A SIMPLE PROOF CHECKER FOR REAL-TIME SYSTEMS

By

Catherine Leung

B. Sc. (Computer Science) University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

....

....

THE UNIVERSITY OF BRITISH COLUMBIA

June 1995

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date:

Abstract

This thesis presents a practical approach to verifying real-time properties of VLSI designs. A simple proof checker with built-in decision procedures for linear programming and predicate calculus offers a pragmatic approach to verifying real-time systems in return for a slight loss of formal rigor when compared with traditional theorem provers. In this approach, an abstract data type represents the hypotheses, claim, and pending proof obligations at each step. A complete proof is a program that generates a proof state with the derived claim and no pending obligations. The user provides replacements for obligations and relies on the proof checker to validate the soundness of each operation. This design decision distinguishes the proof checker from traditional theorem provers, and enhances the view of "proofs as programs". This approach makes proofs robust to incremental changes, and there are few "surprises" when applying rewrite rules or decision procedures to proof obligations. A hand-written proof constructed to verify the timing correctness of a high bandwidth communication protocol was verified using this checker.

Table of Contents

Abstract ii

List of Tables viii

List of Figures ix

Acknowledgement x

1 Introduction 1

1.1 Verifying Timing Properties with the Proof Checker 2

1.2 Theorem Provers 4

1.3 Verification Tools and Real-time Properties 6

1.4 Thesis Overview 7

2 Proof Checker Specification 8

2.1 Structure of the Proof Checker 8

2.1.1 Proof State 9

2.1.2 Proof Rules 11

2.2 The Proof Rules and their Soundness 12

2.2.1 Linear Programming Rule 13

2.2.2 Predicate Calculus Rule 14

2.2.3 Instantiation Rule 15

2.2.4 Skolemization Rule 15

2.2.5 Induction Rule 18

2.2.6 Definition Rule . . . 19
2.2.7 Postponement Rules . . . 20
2.2.8 Equality Rule . . . 23
2.2.9 If Rule . . . 23
2.2.10 Discrete Rule . . . 24
2.3 Conclusion . . . 24

3 Implementation of the Proof Checker **25**
3.1 Abstract Data Type for Proof State . . . 25
3.2 The Proof Rules and some Implementation Techniques . . . 26
3.2.1 Defining the Concrete Types . . . 27
3.2.2 Pattern Matching . . . 28
3.2.3 Failures . . . 29
3.3 Linear Programming . . . 29
3.3.1 Simplex Method . . . 30
3.3.2 Strict Inequalities ($>$ and $<$) . . . 35
3.3.3 Not-equal-to Relations ($\neq$) . . . 36
3.3.4 Special Cases . . . 37
3.4 Implementation of Proof Rules . . . 39
3.4.1 Linear Programming Rule . . . 40
3.4.2 Predicate Calculus Rule . . . 41
3.4.3 Skolemization Rule . . . 42
3.4.4 Instantiation Rule . . . 43
3.4.5 Induction Rule . . . 44
3.4.6 Definition Rule . . . 45
3.4.7 Postponement Rules . . . 45

3.4.8 Equality Rule 47
3.4.9 If Rule 48
3.4.10 Discrete Rule 48
3.5 User Interface 49
3.5.1 Case Analysis over booleans 49
3.5.2 Case Analysis over integers 50
3.5.3 Discharged by Unchanged 50
3.5.4 Printing a State 52
3.5.5 Print Abbreviation 52
3.6 Conclusion 53

4 Verification of Real-time Properties **54**
4.1 Synchronized Transitions: a hardware description language 55
4.2 Safety Properties and Invariants 56
4.3 Expressing Real-time Properties 59
4.4 Summary 63

5 Verifying STARI **64**
5.1 STARI Interfaces 64
5.1.1 Self-timed FIFOs for STARI 66
5.1.2 A schedule for STARI 68
5.2 An ST Program for STARI 72
5.2.1 The invariant 74
5.3 The STARI Proof 77
5.3.1 A snapshot from the proof 78
5.3.2 Some Proof Techniques 82
5.3.3 Flaws from Manual Proof 84

5.4 Observations and Experiences 84
5.4.1 Verified Proof versus Manual Proof 84
5.4.2 FL as a meta-language 85
5.5 Evaluating the Proof and the Proof Checker 86

6 Conclusion 88

6.1 The Simple Approach to Proof Checking 88
6.2 Proofs as Programs 90
6.3 The Postponement Rules 91
6.4 Variable skew version of STARI proof 92
6.5 Summary 92

Bibliography 94

Appendices 96

A User Manual 97

A.1 Structure of Proof Checker 97
A.2 How to Start/Exit the System 98
A.3 Syntax Used in the Checker 99
A.4 Proof Rules 102
A.4.1 To Start/End a proof: (`Start_proof/Done`) 102
A.4.2 The Ten Proof Rules 104
A.4.3 Proof Debugging: debug mode 114
A.5 User Interface 115
A.5.1 Interface Functions 115
A.5.2 Auxiliary Functions 120
A.6 Example 122

