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.

* To appear in *Proceedings of the 1995 Asian Computing Conference.*

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 $\Box$ denotes the always-operator from LTL.

$$\frac{\Theta \rightarrow Inv, \quad \{Inv\}\tau_i\{Inv\}, i = 1, \ldots, n, \quad Inv \rightarrow I}{Prog \vdash \Box I} \quad \text{for } Prog = \langle \Theta, \{\tau_1, \ldots, \tau_n\} \rangle$$

Here Θ is the initial condition and $\tau_1, \ldots, \tau_n$ are the actions of program $Prog$.

The formula $\{p\}\tau\{q\}$ denotes a Hoare triple interpreted as usual: if state formula p holds before action τ is executed, then state formula q holds after. Using COQ and SPIN, we prove the premises of the S_Inv rule for an arbitrary binary tree. Formula I expresses that whenever the root has been notified that all nodes have recorded the value broadcast, all nodes have indeed recorded that value. There are four parts to the proof:

1. Definitions are given in COQ to specify programs as well as the syntax and inference rules for the fragment of LTL needed for our example.
2. The structure of the network is formally specified by defining binary trees of arbitrary size using the built-in inductive types of COQ.
3. The definition of trees is used to define two functions, one which maps a binary tree to a set of actions expressing the program for that tree, and one which maps a binary tree to an LTL formula which expresses the invariant Inv for that tree.
4. The premises of the S_Inv rule are established by structural induction on binary trees.

As mentioned, correctness of the PIF-protocol also involves proving a stable property and a liveness property. The stable property has been established in the same way as the formula in (a) above by application of a proof rule similar to rule S_Inv. We have not done the proof of the liveness property. This proof will be similar to the other two because it again involves reasoning about the program's actions.

The premises of the S_Inv rule are proved by three inductive arguments, one each for the first and last premises, and one for all of the remaining premises. The first two do not involve reasoning about actions. SPIN is used to handle some tedious but straightforward propositional reasoning. For the third inductive argument, the base case (for the one-node tree) involves reasoning about a program containing two actions. SPIN easily verifies that the invariant holds for each action. In the induction step, we assume that the invariant holds for the program of a tree t, and we must show that a slightly larger invariant holds for a slightly larger program obtained from tree t with two new nodes attached at some leaf. We decompose the inductive case into many cases, of which twenty-six are verified by SPIN. These cases are generally obtained from subgoals of the form $\{p \wedge q\}\tau\{p \wedge q\}$, for state formulas p, q and action τ. These cases can be split into two subgoals $\{p\}\tau\{p\}$ and $\{q\}\tau\{q\}$ such that the former can be proved easily using the theorem prover, and the latter can be mapped directly to a PROMELA program and verified using SPIN. The formula q in these cases is quite large and a direct proof in COQ involves a lot of detailed repetitive reasoning, which we avoid because of our use of the model checker.

In related work, COQ is used in [1, 14] to verify the Alternating Bit Protocol and a data link protocol without the aid of a model checker. In both these proofs the network is fixed. Chou [4] verifies the PIF-protocol for arbitrary connected graphs in the HOL theorem prover [11] again without the aid of a model checker. His proof uses abstraction to reduce the concrete version of the problem to

an abstract one. In particular, he defines an abstract version of the concrete program, shows that the property holds for the abstract program, and shows that any property that holds for the abstract program also holds for the concrete one. In contrast to his proof, our proof does not use abstraction; ours is direct and, in addition, supported by a model checker. It is straightforward to extend our proof to cope with arbitrary connected graphs.

As mentioned, mechanical assistance in proofs is also offered by model checkers [5, 15, 21, 16]. They establish validity of formulae in a model, are fully automated, and are extremely fast for reasonably sized models. All model checkers suffer from the state explosion problem, which has been attacked in [10, 20, 24, 28]. Model checkers have been used to verify a number of complex systems, see for example [21]. Several methods for inductive reasoning about systems consisting of an arbitrary number of (identical or similar) processes have been proposed in the literature. German and Sistla [9] present a fully automatic method. Their algorithm is doubly exponential in the size of the system, and therefore inefficient. Induction principles based on equivalences between systems have been proposed by Browne, Clarke, and Grumberg [2] and by Shtadler and Grumberg [27]. Pre-orders, rather than equivalences, between systems are used in the methods of Kurshan and McMillan [18] and of Wolper and Lovinfosse [29]. In contrast to our use of Coq's built-in structural induction, each of the above mentioned induction principles is tailored to a specific application.

Kurshan and Lamport [17] and others have investigated how to integrate theorem proving and model checking to verify programs when pure model checking fails. In [17] a 64-bit multiplier is proved correct. Rajan, Shankar, and Srivas [25] and Müller and Nipkow [22] combine theorem proving and model checking to verify infinite state systems. In these two papers, the underlying idea is to reduce an infinite state system to a finite one using abstraction techniques as in [4]. Unlike [4], in [25] and [22] the finite state system is verified by a model checker, whereas the reduction is verified using the theorem prover. In our example presented here, instead of abstraction, we handle the arbitrary parameter (in our case the number of nodes) by a direct inductive argument and use model checking whenever applicable on the subcases.

