The OCaml language, being perfectly suited for symbolic manipulations, is used extensively by the Coq proof assistant which is used extensively for the verification of purely functional programs. Similarly, the SLAM verification system, proposed by Microsoft, also used OCaml programming language. The ACL2 ("A Computational Logic for Applicative Common Lisp") theorem prover is also composed of a first-order, purely functional subset of Common Lisp.

With the recent paradigm shift in networking brought on by SDN, a clear trend of preferring high-level declarative languages, domain-specific languages (DSL), functional languages—and more specifically, *functional reactive programming*—for programming SDNs (both the SDN controller as well as SDN applications) is emerging. Much of the recent work in SDN programming has followed the declarative programming coupled with the FRP paradigm [189] [167] [190]. This trend is helped by ample foundational research in these fields in the programming and databases community, and by the verifiability properties of functional languages.

Nettle [191] is a SDN specific language implemented as a domain-specific language in the functional programming language Haskell. Nettle adopts the design methodology of domain-specific languages (DSL) research, and is built in the paradigm of *functional reactive programming* (FRP) [192]. Nettle has been used for providing a comprehensive abstraction calculation constructs for configuring BGP policies. In a similar work, Procera [189] is a domain-specific language embedded in Haskell that can be used to specify high-level dynamic reactive network control policies. In other work, the Frenetic project [193] defines a family of domain-specific languages for specifying high-level network policies. In the initial work in the Frenetic project [167], two sub-languages were proposed: *i)* a high-level declarative network query language—which enable Frenetic programs to read the network state using constructs for filtering, grouping, splitting, limiting, aggregating, etc., and *ii)* a general purpose FRP-based network policy management library—using which the policy to govern the forwarding of packets through the network can be defined. The Frenetic framework borrows extensively from the FRP languages like Yampa [194], etc., and reuses many of the proposed primitives. In more recent work, Pyretic [190] is an example DSL in the Frenetic family which supports composable policies constructed from a set of fundamental constructs such as basic policies and combinators along with associated techniques for compiling these techniques to OpenFlow switches.

VII. Applications of Formal Methods in Communication Networks

In the history of the Internet, formal correctness has mostly taken a backseat to practical expediency and pragmatic considerations. The development, standardization, and deployment process is cumbersome and inflexible leading to an environment which only just works [195]. As an example of the unfortunate adhocism that pervades the culture of network protocols, it is noted that BGP, despite any lack of convergence guarantees, is often used in service as an interior-gateway routing protocol (IGP) [76]. While there were some initial successes in the application of formal methods to networking [149], the networking enterprise quickly transformed into a complex behemoth impervious to any attempts at formal analysis and verification. In addition to the inherent complexity of networking protocols, the vertical integration of control and data planes meant lack of modularity and a paucity of useful abstractions [4]. With the tools of formal methods unable to tame the staggering complexity of networking, the resulting frustration bred skepticism leading to a widespread critical view, enunciated by Vint Cerf [15], that formal methods are "overblown, verbose, hard to use, (and) hard to understand". Fortunately, modern attempts at redesigning the Internet ossified architecture—and more specifically, the SDN movement—create new abstractions by separating the control and data

planes and thus allow a great opportunity for incorporating formal methods in networking. With the utility of Internet firmly entrenched in all aspects of modern life, the use of formal methods for ensuring correctness of specification and operation is anticipated to be incorporated into mainstream Internet operations.

In the remainder of this section, we will describe various applications of formal verification methods to networking. We will discuss protocol verification in section VII-A. We will follow it up in section VII-B with a discussion on property verification and discuss reachability analysis, loop detection, and isolation verification, in sections VII-B1, VII-B2, and VII-B3, respectively. We will then discuss the use of formal verification for network configuration management, and network security in sections VII-C and VII-D. We will discuss various issues related to generic network verification in section VII-E; in particular, we will discuss declarative verification, hardware verification, formal specification and synthesis, and implementation variation in sections VII-E1, VII-E2, VII-E3, VII-E4. Finally, we will wrap up this section with application of formal methods specifically to SDN in section VII-F. More specifically, we will discuss the new opportunities created by SDN in section VII-F1, and follow it up with discussions on SDN programming languages, data plane verification, control plane verification, and network debugging in sections VII-F2, VII-F3, VII-F4, and VII-F5, respectively. A tabulated summary of various applications of formal methods in the context of networking is presented in table II. A summary of various tools that are used in this regard is presented separately in table III.

A. Protocol Verification

In layered communication networks, protocols define the set of rules governing exchange of messages between interacting processes which serve two related goals: firstly, to provide service to the local protocol layers above, secondly, to interact according to a defined protocol with remote peer partners on other machines. In terms of specification, the former goal is defined through service specification, while the latter is defined through protocol specification. Both these specifications—service-specification and protocol-specification—can be verified against their design or implementation. Verification at the design stage is more useful as it can avoid unnecessary incorrect implementation [211].

Holzmann [47] [8] lists three ways in which protocols can fail: *deadlocks*—when all the protocols stall waiting for conditions that can never be fulfilled; *livelocks*—when execution sequences keep getting repeated indefinitely without the protocol making any effective progress, and *improper terminations*—when the protocol completes execution without satisfying the proper terminating conditions. The general problem of finding deadlocks in protocols is known to be complex, i.e., PSPACE-complete at best which makes it undecidable for unbounded message queues. Thus any method that relies exclusively on an exhaustive search of state space method is bound to fail, thus prompting much research on alternate non-exhaustive methods that exploit symmetry and abstraction. Also, due to the inherent complexity of the problem, we set a more conservative target in protocol verification of detecting the presence of errors—should they exist—with high probability instead of striving to prove the absence of errors with certainty.

Various works have been proposed for protocol verification including rigorous specification and conformance testing techniques for network protocols [212], rigorous treatment of the TCP protocol [165] [196], verification of ad-hoc routing protocols for wireless sensor networks: [213]. There have been a few survey papers written focusing on communication protocols [214] [7], including a FSM-based protocol verification survey [211] and a survey documenting experience with protocol description [5]. Various tools have been used for protocol verification including the theorem proving tool Isabelle [135] for BGP policy verification and the proof assistant Coq tool [138] for creating a featherweight version of the OpenFlow protocol [139].

B. Property Verification

There is great interest and intent in the research community to develop technological support for automatic verification of various properties of protocols and systems. When we are verifying the property of a system, we are essentially interested in two kinds of properties: *i) Safety property* where we are mainly interested that 'bad' things will not occur, *ii: Liveness property:* where we are mainly interested in that 'good' things will eventually occur [100]. In general, safety property bugs are easier to discover by finding a counterexample, while liveness property violations are difficult to obtain—in particular, a liveness violation example would require finding an infinitely long execution trace in which the desired 'good' property never happens. [131]. Recent works such as Anteater [70], Header Space Analysis (HSA) [79], FlowChecker [80], VeriFlow [198] use an automatic solver to check properties of a logical representation of switch configurations.

In the remainder of this section, we will cover example properties of reachability analysis, loop detection, and isolation verification, packet destination control.

1) Reachability Analysis: Reachability analysis is a powerful method widely used for formal verification of protocols [78] and concurrent distributed systems. Unfortunately, reachability analysis suffers, like all methods based on finite state machines, from the state-explosion problems. Reachability analysis can benefit from symbolic methods which work without inspecting all the reachable states of the system to scale to large networks—e.g., BDD-based symbolic traversals have been proposed for reachability analysis of large finite state machines [215]. An example work that utilizes BDD-based symbolic model checking for reachability analysis is the 'Network configuration in a box' project by Al-Shaer et al. [50].