B Proof Script for STARI **127**

B.1 Proof Script for the Transmitter Transition 127

B.2 Proof Script for the FIFO Transition 148

B.3 Proof Script for the Receiver Transition 171

B.4 Proof Script for the Protocol 193

List of Tables

3.1 Linear Programming Rule 40

3.2 Predicate Calculus Rule 41

3.3 Skolemization Rule 42

3.4 Instantiation Rule 43

3.5 Induction Rule 44

3.6 Definition Rule 45

3.7 Postponement Rules 46

3.8 Equality Rule 47

3.9 If Rule 48

3.10 Discrete Rule 49

3.11 Case Analysis over Booleans 50

3.12 Case Analysis over Integers 51

3.13 Discharged by Unchanged 52

List of Figures

2.1 An example of a proof tree. 9

3.2 A system of linear relations. 30

3.3 Pseudocode for Linear Programming. 38

4.4 A synchronous communication circuit. 55

5.5 STARI communication 65

5.6 A self-timed FIFO 67

5.7 Stage-to-stage transfer times 71

5.8 A Synchronized Transitions program for STARI 75

5.9 The invariant for STARI 76

5.10 A branch from the STARI proof tree. 79

6.11 Identity Properties and Cancellation Law of reals 89

A.12 Definition of `Boolean` type, `Integer` type, and `Real` type. 101

Acknowledgement

I would like to thank my supervisor, Mark Greenstreet, for his time and patience, and his support. What he has taught me is beyond the technical material from the M.Sc. program. This thesis would not be here without him. Thank you, Mark.

I am grateful to Scott Hazelhurst and Carl Seger for their assistance in using FL, and for many useful discussions and suggestions throughout the course of this research. My second reader, Norm Hutchinson, has provided valuable comments for this thesis. Special thanks to Scott for being a friend, and for being there during the good and bad times.

Thanks to Jeff Joyce and Nancy Day for taking the time to discuss HOL with me. Nancy took the time to generate a proof for the example in the User Manual using HOL as a comparison to my proof checker. Thank you, Sree Rajan, for the discussions on PVS and taking the time to verify the same proof in PVS.

I would like to thank Jack Snoeyink for his guidance in my early days as a graduate student. Many thanks to Helene Wong, Xiaomei Han, Mohammad Darwish, and all members of ISD lab for all the supports and encouragements.

Chapter 1

Introduction

Verification is an essential part of the design process. To ensure that a system is functionally correct, designers try to systematically capture requirements and show that they are satisfied. Formal methods can assist this process when the specification is amenable to mathematical formalization and practical techniques are available to carry out the proofs. In particular, this thesis examines the application of formal methods to the verification of real-time systems. Specifications of timing correctness can often be expressed using simple predicates that includes linear inequalities. These are readily expressed in precise and familiar mathematical notation. On the other hand, the proofs that these requirements are satisfied are often lengthy. This thesis presents a proof checker that can be used to ensure the soundness of such proofs.

The work presented in this thesis is motived by a manual proof constructed to verify the timing correctness of a high bandwidth communication protocol, STARI [16]. STARI (Self-Timed At Receiver's Input) is a signaling technique for interchip communication that combines synchronous and asynchronous design methods. Although STARI is interesting in its own right, the manual proof is more tedious than it is profound, and its length makes it untrustworthy. Hand-written proofs often contain implicit assumptions and unstated arguments. Both can lead to errors. Even stated arguments can be wrong. This motivates developing mechanized tools to verify such proofs. Examining the manual proof for STARI, it appears that only a few, simple proof techniques were employed which suggests that a simple proof checker could be written to certify such proofs. To test this hypothesis, such a proof checker was written.

A proof checker takes a proof as input, verifies each step of the proof and certifies the resulting proof. This thesis presents a proof checker designed to verify proofs of real-time properties. The remainder of the introduction includes a discussion on some techniques used to formulate proofs for verifying real-time properties and a survey on existing theorem provers. The chapter concludes with an overview of the thesis.

1.1 Verifying Timing Properties with the Proof Checker

Many existing theorem provers are either extremely tedious and/or require skilled users. The thesis presented here is that a simpler proof checker, with a minimal set of inference rules, is powerful enough to verify correctness proofs for real-time systems. This proof checker, unlike many other traditional theorem provers which embed profound mathematical theories, is more accessible to engineers who are more interested in the result of the verification than the proofs involved. The fact that there is a simple mapping between the structures of proofs constructed from the proof checker and those of the manual proofs simplifies proof construction. The proof checker is domain specific. It is implemented to verify real-time properties in VLSI design. A decision procedure for linear inequalities is incorporated into the system for this purpose.

A theorem prover takes a theorem statement as input, applies different inference rules, and outputs a proof. Often, the built-in inference rules correspond to fundamental axioms of mathematics, allowing the theorem prover to be used to develop a wide variety of theories. Automated application of these inference rules releases users from tedious reasoning, and allows them to focus on more high-level issues. Some theorem provers, which place emphasis on automation, have built-in heuristics to search through inference rules and decide which ones to apply for different scenarios. These theorem provers make multiple proof steps with minimal human interaction. Others, focusing on generality, require more human guidance.