The rest of this paper is organized as follows: The PIF-protocol is described in Sect. 2. In Sect. 3, we briefly present Coq and Spin. In Sect. 4, we outline how our correctness proof has been carried out using a combination of these two systems. Finally, Sect. 5 draws some conclusions.

2 The PIF-Protocol

In this section we specify and implement the PIF-protocol as analyzed in the rest of this paper.

2.1 Specification

Consider a fixed, but arbitrary network constituting a non-empty, finite, binary tree. Nodes in the tree are identified with processes; edges with communication

channels. One node R is identified with the tree's root. Assume that R has recorded some value V. The informal specification of the PIF-protocol is:

(1) Eventually every process in the network records V.
(2) Eventually R is notified that all processes have recorded value V; and once this notification has taken place, all processes continue to record that value.

For a given graph (N, E), where N is a set of nodes and E is a set of edges, let $Tree(N, E)$ denote that this graph is a non-empty, finite, binary tree. Let $R \in N$ denote the tree's root. Every process $n \in N$ has its own variable v_n for recording the broadcast value V. Initially $v_R = V$ holds, *i.e.*, process R has recorded value V, whereas the initial values of variables v_n, for processes n different from R, are irrelevant. The root also has its own variable $done_R$ used to record whether all processes in the network have recorded value V. Initially, $done_R = 0$ holds. (Actually, for nodes n different from R, we have introduced $done_n$ to allow generic descriptions of the processes, but they are never used.)

Using the always-operator $\Box$ and the eventual-operator $\Diamond$ from LTL, it is required that the following holds: If $R \in N \wedge v_R = V \wedge done_R = 0$ holds initially, then

$$\begin{aligned} Tree(N, E) \rightarrow \quad & \Box(done_R = 1 \rightarrow \forall n \in N.v_n = V) \\ \wedge\ & \Box(done_R = 1 \rightarrow \Box done_R = 1) \\ \wedge\ & \Diamond done_R = 1 \end{aligned}$$

is true. That is, it is always the case that all processes in the network have recorded value V if $done_R = 1$ holds; once $done_R = 1$ holds, it continues to hold ($done_R = 1$ is stable); and eventually $done_R = 1$ holds. These three conjuncts correspond to properties (a), (b), and (c) mentioned in the previous section. The proof described in the current paper is that of the first conjunct, property (a).

2.2 Implementation

A program consists of two parts (*cf.* [3]): a state formula and a (finite) collection of guarded actions. The formula characterizes the initial states in which the program may start its execution. A guarded action is of the form $g \rightarrow x_1 := e_1, \ldots, x_m := e_m$ for some natural number $m > 0$, consisting of *guard* g and *body* $x_1 := e_1, \ldots, x_m := e_m$. Here, x_i are distinct variables (to avoid name-clashes) and e_i are expressions ($i = 1, \ldots, m$). Guard g is a boolean expression without quantifiers. An action is *enabled* in a state if its guard evaluates to true in that state. If in some state during execution no action of the program is enabled, then the program is considered terminated as in [19]. Otherwise, an enabled action $g \rightarrow x_1 := e_1, \ldots, x_m := e_m$ is nondeterministically chosen for execution. Execution of this action means that the assignments $x_1 := e_1, \ldots, x_m := e_m$ are executed atomically and simultaneously.

The actions of the program implementing the PIF-protocol are given in Fig. 1. There n ranges over the nodes in the tree; *par* denotes the parent of n, provided that n has a parent; and l and r denote the left and right child of n, respectively. As described above, each node n maintains variables v_n and

$a0 :: cc_n = 0 \wedge pc_n = 1 \rightarrow pc_n := 4, done_n := 1$
$a2\_down :: cc_n = 2 \wedge pc_n = 1 \wedge pc_l = 0 \wedge pc_r = 0 \rightarrow pc_l := 1, v_l := v_n, pc_r := 1, v_r := v_n$
$a3\_down :: cc_n = 3 \wedge pc_n = 1 \wedge pc_l = 0 \wedge pc_r = 0 \rightarrow pc_l := 1, v_l := v_n, pc_r := 1, v_r := v_n$
$a1\_up :: cc_n = 1 \wedge pc_n = 1 \rightarrow pc_{par} := pc_{par} + 1, pc_n := 4$
$a3\_up :: cc_n = 3 \wedge pc_n = 3 \rightarrow pc_{par} := pc_{par} + 1, pc_n := 4$
$a2\_term :: cc_n = 2 \wedge pc_n = 3 \rightarrow pc_n := 4, done_n := 1$

Fig. 1. Actions executed by every node in the tree. The collection of these actions, for all nodes in the tree, constitutes the PIF-protocol.