In a networking context, reachability analysis was first proposed for IP networks by Xie et al.[156]. The technique proposed utilized a static snapshot of network configuration, culled from configuration state from each of the network routers, for determining reachability between applications running on end-hosts. This reachability information is very useful

TABLE II. SUMMARY OF **APPLICATIONS** OF FORMAL VERIFICATION IN NETWORKING

Project and Reference	*Technique*	*Brief Summary*
Protocol Verification		
Bishop et al. [165]	Symbolic Evaluation	Proposed symbolic evaluation testing of TCP implementation against a HOL specification
Ridge et al. [196]	HOL proof assistant	Proposed a rigorous approach for modeling and verifying TCP using the HOL proof assistant
Reachability Analysis		
Xie et al.[156]	Static Checking	Proposed a graph-theoretic algorithm (transitive closure) for static analysis of IP networks with support for ACL policies
Khakpour et al. [197]	Static Checking	Proposed a tool Quarnet comprising algorithms for quantifying reachability based on network configuration (incorporated ACL) and for querying network reachability
Al-Shaer et al. [50]	(Symbolic) Model Checking	Proposed a BDD/ CTL based symbolic model checking approach for performing 'network configuration in a box'
Lopes et al. [22]	SAT solvers	New SAT based solutions for the reachability set predicate
HSA [164]	SAT solvers; Static Checking	HSA provides a protocol-agnostic method for finding data plane bugs in networks by jointly studying the header space of packets and transformations applied to it by networking boxes
NetPlumber [79]	SAT solvers; Static Checking	HSA-based real-time policy checker for networks that works with incremental recomputation
VeriFlow [198]	Mininet, Depth-first search, Tries	Implemented as a layer between SDN controller and switches, VeriFlow verifies network-wide invariants in real-time dynamically as a forwarding rule is added
AP Verifier [93]	Atomic Predicates Verifier	AP verifier reduces the set of predicates representing packet filters to minimal atomic predicates, using which AP verifier dramatically improves computation of network reachability
Loop Detection		
HSA, NetPlumber, VeriFlow, AP Verifier	See above	These tools provide support for loop checking as well.
Isolation Verification		
AP Verifier [93]	Atomic Predicates Verifier	AP verifier, discussed above, can also be used to verify slice isolation [93]
"Splendid Isolation" [51]	Model Checking	Proposes a slices abstraction for SDNs with automatically verifiable formal isolation properties (expressed in CTL and checked through NuSMV tool)
Configuration Management		
rcc [152]	Static Analysis	Static analysis tool for detecting BGP configuration faults proactively before deployment
Qie et al. [158]	Static Analysis	Proposed using service grammar, incorporating a requirements language containing global high-level constraints, for detecting BGP configuration errors
Narain et al. [148]	(Lightweight) Model Checking; Scenario Finding	Proposed a method for managing (i.e., formalizing, and automating, reasoning about) network configuration with 'model finding' using the Alloy analyzer
ConfigChecker [50]	(Symbolic) Model Checking	Performs firewall verification with BDD-based model checking to perform symbolic reachability analysis
FlowChecker [80]	(Symbolic) Model Checking	The FlowChecker tool can be used to verify the correctness of OpenFlow federated infrastructures and debug reachability and security problems
Anteater [70]	SAT solvers; Static Checking	Anteater, builds upon Xie et al. [156], *implements* a tool for checking invariants in the data plane by transforming invariants into SAT instances to be checked against network state by a SAT solver
Paulson et al. [135]	Theorem Prover	BGP policy verification was performed using Isabelle/ HOL prover
Network Security		
FLOVER [69]	Model-Checking; SMT solvers	Verifies that the aggregate of flow policies instantiated within an OpenFlow network does not violate the networks security policy
MulVAL [199]	Logic-based Analysis	A logic-based network security analyzer
Zhang et al. [67]	SAT and QBF solvers	A SAT based technique for comparing the equivalence and inclusion relationship between two firewalls, and also propose Quantified Boolean Formula (QBF) based ACL optimization
Margrave [157]	SAT solvers & Scenario Finding	Firewall analysis tool that allows tracing behavior to specific rules and verification against security goals
Al-Shaer et al. [200]	Tree based model	Proposed a "Firewall Policy Advisor" for managing firewall filtering rules, and for detecting all anomalies in single or multiple firewall environments
Kothari et al. [201]	Symbolic Execution & Model Checking	Studies protocol manipulation attacks in which adversaries induce honest players into undesirable behaviors
Ritchey et al. [202]	Model Checking	Uses model checking to analyze network vulnerabilities
Gouda et al. [203]	Firewall Decision Diagrams	Presented a structured firewall design ensuring consistency, completeness and compactness. Also, proposed firewall decision diagram (FDD) for modeling firewall specification formally
Automatic Synthesis		
FVN project [204]	Logic-based framework	FVN presents a approach towards unifying the design, specification, implementation, and verification of networking protocols based on a logic language NDLog
Noyes et al. [205]	Model Checking	Proposed techniques for synthesis of network updates using NuSMV and OCaml tools
Wang et al. [206]	Reactive Synthesis & Model Checking	Proposed techniques for automated synthesis of reactive controllers for SDNs
Data Plane Verification		
FlowChecker [80], Anteater [70], HSA [164]	Static Checking	These tools perform static verification of the data plane of a network based on a snapshot of network state
NetPlumber & VeriFlow [198]	Dynamic Checking	These tools perform dynamic verification of the data plane using incremental recomputation techniques
ATPG [207]	Automatic Testing	ATPG is an automatic testing tool for generating test packets
NetSigtht, ndb [208]	Interactive Debuggers	Interactive debugging tools that operates passively
Control Plane Verification		
Scott et al. [209]	'Retrospective Causal Inference'	Proposed improving SDN troubleshooting by automatically identifying the minimum sequence of inputs responsible for causing a control software bug
Guha et al. [168]	Theorem Proving	Proposed a featherweight version of the OpenFlow protocol, and used the Coq tool for verifying the network controller [139]
Reitblatt et al.	Theorem Proving	Ensuring per-packet and per-flow consistency of network updates using the Coq prover
NICE [130]	Symbolic Execution & Model Checking	Performs symbolic execution of OpenFlow applications while applying model checking to explore the entire state space of the network
FlowLog [180]	Model Checking	Proposed a declarative finite-state language for programming SDN controllers that balances expressiveness and analysis and is amenable to model checking
Sethi et al. [210]	Model Checking	Proposed new abstractions for model checking SDN controllers

TABLE III. REPRESENTATIVE SUMMARY OF VARIOUS FORMAL VERIFICATION **TOOLS**

Technique	*Tool*	*Brief Summary*
Model Checkers		
	SPIN [46]	SPIN is a mostly automated tool for verifying distributed and concurrent systems
	NuSMV [49]	NuSMV, an extension of the original symbolic model checking tool SMV, is a model checking tool based on BDDs
	Alloy [52]	Alloy analyzer is a light-weight formal method that can analyze user specified properties of a (partial) model
	PRISM [122]	PRISM is a probabilistic model checker suitable for systems that exhibit probabilistic behavior
	UPPAAL [85]	UPPAAL is a model-checking based tool-box, based on timed-automata formalism, used for verification of real-time systems
Theorem Provers		
	Edinburgh LCF [188]	Interactive theorem prover proposed in 1972 which introduced ML language as a metalanguage for writing proving tactics
	HOL [136]	HOL represents a family of interactive theorem provers that are based on higher-order logics and strategies
	Isabelle [135]	A popular LCF-style theorem prover (written in Standard ML) that can work with various logics
	ACL2 [134]	ACL2 is a mechanical theorem prover with a Common Lisp-variant programming language, and an extensive first-order logic based theory
	PVS [137]	PVS is an automated theorem prover with an integrated specification language with multiple support tools
	Coq [138]	Coq is an interactive theorem prover that assists in finding proofs, and in extracting a certified program from the constructive proof
SAT and SMT solvers		
	Microsoft's Z3 [66]	Z3 is a state of the art SMT solver from Microsoft Research
	Kodkod [53]	Kodkod is a SAT-based constraint solver that can work with first-order logic with relations, transitive closure, etc.
	YICES [68]	Yices is an efficient SMT solver that can also act as a SAT and MaxSAT solver

in network troubleshooting and management for verifying the implementation of the intent of the network designer, and for troubleshooting reachability problems. Xie et al. reduce the reachability problem to a classical graph theoretic problem which can be solved in polynomial time by computing the transitive closure[12] to set union and intersection operations on the representation of reachability set. Recently, advances in SAT technology has led to its use for reachability analysis problems [71] [22].

Kazemian et al. have proposed a general protocol-agnostic static checking framework for networks based on header space analysis (HSA) [164]. Kazemian et al. have proposed a library of tools, called Hassel, which implements their proposed HSA based framework to identify important classes of failures which also includes forwarding loop detection, traffic isolation failure, beside reachability failure. The basic insight of HSA is to model a packet by its header by treating the entire header field as a concatenation of bits without any associated semantics—instead, the packet may be considered as a (geometric) point in the $0, 1^L$ geometric space where L is the maximum length of the packet header. The network is then modeled as being composed of network boxes (such as routers, switches, etc.) that transform packets from one point to another point, or possibly set of points (assuming multicast). This geometric approach taken by HSA (i.e., of representing the packet as a point in a subspace) allows the Hassel tools to work in a protocol-agnostic manner. Using HSA, we can easily *i)* find all packets that can reach from a point A to another point B, *ii)* find loops regardless of the protocol/ layer, *iii)* prove that two slides are isolated. Unlike model checking, HSA is not limited to providing a single counterexample in case of a failure detection, but can importantly provide information about the full set of failed packets.