From a survey of existing theorem provers, it was noted that unpredictable output from inference rules can be frustrating in proof development. The proof checker described here avoids this problem because the user provides the expected result of each step. The use of a functional meta-language as the user interface to the proof checker makes this approach practical: the user does not have to repeatedly type enormous expressions; instead, functions can be written in the metalanguage to compute intended results and other inputs to the checker. Inference rules are only used to verify if the suggested output is a valid replacement of the preceding formula. This allows the user to control the exact structure output from an inference rule. This design decision eases the construction and manipulation of expressions, allows the user to locate the problem when a proof breaks down, and enhances the process of proof debugging.

The proof checker contains functions which allow the user to define abbreviations for large expressions. The pretty-printer, when printing a formula, replaces large expressions with equivalent user-defined abbreviations. This avoids printing out large, incomprehensible expressions, and allows user to better understand the meaning of expressions instead of confusing them with uninformative details.

Proof scripts can be written in modules that can be instantiated and reused. Thus if similar reasoning is required in various places in the proof, only one piece of 'code' needs to be constructed and similar arguments can be expressed as instantiations of this single module definition. In addition to reducing the tedium of proof construction, this also allows the proof to be structured hierarchically.

When verifying timing properties of VLSI designs, the system is modeled as a Synchronized Transitions program, and invariants are used to establish safety properties. A continuous model of time is employed: times are represented as real numbers, not integers. Unlike discrete models of time, no time interval can be overlooked. Real-time constraints are enforced by adding real-valued auxiliary variables, which are used for bookkeeping in the verification and not represented by wires or voltage in the implementation. The same approach is presented in [11] where the auxiliary variables are called *timers*.

1.2 Theorem Provers

The popularity of proof checking and theorem proving tools has increased as formal methods have come to play an increasingly important role in hardware design and verification. Existing theorem provers are distinguished by the mathematical formalisms that they are based upon, the algorithms that are used to reason about these formulas, and the choice of batch-oriented versus interactive user interfaces.

HOL [7], the Higher Order Logic system, was developed at Cambridge University in the early 1980's. It is an LCF-based [6, 12] [1] theorem prover for formal specification and verification in higher-order logic. The entire system is based on the five fundamental Peano axioms and the abstraction axiom; users typically extend the system with built-in decision procedures to suit the application. There are no pre-determined application-specific concepts built into the system. For these reasons, the system is general and flexible. However, for the same reason, the system requires highly skilled users to guide the proof.

EHDM [22] and PVS [10, 22, 26, 27] were developed in SRI International at 1984 and 1991 respectively. EHDM uses a specification language based on typed higher order logic with a rich type system. The verification system includes a parser, pretty-printer,

[1] LCF (Logic for Computable Functions) is an interactive reasoning tool which uses abstract data types to protect the soundness of theorems manipulated by the inference rules. Proof tactics or strategies are communicated to the system through a metalanguage (In the HOL system, ML is used as the metalanguage).

type-checker, proof checker, and various documentation aids. The proof checker involved is not interactive; instead, it is guided by proof descriptions which are included as part of the specification text by the user. EHDM allows modularization of specifications which supports a form of hierarchical verification. PVS is an LCF-style theorem prover based on many of the concepts of EHDM. The PVS specification language has an even richer type system including dependent types and predicate sub-types. Decision procedures in PVS include arithmetic, equality, predicate calculus, and a simple form of temporal logic.

The Boyer-Moore Prover [4] is a batch-oriented, heuristic theorem prover. The Boyer-Moore theorem prover deals with a subset of quantifier free first-order logic and consists of an *ad hoc* collection of heuristic proof techniques. Decision procedures are embedded into the system to increase its efficiency and predictability. To prove a theorem, the system assumes the negation of this theorem; in a series of *simplifications*, this negation is broken into a set of supposedly simpler formulas. Recursively the *simplifier* tries to write the hypotheses to non-`F` (a predicate not logically equivalent to the constant `False`) by a form of backwards chaining. When the goal to be proven is not suitable for these techniques, this approach can spend large amounts of time failing to find a proof. This complicates the addition of new decision procedures [3]. Furthermore, a significant amount of tedious human effort can be required in the exploratory phase of proof development to find an initial decomposition of the theorem that is amenable to the prover's heuristics.

The Larch Prover [13], like the Boyer-Moore Prover, deals with a subset of first-order logic and is based on equational term-rewriting. It does not employ heuristics to derive subgoals automatically. The Larch Prover was originally used to debug a specification or a set of invariants, therefore its focus is aimed at locating where and when a proof breaks down. The theorem prover works efficiently with large sets of large equations, however, the inference rules can yield huge expressions as a result.