$done_n$. In addition, every node maintains a variable pc_n, which can be thought of as n's program counter. Initially, pc_R=1 holds, whereas pc_n=0 holds for all nodes n different from R. We have also used variables cc_n for nodes n in the tree. Variables cc_n cannot be changed by any action and represents the number of n's neighbors in the tree. Thus, for the root of the tree either cc_R=0 or cc_R=2 holds. In the first case, R is the only node in the tree; in the second case, the tree consists of more than one node. There exists exactly one node n in the tree satisfying cc_n=0 or cc_n=2. We identify this node with the root R. For other nodes n in the tree, we have that either cc_n=1 (n is a leaf) or cc_n=3 (n is an internal node) holds. The initial values of the cc variables, the pc variables, $done_R = 0$, $v_R = V$, and $R \in N$ characterize the states in which the execution of the program may start. Action $a0$ in Fig. 1 can be executed only if the tree consists of one node. In this case, the node sets its variable pc_R to 4 and its variable $done_R$ to 1 and the program terminates. If the tree consists of more than one node, the root initiates the program by passing on value V to its neighbors (action $a2\_down$). After an internal node has received value V, it passes V on to its children (action $a3\_down$). When a leaf has received value V, it informs its parent about this (action $a1\_up$). After an internal node has been informed that both its children have received the value, the node itself informs its parent (action $a3\_up$). Eventually, when the root gets the information that its children (hence, all other nodes in the tree) have received value V, it sets its variable $done_R$ to 1 and the program terminates (action a2_term).

3 Coq and Spin

We briefly introduce the Coq Proof Development System and the Spin Verification System.

3.1 The Coq Proof Development System

As stated, Coq is an implementation of the Calculus of Inductive Constructions (CIC). Familiarity with CIC is not required for understanding the proofs in the next section. We simply introduce the syntax used there. Let x represent

variables and M, N represent terms of CIC. The syntax of terms is as follows.

$$Prop \mid Set \mid Type \mid x \mid MN \mid \lambda x : M.N \mid \forall x : M.N \mid M \to N \mid$$
$$M \wedge N \mid M \vee N \mid \exists x : M.N \mid \neg M \mid M = N \mid Ind\, x : M\, \{N_1 | \cdots | N_n\} \mid$$
$$Rec\ M\ N \mid Case\ x : M\ of\ M_1 \Rightarrow N_1, \ldots, M_n \Rightarrow N_n$$

$Prop$ is the type of logical propositions, whereas Set is the type of data types. $Type$ is the type of both $Prop$ and Set. In CIC, variables and constants are not distinguished. In Coq, a new constant can be introduced and given with its type using the Parameter keyword. It is also possible to introduce new constants via definitions. The Definition keyword is used for this purpose.

Application is represented as juxtaposition of terms. Abstraction is represented as usual where the bound variable is typed. The logical operators $\forall, \to, \wedge, \vee, \exists, \neg, =$ are the familiar ones from higher-order logic.

The Ind constant is used to build inductive definitions where M is the type of the class of terms being defined and $N_1, \ldots, N_n$ where $n \geq 0$ are the types of the constructors. In Coq, inductive definitions are introduced with an Inductive declaration where each constructor is named and given with its type separated by vertical bars. Rec and $Case$ are the operators for defining recursive and inductive functions, respectively, over inductive types.

3.2 The Spin Verification System

As stated, Spin [15] is a tool for establishing temporal properties of systems modeled in a guarded commands-like language called Promela. Spin has been used to prove properties of communication protocols and asynchronous hardware. It can also be used to prove termination of systems. As we have noted, model checking provides complete automation. The algorithms underlying a model checker such as Spin suffer from scalability: They are PSPACE hard. Consequently, one quickly runs out space as the size of the model increases.

A Promela program consists of a section in which variables are declared and statements. In essence, statements are built up from assignments, the empty statement *skip*, sequential composition, assert statements, conditional statements, and loops. (We will not use loops in this paper.) The conditional statements we use in this paper are of the form $if\ ::g_1 \to S_1 :: \cdots ::g_n \to S_n\ fi$, where symbol "::" separates the *guarded actions* $g_i \to S_i$ where g_i is a guard and S_i is a statement $(i = 1, \ldots, n)$. (Conditional statements in Promela are more general.) If a guard is the constant *true* then it may be omitted. These conditional statements have the same interpretation as, for example, Dijkstra's conditional statements with the exception that in Promela the process blocks (and does not abort) when none of its guards is enabled. An assert statement is of the form $assert\{g\}$, for a guard g. This statement acts like *skip* when executed in a state satisfying g; otherwise the execution is aborted.

For a finite set of states, one can generate an arbitrary state by means of a conditional statement. For example, $if :: x{:=}0 :: x{:=}1\ fi$; $if :: y{:=}1 :: y{:=}2\ fi$

generates some state in the set characterized by predicate $(x = 0 \vee x = 1) \wedge (y = 1 \vee y = 2)$.