Lopes et al. [22] have recently proposed extending the reachability predicate ("Can a packet from node A reach node B?") to a generalized abstraction of *reachability set* ("What are all the packets that can reach node B from node A?"). It is highlighted in [22] that reachability sets are useful for two reasons: *incremental computation* and *intelligibility*. In general, the tools for calculating reachability sets are less developed, although some languages like Datalog provide out of the box support for computation of reachability sets. The technique of incremental computation is useful for dynamic verification (i.e., when a new rule is being added) and has been recently proposed for real-time verification of SDN networks [198] [79]. The main insight underlying such an approach is the realization that a single rule change is unlikely to change the underlying network state machine drastically. Therefore, small modifications are necessary to the "reachability set" to incorporate the changes introduced by the addition of the new rule. Reachability set is also more intelligible as it produces a more general counter example—e.g., it can provide a set of packets being dropped.

In a promising recently proposed work [93], Yang and Lam present "Atomic Predicate (AP) Verifier", which reduces the set of predicates representing network packet filters to a set of atomic predicates that is provably both minimum and unique, which can be used to dramatically improve the computation of network reachability. The basic insight of this work is that atomic predicates have the following key property: Any given predicate is equal to the disjunction of the a subset of atomic predicates, and thus can be stored and represented as a set of integers identifying the atomic predicate. The conjunction (or the disjunction) of two predicates can be computed quickly as the intersection (or union) of two sets of integers. As an example, Yang and Lam show that while the Stanford network has 71 ACLs and 1584 rules, there were only 21 atomic predicates for these ACLs and rules (due to great redundancy in the forwarding and ACL rules). By encoding the rules in terms of atomic predicates (in the form of BDDs which can be manipulated through well-known graph BDD algorithms), this unnecessary redundancy is removed leading to much greater

[12]Transitive closure of G has an directed edge from x to y iff there is a directed path from x to y in G. Transitive closure is a standard graph-theoretic technique which, intuitive speaking, provides an efficient method for answering the reachability question 'where can we get from here?'

space and time efficiency. In their performance evaluation, Yang and Lam compare AP Verifier with Hassel in C and NetPlumber to demonstrate that AP verifier is significantly more time and space efficient [93].

Property verification also includes questions about *packet destination control:* Can a packet *i)* get out of the network, *ii)* get dropped, *iii)* go through certain switches, or *iv)* never pass through certain links. Model checking as well as ternary symbolic simulation techniques can be used for packet destination control [21].

2) Loop Detection: Header Space Analysis (HSA) [164] defines a "network algebra" which captures the manipulation of packet headers by network routers and switches. In the HSA framework, packet headers, represented as n-dimensional bit fields, are operated upon by the function defined by routers and switches which effectively transform the packet headers. HSA [164], and its enhancement NetPlumber [79], can verify a range of properties such as connectivity, reachability between ports, absence of any loops, and isolation between groups, etc. Various other approaches have been proposed in literature for loop detection including ConfigChecker [50], AP Verifier [93], etc.

3) Isolation Verification: For various reasons (such as security, confidentiality, etc.), it is sometimes desirable to ensure that certain kinds of traffic are isolated from each other. In current Internet, this is managed by various ad-hoc mechanisms often requiring manual intervention. For example, techniques used for ensuring isolation include: *i)* low level mechanisms such as VLANs or ACLs requiring configuration, *ii)* special purpose devices such as firewalls, *iii)* or complex hypervisors such as the FlowVisor system [216] for OpenFlow networks. It is desirable to have more fundamental abstractions that can be exploited to provide verifiable isolation between traffic as desired. An initial work in this regard has been presented for SDNs in the "splendid isolation" project [51] proposed as part of the Frenetic project [193]. In this work, a slice abstraction is presented and algorithms for compiling slices is presented along with a tool for automatic verification of formal isolation properties. In other works, AP Verifier can also verify slice isolation as reported in [93].

C. Network Configuration Management

Configuration errors can create numerous connectivity, security, performance, and reliability problems. It has been pointed out in literature that the bulk of network downtime is in fact due to manual errors [217] and misconfiguration of devices [21]. The problem is especially acute since it is not far fetched for a misconfiguration of a single device to cripple an entire network. Various problems can arise from bugs due to misconfiguration including access control failures, isolation guarantee failures, routing loops, reachability failures, blackholes, etc. The presence of such problems can have debilitating effect on network performance and efficiency, thus motivating a more rigorous and formal management of network configuration. In configuration management, we would like to have multiple abstractions, incorporating correctness checks, between the high-level global end-to-end requirements and low-level distributed configuration at individual devices.

Static analysis has been used extensively for detecting configuration faults. Feamster et al. proposed a static analysis tool *rcc* for detecting BGP configuration faults [152]. The *rcc* tool allowed proactive analysis of network configurations before deployment in an operational network by checking that BGP configuration satisfies a set of constraints, based on the correctness specification. The *rcc* tool, like most practical static analysis tools, is neither complete nor sound—i.e., it can miss problematic configurations, and may complain about harmless deviations from the best practices. Nevertheless, *rcc* was able to find many important classes of errors to make it useful in practice. Qie et al. [158] proposed an approach based on "service grammar" for BGP which incorporated a requirements language using which the network operator can specify high-level requirements against which the system may be checked. Unfortunately, the proposed grammar was rather low-level thus having possibilities of erroneous specification. In another work, Narain et al. proposed managing network configuration through model finding [148] while using the Alloy analyzer [52]. In yet other work, ConfigChecker [50] performs firewall verification with BDD-based model checking to perform symbolic reachability analysis. Configuration management has been a fertile area for application for formal verification methods with various proposals in literature [218] [148] [80] [70].

D. Network Security

There are various important subproblems of firewall verification and synthesis [67]. Firstly, the *firewall equivalence checking* problem focuses on determining if two firewalls have identical behavior—i.e., they drop and permit the same set of packets. Secondly, the *firewall inclusion checking* problem compares two firewall policies and can verify that one policy is inclusive, i.e., more strict, than the other policy. Thirdly, the *firewall rule redundancy checking* problem focuses on determining redundant rules—i.e., rules that can be deleted without affecting the behavior of the firewall. Lastly, the *firewall synthesis* problem focuses on synthesizing a firewall with minimum number of rules install that matches exactly the behavior of another given firewall.

There has been a lot of work in firewall verification and synthesis (e.g., [67] [157] [200] [201] [203]) and vulnerability analysis (e.g., [201] [202] [219]) and a variety of techniques have been utilized including static analysis [220], model based analysis [219] [221], logic-based analysis [199], SAT solvers, model checking [69] [202], new abstractions (e.g., firewall decision diagrams or FDD [203], atomic predicates (AP) verifier [93], etc.). The interested reader is referred to a detailed description of related work in [157] and [203].

E. Network Verification

Traditionally, the focus of formal verification community has been on hardware systems or software systems, and relatively less on network verification. Networked systems comprise a

software component (implementing the node OS, protocols, applications, etc.) and a hardware component (featuring the range of hardware configuration such as microprocessors, general purpose processors, DSPs, ASICs, etc.). Networked system are in fact distributed systems composed of end hosts that use the network as well as networking nodes (such as routers, switches, and various middleboxes such as firewall, load balancers etc.) that implement the network. In previous work, network verification is considered as essentially a state machine verification problem [22]—i.e., a communication network can be visualized as a finite network of FSMs. Although, this problem is quite complex theoretically—PSPACE-complete for the general problem of verification of network of FSMs—structural properties of networks fortunately enable techniques like Anteater [70] and HSA [164] to work satisfactorily in practice.

1) Declarative Network Verification: As pointed out earlier, there is an increasing trend in using declarative programming techniques, and techniques that have been successful in deductive databases community, in networking. The use of such techniques also enables importantly the ability to perform network verification. There has been some work in this regard [181] in which the task of formal specification is performed through declarative networking code, using Network Datalog (NDLog), a distributed variant of Datalog, while verification is done through a general-purpose theorem prover.

2) Hardware Verification: Formal verification methods have been used for hardware verification of networking devices. A general survey of formal verification in hardware design can be seen at [6]. Some sample works in hardware verification in networking include verification of: *i* the lookup machine of a hardware router [222], *ii* the Fairisle ATM swithing element [223], *iii)* network-on-chip [224] [225].

3) Formal Specification and Synthesis: There are many benefits in formally specification including the clarity accompanying rigorous specification of high-level specification of the target networking problem along with the ability to employ mechanized correctness checking to weed out trivial mistakes through techniques such type-checking. It has been shown in research that informal specification of protocols can lead to incorrect reasoning and implementation [226] and ambiguity [129].