1.3 Verification Tools and Real-time Properties

Several proof techniques have been developed to model real-time systems and verify their timing properties using the theorem provers described in the previous sections. For example, the semantics of Duration Calculus has been encoded in the logic of PVS. Duration Calculus is an interval temporal logic for reasoning about real-time systems. This approach has been applied to a few small examples. For example, safety properties of a design of a leaking gas burner have been verified using this tool. [27, 26]

The Larch Prover has been used to verify safety properties of circuits using invariants. The system to be verified is modeled as a Synchronized Transitions program [29]. Synchronized Transitions is a guarded command language, which is also used in the approach presented in this thesis. (See section 4.3.) Protocols are used to capture essential properties of the transitions in the program. This approach can be extended to model real-time system as is explained in greater detail in Chapter 4.

UNITY is a guarded command language based on an interleaving model of concurrency. It has many features in common with Synchronized Transitions. In [8], it is shown how UNITY can be used to specify designs ranging from architecture independent programs to architecture specific ones. This language has been used to specify a real-time design which was then verified by the Boyer-Moore theorem prover. [14]

HOL-UNITY is an implementation of the logic for UNITY in the HOL theorem prover. UNITY programs and properties have been expressed in higher order logic in HOL [2]. In UNITY logic, there are two safety properties: **unless** and **invariant** and two progress properties: **ensures** and **leadsto**. A tactic for automating proofs of such properties was developed in HOL-UNITY. Although the proof of the progress properties of the lift-control program presented in [2] does not involve real-time properties, it might be possible to extend this approach to reason about real-time properties using methods like those present in [14]. However, this would require a practical theory of the reals constructed from the HOL axioms. Researchers have explored ways of implementing a decision procedure for elementary real algebra in HOL. In [18], the difficulties of constructing such a theory are described along with a solution. It explains how a theory rich enough to reason about polynomial inequalities can be implemented in HOL. With a theory of elementary real arithmetic, HOL could be used to reason about timing relations in real-time systems.

Time separation of events in concurrent systems can be determined by modeling the system as a cyclic connected graph. An initial graph is formulated with its nodes representing events and its arcs labeled with delay information. Tight upper and lower bounds for each event can be determined using an algorithm presented in [20]. This approach has been used to verify specific instances of STARI [19].

1.4 Thesis Overview

The remainder of this thesis explains the theory behind the proof checker and the verification technique, and presents an example of how the checker is applied to verifying STARI. Chapter 2 describes how a proof is structured, presents the ten proof rules and two decision procedures in the proof checker, and presents arguments for their soundness (and thus the soundness of the checker). Chapter 3 describes the implementation of the checker and shows that it implements the specification presented in Chapter 2. Chapter 4 discusses the approach employed to model real-time systems and verify their timing properties. The STARI example is presented in Chapter 5. Chapter 6 summarizes this investigation and suggests possible enhancements to the proof checker.

Chapter 2

Proof Checker Specification

A proof checker is a program that verifies the soundness of a proof. A proof is represented by a sequence of proof states that are manipulated by a small set of proof rules. The soundness of the checker depends only on the soundness of these rules. This chapter describes the structure of the proof checker and justifies the soundness of each proof rule.

2.1 Structure of the Proof Checker

The proof checker is implemented as an LCF style theorem prover [12, 6]: proof states are represented by an abstract data type, and these states are created and manipulated by a small set of rules. A functional metalanguage allows the user to define other proof methods using the fundamental proof rules of the checker. By protecting the proof state with an abstract data type, the soundness of a proof depends only on the soundness of the built-in rules and not on any machinery that the user may build on top of them.

The checker verifies backward proofs. A proof is viewed as a tree: the claim is the root; edges are labeled by proof rules; and the leaves represent simple tautologies. The conjunction of all the children of a node implies the node itself. A proof starts with the claim of the theorem as the one pending proof obligation to be discharged; proof rules are applied to reduce the claim into simple obligations that are decidable by the built-in procedures of the checker. Figure 2.1 shows an example of a proof tree. P is the claim to be proven, $Q \wedge R$ implies P by rule#1, and P is broken down into Q and R. By rule#2, Q is rewritten as S, and by rule #3 and #4, S and R are verified to be tautologies.

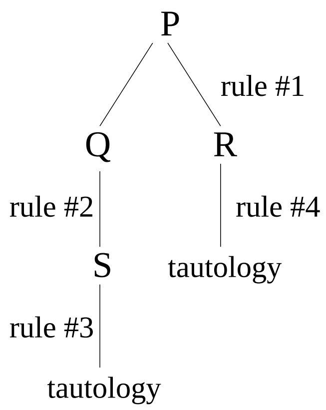


Figure 2.1: An example of a proof tree.

A proof script defines a traversal of the proof tree. Such traversals can be in an arbitrary order starting from the root, which allows the user to choose the order in which obligations are simplified and discharged. At each step of the proof, the pending proof obligations are maintained as a list. These obligations correspond to non-leaf nodes that have not yet been broken down into simpler obligations. Although the tree structure is not explicitly represented by the proof state, it could be reconstructed from the sequence of proof rules in the proof script.

