SNAP: A Validator/Annotator for Proofs of Distributed Programs

R. Bharadwaj* F. A. Stomp†

Abstract

Distributed programs are notorious for subtle errors, requiring proofs to establish their correctness. Correctness proofs carried out on paper tend to be error-prone. This paper presents system SNAP (Simple Notion of Annotating Proofs) being developed to validate correctness proofs in Linear Time Temporal Logic (LTL). SNAP supports *forward proofs* and can be used either *interactively* or in *batch mode*. The system validates proofs entered by users — in a representation close to handwritten proofs — by providing justifications for every proof step. Justifications are in essence theorems or meta rules in the system's rule-base; together with the proof, they constitute a *proof annotation*. Theorems proved elsewhere may be loaded into the system's rule-base *without having to redo proofs*, a mechanism to raise the abstraction level of proofs. SNAP supports proofs of theorems and certain meta rules, all of which may be dynamically added to its rule-base. Since the design goal of SNAP is to validate and annotate correctness proofs of programs, we have sacrificed generality for efficiency and ease of use. For example, the system does not support higher-order logics.

To demonstrate the use of SNAP, we discuss how the proof of a safety property for Dekker's algorithm (an algorithm for mutual exclusion) has been validated by the system.

1 Introduction

Distributed programs are notorious for subtle errors, requiring proofs to establish their correctness. Proofs of distributed programs carried out on paper tend to be error-prone. We are developing system SNAP (Simple Notation of Annotating Proofs) to validate correctness proofs of imperative programs carried out in Linear Time Temporal Logic (LTL) [MP91b].

LTL is built upon first-order predicate logic. Our current implementation of SNAP supports *forward proofs* of formulae expressed in quantifier-free predicate logic (without temporal operators). One of the main design objectives of SNAP has been to handle proofs using a representation that is close to handwritten proofs — to use the system, merely reading any textbook on formal proofs should suffice. A proof supplied by the user is validated by the system by providing justifications for every proof step. These justifications are in essence theorems or meta rules in the system's rule-base; together with the proof, they constitute a *proof annotation*. The key idea of the user supplying a

*CRL, McMaster University, Hamilton, ON, Canada L8S 4K1. Email: ramesh@triose.crl.mcmaster.ca

†AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA. Email: frank@research.att.com

proof, and the system providing its annotation, eliminates the need for users to memorize names of rules to conduct proofs, as in virtually all other systems. Another characteristic of SNAP is that a theorem, once proved, is immediately available for use as justifications in future proofs: it is dynamically added to the rule-base *without its associated proof.*

A conventional interactive system is typically instructed to carry out one proof step at a time — users must peruse system response before being able to determine the next step. In SNAP, on the other hand, proof steps may be provided one at a time, i.e., interactively, or all at once, i.e., in batch mode. There is no need to examine the system's response after every step, because proofs provided to SNAP are in the same style as handwritten proofs. Users may also load theorems and meta rules (with or without their associated proofs) into SNAP's rule-base, allowing them to carry out proofs at a *higher level of abstraction* — it is not required to prove every theorem, as in some systems. For example, one may include a theorem proved elsewhere (without its proof) in the system's rule base. Nested formulations of theorems and their proofs are possible, to support the common practice of formulating, proving, and applying lemmata when doing another proof. The system also offers automation [Bha94]. Consequently, proofs may be given using any combination of interactive, batch, and automated system modes.

SNAP allows theorems and certain meta rules to be proved and used in other proofs. For instance, for a set , of predicates, and predicates P, Q, we demonstrate in section 2 how the cut-rule "if , $\vdash P$ and , $\cup \{P\} \vdash Q$ then , $\vdash Q$" is derived from a very basic set of rules.

There are many systems for machine aided verification. Systems used for program verification are of two kinds: *model checkers* and (interactive/automated) *theorem provers.*

Model checkers [CES86, Hol91, McM93, Kur89] establish properties by checking for their validity. By nature, model checking can be done only for programs that generate a finite state space. They are extremely fast for programs with state spaces of reasonable size, and have been used to verify complex systems, e.g., see [MS91]. Another versatile tool is the Concurrency Workbench of Cleaveland, Parrow, and Steffen [CPS93], an automated tool for analyzing finite-state distributed programs.