In order to create a correctly performing implementation, it is worthwhile to invest time and effort in *design verification*. Various approaches can be explored including specialized meta-theories specific to routing and forwarding [76], axiomatic logic-based formalisms [227], or declarative programming frameworks [173] [191], to specify the design. These formalisms can then be analyzed used methods like theorem provers, model checking, SAT/ SMT solvers, lightweight formal methods etc. to verify the correctness of the design and thereby guide the implementation.

There also has been work in synthesizing protocol implementations from formal specifications. An example work in this regard is the "formally verifiable networking" (FVN) project [204]. In another work, the synthesis of network updates have been proposed [205]. Recently Lopes et al. [22] have indicated building a synthesis tool for Microsoft Azure firewalls as their future work—such a tool can enable synthesis of low-level rules from a high-level specification and thus network operators can forego the error-prone access control list (ACL) configuration CLI.

4) Implementation Verification: Having studied techniques that can be used to verify design in previous subsections, we will now see that a variety of techniques, described earlier in section V, can be used to verify *implementations*. In particular, we can make use of static checking as well as dynamic checking. In static validation, correctness properties are defined as invariants or constraints which are then checked to find out any system faults. In certain cases, a pre-processing stage may be necessary to transform the real system into an intermediate more checkable form. Static analysis and model checking are static validation tools. While most model checking tools work with specification models, some model checking tools (such as MaceMC [131], VeriSoft [12] and CMC [97]) can work directly with implementation code making them very valuable for verifying implementations. In dynamic validation, on the other hand, we rely on runtime verification and testing—which per se are not really formal verification tools but nonetheless perform a complementary role.

F. Applications in SDN

In this section, we will discuss new opportunities offered for incorporating programming and verification advances into the networking context by the SDN architecture. We will initially discuss the new degrees of freedom offered by SDN in section VII-F1. We will discuss SDN programming languages in section VII-F2 and will thereafter talk about data plane and control plane verification in sections VII-F3 and VII-F3, respectively. Finally, we will discuss SDN debugging tools in section VII-F5.

1) What is new about SDN?: In traditional networking, the complex intricacies of a vertically integrated network architecture largely ruled out applications of formal methods to the domain of networking. This resulted in ad-hoc management of networks by "masters of complexity" [23]—network administrators who kept networks running mainly through intuition and judgment honed through experience with a very limited tool-set. Fortunately, the recent SDN architecture is much cleaner and offers an opportunity at rethinking networking management and troubleshooting [228]. There are three reasons for the optimistic evaluation of verification prospects of SDNs: firstly, the control plane that previously ran as distributed algorithms across individual devices has now been refactored into a single program that runs on the controller; secondly, the heterogeneity in traditional networking—in devices, configuration interfaces, vendors, and softwares—has given way to stock programmable switches supporting standard interfaces with precise semantics [168]; lastly, it is envisioned that the core network, or the *fabric*, in the new SDN architecture will be purely hardware (finite state) and is thus amenable to efficient application of verification techniques

[21]. These new degrees of freedom enabled by SDN have ignited a renewed resolve in the networking community of applying formal methods to networking and to put networking on a solid theoretical foundation [139] [229] [230].

2) SDN Programming Languages: Various SDN specific programming languages have been proposed recently (e.g., Frenetic [193], NetCore [231] [168], Pyretic [190], and NetKat [73], etc.). These network programming languages enable programmers, in line with the vision of software defined networks, to define the desired network behavior at a high-level and the compiler then translates the high level abstract description to rules that are installed on the underlying hardware devices. The NetCore language [231] was initially designed to provide support for parallel composition and was later extended by Pyretic [190] for sequential composition. NetCore provides a rich set of programming primitives including predicates for filtering packets, actions for modifying and forwarding packets, and (parallel and sequential) composition operators for building elaborate policies from simpler ones. NetCore has even been formalized in Coq. NetKat is similar to NetCore and Pyretic, but additionally provides formal axiomatic semantics and a compiler based on an equational theory for reasoning about programs. NetKat is based on *Kleene algebra with tests* which is a mature framework that combines Kleene algebra—useful for reasoning about network structure—and Boolean algebra which is useful for reasoning about the predicates that define switch behavior. NetKat provides consistent reasoning principles that other network programming languages lack. In contrast to fore-mentioned languages, which have a functional bent and are suited for programming of centralized controllers, the DataLog[13] based declarative network programming language *NDLog* [178] [173] is a logic programming language suited to distributed programming.

3) Data Plane Verification: Various approaches have been proposed for data plane verification including *i)* static checking–in which the correctness is verified independently, *ii)* dynamic checking—in which new forwarding state is checked before being added, *iii)* automatic testing—where the correct behavior of the dataplane is checked automatically, and *iv)* interactive debugging—which aims at finding bugs in operational networks. The Anteater [70], FlowChecker [80], and Hassell [164] tools are example *static checking tools*. Various real-time *dynamic checking tools* have been proposed in literature including NetPlumber [79] and VeriFlow [198]. The NetPlumber tool uses a novel header space analysis for performing a real time network policy check, while the VeriFlow tool verifies network invariants—e.g., lack of access control violations, absence of routing loops, blackholes, etc.—in real time and presents a diagnostic report in case of a violation. The Automatic Test Packet Generation (ATPG) tool is an automatic testing tool that automatically generates test packets [207]. The ATPG verifies full reachability in a network, using minimal network of test packets by using a heuristic solver for the min-set-cover problem, and detects anomalies by looking for persistent packet drops that are indicative of some software or hardware errors. Finally, the NetSigtht and the Network Debugger (ndb) tools are *interactive debugging tools* that operate passively without generating any new packets unlike the ATPG tool. The ndb tool [208]—the analogue of gdb debugger for programming—is like a network-wide path-aware tcpdump that builds packet histories which can be exploited by network analysis applications to verify the policy compliance of network data plane behavior.

4) Control Plane Verification: Various projects have aimed at verification of the control plane functionality of SDNs. In the SDN architecture, it is envisioned that network programs will run as SDN applications on top of a northbound API exposed by SDN controller. This will allow SDN applications to leverage the services of the SDN controller, which will be responsible for managing the distributed state through a southbound API like OpenFlow, while the SDN application can focus on using the state for the task it wishes to perform. It is anticipated that this architecture will allow innovation to flourish and the development of numerous network based applications. In such an environment, it is necessary to ensure that we have tools available for testing and verifying such SDN applications. Canini et al. present their NICE framework for testing OpenFlow applications [130]. Kuzniar et al. have proposed another framework, named SOFT, for verifying OpenFLow switch interoperability. There also has been work on computationally verifying network programs in the Coq mechanical proof tool [232].

There also has been work on isolating fault inducing inputs to SDN control software [209], controller verification [168], and ensuring per-packet and per-flow consistency of network updates [48]. The problem of verifying a generic SDN controller—which in its general setting is Turing complete (e.g., NOX, Floodlight, etc.)—is undecidable. Guha et al. have proposed a method of using for machine verification of network controllers [168]. FlowLog [180] is a declarative, finite-state, language for programming SDN controllers that balances expressiveness and analysis and is amenable to model checking. In another model checking based work, Sethi et al. [210] have proposed new data state and network state abstractions that can be used for model checking SDN controllers more efficiently. The Frenetic framework [193] incorporates features to help achieve per-packet and per-flow consistency during network updates [48]. The safe update protocol proposed in [48] builds upon approaches that use incremental recomputation (e.g., Anteater [70], VeriFlow [198], etc.), which may have a transient stage in which the property to be verified may be violated, by ensuring that the property under check also holds during the transient stage.

5) Network Debugging: As mentioned before, networks are composed of both hardware and software components and are managed in many cases manually. Due to this reason, networks can fail in a variety of ways making the job of debugging and troubleshooting a network very complex. Traditionally, networking has a very primitive toolset for troubleshooting comprising few ad-hoc tools such as ping, traceroute, etc.