2.1.1 Proof State

A proof state in the checker is composed of a claim, a hypothesis list, an obligation list, and a postponed list.

- The claim is the main goal or theorem to be proven. This field associates the theorem to be proven with its proof.
- The hypothesis list contains the hypotheses of the proof. These are stated at the beginning of the proof. No element can be added to or removed from this list once the proof is started.

- The obligation list is the list of pending proof obligations that must be discharged before the claim can be declared proven. Initially, this list contains exactly one element: the claim. The size of the list changes as obligations are broken down or discharged. The proof is complete when this list becomes empty.

- The postponed list contains all unverified assumptions made along the course of the proof. Initially, this list is empty. An obligation can be moved to or removed from this list with the `Postponement_rules` described in Section 2.2.7. Moving a proof obligation to the postponed list is the only way a proof obligation can be discharged without actually proving it. When a proof is completed, all obligations remaining on the the postponed list are printed, and it is the user's responsibility to verify them.

Each proof state represents an implication: $\forall v.(Hyp(v) \wedge Post(v) \Rightarrow Obl(v))$, where $Hyp(v)$ represents the list of hypotheses; $Post(v)$, the list of objects being postponed; $Obl(v)$, the list of obligations, and v is the set of variables over these three predicates. The pending obligations are implied by the hypotheses and the postponed objects. It states what remains in order to prove the theorem.

Initially, a proof state contains one obligation, the claim. The hypothesis list gives the context of the proof and defines the variables that appear in the proof. As mentioned above, the postponed list is initially empty. Therefore, an initial proof state can be viewed as the implication, $\forall v.(Hyp(v) \Rightarrow Claim(v))$. This is the theorem to be verified.

The proof is complete when all obligations are discharged and no postponed object remains on the postponed list. The last state of a proof gives the implicit implication of the form, $\forall v.(Hyp(v) \wedge \emptyset \Rightarrow \emptyset)$. An empty list is equivalent to the boolean value `True`; accordingly, the implication above is logically equivalent to `True`.

2.1.2 Proof Rules

There are two types of proof rules: discharge rules and replacement rules. Discharge rules verify that an obligation is a tautology and remove it from the obligation list. Replacement rules, after verifying that the replacement is sound, replace one or more pending obligations with one or more new obligations provided by the user. A replacement is sound if and only if the new proof state *implies* the old one. Replacement rules often substitute a set of old obligations with a new set, where the new set logically implies the old set, and leave the remaining elements of a proof state unchanged. The new obligations are not required to be equivalent to the old obligations. Thus, the proof checker is conservative, i.e., failure to verify a proof does not imply the negation of the theorem.

Using the notation introduced in the previous section, replacement rules can replace a set of old obligations with a set of new obligations only if the following holds:

$$(\forall v'.(Hyp(v') \wedge Post(v')) \Rightarrow Obl'(v')) \Rightarrow (\forall v.(Hyp(v) \wedge Post(v)) \Rightarrow Obl(v))$$

where $Hyp(v)$, $Post(v)$ and $Obl(v)$ are the list of old obligations, the list of old postponements and the list of old hypotheses over the variable set, v, and

$Hyp'(v')$, $Post'(v')$ and $Obl'(v')$ are the list of new obligations, the list of new postponements and the list of new hypotheses over the variable set, v'.

The set of variables v and v' can differ, since new variables (i.e. skolem constants) can be introduced by the `Skolem_rule`. (See Section 2.2.4).

We conclude that $(\forall v'.(Hyp(v') \wedge Post'(v')) \Rightarrow Obl'(v')) \Rightarrow (\forall v.(Hyp(v) \wedge Post(v)) \Rightarrow Obl(v))$ holds throughout the course of a proof. We can extend this to view the entire proof as a sequence of implications:

$$True$$

$$
\begin{aligned}
&\equiv \quad \forall v_n.(Hyp(v_n) \wedge Post_n(v_n) \Rightarrow True) \\
&\equiv \quad \forall v_n.((Hyp(v_n) \wedge Post_n(v_n)) \Rightarrow Obl_n(v_n)) \\
&\Rightarrow \quad \forall v_{n-1}.((Hyp(v_{n-1}) \wedge Post_{n-1}(v_{n-1})) \Rightarrow Obl_{n-1}(v_{n-1})) \\
&\Rightarrow \quad \ldots \\
&\Rightarrow \quad \forall v_1.((Hyp(v_1) \wedge Post(v_1)) \Rightarrow Obl(v_1)) \\
&\equiv \quad \forall v_1.(Hyp(v_1) \Rightarrow Claim(v_1)).
\end{aligned}
$$

Thus, a complete proof establishes $True \Rightarrow (\forall v_1.(Hyp(v_1) \Rightarrow Claim(v_1)))$, which is the original claim.

