Formalizing Inductive Proofs of
Network Algorithms

Ramesh Bharadwaj\(^1\), Amy Felty\(^2\), Frank Stomp\(^2\)

\(^1\) CRL, McMaster University, 1280 Main St. West, Hamilton, ON, Canada L8S4K1
\(^2\) AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA

Abstract. Theorem proving and model checking are combined to fully formalize a correctness proof of a broadcasting protocol. The protocol is executed in a network of processors which constitutes a binary tree of arbitrary size. We use the theorem prover COQ and the model checker SPIN to verify the broadcasting protocol.

Our goals in this work are twofold. The first one is to provide a strategy for carrying out formal, mechanical correctness proofs of distributed network algorithms. Even though logical specifications of programs implementing such algorithms are often defined precisely enough to allow a human verifier to prove the program's correctness, the definition of the network is often only informal or implicit. Our example illustrates how an underlying network can be formally defined by means of induction, and how to reason about network algorithms by structural induction.

Our second goal is to integrate theorem proving and model checking to increase the class of algorithms for which mechanical verification is practical. Theorem provers are expressive and powerful, but require sophisticated insight and guidance by the user. Model checkers are fully automatic and effective for verifying finite state automata, but limited to finite spaces of a certain size. We provide a proof strategy which draws on the strengths of both techniques.

1 Introduction

In general, distributed network algorithms are designed to function properly for a specific class of networks, such as rings or complete networks. In most cases the size of the network is unknown and the algorithms are described in a generic way. The (topology of the) underlying network is crucial for the correctness of an algorithm. However, the definition of the network is often left out of the logical specification of the program implementing the algorithm; it is often informal and only implicitly defined. As a consequence, it is not directly possible to mechanically check whether a correctness proof (constructed manually) itself is correct. The current paper addresses this problem, and shows how a combination of model checking and theorem proving can be used to reason about programs executed in a specific class of networks when the size and exact shape of the network are unknown.

Model checking has been used to verify a number of distributed network algorithms and protocols. It is a powerful verification technique that provides
full automation. However, model checkers cannot handle networks of arbitrary size. Theorem provers, on the other hand, generally implement very expressive logics which can handle infinite or arbitrary parameters, such as the number of processes. But they require sophisticated insight and guidance by the user. In this paper, we present an integration of theorem proving and model checking such that structural induction over the network is done within a theorem prover, whereas the base case and many of the subcases of the induction step are verified using a model checker.

In our combined approach, we use the COQ Proof Development System [6] and the SPIN Verification System [15]. COQ is an interactive tactic-style theorem prover which implements the Calculus of Inductive Constructions (CIC), a higher-order type theory that supports inductive types. When a type is defined inductively in COQ, a principle of structural induction and an operator for defining functions recursively over that type are automatically generated. SPIN is a model checker for establishing temporal properties of systems modeled in a guarded commands-like language called PROMELA.

The example we consider to demonstrate our techniques is the PIF-protocol, a broadcasting algorithm developed by Segall [26], executed in a network that constitutes a binary tree. ("PIF" stands for Propagation of Information with Feedback.) The size of the tree is left unspecified. The PIF-protocol is important because it can be identified in many distributed network algorithms, such as the spanning tree algorithm in [8] and the minimum path algorithms in [26]. Intuitively, the PIF-protocol achieves the following: A value, initially recorded by the root of the tree, has to be broadcast and eventually every node in the tree should record this value. Also, the root should eventually be notified that every node has recorded the value.

We specify the PIF-protocol in Manna and Pnueli's Linear Time Temporal Logic (LTL) [19]. The program implementing the PIF-protocol is a pair consisting of a state formula and a finite set of actions formulated as in UNITY [3]. (A state formula is an LTL formula without temporal operators.) The formula characterizes the states in which the program may start its execution. Our correctness proof of the PIF-protocol can be decomposed into three parts: (a) a proof that some state formula continuously holds; (b) a proof that some state formula is stable (once the formula holds, it continues to hold); and (c) a proof of a liveness property.

For part (a) we have applied (a variant of) the S_Inv rule of Manna and Pnueli [19]. This rule states that state formula I is always true if there exists a state formula Inv such that Inv holds initially; it is preserved under every action of the program; and it is stronger than I. The technical formulation of this rule is as follows, where \( \square \) denotes the always-operator from LTL.

\[
\Theta \rightarrow \text{Inv}, \quad \{\text{Inv}\}_i \rightarrow \text{Inv}, \quad \ldots, \quad \text{Inv} \rightarrow I \\
\text{Prog} \vdash \square \Theta \\
\text{for } \text{Prog} = (\Theta, \{\tau_1, \ldots, \tau_n\})
\]

Here \( \Theta \) is the initial condition and \( \tau_1, \ldots, \tau_n \) are the actions of program \( \text{Prog} \). The formula \( \{p\}_\tau \{q\} \) denotes a Hoare triple interpreted as usual: if state formula \( p \) holds before action \( \tau \) is executed, then state formula \( q \) holds after. Using COQ