[13]Datalog is a declarative logic programming language used as a query language for deductive databases. It is a simplified form of Prolog, and can be envisioned as a subset of Prolog sans the complex terms allowed by Prolog.

usually complemented by the painstaking manual process of inspecting log files. Broadly speaking, debugging can take place either statically or dynamically. Static debugging—akin to compile-time checking—works by inspecting network configuration and settings through static analysis tools, model checking, SAT solvers, etc. Dynamic checking—similar to run-time checking—works by checking if the data plane is behaving as it should (techniques for data plane verification have earlier been discussed in section VII-F3). Dynamic checking can catch errors that arise from reasons other that erroneous configurations, e.g., it can help in the case of *i)* hardware errors, *ii)* link failure, *iii)* congestion, *iv)* intermittent problems, etc. Heller et al. have proposed systematic troubleshooting of SDNs by establishing equivalence of network views at different layers [228]. In particular, Heller et al. proposed comparing *i)* actual network behavior vs. policy, *ii)* the policy vs. device state, *iii)* the device state vs. the hardware state, etc. By comparing these diverse network views systematically, more efficient troubleshooting can be performed which will allow identification of faults and systematic tracking down root causes.

Handigol et al. [208] have proposed the ndb (network debugger) tool, analogous to the software debugging gdb tool, that aims to capture and reconstruct the sequence of events that leads to buggy behavior. In particular, it allows users to define a 'network breakpoint' in the form of (header, switch) filter to identify the errant behavior, and then produces a packet backtrace, which includes historical information about the path taken by the packet as well as the state of the flow tables at each switch, to aid in troubleshooting of networks [233]. In a similar vein, Wundsam et al. [234] have proposed the OFRewind framework which is useful for capturing and reproducing the sequence of problematic OpenFlow command sequence. In another work, Scott et al. have proposed using correspondence checking and simulation based causal inferencing to isolate and localize software faults in SDN [235]. In networked systems, erroneous behavior can manifest itself due to the various issues related to distributed computing such as asynchrony, concurrency, and partial failures leading to time-consuming troubleshooting and considerable angst [209]. Various debugging tools have been proposed for debugging general distributed systems: e.g., Pip [236], etc., and automatic debugging techniques specific to SDN have been proposed in [237].

VIII. Open Issues and Future Work

The area of formal methods and verification is vast with various mature tools and techniques available. With networking being fundamentally important to all aspects of life including government, defence, industry, finance, etc., networks are in dire need of provably correct mechanisms. Notwithstanding the lack of any major breakthroughs made by formal methods in traditional networking, architectural support from SDN along with its clean abstractions provide a source of optimism for the future of formal methods in networking. The nascent field of network verification is wide open and is ripe for further exploration. In this section, we will point a few important open issues and highlight possible future work.

A. Scalable Formal Verification For Large Networks

Advanced in technologies such as BDDs and SAT solvers have extended the state of the art considerably in recent years. However, more work needs to be done for current formal verification techniques to scale to large networks and to verify large software systems (such as network applications and protocol implementations). An approach that has been proposed in literature for scaling to large networks is to utilize incremental recomputation thereby avoiding the overhead of redoing expensive static calculations. For example, NetPlumber [164] improves HSA [164], and Veriflow [198] improves Anteater [70], by supporting incremental computation. The incorporation of incremental recomputations techniques have allowed these tools to scale to reasonably large networks. In recent work, Yang and Lam have proposed an efficient real-time verifier of network properties using atomic predicates [93]. More work is needed in this area to exploit these recent works so that network verification for large networks can become both practical and efficient.

B. Automated Synthesis

Synthesis which promises to automatically derive implementations from specifications is an extremely important future goal that can improve programmer productivity. The problem of automated synthesis is at the frontier of verification research today [22]. Some important works in this regard include synthesis of network updates [205], synthesis of network controllers [206], synthesis of finite state controllers from temporal logic specifications, and synthesis of programs from examples by exploiting domain specific knowledge, etc. [124]. In the context of networking, more work needs to be done so that subsystems such as protocols, configurations, hardware may be synthesized through a high-level formal specification only in a user-friendly manner.

C. Selection of the Right Formal Method for the Task

As highlighted in this work, there is a vast amount of work that has been done in the field of formal methods. There are various logics, notations, technologies and tools available, each making its own claim of superiority, that may be utilized. Many of the claims are valid in that certain tools do certain excel in niche areas; however, each tool has its disadvantages as well. As Keshav pointed out in [238], the choice of the most appropriate tool is certainly not trivial even for an established researcher, let alone for a graduate student. It is important to use the most appropriate specification language for the task, as noted in the 10 commandments stated in [239]. With research in network verification recently starting to flourish, it is important to determine the right tools for various verification tasks in network verification. Two tools that are immediately useful for a networking researcher are Alloy and SPIN: a practical comparison of these two tools is presented in [240].

D. Specialized Network Verification Tools

In contrast to sophisticated well-honed design automation tools that are available for general hardware[14] and software industries, networking industry has almost no rigorous tools for verification. The vision of building a network CAD was articulated by McKeown. Encouragingly, as the SDN architecture is becoming mainstream, there is renewed interest in building specialized tools that will allow automated debugging, verification, and analysis. Some important issues that need to be addressed before such a vision can be realized are [228]: *i)* incorporating program semantics into network troubleshooting tools; *ii)* improved techniques for checking invariants; *iii)* development of new abstractions, especially in the SDN context, to facilitate troubleshooting.

E. Verification for Concurrent and Parallel Programming

With the emergence of data centers and cloud computing, the programming world is undergoing a silent revolution with a growing trend towards parallel programming. Although, there are various approaches that have been proposed to support verification of concurrent programs, more research needs to be done to propose new clean simplified abstractions for building verified concurrently executing programs that can exploit modern multi-core and multi-processor architectures, and parallel programming style suited to data centers and cloud computing.

IX. Conclusions

We are in an exciting time in the networking world with recent innovations such as software defined networking and cloud computing fundamentally altering the landscape of the networking world. Keeping in mind the criticality of the Internet infrastructure, assuring the correct behavior of various subsystems of the Internet has become essential. There is great interest in applying the vast amount of work that has been done in the community of formal methods and verification to networks. The work in formal methods draws upon many diverse fields such as logic, theoretical computer science, programming languages, mathematics, etc., and hence appears daunting to a non-specialist. In this work, we present a detailed tutorial on the various methods and techniques used in formal methods and verification while providing necessary background and references to important works. We also present a detailed survey of the application of formal methods in the networking context. We have also identified some important research directions that can be pursued in future work.

References

[1] J. Day, *Patterns in network architecture: a return to fundamentals*. Pearson Education, 2007.

[2] S. Garfinkel, "History's worst software bugs," *Wired News, Nov*, 2005.

[3] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, 2002.

[4] J. Rexford and P. Zave, "Report of the DIMACS working group on abstractions for network services, architecture, and implementation," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, pp. 56–59, 2012.

[5] P. Zave, "Experiences with protocol description," in *Workshop on Rigorous Protocol Engineering (WRIPE'11)*, 2011.

[6] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, 1999.

[7] G. Bochmann and C. Sunshine, "Formal methods in communication protocol design," *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 624–631, 1980.

[8] G. J. Holzmann, "Design and validation of protocols: a tutorial," *Computer Networks and ISDN Systems*, vol. 25, no. 9, pp. 981–1017, 1993.

[9] K. Bhargavan, D. Obradovic, and C. A. Gunter, "Formal verification of standards for distance vector routing protocols," *Journal of the ACM (JACM)*, vol. 49, no. 4, pp. 538–576, 2002.

[10] J. Jürjens, *Secure systems development with UML*. Springer, 2005.

[11] D. M. Buede, *The engineering design of systems: models and methods*, vol. 55. John Wiley & Sons, 2011.

[12] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 174–186, ACM, 1997.

[13] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan, "Verisim: Formal analysis of network simulations," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 129–145, 2002.

[14] H. Chen, D. Dean, and D. Wagner, "Model Checking One Million Lines of C Code," in *NDSS*, vol. 4, pp. 171–185, 2004.

[15] "Running code vs. formal testing methods [Online]." http://www.ietf.org/mail-archive/web/ietf/current/msg10577.html. Accessed: 2013-09-30.

[16] G. Holzmann, "OOPSLA keynote: Scrub and Spin: Stealth Use of Formal Methods in Software Development," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, p. 4, ACM, 2009.

[17] "International Conference on Formal Techniques for Networked and Distributed Systems [Online]." http://www.informatik.uni-trier.de/~ley/db/conf/forte/. Accessed: 2013-09-12.

[18] http://www.cs.cornell.edu/conferences/formalnetworks/. Accessed: 2013-10-1.

[19] F. Babich and L. Deotto, "Formal methods for specification and analysis of communication protocols," *Communications Surveys & Tutorials, IEEE*, vol. 4, no. 1, pp. 2–20, 2002.

[20] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 19, 2009.

[21] S. Zhang, S. Malik, and R. McGeer, "Verification of computer switching networks: an overview," in *Automated Technology for Verification and Analysis*, pp. 1–16, Springer, 2012.