The user is required to provide the rewritten forms of pending obligations for replacement rules. This feature prevents surprises as to how obligations will be rewritten and provides robustness to proofs. Sometimes, the exact form of an obligation is critical to applying a proof rule. With this feature, the user always knows the exact form of each expression. As the system is enhanced, old proofs will not break because expressions will still be rewritten to the same form. This feature also facilitates proof debugging. After correcting an error, the user can re-execute previously verified parts of the proof script in "gullible mode" where proof steps replace obligations quickly without checking for soundness. Any proof states derived from proof rules executed in gullible mode are marked as untrustworthy. Thus, when the entire proof is debugged, it must be executed again with every step checked for the theorem to be certified by the checker.

2.2 The Proof Rules and their Soundness

This section presents the proof rules and gives justifications for each one. Appendix A.4 presents the syntax and usage of the proof rules.

2.2.1 Linear Programming Rule

Linear programming is built into the checker to provide a decision procedure for systems of linear inequalities. In a real-time system, timing constraints can be checked by this proof rule. Linear programming is also used to verify ranges in case analysis. Many arithmetic relations (or equalities) can be verified by linear programming as well.

Linear programming [23] is a continuous optimization technique, typically with an uncountable number of feasible points. A feasible point in a system of linear inequalities is a point which satisfies each of the inequalities. A linear program is infeasible if no such point exists. In this implementation, the coefficients for linear inequalities are expressed as rational numbers. The existence of a feasible point implies the existence of a rational feasible point in the same system; therefore operations over the set of rational numbers are sufficient to determine a system's feasibility.

Systems of linear inequalities are represented as sets of predicates. For example, $x \geq z$ can be deduced from $x \geq y$ and $y \geq z$. In this example, $x \geq y$, $y \geq z$, and $x \geq z$ are viewed as three boolean predicates. The `LP_rule` is used to reason about systems of the form $a \wedge b \wedge c \Rightarrow d$, where a, b, c, and d are linear inequalities. (The number of inequalities in a system is not fixed.) We note that if d is a linear inequality, so is $\neg d$. The conjunction of inequalities $a \wedge b \wedge c \wedge \neg d$ is a linear program which is feasible if and only if there is some assignment to the variables appearing in a, b, c, and $\neg d$, such that all four inequalities are satisfiable. Likewise, if the linear program is infeasible, then $\neg(a \wedge b \wedge c \wedge \neg d)$ is a tautology (i.e. it holds for all assignments of the variables appearing in the inequalities), and $(a \wedge b \wedge c) \Rightarrow d$ can be discharged. Discharging an obligation can be expressed in the notation we introduced earlier as follows. The proof state:

$$s = (\forall v.(Hyp(v) \wedge Post(v)) \Rightarrow Obl(v))$$

$$\text{where } Obl(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_i(v) \wedge \cdots \wedge o_n(v), \text{ and}$$

$$o_i(v) = \text{the clause to be discharged.}$$

can be rewritten as

$$s' = (\forall v.(Hyp(v) \wedge Post(v)) \Rightarrow Obl'(v))$$
$$\text{where } Obl'(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_{i-1}(v) \wedge o_{i+1}(v) \wedge \cdots \wedge o_n(v)$$

after $o_i(v)$ is verified to be a tautology. With the same reasoning given in Section 2.1.2, it can be seen that $s' \Rightarrow s$ since $Obl'(v) \Rightarrow Obl(v)$ (in this case, $Obl'(v)$ is logically equivalent to $Obl(v)$).

2.2.2 Predicate Calculus Rule

Boolean manipulation is essential in constructing proofs. It allows reasoning in a subset of first order logic. The `PC_rule` takes a list of obligations ($[a, b, c]$) and replaces it with another list of obligations ($[d, e]$), upon verification that $(d \wedge e) \Rightarrow (a \wedge b \wedge c)$ is a simple tautology. The list of replacement predicates can be empty in which case the original obligations are discharged if their conjunction is a simple tautology.

The soundness of the `PC_rule` can be justified in the following fashion. Note that the obligation list represents a conjunction of obligations. Since conjunction is commutative and associative, the order of the obligations in the list is not significant. Therefore, without loss of generality, $o_i(v) \cdots o_{i+m}(v)$ are selected to be the old obligations to be replaced.

Consider a state s of the form

$$s = \forall v.(Hyp(v) \wedge Post(v) \Rightarrow Obl(v))$$
$$\text{where } Obl(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_i(v) \wedge \cdots o_{i+m}(v) \wedge \cdots \wedge o_n(v).$$

Soundness requires that the successive state implies the current state. The proposed successive state, s', is of the form

$$s' = \forall v.(Hyp(v) \wedge Post(v) \Rightarrow Obl'(v))$$
$$\text{where } Obl'(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_i'(v) \wedge \cdots \wedge o_{i+m}' \wedge \cdots \wedge o_n(v)$$
$$\text{such that } (o_i'(v) \wedge \cdots \wedge o_{i+m}'(v)) \Rightarrow (o_i(v) \wedge \cdots \wedge o_{i+m}(v)).$$

The new list of obligations is inserted in place of the old obligation with the lowest index.