Partial correctness $\{p\}\ T\ \{q\}$ of program T w.r.t. precondition p and post-condition q is interpreted as usual (*cf.* Sect. 1). We have that $\{p\}\ g \rightarrow a\ \{q\}$ holds iff $\{p \wedge g\}\ a\ \{q\}$ holds. In our proof, we often need to prove such a partial correctness formula, where a always terminates. Consider the finite set of states corresponding to the possible combinations of values that the variables may take. Let S be the PROMELA program that generates an arbitrary state from this set in the manner described above. The partial correctness formula $\{p \wedge g\}\ a\ \{q\}$ can be shown to be equivalent to termination of the PROMELA program

$S;\ if :: p \wedge g \rightarrow a;\ assert\{q\} :: \neg(p \wedge g) \rightarrow skip\ fi.$

Similarly, validity of the implication $p \rightarrow q$ can be translated into the question of whether or not the PROMELA program

$S;\ if :: p \rightarrow assert\{q\} :: \neg p \rightarrow skip\ fi$

always terminates. Validation of such implications and of partial correctness formulas of single actions are the only two ways in which we use SPIN. (In our example p and q are generally very large.)

4 Correctness Proof of the PIF-Protocol

In this section we outline our correctness proof of the PIF-protocol executed in an arbitrary binary tree.

4.1 Specification of State Formulas and Actions

First, we give definitions in COQ specifying the syntax of state formulas and actions. State formulas are formed from atomic formulas expressing equality between terms and the logical connectives $\wedge, \vee, \rightarrow, \neg$. Terms are formed from variables, the constant zero, and the successor function. These are the only expressions needed for our example. Variables, terms, and state formulas are specified as inductive types in COQ. Processes or nodes in the tree are uniquely identified with a natural number using *nat*, the predefined type of natural numbers in COQ. Variables will take an argument of type *nat* indicating the process to which it belongs. There are four variables for each process defined as follows.

$\mathsf{Inductive}\ var := pc : nat \rightarrow var \mid v : nat \rightarrow var \mid cc : nat \rightarrow var \mid done : nat \rightarrow var.$

The logical operators $\rightarrow, \wedge, \vee, \neg, =$ appear both in CIC expressions and in state formulas which we want to encode in CIC. To avoid confusion, we superscript many of the symbols in the COQ definitions of state formulas with a "$*$". Terms and formulas are defined as follows.

$\mathsf{Inductive}\ tm := 0^* : tm \mid s^* : tm \rightarrow tm \mid x : var \rightarrow tm.$

$\mathsf{Inductive}\ form := False : form \mid \neg^* : form \rightarrow form$
$\mid \wedge^* : form \rightarrow form \rightarrow form \mid \vee^* : form \rightarrow form \rightarrow form$
$\mid \rightarrow^* : form \rightarrow form \rightarrow form \mid =^* : tm \rightarrow tm \rightarrow form.$

We adopt the usual convention that the constructor $\rightarrow$ associates to the right. For readability, we abbreviate both the variable $(pc\ n)$ and the term $(x\ (pc\ n))$

as pc_n, and similarly for the other three kinds of variables. It will always be clear from context which is meant. In addition we use infix notation for the binary connectives $\wedge^*, \vee^*, \rightarrow^*, =^*$. For example, the state formula $pc_n = 0$ is represented by the term $(=^* \ (x \ (pc \ n)) \ 0^*)$ of type *form*, which we write as $(pc_n \ =^* \ 0^*)$. We introduce a parameter V for the value passed through the network. By making it a parameter, our theorems will hold for any instantiation of V. We also define 1^* for convenience later. The terms $2^*, 3^*, 4^*$ are defined similarly.

Parameter V : *tm*.
Definition $1^* := (s^* \ 0^*)$.

We do not include any temporal operators here since they are not needed to prove the premises of the S_Inv rule, which contain only state formulas. We express provability of state formulas via an inductive definition of a predicate *prov* of type $form \rightarrow Prop$. We do not give its definition here. It specifies a natural deduction inference system for the fraction of first-order classical logic that we need and is similar to specifications given in [23, 13, 7]. From this definition, we can prove for example:

Lemma *provable_and_i* : $\forall A, B : form.((prov \ A) \wedge (prov \ B)) \rightarrow (prov \ (A \wedge^* B))$.

Actions consist of a guard and a list of assignment statements. The formulas that can occur in guards are the same as state formulas defined by the type *form*. Assignment statements and actions are defined below. The latter uses the built-in *list* type of Coq.

Inductive $Assign : Set := assign : var \rightarrow tm \rightarrow Assign$.
Inductive $Action : Set := action : form \rightarrow (list \ Assign) \rightarrow Action$.