[22] N. Lopes, N. Bjørner, P. Godefroid, and G. Varghese, "Network verification in the light of program verification,"

[23] "Scott Shenker's talk at Stanford (2013) [Online]." http://www.youtube.com/watch?v=WabdXYzCAOU. Accessed: 2013-09-12.

[24] H. Scholz, *Concise history of logic*, vol. 94. Philosophical Library, 1961.

[25] M. Ben-Ari, *Mathematical logic for computer science*. Springer, 2012.

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

[14] The electronic design automation (EDA) industry in hardware design is a big market catering to a multi-billion dollar industry.

[27] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu, "On the unusual effectiveness of logic in computer science," *The Bulletin of Symbolic Logic*, vol. 7, no. 2, pp. 213–236, 2001.

[28] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[29] S. Russell, *Artificial intelligence: A modern approach, 2/E*. Pearson Education India, 2003.

[30] K. J. Devlin, *Logic and information*. Cambridge University Press, 1995.

[31] J. Franco and J. Martin, "A history of satisfiability.," *Handbook of Satisfiability*, vol. 185, pp. 3–74, 2009.

[32] D. Makinson, *Sets, logic and maths for computing*. Springer, 2012.

[33] K. Devlin, *Sets, functions, and logic: an introduction to abstract mathematics*. CRC Press, 2003.

[34] O. Strichman, *Decision procedures: an algorithmic point of view*. Springer, 2010.

[35] J. Harrison, "The HOL Light Theory of Euclidean Space," *Automated Reasoning*, vol. 50, no. 2, pp. 173–190, 2013.

[36] T. Mhamdi, O. Hasan, and S. Tahar, "Formalization of measure theory and lebesgue integration for probabilistic analysis in hol," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1, pp. 13:1–13:23, 2013.

[37] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[38] K. R. Apt, "Ten years of Hoare's logic: A surveypart I," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 3, no. 4, pp. 431–483, 1981.

[39] E. A. Emerson, "Temporal and modal logic.," *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, vol. 995, p. 1072, 1990.

[40] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pp. 46–57, IEEE, 1977.

[41] 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 (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[42] L. Lamport, "What good is temporal logic?," in *IFIP congress*, vol. 83, pp. 657–668, 1983.

[43] D. M. Gabbay, I. Hodkinson, M. Reynolds, and M. Finger, *Temporal logic: mathematical foundations and computational aspects*. Clarendon Press, 2000.

[44] Z. Manna and A. Pnueli, *Temporal Logic*. Springer, 1992.

[45] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.

[46] G. J. Holzmann, "The model checker SPIN," *Software Engineering, IEEE Transactions on*, vol. 23, no. 5, pp. 279–295, 1997.

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

[48] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 323–334, ACM, 2012.

[49] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*, pp. 359–364, Springer, 2002.

[50] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*, pp. 123–132, IEEE, 2009.

[51] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 79–84, ACM, 2012.

[52] D. Jackson, "Software abstractions: Logic," *Language, and Analysis, The MIT Press*, 2006.

[53] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 632–647, Springer, 2007.

[54] N. Feamster and H. Balakrishnan, "Towards a logic for wide-area internet routing," in *ACM SIGCOMM Computer Communication Review*, vol. 33, pp. 289–300, ACM, 2003.

[55] A. Biere, *Handbook of satisfiability*, vol. 185. IOS Press, 2009.

[56] S. Malik and L. Zhang, "Boolean satisfiability from theoretical hardness to practical success," *Communications of the ACM*, vol. 52, no. 8, pp. 76–82, 2009.

[57] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[58] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories.," *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[59] L. Bordeaux, Y. Hamadi, and L. Zhang, "Propositional satisfiability and constraint programming: A comparative survey," *ACM Computing Surveys (CSUR)*, vol. 38, no. 4, p. 12, 2006.

[60] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, *Algorithms for the satisfiability (SAT) problem*. Springer, 1999.

[61] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[62] M. K. Ganai and A. Gupta, "SAT-Based Verification Framework," *SAT-Based Scalable Formal Verification Solutions*, pp. 247–261, 2007.

[63] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Foundations of Artificial Intelligence*, vol. 3, pp. 89–134, 2008.

[64] N. Een and N. Sörensson, "Minisat: A sat solver with conflict-clause minimization," *Sat*, vol. 5, 2005.

[65] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, ACM, 2001.

[66] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[67] S. Zhang, A. Mahmoud, S. Malik, and S. Narain, "Verification and synthesis of firewalls using SAT and QBF," in *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pp. 1–6, IEEE, 2012.

[68] B. Dutertre and L. De Moura, "The YICES SMT solver," *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, vol. 2, p. 2, 2006.

[69] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in openflow,"

[70] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 290–301, 2011.

[71] S. Zhang and S. Malik, "Sat based verification of network data planes," in *Automated Technology for Verification and Analysis*, pp. 496–505, Springer, 2013.

[72] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 3, pp. 427–443, 1997.

[73] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic Foundations for Networks," 2013.

[74] J. S. Baras and G. Theodorakopoulos, "Path problems in networks," *Synthesis Lectures on Communication Networks*, vol. 3, no. 1, pp. 1–77, 2010.

[75] J. L. Sobrinho, "An algebraic theory of dynamic network routing," *IEEE/ACM Transactions on Networking (TON)*, vol. 13, no. 5, pp. 1160–1173, 2005.

[76] T. G. Griffin and J. L. Sobrinho, "Metarouting," in *ACM SIGCOMM Computer Communication Review*, vol. 35, pp. 1–12, ACM, 2005.

[77] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[78] G. J. Holzmann, "An improved protocol reachability analysis technique," *Software: Practice and Experience*, vol. 18, no. 2, pp. 137–161, 1988.

[79] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[80] E. Al-Shaer and S. Al-Haj, "Flowchecker: Configuration analysis and verification of federated openflow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pp. 37–44, ACM, 2010.

[81] J. E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation, 3/E*. Pearson Education India, 2008.

[82] R. Alur, "Timed automata," in *Computer Aided Verification*, pp. 8–22, Springer, 1999.

[83] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[84] J. Billington, M. Diaz, and G. Rozenberg, *Application of Petri nets to communication networks: advances in Petri nets*. No. 1605, Springer, 1999.

[85] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[86] C. Wang, G. D. Hachtel, and F. Somenzi, *Abstraction refinement for large scale model checking*. Springer, 2006.

[87] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT press, 1999.

[88] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[89] H. R. Andersen, "An introduction to binary decision diagrams," *Lecture notes, available online, IT University of Copenhagen*, 1997.

[90] D. Knuth, "The art of computer programming: Bitwise tricks & techniques; binary decision diagrams, volume 4, fascicle 1," 2009.

[91] R. E. Bryant, "Binary decision diagrams and beyond: Enabling technologies for formal verification," in *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, pp. 236–243, IEEE, 1995.

[92] R. McGeer, "New results on BDD sizes and implications for verification," in *Proceedings of the International Workshop on Logic Synthesis (June 2012)*, 2012.

[93] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *ICNP, the IEEE International Conference on Network Protocols*, 2013.

[94] G. J. Holzmann and M. H. Smith, "Automating software feature verification," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 72–87, 2000.

[95] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Computer aided verification*, pp. 260–264, Springer, 2001.

[96] Ball, Thomas and Majumdar, Rupak and Millstein, Todd and Rajamani, Sriram K, "Automatic predicate abstraction of c programs," in *ACM SIGPLAN Notices*, vol. 36, pp. 203–213, ACM, 2001.

[97] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 75–88, 2002.

[98] S. Budkowski and P. Dembinski, "An introduction to estelle: a specification language for distributed systems," *Computer Networks and ISDN systems*, vol. 14, no. 1, pp. 3–23, 1987.

[99] J. Jacky, *The way of Z: practical programming with formal methods*. Cambridge University Press, 1996.

[100] P. Camurati and P. Prinetto, "Formal verification of hardware correctness: Introduction and survey of current research," *Computer*, vol. 21, no. 7, pp. 8–19, 1988.

[101] F. Belina and D. Hogrefe, "The CCITT-specification and description language SDL," *Computer Networks and ISDN Systems*, vol. 16, no. 4, pp. 311–341, 1989.

[102] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networks and ISDN systems*, vol. 14, no. 1, pp. 25–59, 1987.

[103] O. Grumberg and H. Veith, *25 years of model checking: history, achievements, perspectives*, vol. 5000. Springer, 2008.

[104] E. A. Emerson, "The beginning of model checking: A personal perspective," in *25 Years of Model Checking*, pp. 27–45, Springer, 2008.

[105] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," in *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*, pp. 394–406, IEEE, 1992.