With some boolean manipulation, we can see that $s' \Rightarrow s$ given that the two predicates are quantified over the same set of variables. Since no new variables are introduced by the `PC_rule`, the implication holds, and therefore the rule is sound.

2.2.3 Instantiation Rule

The `Instantiate_rule` provides a way to extract specific cases from universally quantified expressions. It provides arguments of the following form:

$$\frac{\begin{array}{l}\text{All } A\text{s are } B \\ C \text{ is an } A\end{array}}{C \text{ is } B.}$$

It discharges obligations of the form $\forall i.f(i) \Rightarrow f(k)$, where k is a constant. This proof rule is often used after retrieving a hypothesis (or hypotheses). It can also be used to replace an existentially quantified obligation with a suitable witness. The justification for the rule is equivalent to that presented in section 2.2.1 for linear programming as both discharge tautologies.

2.2.4 Skolemization Rule

The `Skolem_rule` is symmetric to the `Instantiate_rule`. It provides arguments of the following form:

$$\frac{\begin{array}{l} A \text{ is } B \\ A \text{ can be anything} \end{array}}{\text{everything is } B.}$$

This rule is often used to remove quantifiers from an expression. Without quantifiers, the expression is built up from linear arithmetic predicates and atomic formulas using simple logical connectives, where reasoning can be done with the other nine proof rules.

Universal quantifiers, in this checker, are always over all integers. Existentially quantified expressions $\exists i.P(i)$ can be defined as $\neg\forall i.\neg P(i)$ and manipulated by the `Instantiate_rule` and the `Skolem_rule` as universally quantified expressions. In this approach, the `Skolem_rule` provides an existential witness given an existentially quantified predicate, and the `Instantiate_rule` discharges an existentially quantified obligation given an instance.

The concept behind skolemizing a universally quantified expression is to choose a constant which can be of any arbitrary value to substitute the quantifier [17]. This constant is called a skolem constant. To avoid clashes between the representation of a skolem constant and previously defined variables, the skolem constant cannot be a free variable in the target obligation, or any of the hypotheses on the hypothesis list.

It is sufficient to check the target obligation and the hypothesis list for free variables. In our notation, skolemizing an expression is viewed as moving the universal quantifier to the outermost scope.

First, consider a state with an empty postponed list and an obligation. We claim that

$$\begin{array}{rl} & \forall z.(Hyp(z) \Rightarrow \forall x.p(x,z)) \\ \equiv & \forall \_x.\forall z.(Hyp(z) \Rightarrow p(\_x,z)) \end{array}$$

provided that $\_x$ and z are disjoint. This shows that it is necessary to examine the hypothesis list for the free variable $\_x$ while skolemizing $\forall x.p(x,z)$.

To justify the need to examine the target obligation for colliding free variables, consider the state $\forall z.(Hyp(z) \Rightarrow (\forall x.\forall y.p(x,y,z)))$.

$$\begin{aligned} & \forall z(Hyp(z) \Rightarrow (\forall x.\forall y.p(x,y,z))) \\ \equiv \quad & \forall \_x.\forall z.(Hyp(z) \Rightarrow (\forall y.p(\_x,y,z))) \end{aligned}$$

It would be illegal to move the quantifier y to the outer scope using the same skolem constant, $\_x$, since

$$\begin{aligned} & \forall \_x.\forall \_x.\forall z.(Hyp(z) \Rightarrow p(\_x,\_x,z)) \\ \equiv \quad & \forall \_x.\forall z.(Hyp(z) \Rightarrow (\forall x.p(\_x,\_x,z))) \\ \neq \quad & \forall z(Hyp(z) \Rightarrow (\forall x.\forall y.p(x,y,z))) \end{aligned}$$

Now, consider a state with two obligations.

$$\begin{aligned} & \forall z.(Hyp(z) \Rightarrow ((\forall x.p(x,z)) \wedge (\forall y.q(y,z)))) \\ \equiv \quad & (\forall \_x.\forall z.(Hyp(z) \Rightarrow p(\_x,z))) \wedge (\forall \_y.\forall z.(Hyp(z) \Rightarrow q(\_y,z))) \\ \equiv \quad & (\forall \_x.\forall z.(Hyp(z) \Rightarrow p(\_x,z))) \wedge (\forall \_x.\forall z.(Hyp(z) \Rightarrow q(\_x,z))) \\ \equiv \quad & \forall \_x.\forall z.(Hyp(z) \Rightarrow (p(\_x,z) \wedge q(\_x,z))) \end{aligned}$$

This shows that using the same skolem constant for two separate obligations is a legal operation in the proof checker.

Note that the postponed list serves as a buffer to hold obligations that are not in focus at the current step. This list is introduced for the convenience of users of the proof checker, and it is not necessary to distinguish the contents of this list from those of the obligation list when reasoning about logical soundness of the proof checker. (See Section 2.2.7).