We specify substitution on terms as a set of equations at the object-level. The Coq term $(subst \ A \ y \ t)$ encodes $[t/y]A$, *i.e.*, the formula obtained from A by replacing every free occurrence of y in A by t. Using the definition of *subst*, we define a function *ht* of type $form \rightarrow Action \rightarrow form \rightarrow form$ which maps a Hoare triple $\{p\}g \rightarrow x_1 := t_1, \ldots, x_n := t_n\{q\}$ to the equivalent state formula $(p \wedge g) \rightarrow [\bar{t}/\bar{x}]q$. Here, $[\bar{t}/\bar{x}]q$ denotes the simultaneous replacement of all free occurrences of x_i in q by t_i, $1 \leq i \leq n$. (We omit the details.)

4.2 Coq Specification of the Network

Binary trees of processors are defined by the following inductive definition.

Inductive $BinTree := root : nat \rightarrow BinTree$
$\quad \mid children : BinTree \rightarrow nat \rightarrow nat \rightarrow nat \rightarrow BinTree$.

Here, $(root \ n)$ is a tree containing only processor n, and $(children \ t \ n_1 \ n_2 \ n)$ is the tree obtained by adding two new children n_1 and n_2 to leaf n in t. We choose this definition of binary trees over the more standard one in which a tree is either a leaf or a node with two subtrees, because it simplifies our proofs by structural induction over trees. Of course, for our definition, we need additional predicates to ensure that a tree is well-formed. For example, in $(children \ t \ n_1 \ n_2 \ n)$, n must occur as a leaf in t, and n_1 and n_2 must be distinct and not already occur in t. For this purpose, we define the sets and predicates below. Instead of giving their

formal definitions, we give a short explanation. They are all defined recursively over the type *BinTree*. The set theory library of Coq is used in these definitions.

- $(troot\ t)$ evaluates to the root of tree t.
- $(pids\ t)$ gives the set of natural numbers (processes) in t.
- $(parents\ t)$ evaluates to the set of nodes in t that occur as parents.
- $(distinct\_nodes\ t)$ holds if all of the process identifiers that occur at the nodes in t are distinct from one another. For a one-node tree, this predicate always holds. The proposition $(distinct\_nodes\ (children\ t\ n_1\ n_2\ n))$ is equivalent to $\neg(n_1 \in (pids\ t)) \wedge \neg(n_2 \in (pids\ t)) \wedge \neg(n_1 = n_2) \wedge (distinct\_nodes\ t)$.
- $(correct\_parents\ t)$ holds if every time two children are added at a node n, n occurs in t and does not already have children. This predicate always holds for one-node trees, and $(correct\_parents\ (children\ t\ n_1\ n_2\ n))$ is equivalent to $(n \in (pids\ t)) \wedge \neg(n \in (parents\ t)) \wedge (correct\_parents\ t)$.

The predicate *tree* which holds only for well-formed trees is defined as follows.

Definition $tree := \lambda t : BinTree.(distinct\_nodes\ t) \wedge (correct\_parents\ t)$.

In Coq, each time an inductive definition is given, a structural induction principle is automatically generated and proved. The induction principle for trees, which plays an essential role in the proofs here, is the following.

$\forall P : BinTree \rightarrow Prop.$
$(\forall n : nat.(P\ (root\ n))) \rightarrow$
$(\forall t : BinTree.(P\ t) \rightarrow \forall n_1, n_2, n : nat.(P\ (children\ t\ n_1\ n_2\ n))) \rightarrow$
$\forall t : BinTree.(P\ t).$

Using this principle, the following theorem, for example, can be proved easily.

Lemma $tree\_subtree : \forall t : BinTree.\forall n_1, n_2, n : nat.$
$(tree\ (children\ t\ n_1\ n_2\ n)) \rightarrow (tree\ t).$

4.3 Basic Definitions for the PIF-Protocol

As stated, we define a function which maps a tree to a set of actions implementing the PIF-protocol and another function which maps a tree to an invariant used in proving properties of this implementation. To define the former, we encode each action as a function from natural numbers (nodes) to actions. For example the $a1\_up$ action is encoded as follows (where brackets are used to denote lists in Coq and commas are used to separate list items).

Definition $a1\_up := \lambda n, par : nat.$
$(action\ ((cc_n =^* 1^*) \wedge^* (pc_n =^* 1^*))\ \ [(assign\ pc_{par}\ (s^*\ pc_{par})), (assign\ pc_n\ 4^*)]).$

We then define a function *actions* of type $BinTree \rightarrow (set\ Action)$ that takes a tree and returns a set containing $a0$, $a2\_down$, and $a2\_term$ for the root, $a3\_down$ and $a3\_up$ for each internal node, and $a1\_up$ for each leaf.

The invariant used in our proofs is defined as a conjunction of state formulas where each conjunct is one of the formulas below instantiated for a particular node. In these formulas, n refers to an arbitrary node, l refers to n's left child, r refers to n's right child, and par refers to n's parent.