Theorem provers establish properties by proofs. Examples of interactive provers are HOL [GM93], COQ [DF+93], the system presented in [Fel93], all of which are designed along the lines of LCF [GMW79]. Proofs may be generated by invoking (built-in or user-defined) *tactics* — procedures to repeatedly apply some theorems or other tactics. Application of tactics entails executing code. In general, to use them requires intimate knowledge of a system's built-in tactics, and familiarity with the system's underlying programming language. Most conventional systems have been developed to carry out *backward* proofs, and employ higher-order logics in which users may encode arbitrary logics.

A key difference between such systems and SNAP is that in the former, users specify the rule to be applied (with the system responding with a set of sub-goals), whereas in SNAP users directly specify the proof steps, with the system searching for a justification for every step. In addition, using SNAP does not entail writing code in any programming language or familiarizing oneself with

existing system code. Another difference is the order in which proofs are carried out (forward vs. backward).

Many theorem provers have been designed to find proofs automatically (see [SuB89] for an overview). Since in most logics the question "is formula P provable" is undecidable, user guidance is often needed. However, to do so requires intimate knowledge of the underlying search algorithms, expertise which is usually not widely available. In addition, we believe that a completely automated proof of a formula does not offer any insight into its validity. However, we have included automation in SNAP to support users who desire automated proofs.

Since SNAP is to be used only in program verification, we were able to implement a very lightweight system (using less than 3000 lines of source statements). For reasons of portability, SNAP has been implemented in the C programming language [KR88] in the UNIX-environment. Our implementation is efficient and frugal in the use of resources: a version runs on a PC (386SX with 1 Megabyte RAM), with acceptable system response time even for interactive use.

The rest of this paper is organized as follows: In the next section we discuss SNAP's user interface, illustrated by examples. In Section 3, we validate the proof of a safety property of Dekker's algorithm using the system. In Section 4, we discuss SNAP's underlying proof annotation algorithm. Finally, Section 5 contains some conclusions and a preview of our future work.

2 User Interface

In this section we discuss the user interface of SNAP, illustrated by small examples. We also outline how SNAP supports program proofs.

In SNAP, we denote the negation of a formula P by $\tilde{}P$. Other logical operators are: "`\/`" (disjunction); "`/\`" (conjunction); "`=>`" (implication); and "`<=>`" (equivalence). Proposition *false* is denoted by "`ff`". For demonstration, we shall assume that the set of proof rules listed in Figure 1 are in SNAP's rule-base. In Figure 1, $P_1, \cdots, P_n \mid\!- \; P$ is to be interpreted as $\{P_1, \cdots, P_n\} \vdash P$, i.e., P is derivable from $P_1, \cdots, P_n$.

The monotonicity property "if Γ, $\subseteq$ Γ' and $\Gamma \vdash P$ hold, then $\Gamma' \vdash P$ holds" is built into the system. Users interact with SNAP by means of commands, each of which must be terminated by the reserved symbol "`;`". The system records the status of each proof as follows: set S records formulae assumed during the proof, and sequence L records the proof steps.

Initiating a Proof: A proof is initiated by keyword `prove`, optionally followed by *premises*, followed by *goal* G. Premises are enclosed within `if` ... `then`, and separated by symbol `and`. Each premise and goal G are of the form $P_1, \cdots, P_n \vdash P$.

```
DeductionThm:   if  P |- Q  then |- P => Q;
contradiction:  if ~P |- ff then |- P;
falsum:       ff |- P;
DirectConseq:  P |- P;
NegDef:    ~P        |- (P => ff);
NegDef:    (P => ff) |- ~P;
ModusPonens: P, P=>Q  |-  Q;
conjElim:   P /\ Q  |-  P;
conjElim:   P /\ Q  |-  Q;
conjIntro:  P,   Q  |-  P /\ Q;
disjDefn:   P \/ Q  |-  ~P => Q;
condDefn:   ~P => Q  |-  P \/ Q;
bicondDefn: (P <=> Q)             |- (P  => Q) /\ (Q => P);
bicondDefn: (P  => Q) /\ (Q => P) |- (P <=> Q);
```

Figure 1: *Basic Set of Proof Rules*

Introducing Assumptions: Command `assume` *Prop*, where *Prop* is of the form $P_1, \cdots, P_n \vdash P$, appends *Prop* to sequence L. Command `assume` P, where P is a formula not containing symbol "$\vdash$", adds P to set S and appends $P \vdash P$ to sequence L.