[106] C. Baier, J.-P. Katoen, *et al.*, *Principles of model checking*, vol. 26202649. MIT press Cambridge, 2008.

[107] E. M. Clarke, "The birth of model checking," in *25 Years of Model Checking*, pp. 1–26, Springer, 2008.

[108] "Webpage on "Model Theory" in Stanford's Encyclopedia of Philosophy [Online]." http://plato.stanford.edu/entries/model-theory/. Accessed: 2013-10-3.

[109] D. Marker, *Model theory: an introduction*. Springer, 2002.

[110] K. L. McMillan, *Symbolic model checking*. Springer, 1993.

[111] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *Computer Aided Verification*, pp. 419–422, Springer, 1996.

[112] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10¡ sup¿ 20¡/sup¿ states and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[113] K. L. McMillan, "The SMV system, symbolic model checking-an approach," tech. rep., Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[114] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *Symbolic model checking without BDDs*. Springer, 1999.

[115] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.

[116] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 1, pp. 69–83, 2009.

[117] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, Springer, 2004.

[118] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Runtime Verification*, pp. 122–135, Springer, 2010.

[119] R. Fagin and J. Y. Halpern, "Reasoning about knowledge and probability," *Journal of the ACM (JACM)*, vol. 41, no. 2, pp. 340–367, 1994.

[120] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, "Model-checking algorithms for continuous-time markov chains," *Software Engineering, IEEE Transactions on*, vol. 29, no. 6, pp. 524–541, 2003.

[121] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model

checking," in *Formal methods for performance evaluation*, pp. 220–270, Springer, 2007.

[122] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: probabilistic model checking for performance and reliability analysis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 40–45, 2009.

[123] M. Kwiatkowska, G. Norman, and J. Sproston, *Probabilistic model checking of the IEEE 802.11 wireless local area network protocol*. Springer, 2002.

[124] R. Alur, T. A. Henzinger, I. Austria, and M. Y. Vardi, "Theory in practice for system design and verification," 2013.

[125] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[126] K. Havelund and T. Pressburger, "Model checking Java programs using Java Pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[127] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *Model Checking Software*, pp. 235–239, Springer, 2003.

[128] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar, "Comparison of model checking tools for information systems," in *Formal Methods and Software Engineering*, pp. 581–596, Springer, 2010.

[129] P. Zave, "Understanding SIP through model-checking," in *Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, pp. 256–279, Springer, 2008.

[130] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A nice way to test openflow applications," *NSDI, Apr*, 2012.

[131] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," *NSDI 07: Networked Systems Design and Implementation*, pp. 243–256, 2007.

[132] C.-L. Chang, R. C.-T. Lee, and R. C.-T. Lee, *Symbolic logic and mechanical theorem proving*, vol. 67. Academic press New York, 1973.

[133] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[134] B. Brock, M. Kaufmann, and J. S. Moore, "Acl2 theorems about commercial microprocessors," in *Formal Methods in Computer-Aided Design*, pp. 275–293, Springer, 1996.

[135] L. C. Paulson and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, vol. 2283. Springer, 2002.

[136] M. J. Gordon, *HOL: A proof generating system for higher-order logic*. Springer, 1987.

[137] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Automated DeductionCADE-11*, pp. 748–752, Springer, 1992.

[138] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, 2004.

[139] A. Guha, M. Reitblatt, and N. Foster, "Formal foundations for software defined networks," in *Open Net Summit*, 2013.

[140] L. Liu, O. Hasan, and S. Tahar, "Formal reasoning about finite-state discrete-time markov chains in hol," *Journal of Computer Science and Technology*, vol. 28, no. 2, pp. 217–231, 2013.

[141] O. Hasan and S. Tahar, "Performance analysis and functional verification of the stop-and-wait protocol in hol," *Journal of Automated Reasoning*, vol. 42, no. 1, pp. 1–33, 2009.

[142] M. Elleuch, O. Hasan, S. Tahar, and M. Abid, "Formal analysis of a scheduling algorithm for wireless sensor networks," in *Formal Engineering Methods*, vol. 6991 of *LNCS*, pp. 388–403, Springer, 2011.

[143] L. Liu, O. Hasan, and S. Tahar, "Formal analysis of memory contention in a multiprocessor system," in *Formal Methods: Foundations and Applications*, vol. 8195 of *Lecture Notes in Computer Science*, pp. 195–210, Springer, 2013.

[144] T. Mhamdi, O. Hasan, and S. Tahar, "Quantitative analysis of information flow using theorem proving," in *Formal Methods and Software Engineering*, vol. 7635 of *Lecture Notes in Computer Science*, pp. 119–134, Springer, 2012.

[145] S. Agerholm and P. G. Larsen, "A lightweight approach to formal methods," in *Applied Formal MethodsFM-Trends 98*, pp. 168–183, Springer, 1999.

[146] D. Jackson, "Lightweight formal methods," in *FME 2001: Formal Methods for Increasing Software Productivity*, pp. 1–1, Springer, 2001.

[147] D. Jackson, "Dependable software by design," *Scientific American*, vol. 294, no. 6, pp. 68–75, 2006.

[148] S. Narain *et al.*, "Network configuration management via model finding.," in *LISA*, vol. 5, pp. 15–15, 2005.

[149] P. Zave, "Formal methods and networking: Former success, current failure," 2012.

[150] Y. Xie, M. Naik, B. Hackett, and A. Aiken, "Soundness and its role in bug detection systems," in *Proc. of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[151] D. Engler and M. Musuvathi, "Static analysis versus software model checking for bug finding," in *Verification, Model Checking, and Abstract Interpretation*, pp. 191–210, Springer, 2004.

[152] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 43–56, USENIX Association, 2005.

[153] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[154] K. R. M. Leino, "Extended static checking: A ten-year perspective," in *Informatics*, pp. 157–175, Springer, 2001.

[155] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *ACM Sigplan Notices*, vol. 37, pp. 234–245, ACM, 2002.

[156] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of ip networks," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 3, pp. 2170–2183, IEEE, 2005.

[157] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis," in *USENIX Large Installation System Administration Conference*, 2010.

[158] X. Qie and S. Narain, "Using service grammar to diagnose BGP configuration errors," *Science of Computer Programming*, vol. 53, no. 2, pp. 125–141, 2004.

[159] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[160] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[161] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[162] R. E. Bryant, "Symbolic simulationtechniques and applications," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 517–521, ACM, 1991.

[163] R. E. Bryant and C.-J. H. Seger, "Formal verification of digital circuits using symbolic ternary system models," in *Computer-Aided Verification*, pp. 33–43, Springer, 1991.

[164] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," *NSDI, Apr*, 2012.

[165] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations," in *ACM SIGPLAN Notices*, vol. 41, pp. 55–66, ACM, 2006.

[166] H. R. Nielson and F. Nielson, *Semantics with applications: an appetizer*. Springer, 2007.

[167] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[168] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controllers.," in *PLDI*, pp. 483–494, 2013.

[169] H. Abelson, G. Sussman, and J. Sussman, "Structure and interpretation of computer programs," *MIT Press, Cambridge, MA, USA*, 1985.

[170] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom analytics: exploring data-centric, declarative programming for the cloud," in *Proceedings of the 5th European conference on Computer systems*, pp. 223–236, ACM, 2010.

[171] S. P. Jones, "The future is parallel, the future of parallel is declarative," 2012.

[172] J. M. Hellerstein, "The declarative imperative: experiences and conjectures in distributed logic," *ACM SIGMOD Record*, vol. 39, no. 1, pp. 5–19, 2010.

[173] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking," *Communications of the ACM*, vol. 52, no. 11, pp. 87–95, 2009.

[174] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pp. 1–10, ACM, 2009.

[175] D. Maier and D. Warren, "Computing with logic," 1988.

[176] S. C. Johnson, *Yacc: Yet another compiler-compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[177] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "binpac: A yacc for writing application protocol parsers," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pp. 289–300, ACM, 2006.

[178] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, "Declarative routing: extensible routing with declarative queries," in *ACM SIGCOMM Computer Communication Review*, vol. 35, pp. 289–300, ACM, 2005.

[179] N. P. Katta, J. Rexford, and D. Walker, "Logic programming for software-defined networks," in *Workshop on Cross-Model Design and Validation (XLDI)*, 2012.

[180] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, analyzable controller programming," 2013.

[181] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky, "Declarative network verification," in *Practical Aspects of Declarative Languages*, pp. 61–75, Springer, 2009.

[182] J. Harrison, "Introduction to functional programming," *Lecture Notes, Cam*, 1997.