(I1) $cc_n = 0 \rightarrow (v_n = V \wedge ((pc_n = 1 \wedge done_n = 0) \vee (pc_n = 4 \wedge done_n = 1)))$

(I2) $cc_n = 1 \rightarrow ((pc_n = 0 \vee pc_n = 1 \vee pc_n = 4) \wedge (\neg pc_n = 0 \rightarrow v_n = V))$

(I3) $cc_n = 2 \rightarrow (v_n = V \wedge (((pc_n = 1 \vee pc_n = 2 \vee pc_n = 3) \wedge done_n = 0) \vee (pc_n = 4 \wedge done_n = 1)))$

(I4) $cc_n = 3 \rightarrow ((pc_n = 0 \vee pc_n = 1 \vee pc_n = 2 \vee pc_n = 3 \vee pc_n = 4) \wedge (\neg pc_n = 0 \rightarrow v_n = V))$

(I5) $(cc_n = 3 \wedge pc_n = 0) \rightarrow (pc_l = 0 \wedge pc_r = 0)$

(I6) $((cc_n = 1 \vee cc_n = 3) \wedge pc_n = 0) \rightarrow (pc_{par} = 0 \vee pc_{par} = 1)$

(I7) $((cc_n = 2 \vee cc_n = 3) \wedge pc_n = 1) \rightarrow ((pc_l = 0 \wedge pc_r = 0) \vee ((pc_l = 1 \vee pc_l = 2 \vee pc_l = 3) \wedge (pc_r = 1 \vee pc_r = 2 \vee pc_r = 3)))$

(I8) $((cc_n = 1 \vee cc_n = 3) \wedge pc_n = 1) \rightarrow (pc_{par} = 1 \vee pc_{par} = 2)$

(I9) $((cc_n = 2 \vee cc_n = 3) \wedge pc_n = 2) \rightarrow ((pc_l{=}4 \wedge (pc_r{=}1 \vee pc_r{=}2 \vee pc_r{=}3)) \vee (pc_r{=}4 \wedge (pc_l{=}1 \vee pc_l{=}2 \vee pc_l{=}3)))$

(I10) $(cc_n = 3 \wedge pc_n = 2) \rightarrow (pc_{par} = 1 \vee pc_{par} = 2)$

(I11) $((cc_n = 2 \vee cc_n = 3) \wedge pc_n = 3) \rightarrow (pc_l = 4 \wedge pc_r = 4)$

(I12) $(cc_n = 3 \wedge pc_n = 3) \rightarrow (pc_{par} = 1 \vee pc_{par} = 2)$

(I13) $((cc_n = 2 \vee cc_n = 3) \wedge pc_n = 4) \rightarrow (pc_l = 4 \wedge pc_r = 4)$

(I14) $((cc_n = 1 \vee cc_n = 3) \wedge pc_n = 4) \rightarrow (pc_{par} = 2 \vee pc_{par} = 3 \vee pc_{par} = 4)$

Each of these formulas is easy to understand. For example, conjunct (I9) states that for an internal node or root n of a tree consisting of more than one node, the following is true: If $pc_n{=}2$ then the pc variable of one of n's children equals 4, and the pc variable of the other child is in the range from 1 to 3. Each of these formulas is encoded in Coq as a function from natural numbers (nodes) to type *form* in the obvious way. For example, the encoding of (I9) is:

```
Definition I9 := λn, l, r : nat.((cc_n =* 2* ∨* cc_n =* 3*) ∧* pc_n =* 2*) →*
  ((pc_l =* 4* ∧*(pc_r =* 1* ∨* pc_r =* 2* ∨* pc_r =* 3*)) ∨*
   (pc_r =* 4* ∧*(pc_l =* 1* ∨* pc_l =* 2* ∨* pc_l =* 3*))).
```

Using these definitions, we define a function Inv of type $BinTree \rightarrow form$ that takes a tree as its argument, and returns a formula that is a large conjunction of each of the fourteen formulas of the invariant included for each internal node in the tree, each of the formulas except those that relate a node to its children for each leaf of the tree, and each of the conjuncts except those that relate a node to its parent for the root. We omit the precise definition here. It uses an auxiliary definition *inv_triple*, where $(inv\_triple\ n\ n_1\ n_2)$ characterizes that part of the invariant that relates the variables of nodes n, n_1, n_2 to each other. Using this definition, the formula $(Inv\ (children\ t\ n_1\ n_2\ n))$ is equivalent to $(Inv\ t) \wedge (inv\_triple\ n\ n_1\ n_2)$.

The initial condition and the safety property of the PIF-protocol that we want to prove are defined in Coq as follows.