Entering Proof Steps: Proof steps are provided by directly entering formulae (without any keyword). If a step is justifiable on the basis of a theorem or meta rule in SNAP's rule-base, the system generates a justification and appends the step to sequence L. Entering $P_1, \cdots, P_n \vdash P$ as a proof step is interpreted as "P is derivable under assumptions $\{P_1, \cdots, P_n\}$ in one step". Additionally, SNAP uses the convention that to enter formula $S \vdash P$, one merely enters P.

A proof is deemed complete if the last step in L is identical to the goal of the current proof; in addition, the assumptions under which this step is valid must be contained in the premises of the proof.

Saving Theorems: When proved, the user may save a theorem (without its proof) in the rule-base, by entering command `qed` followed by the name to be assigned to the corresponding theorem in the rule-base.

To demonstrate the use of SNAP, let us first examine how a proof of $\neg\neg P \vdash P$ is given using pencil and paper:

> Assume that $\neg\neg P$ holds. Then, $\neg P \Rightarrow ff$ holds, too (by proof rule `NegDef`). Let us assume that $\neg P$ also holds. We may now derive ff (by proof rule `ModusPonens`). Thus, $\neg\neg P \vdash P$ follows (by proof rule `contradiction`). This concludes the proof.

To carry out this proof in SNAP, one enters the commands below. Note the direct correspondence between the proof on paper and the proof provided to SNAP.

```
prove ~~P |- P;
assume ~~P;
~P=>ff;
assume ~P;
ff;
~~P|-P;
qed Double_Neg_Elim;
```

The (un-edited) output produced by the system is give below. Here, "FOL" refers to the name of the module which contains the basic set of rules.

```
Double_Neg_Elim: (~(~P))|- P
Proof:

  1: (~(~P))
                                                              (Assumption)
  2: ((~P) => ff)
                                                    (FOL.NegDef, 1)
  3: (~P)
                                                              (Assumption)
  4: ff
                                                  (FOL.ModusPonens, 3, 2)
  5: (~(~P))|- P
                                                   (FOL.contradiction, 4)
QED
```

As another example, we present a proof of the cut-rule:

```
prove if |-P and P|-Q then |-Q;
assume |-P;
assume P|-Q;
|-P=>Q;
|-Q;
qed Cut_Rule;
```

The (un-edited) output produced by the system is:

```
Cut_Rule: if |- P  and  P|- Q then |- Q
Proof:

  1: |- P
                                                              (Assumption)
  2: P|- Q
                                                              (Assumption)
  3: (P => Q)
                                                   (FOL.DeductionThm, 2)
  4: Q
                                                 (FOL.ModusPonens, 1, 3)
QED
```

In addition to the commands discussed above, SNAP also offers commands to view proof steps (with associated justifications), inference rules, assumptions, and the proof goal.

In the remainder of this section we discuss how SNAP can be applied to correctness proofs of programs.

Programs are represented as a (finite) set of transitions (as in UNITY [CM88]). Each transition consists of a boolean expression (guard) and a (finite) sequence of assignments (body). A transition with guard g and body B is denoted as $g \longrightarrow B$. Assignments in body B are separated by commas. If g is always true, the transition may be written as B. As in UNITY, the transition's body is executed as a multiple assignment. For example, transition "x:=y, y:=x" interchanges the values of the variables x and y.

The problem of establishing an LTL-property for a program can be reduced to the problem of proving a finite set of first-order formulae [MP91a]. Some of these formulae are first-order predicates; others refer to individual transitions of the program under consideration. The latter are of the form $\{p\}\tau\{q\}$, where p and q are predicates and τ is a transition. The interpretation of $\{p\}\tau\{q\}$ is that whenever the guard of τ is true in a state in which p holds, then q holds after the execution of the body of τ.

Now, each such formula can be translated into a first-order implication using Dijkstra's notion of *weakest precondition* [Dij76]. Let τ denote $g \longrightarrow B$. Denoting by $wp(B, q)$ the weakest precondition of body B w.r.t. postcondition q, $\{p\}\tau\{q\}$ is equivalent to $(p \wedge g) \Rightarrow wp(B, q)$. Note that the weakest precondition of sequence $x_1 := e_1, \cdots, x_n := e_n$ of assignments w.r.t. postcondition q is $q[e_1/x_1, \cdots, e_n/x_n]$, where $q[e_1/x_1, \cdots, e_n/x_n]$ denotes the simultaneous substitution of e_i for x_i in q $(i = 1, \cdots, n)$. When one inputs a formula of the form $\{p\}\tau\{q\}$ in SNAP, it is automatically transformed into $(p \wedge g) \Rightarrow wp(B, q)$, where τ, g, and B are as above. Consequently, proofs as in the examples above can used to reason about transitions.