[183] D. S. Scott, "λ-calculus: Then & now," in *ACM Turing Centenary Celebration*, p. 9, ACM, 2012.

[184] R. Herken, *The universal Turing machine: a half-century survey*, vol. 2. Springer, 1995.

[185] Z. Luo, *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., 1994.

[186] S. Thompson, *Haskell: the craft of functional programming*, vol. 2. Addison-Wesley, 1999.

[187] L. C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1996.

[188] M. J. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF: a mechanised logic of computation*, vol. 78. Springer, 1979.

[189] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 43–48, ACM, 2012.

[190] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," *NSDI, Apr*, 2013.

[191] A. Voellmy and P. Hudak, "Nettle: Taking the sting out of programming network routers," in *Practical Aspects of Declarative Languages*, pp. 235–249, Springer, 2011.

[192] Z. Wan and P. Hudak, "Functional reactive programming from first principles," in *ACM SIGPLAN Notices*, vol. 35, pp. 242–252, ACM, 2000.

[193] "The Frenetic Research Project [Online]." http://www.frenetic-lang.org. Accessed: 2013-09-12.

[194] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *Advanced Functional Programming*, pp. 159–187, Springer, 2003.

[195] M. Handley, "Why the internet only just works," *BT Technology Journal*, vol. 24, no. 3, pp. 119–129, 2006.

[196] T. Ridge, M. Norrish, and P. Sewell, "A rigorous approach to networking: TCP, from implementation to protocol to service," in *FM 2008: Formal Methods*, pp. 294–309, Springer, 2008.

[197] A. R. Khakpour and A. X. Liu, "Quantifying and querying network reachability," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pp. 817–826, IEEE, 2010.

[198] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[199] X. Ou, S. Govindavajhala, and A. W. Appel, "MulVAL: A logic-based network security analyzer," in *14th USENIX Security Symposium*, pp. 1–16, 2005.

[200] E. S. Al-Shaer and H. H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4, pp. 2605–2616, IEEE, 2004.

[201] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," *SIGCOMM-Computer Communication Review*, vol. 41, no. 4, p. 26, 2011.

[202] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pp. 156–165, IEEE, 2000.

[203] M. G. Gouda and A. X. Liu, "Structured firewall design," *Computer Networks*, vol. 51, no. 4, pp. 1106–1120, 2007.

[204] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu, "Formally verifiable networking," in *HotNets*, ACM Sigcomm, 2009.

[205] A. Noyes, T. Warszawski, and N. Foster, "Toward synthesis of network updates," in *Workshop on Synthesis (SYNT)*, 2013.

[206] U. T. B. T. L. Anduo Wang, Salar Moarref and A. Scedrov., "Automated synthesis of reactive controllers for software-defined networks," *The 3rd International Workshop on Rigorous Protocol Engineering, WRIPE 2013*, 2013.

[207] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pp. 241–252, ACM, 2012.

[208] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "Where is the debugger for my software-defined network?," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 55–60, ACM, 2012.

[209] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker, "How did we get into this mess? isolating fault-inducing inputs to sdn control software," tech. rep., Technical Re-

port UCB/EECS-2013-8, EECS Department, University of California, Berkeley, 2013.

[210] D. Sethi, S. Narayana, and S. Malik, "Abstractions for Model Checking SDN Controllers," in *Formal Methods in Computer Aided Design*, 2013.

[211] M. C. Yuang, "Survey of protocol verification techniques based on finite state machine models," in *Computer Networking Symposium, 1988., Proceedings of the*, pp. 164–172, IEEE, 1988.

[212] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets," in *ACM SIGCOMM Computer Communication Review*, vol. 35, pp. 265–276, ACM, 2005.

[213] Z. Chen, D. Zhang, R. Zhu, Y. Ma, P. Yin, and F. Xie, "A review of automated formal verification of ad hoc routing protocols for wireless sensor networks," *arXiv preprint arXiv:1305.7410*, 2013.

[214] R. Lai, "A survey of communication protocol testing," *Journal of Systems and Software*, vol. 62, no. 1, pp. 21–46, 2002.

[215] G. Cabodi, P. Camurati, and S. Quer, "Improved reachability analysis of large finite state machines," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pp. 354–360, IEEE Computer Society, 1997.

[216] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, *et al.*, "Carving research slices out of your production networks with openflow," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 129–130, 2010.

[217] "What's Behind Network Downtime? - IBM." www-05.ibm.com/uk/juniper/pdf/200249.pdf. Accessed: 2013-09-30.

[218] N. Feamster, "Practical verification techniques for wide-area routing," *ACM Sigcomm Computer Communication Review*, vol. 34, no. 1, pp. 87–92, 2004.

[219] C. Ramakrishnan and R. Sekar, "Model-based analysis of configuration vulnerabilities," *Journal of Computer Security*, vol. 10, no. 1, pp. 189–209, 2002.

[220] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.

[221] D. M. Nicol, W. H. Sanders, and K. S. Trivedi, "Model-based evaluation: from dependability to security," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 48–65, 2004.

[222] D. Antoš, V. Rehak, and J. Korenek, "Hardware router's lookup machine and its formal verification," in *ICN'2004 Conference Proceedings*, vol. 2, pp. 1–002, Citeseer, 2004.

[223] P. Curzon, *The formal verification of the Fairisle ATM switching element*. Citeseer, 1994.

[224] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz, "A formal approach to the verification of networks on chip," *EURASIP Journal on Embedded Systems*, vol. 2009, p. 2, 2009.

[225] T. van den Broek and J. Schmaltz, "Towards a formally verified network-on-chip," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pp. 184–187, IEEE, 2009.

[226] P. Zave, "Using lightweight modeling to understand chord," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 49–57, 2012.

[227] M. Karsten, S. Keshav, S. Prasad, and M. Beg, "An axiomatic basis for communication," in *ACM SIGCOMM Computer Communication Review*, vol. 37, pp. 217–228, ACM, 2007.

[228] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, *et al.*, "Leveraging SDN layering to systematically troubleshoot networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 37–42, ACM, 2013.

[229] M.-K. Shin, K.-H. N. M. Kang, and J.-Y. Choi, "Formal specification and programming for sdn," *IETF 84 Proceedings*, 2012.

[230] R. W. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury, "Verifiably-safe software-defined networks for CPS," in *Proceedings of the 2nd ACM international conference on High confidence networked systems*, pp. 101–110, ACM, 2013.

[231] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN Notices*, vol. 47, pp. 217–230, ACM, 2012.

[232] G. Stewart, "Computational verification of network programs in coq," in *Certified Programs and Proofs*, 2013.

[233] N. A. Handigol, *Using packet histories to troubleshoot networks*. PhD thesis, Stanford University, 2013.

[234] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "Ofrewind: enabling record and replay troubleshooting for networks," in *USENIX ATC*, 2011.

[235] R. C. Scott, A. Wundsam, K. Zarifis, and S. Shenker, "What, Where, and When: Software Fault Localization for SDN," tech. rep., Technical Report UCB/EECS-2012-178, EECS Department, University of California, Berkeley, 2012.

[236] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems.," in *NSDI*, vol. 6, pp. 115–128, 2006.

[237] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker, "Automatic troubleshooting for sdn control software," 2013.

[238] S. Keshav, "Editor's message: Modeling," *Computer Communication Review*, vol. 42, no. 3, p. 3, 2012.

[239] J. P. Bowen and M. G. Hinchey, "Ten commandments of formal methods," *Computer*, vol. 28, no. 4, pp. 56–63, 1995.

[240] P. Zave, "A Practical Comparison of Alloy and SPIN," in *IFIP Working Group 2.3 Programming Methodology*, pp. 1–9, 2012.

Junaid Qadir He is an Assistant Professor at the School of Electrical Engineering and Computer Sciences (SEECS), National University of Sciences and Technology (NUST), Pakistan. He is also the Director of the Cognet Lab at SEECS. He completed his BS in Electrical Engineering from UET, Lahore, Pakistan and his PhD from University of New South Wales, Australia in 2008. His research interests include networking/ algorithmic issues in cognitive radio networks, wireless networks, and software-defined networks.

Osman Hasan He is an Assistant Professor at the School of Electrical Engineering and Computer Sciences (SEECS), National University of Sciences and Technology (NUST), Pakistan. He is the lab director of System Analysis and Verification (SAVe) Lab at NUST SEECS. His main research interests include formal verification, interactive theorem proving and higher-order logic. Prior to joining NUST SEECS, he did his PhD and post-doctoral fellowship from the hardware verification group (HVG) at Concordia University, Montreal, Canada.