Definition $Init := \lambda t : BinTree.\forall n : nat.(n \in (pids\ t)) \rightarrow$
$(prov\ (((cc_n =^* 0^*) \vee^* (cc_n =^* 2^*)) \rightarrow^*$
$((pc_n =^* 1^*) \wedge^* (v_n =^* V) \wedge^* (done_n =^* 0^*))) \wedge^*$
$(((cc_n =^* 1^*) \vee^* (cc_n =^* 3^*)) \rightarrow^* (pc_n =^* 0^*)))$.
Definition $I := \lambda t : BinTree.(prov\ (done_{(troot\ t)} =^* 1^*)) \rightarrow$
$\forall n : nat.(n \in (pids\ t)) \rightarrow (prov\ (v_n =^* V))$.

4.4 Discussion of our Correctness Proof

We next discuss our correctness proof of the PIF-protocol. We concentrate on application of the S_Inv rule to establish the invariance of property $(I\ t)$ for arbitrary tree t.

As preparation we derive the values that each variable may take. For example, the values of every pc variable are 0, 1, 2, 3, or 4. (Of course, we derive these properties by means of theorem proving.) We need these properties, because model checkers can deal only with variables whose values are in a certain (finite) range.

The following theorems correspond to the first and last premises of the S_Inv rule. They are proved by induction, using SPIN for some propositional reasoning.

Theorem $init\_imp\_inv : \forall t : BinTree.(tree\ t) \rightarrow (Init\ t) \rightarrow (prov\ (Inv\ t))$.
Theorem $inv\_imp\_I : \forall t : BinTree.(tree\ t) \rightarrow (prov\ (Inv\ t)) \rightarrow (I\ t)$.

The next theorem is the most complex, establishing the premises of the S_Inv rule that deal with the actions of the program.

Theorem $invariant\_actions : \forall t : BinTree.(tree\ t) \rightarrow \forall a : Action.$
$(a \in (actions\ t)) \rightarrow (provable\ (ht\ (Inv\ t)\ a\ (Inv\ t)))$.

The proof is by induction on t. For the basis of induction we have two cases, one for action $a0$ and one for action $a2\_term$. Each of this cases is easily model checked. For the induction step, when tree t is of the form $(children\ t'\ n_1\ n_2\ n)$, we have to show that $(Inv\ (children\ t'\ n_1\ n_2\ n))$ is preserved by every action that can be executed by nodes in t. By theorem proving we deduce that this holds if (1) and (2) below both hold.

(1) $(Inv\ (children\ t'\ n_1\ n_2\ n))$ is preserved by each of the six "new" actions whose execution involves one of the nodes n_1, n_2 (and n).
(2) $(Inv\ (children\ t'\ n_1\ n_2\ n))$ is preserved by "old" actions whose execution involves only nodes in t'.

For both (1) and (2), we decompose the reasoning by some simple Hoare rules, which we have proved using COQ. For ease of exposition, we consider one kind of subcase that arises when proving (1) in which t is of the form shown in Fig. 2. This kind of subcase results from another inductive argument.

(We omit the details of this subinduction.) Thus, node n and at least one of the nodes $n1, n2$ are involved in the execution of action a. Let t' be the subtree of t consisting of tree y, as in Fig. 2, and the nodes $n, n3$. We then use COQ to show that $(Inv\ (children\ t'\ n_1\ n_2\ n))$ is equivalent to the conjunction of $(inv\_triple\ n\ n_1\ n_2)$, $(inv\_triple\ par\ n_3\ n)$, and some formula J, where J does

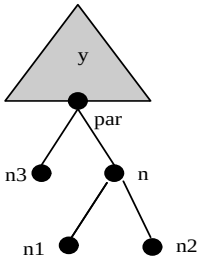

Fig. 2. Binary tree to illustrate our proof strategy.

not refer to variables of the nodes n, n_1, n_2. Using the theorem prover we show that J is preserved by action a, because J does not refer to variables that can be modified by a. Then we prove property $\{P\}a\{P\}$, for P defined as the conjunction of $(inv\_triple\ n\ n_1\ n_2)$ and $(inv\_triple\ par\ n_3\ n)$. This is done by showing that for action $a \equiv g \rightarrow bd$ the PROMELA program

S; $if :: P \wedge g \rightarrow bd$; $assert\{P\} :: \neg(P \wedge g) \rightarrow skip\ fi$

always terminates (*cf.* Sect. 2.2). As before, S is a program that generates possible values of the variables. Application of Hoare rules then completes this subcase.

In total, (1) and (2) consist of about fifteen problems. Intuitively, we have used COQ to decompose –without much effort, because theorem provers do this well– each of the problems into subproblems so that each of these subproblems is solved by a model checker. We have identified twenty-six cases which could be model checked. The amount of time taken by SPIN to validate the subproblems ranges from a few seconds to about one hour; the amount of states enumerated needed to do so ranged from about 200 to 200,000,000. We could only apply model checking to problems in which the predicates and actions were "concrete".

We have also verified the PIF-protocol by theorem proving techniques only, and found that the use of a model checker significantly simplifies the size of the proof as well as the effort that we, as human verifiers, have to invest. Even though some of the problems required about an hour to be model checked, constructing a proof requires a lot more effort.