Consider the following example:

```
prove |-{x=X/\y=Y} x:=y, y:=x {x=Y/\y=X};
assume x=X/\y=Y;
x=X;
y=Y;
y=Y/\x=X;
|- (x=X/\y=Y) => (y=Y/\x=X);
|- {x=X/\y=Y} x:=y, y:=x {x=Y/\y=X};
qed swap;
```

The (un-edited) output produced by SNAP is:

```
swap: |- (((x = X) /\ (y = Y)) => ((y = Y) /\ (x = X)))
Proof:

  1: ((x = X) /\ (y = Y))
                                                              (Assumption)
  2: (x = X)
                                                          (FOL.conjElim, 1)
  3: (y = Y)
                                                          (FOL.conjElim, 1)
  4: ((y = Y) /\ (x = X))
                                                     (FOL.conjIntro, 3, 2)
  5: |- (((x = X) /\ (y = Y)) => ((y = Y) /\ (x = X)))
                                                     (FOL.DeductionThm, 4)
QED
```

3 Application to Dekker's Algorithm

In this section we show how SNAP may be used to validate correctness proofs of programs. For this purpose, we consider Dekker's algorithm, a concurrent program that ensures mutual exclusion for two processes. We shall concentrate on proving a safety property which asserts that the two processes cannot be in their critical section at the same time. The proof is from [MP83].

An abstract version of Dekker's algorithm is presented in Figure 2. It has been taken from [BA82] and is an abstraction in the sense that the code in the bodies of the critical sections has been removed. For this program, we wish to establish the LTL formula $\Box\neg(pc1 = L5 \wedge pc2 = M5)$, where $pc1$ represents the first process's program counter, $pc2$ represents the second process's program counter, and $L5$, $M5$ are locations in the program. Thus, the locations $L5$ and $M5$ play the roles of critical sections.

The program has three shared variables: $c1$, $c2$, and $turn$. Initially, each of these variables has value 1. If the first process wishes to enter its critical section, it indicates this by assigning 0 to

variable $c1$. Variable $c2$ is used for the same purpose by the second process. Variable *turn* is used to resolve the conflict that arises when both processes have indicated their intention to enter their critical sections.

Code executed by process $P1$

$L0$: $c1{:=}0$;
$L1$: **if** $c2 \neq 0$ **then goto** $L5$ **fi**;
$L2$: **if** $turn{=}1$ **then goto** $L1$ **else goto** $L3$ **fi**;
$L3$: $c1{:=}1$;
$L4$: **if** $turn{=}2$ **then goto** $L4$ **else goto** $L0$ **fi**;
$L5$: $turn := 2$; $c1 := 1$; **goto** $L0$;

Code executed by process $P2$

$M0$: $c2{:=}0$;
$M1$: **if** $c1 \neq 0$ **then goto** $M5$ **fi**;
$M2$: **if** $turn{=}2$ **then goto** $M1$ **else goto** $M3$ **fi**;
$M3$: $c2{:=}1$;
$M4$: **if** $turn{=}1$ **then goto** $M4$ **else goto** $M0$ **fi**;
$M5$: $turn := 1$; $c2 := 1$; **goto** $M0$;

Figure 2: *Dekker's Algorithm*

Recall that the safety property we wish to prove for this program is $\Box\neg(pc1 = L5 \wedge pc2 = M5)$. In order to do so, we apply rule S_Inv of Manna and Pnueli. This rule states that one can establish $\Box P$ for a program and a state-formula P by showing that there exists a state-formula I stronger than P such that I holds initially, and I is preserved under every atomic action of the program. The technical formulation of this rule is given in Figure 3. There $\vdash$ denotes provability in first-order predicate logic, and $Prog \Vdash\!\!- \Box P$ denotes that $\Box P$ is provable for program *Prog* within Manna and Pnueli's system.

$$\frac{\begin{array}{l}\vdash Init \Rightarrow I \\ \vdash \{I\}\ \tau\ \{I\}, \text{ for every atomic transition } \tau \text{ of program } Prog \\ \vdash I \Rightarrow P\end{array}}{Prog \Vdash\!\!- \Box P}$$

Figure 3: *Manna and Pnueli's S_Inv-rule*

In order to apply the S_Inv-rule to Dekker's algorithm, we have to specify the program in Figure 2 in terms of atomic actions. This is done in Figure 4. (A transformation to obtain these actions has been described in [S94a].) Initially, $pc1{=}L0$, $pc2{=}M0$, and $c1{=}c2{=}turn{=}1$ hold.

τ_1: $pc1{=}L0 \longrightarrow c1 := 0,\ pc1 := L1$
τ_2: $pc1{=}L1 \wedge c2 = 0 \longrightarrow pc1 := L2$
τ_3: $pc1{=}L1 \wedge c2 \neq 0 \longrightarrow pc1 := L5$
τ_4: $pc1{=}L2 \wedge turn = 1 \longrightarrow pc1 := L1$
τ_5: $pc1{=}L2 \wedge turn \neq 1 \longrightarrow pc1 := L3$
τ_6: $pc1{=}L3 \longrightarrow c1 := 1,\ pc1 := L4$
τ_7: $pc1{=}L4 \wedge turn = 2 \longrightarrow pc1 := L4$
τ_8: $pc1{=}L4 \wedge turn \neq 2 \longrightarrow pc1 := L0$
τ_9: $pc1{=}L5 \longrightarrow turn := 2,\ c1 := 0,\ pc1 := L0$

τ_{10}: $pc2{=}M0 \longrightarrow c2 := 0,\ pc2 := M1$
τ_{12}: $pc2{=}M1 \wedge c1 = 0 \longrightarrow pc2 := M2$
τ_{13}: $pc2{=}M1 \wedge c1 \neq 0 \longrightarrow pc2 := M5$
τ_{14}: $pc2{=}M2 \wedge turn = 2 \longrightarrow pc2 := M1$
τ_{15}: $pc2{=}M2 \wedge turn \neq 2 \longrightarrow pc2 := M3$
τ_{16}: $pc2{=}M3 \longrightarrow c2 := 1,\ pc2 := M4$
τ_{17}: $pc2{=}M4 \wedge turn = 1 \longrightarrow pc2 := M4$
τ_{18}: $pc2{=}M4 \wedge turn \neq 1 \longrightarrow pc2 := M0$
τ_{19}: $pc2{=}M5 \longrightarrow turn := 1,\ c2 := 1,\ pc2 := M0$

Figure 4: *The Atomic Transitions in Dekker's Algorithm. Transitions* $\tau_1, \cdots, \tau_9$ *are executed by the first process; transitions* $\tau_{10}, \cdots, \tau_{19}$ *are executed by the second process.*

To apply rule S_Inv with $Init \equiv pc1 = L0 \wedge pc2 = M0 \wedge c1 = 1 \wedge c2 = 1 \wedge turn = 1$, and $P \equiv \neg(pc1 = L5 \wedge pc2 = M5)$, we define I as the conjunction of:

(a) $turn = 1 \vee turn = 2$, and

(b) $c1 = 0 \vee c1 = 1$, and

(c) $c2 = 0 \vee c2 = 1$, and

(d) $(pc1 = L1 \vee pc1 = L2 \vee pc1 = L3 \vee pc1 = L5) \Rightarrow c1 = 0$, and

(e) $(pc2 = M1 \vee pc2 = M2 \vee pc2 = M3 \vee pc2 = M5) \Rightarrow c2 = 0$, and

(f) $(pc1 = L0 \vee pc1 = L4) \Rightarrow c1 = 1$, and

(g) $(pc2 = M0 \vee pc2 = M4) \Rightarrow c2 = 1$, and

(h) $pc1 = L5 \Rightarrow (pc2 = M1 \vee pc2 = M2 \vee c2 = 1 \vee turn = 1)$, and

(i) $pc2 = M5 \Rightarrow (pc1 = L1 \vee pc1 = L2 \vee c1 = 1 \vee turn = 2)$.

Predicate I expresses that variable *turn* can take only 1 or 2 as its value (clause (a)); that the variables $c1, c2$ take the values 1 or 2 (clauses (b) and (c)); that $c1 = 0$ if the program counter of

the first process is at location $L1$, $L2$, $L3$, or $L5$ (clause (d)); that $c1 = 1$ if the program counter of the first process is at location $L0$ or $L4$ (clause (f)); and that the following is true: if the program counter of the first process is at location $L5$, then the program counter of the second process is at location $M1$ or $M2$, or the value of variable $c2$ is 1, or the value of variable *turn* is 1 (clause (h)). The intuitive explanation of the clauses (e), (g), and (i) is along the same lines.