5 Conclusion

Model checking and theorem proving have been combined to show that inductive reasoning about network algorithms can be carried out to mechanically verify network algorithms. As an example we proved correctness of the PIF-protocol when the underlying network constitutes a binary tree. The proof is by structural induction on the binary tree. Induction is handled by the theorem prover, and the base case as well as many subcases in the induction step are handled by the model checker. Although we have used the model checker SPIN and the theorem prover COQ, our results would not be affected by another choice of model checker or higher-order tactic-style theorem prover.

Model checkers are attractive because they provide complete automation. On their own, they cannot verify the PIF-protocol, because the state space is of arbitrary, although finite size. Theorem provers are attractive because of their generality, and can be used to prove correctness of the PIF-protocol. Yet such proofs require sophisticated insight and guidance by the user. Combining both techniques as we have offers the advantages of each of them, while overcoming their drawbacks. We have identified those subproblems where model checking applies. The theorem prover has been used only to tackle those subproblems that are out of reach of model checkers, or to bring a subproblem into a form that is within the reach of a model checker.

We plan to formulate more general induction principles and to show that our approach scales up by proving correctness of larger algorithms. We also plan to analyze algorithms whose correctness proofs can be structured so that model checking is not only applied to single-step programs, as in the current paper, but to more complicated ones in order to take fuller advantage of model checking techniques. An example of a proof rule that allows such structuring is the rule in [12] for proving strongly-fair termination of programs, because the rule must be applied recursively to smaller programs. One of the premises of the rule requires proving strongly-fair termination of a smaller program, which can be model checked if feasible. Otherwise, the theorem prover must be used to repeatedly apply the rule to decompose the problem into subproblems until a program is obtained that is small enough to be model checked.

References

1. Marc Bezem and Jan Friso Groote. A formal verification of the alternating bit protocol in the calculus of constructions. Technical Report Logic Group Reprint Series No. 88, Utrecht University, 1993.
2. M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical processes. In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, 1986.
3. K. M. Chandy and J. Misra. *Parallel Program Design—A Foundation*. Addison-Wesley, 1988.
4. Ching-Tsun Chou. Mechanical verification of distributed algorithms in higher-order logic. *The Computer Journal*, 1995. To appear.
5. 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*, 8(2):244–263, 1986.
6. Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, Christine Paulin-Mohring, César Muñoz, Chetan Murthy, Catherine Parent, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1995.
7. Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.
8. R. T. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

9. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the Association for Computing Machinery*, 39(3):675–735, 1992.
10. Patrice Godefroid. Using partial orders to improve automatic verification methods (extended abstract). In *Proceedings of the 2nd International Workshop on Computer-Aided Verification*, pages 176–185. Springer Verlag Lecture Notes in Computer Science 513, 1990.
11. M. J. C. Gordon and T. F. Melham. *Introduction to HOL—A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
12. O. Grumberg, N. Francez, J. A. Makowsky, and W. P. de Roever. A proof rule for fair termination of guarded commands. *Information and Control*, 66(1/2):83–102, July/August 1985.
13. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.
14. L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. Proof-checking a data link protocol. In *Proceedings of the ESPRIT BRA Workshop on Types for Proofs and Programs*, 1994.
15. Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall Software Series, 1991.
16. R. P. Kurshan. Analysis of discrete event coordination. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (REX Workshop)*, pages 414–453. Springer Verlag Lecture Notes in Computer Science 430, 1989.
17. R. P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In *Proceedings of the 5th International Workshop on Computer-Aided Verification*, pages 166–179. Springer Verlag Lecture Notes in Computer Science 697, 1993.
18. R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.
19. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1991.
20. K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of the 4th International Workshop on Computer-Aided Verification*, pages 164–177. Springer Verlag Lecture Notes in Computer Science 663, 1992.
21. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
22. Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O-automata. In *Proceedings of the First Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12. Technical Report NS-95-2,BRICS Notes Series, Aarhus, 1995.
23. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
24. Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Workshop on Computer-Aided Verification*. Springer Verlag Lecture Notes in Computer Science 801, 1994.
25. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model-checking with automated proof checking. In *Proceedings of the 7th International Workshop on Computer-Aided Verification*. Springer Verlag Lecture Notes in Computer Science, 1995.
26. A. Segall. Distributed network protocols. *IEEE Trans. on Inf. Theory*, IT29(1), 1983.
27. Z. Shtadler and O. Grumberg. Network grammars, communication behavior, and automatic verification. In *Proceedings of the Workshop on Automatic Verification*

Methods for Finite State Systems, pages 151–165. Springer Verlag Lecture Notes in Computer Science, 1989.

28. Antti Valmari. A stubborn attack on state explosion (abridged version). In *Proceedings of the 2nd International Workshop on Computer-Aided Verification*, pages 156–165. Springer Verlag Lecture Notes in Computer Science 513, 1990.
29. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80. Springer Verlag Lecture Notes in Computer Science, 1989.