For this choice of I, we applied rule S_Inv. The proof was carried out under the assumption that all locations in the program are distinct, and has been validated and annotated by SNAP. The proof consists of 20 lemmata: 18 to show that the invariant is preserved under all transitions, one to show that the invariant holds initially, and one to show that the invariant implies the formula expressing mutual exclusion.

4 The Proof Annotation Algorithm

In this section we describe the algorithm responsible for the generation of justifications of proof steps.

Our proof annotation algorithm relies on a function we call `match` (also known as one-way unification), described in [Sie90]. Let a substitution be a sequence of tuples $\langle x, e\rangle$, where x is a variable and e an expression. Variable x can stand either for an arbitrary predicate or term. Expression e is either a predicate or a term. (The actual types of x and e are inferred from their contexts and determined by the parser.) Given two expressions e_1, e_2 and a substitution σ, $\texttt{match}(e_1, e_2, \sigma)$ returns *true* if there exists a substitution σ' extending[1] σ such that $\sigma'(e_1) = e_2$ (cf. [Sie90]). (Here $\sigma'(e_1)$ denotes the expression obtained from e_1 by simultaneous replacement of every variable x by e when $\langle x, e\rangle$ is in σ'.) If this is the case, then, in addition, variable σ is assigned such a value σ'. Otherwise, i.e., when no such substitution σ' exists, `match` returns *false* and leaves σ unchanged.

Let *Expn* denote the type of expressions. Expressions will be denoted by e, possibly subscripted or primed. We also define a type *Gamma*, whose domain is interpreted as a (finite) set of expressions. For convenience, in the discussion below we represent sets as lists. From now on, $,_1, \cdots, ,_n$ denote variables of type *Gamma*. A proof rule R is a non-empty (finite) sequence of elements of type $\mathit{Gamma} \times \mathit{Expn}$, represented as a list. Sequence $\langle ,_1, e_1\rangle, \cdots, \langle ,_n, e_n\rangle$ has the following interpretation: $,_1 \vdash e_1$ is derivable under the assumptions $,_2 \vdash e_2, \cdots, ,_n \vdash e_n$. In case $n = 1$, it means that $,_1 \vdash e_1$ is derivable under no assumptions. The above sequence has an associated identifier, its *name*. We call $,_1 \vdash e_1$ the conclusion of the proof rule, and denote this by $\mathit{concl}(R)$. We call $,_2 \vdash e_2, \cdots, ,_n \vdash e_n$ the premises of the proof rule. For a pair $P = \langle , , e\rangle$, e is denoted by $\mathit{expn}(P)$ and $,$ by $\mathit{hyp}(P)$.

Recall that every proof is represented as a (finite) sequence L, as described in Section 2, that corresponds to the proof steps so far. For the description of the annotation algorithm, we assume that the user has input a proof step of the form $, \vdash e$. This is converted by the system into a *step*

[1] The extension of a substitution –a sequence– has its usual meaning.

$\langle , , e\rangle$. In the following, we only consider checking for applicability of a single rule R. (It is a trivial extension to check for a set of rules.)

For a rule R, identified by $, _1 \vdash e_1, \cdots, , _n \vdash e_n$, the annotation algorithm checks whether e_1 can be matched with e under substitution σ. Initially, σ is the empty substitution ϵ. An attempt is then made to match each element of $, _1$ with the expression part of a step in the proof. For every such match found, at say position i in sequence L, the algorithm proceeds to check that the hypothesis of that step is a subset of $,$ (modulo substitution). If this is not the case, the algorithm checks whether every element e' of $, _1$ can be directly matched with an element of $, .$ (Recall that we assume the monotonicity property: if $, \subseteq , '$ and $, \vdash P$ hold, then $, ' \vdash P$ holds.) If rule R also has premises, the algorithm checks that all of them can be matched in a similar way as above with steps in sequence L.

If the above procedure is successful, the algorithm returns the name of the matched rule. In addition, if all the premises of rule R have been justified on the basis of steps in sequence L, the algorithm also returns a (finite) sequence of natural numbers indicating those steps in L.

If the above process is unsuccessful in finding a justification, the algorithm reports failure.

The following is a description in pseudo-code of the algorithm outlined above. As discussed above, a sequence of natural numbers may be provided as part of the justification. In order to concentrate on the essentials of the algorithm, we have not shown how such sequences are generated in the code. Also, the function merely checks whether a given rule applies, and does not return the rule's name if it applies. Below, *Subst* denotes the type of substitutions. For list ℓ, $\mid \ell \mid$ denotes its length, and $\ell[i]$ denotes its i^{th} element. Operations on sets are denoted as usual; the notation $, -e$ denotes $, -\{e\}$. The annotation function is called `is_justification`. We also define four auxiliary functions. (Observe how these functions rely heavily on backtracking.)

Function `extra_check`$(, _1, , _2, , , \sigma)$ returns *true* iff the elements e_2 of $, _2$ for which no element of $, _1$ matches e_2 under substitution σ constitute a subset of $, .$

integer function `extra_check`$(, _1, , _2, , , \sigma)$:
    Gamma $, '_2 :=, _2$;
    for $i := 1$ **to** $\mid , _2 \mid$
    do $j := 1$;
        while $j \leq \mid , _1 \mid \wedge \neg$`match`$(, _1[j], , _2[i], \sigma)$;
        do $j := j + 1$ **od**;
        if $j \leq \mid , _1 \mid$ **then** $, '_2 := , '_2 - , _2[i]$ **fi**
    od;
    return $(, '_2 \subseteq ,)$

Function `match_steps`$(, _1, , _2, L, N, \sigma)$ checks, for each of the first N elements e of $, _1$, whether there exists a step *step* in L such that the gamma part of *step* is a subset of $, _2$ and e matches the expression part of *step* under substitution σ.

integer function `match_steps`$(, _1, , _2, L, N, \sigma)$:
    Subst σ';
    if $N = 0$ **then return** *true* **fi**;
    for $i := 1$ **to** $\mid L \mid$
    do $\sigma' := \sigma$;
        if $hyp(L[i]) \subseteq , _2$ **and** `match`$(, _1[N], expn(L[i]), \sigma)$
        then if `match_steps`$(, _1, , _2, L, N-1, \sigma)$
                then return *true*
                fi
        fi;
        $\sigma := \sigma'$
    od;
    return *false*

Function `match_gamma`$(, _1, , _2, N, \sigma)$ checks whether each of the first N elements of $, _1$ matches at least one element of $, _2$ under substitution σ.

integer function `match_gamma`$(, _1, , _2, N, \sigma)$:
    Subst σ';
    if $N = 0$ **then return** *true* **fi**;
    for $i := 1$ **to** $\mid , _2 \mid$
    do $\sigma' := \sigma$;
        if `match`$(, _1[N], , _2[i], \sigma)$
        then if `match_gamma`$(, _1, , _2, N-1, \sigma)$
                then return *true*
                fi
        fi;
        $\sigma := \sigma'$
    od;
    return *false*

Function `match_premises`$(R, M, , , step, \sigma)$ checks whether, for every element $, ', e'$ in sequence $R[M], \cdots, R[\mid R \mid]$, e' matches the expression part of *step* and whether `extra_check`$(, ', hyp(step), , , \sigma)$ holds.

```
integer function match_premises(R, M, , , step, σ)
    subst σ′:=ϵ;
    if match(expn(R[M]), expn(step), σ) and
        extra_check(hyp(R[M]), hyp(step), , , σ)
    then if M <| R |
            then for i := 1 to | L |
                    do σ′ := σ;
                        if match_premises(R, M + 1, , , step, σ)
                        then return true
                        fi;
                    σ := σ′
                    od;
                    return false
            else return true
    else return false
```

Finally, we present the main function `is_justification`$(R, , , e)$, whose description has already been given.

```
integer function is_justification(R, , , e):
    subst σ, σ′:=ϵ;
    if match(expn(R[1]), e, σ) and
        (match_steps(hyp(R[1]), , , L, | hyp(R[1]) |, σ) or
        match_gamma(hyp(R[1]), , , | hyp(R[1]) |, σ))
    then if | R |> 1
            then for i := 1 to | L |
                    do σ′ := σ;
                        if match_premises(R, 2, , , L[i], σ)
                        then return true
                        fi;
                    σ := σ′
                    od;
                    return false
            else return true
    else return false
```

5 Conclusion and Future Work

We have presented system SNAP which may be used for validating proofs in Linear Time Temporal Logic (LTL). The primary motivation to build the system was our desire for mechanically validating correctness proofs of distributed programs, using a representation close to handwritten proofs. In

the design of our system, we have sacrificed generality for efficiency and ease of use. Our system is also applicable to problems in other areas of Software Engineering. For example, the first author is currently investigating its use in validating proofs of theorems discussed in [Par93a]. The underlying logic, presented in [Par93b], is used for reasoning about specifications of safety-critical systems.

Another motivation was to support *forward* proofs. We also wanted to eliminate the need for users having to remember names of proof rules to carry out proofs. In contrast to other systems, users of SNAP directly input proof steps instead of specifying rules to be applied. Our system can be used interactively or in batch mode, and offers automation as an option. SNAP generates proof annotations, enhancing the readability of proofs. The system can be used without having to learn system-specific proof paradigms or a new programming language.

The use of SNAP in correctness proofs of programs has been demonstrated by establishing a safety property of Dekker's algorithm, as sketched in this paper. To use our system to validate correctness proofs of larger distributed programs, a lot remains to be done. For instance, we have to include quantifiers and LTL proof rules. Also to be included are finite sequences, since sequences are often needed in correctness proofs that reason about a program's communication behavior. Further, we need a representation for programs in order to directly encode LTL proof rules that refer to programs.

We realize that model checking algorithms might have done better, in terms of time, for establishing the safety property for Dekker's algorithm. This is so, because the size of the state space is small for this problem. One should realize, however, that the computational overhead of proof annotation *is independent of the size of the program's state space.* The time to validate a proof is only dictated by the size of the proof, and the length of formulae that occur in it. For example, our system has validated a correctness proof for the algorithm to compute the greatest common divisor (gcd) of two strictly positive integers. This proof consists of eight lemmata, four of which deal with a safety property and four with a liveness property. Our system performed the annotation in under a second. Note that the gcd algorithm generates an infinite state space.

In our future work we plan to include support for backward proofs, to allow a mixture of forward and backward proof styles. We also plan to extend the user interface to allow direct application of specific proof rules.

Acknowledgements

We thank Gerard Holzmann for discussions during the initial phases of this project. Thanks are also due to Dennis Ritchie for hosting the first author in his group, during which this project was executed. The first author also thanks the Telecommunications Research Institute of Ontario for support. We also thank Amy Felty, Tim Griffin, and David Parnas for their constructive comments.

References

[BA82] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice-Hall International Inc., 1982.

[Bha94] R. Bharadwaj. *TOP/Snap – A Transition Oriented Prover for PROMELA.* Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley, 1988.

[CES86] 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.

[CPS93] R. Cleaveland, J. Parrow and B. Steffen. *"The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems".* ACM Transactions on Programming Languages and Systems 15(1):36–72, 1993.

[DF+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, B. Werner". *The Coq Proof Assistant User's Guide.* Technical Report 154, INRIA, 1993.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976

[Fel93] A. Felty. *"Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language".* Journal of Automated Reasoning, 11(1):43–81, 1993.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[GMW79] M. J. Gordon, A. J. Milner and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Lecture Notes in Computer Science 78, Springer-Verlag, 1979.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall Software Series, 1991.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, N. J. 2^{nd} ed. 1988.

[Kur89] R. P. Kurshan. *"Analysis of Discrete Event Coordination".* Lecture Notes in Computer Science 430, pp. 414–453.

[MP83] Z. Manna and A. Pnueli. *"Verification of Concurrent Programs: A Temporal Proof System".* Foundations of Computer Science IV, Distributed Systems: Part 2, Mathematical Centre Tracts 159, Center for Mathematics and Computer Science, Amsterdam, 163–255, 1983.

[MP91a] Z. Manna and A. Pnueli. *"Completing the Temporal Picture"*. Theoretical Computer Science, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[MS91] K. L. McMillan and J. Schwalbe. *"Formal Verification of the Encore Gigamax Cache Consistency Protocol"*. International Symposium on Shared Memory Multiprocessors, 1991.

[Par93a] D. L. Parnas. *"Some Theorems We Should Prove"*. Proc. Int'l Meeting on Higher Order Logic, Theorem Proving and its Applications, Vancouver, BC, pp. 156 – 163, 1993.

[Par93b] D. L. Parnas. *"Predicate Logic for Software Engineering"*. IEEE Transactions on Software Engineering, 19(9):856–862, Sep 1993.

[Sie90] J. H. Siekmann. *"An Introduction to Unification Theory"*. Formal Techniques in Artificial Intelligence, R. B. Banerji, Editor, Elsevier Science Publishers BV (North-Holland, 1990.

[S94a] F. A. Stomp and W.-P. de Roever. *Principles for Sequential Reasoning about Distributed Algorithms.* To appear at the ftp-site of Formal Aspects of Computing (1994).

[SuB89] P. A. Subrahmanyam, Graham Birtwistle, Editors. *Current Trends in Hardware Verification and Automated Theorem Proving* Springer Verlag, 1989.