Consider a proof state,

$$s = \forall v.(Hyp(v) \wedge Post(v) \Rightarrow Obl(v))$$
$$\text{where } Obl(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge (\forall \_z.P(\_z)) \wedge \cdots \wedge o_n(v).$$

Applying the `Skolem_rule` produces

$$s' = \forall v'.(Hyp(v') \wedge Post(v') \Rightarrow Obl'(v'))$$
$$\text{where } Obl(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge P(\_z) \wedge \cdots \wedge o_n(v).$$
$$v' = v \cup \_z.$$

This transformation is sound given the reasoning above, because the hypotheses and the set of variables appearing in these hypotheses (a subset of v) do not change.

2.2.5 Induction Rule

Mathematical induction provides another way to reason about universally quantified assertions. Given an assertion $P(k)$ that is universally quantified over the integer variable k, we do three things to prove it by induction. Prove that the base case, $P(b)$, is a tautology. Then, prove that given that the expression holds for cases from b to n, (where $n > b$) $P(n+1)$ holds too. We call this inducting up. The final step is to induct downwards by proving that cases b down to n imply $P(n-1)$ (where $n < b$). This is called *strong induction*. As opposed to weak induction, the induction step is implied by all previous cases. Strong induction is equivalent to weak induction [9].

The `Induction_rule` takes a universally quantified obligation, $\forall i.P(i)$, and breaks it into three clauses:

1. The base case, $P(base)$.

2. Induction step going upwards,
   $\forall n.(n > base) \wedge (\forall i \in \{base, n-1\}.P(i)) \Rightarrow P(n)$,
   where i and n are not free variables in P.

3. Induction step going downwards,

   $\forall n.(n < base) \wedge (\forall i \in \{n+1, base\}.P(i)) \Rightarrow P(n)$,

   where i and n are not free variables in P.

We claim that the conjunction of these three clauses is logically equivalent to the initial obligation. Therefore, the replacement is sound.

Consider a state s.

$$s = \forall v.((Hyp(v) \wedge Post(v)) \Rightarrow Obl(v))$$
$$\text{where } Obl(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_i(v) \wedge \cdots \wedge o_n$$

Given that $o_1^{'} \wedge o_2^{'} \wedge o_3^{'} \Leftrightarrow o_i(v)$,

$$s' = \forall v.((Hyp(v) \wedge Post(v)) \Rightarrow Obl'(v))$$
$$\text{where } Obl'(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_1^{'}(v) \wedge o_2^{'}(v) \wedge o_3^{'}(v) \wedge \cdots \wedge o_n$$

is logically equivalent to s by the reasoning given in Section 2.1.2.

2.2.6 Definition Rule

In any proof, there is a set of hypotheses which gives the context of the proof and defines the variables that appear in the proof. The `Definition_rule` takes a hypothesis, H, from the hypothesis list and rewrites an obligation, O, as $H \Rightarrow O$.

Soundness of this rule is shown as follows. Consider states s and s'.

$$s = \forall v.((Hyp(v) \wedge Post(v)) \Rightarrow Obl(v))$$
$$s' = \forall v.((Hyp(v) \wedge Post(v)) \Rightarrow Obl'(v))$$
$$\text{where } Hyp(v) = h_1(v) \wedge \cdots \wedge h_i(v) \wedge \cdots \wedge h_n(v),$$
$$Obl(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_j(v) \wedge \cdots \wedge o_m, \text{ and}$$
$$Obl'(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge (h_i(v) \Rightarrow o_j(v)) \wedge \cdots \wedge o_m.$$

It can be shown by simple predicate calculus that s and s' are logically equivalent, since all variables within the two expressions are within the same scope.

2.2.7 Postponement Rules

The set of `Postponement_rules` increases the flexibility of proof checking. It allows the user to discharge an obligation without verifying it with the built-in proof rules when the required reasoning is outside the scope of the proof checker. When the prove is done, a list of such obligations is produced, and it is up to the user to verify them using other methods. The `Postponement_rules` also provide a lemma mechanism. When a lemma appears more than once in a proof, the lemma can be moved to the postponed list. The lemma can be retrieved from this list each time it is needed. After the last use, the postponed lemma can be moved back onto the obligation list to be discharged with one sequence of proof steps. These rules can be used when sketching out basic structures of proofs. Tedious proof steps can be left unjustified until the exact components of a proof are formulated. Note that the content of the postponed list requires verification given the hypotheses.

Each postponed object in the list is tagged with a name. An obligation is tagged with a name before it is put onto the list, and these 'lemmas' are referenced by names instead of indices. The rules for manipulating the postponed list are described below:

Rule #1 discharges an obligation by moving it to the postponed list. The user provides a name with which to tag this obligation; The rule moves the obligation from the obligation list to the postponed list if the name is not already used or if the obligation implies the postponed object with the same name. If there is an object on the postponed list with the same name, and this object implies the obligation, then the obligation is removed from the obligation list and the postponed list is unchanged. If the name refers to a postponed object and neither of the relations hold, the rule fails.

The soundness of this rule is presented for each case separately. For the case where the proposed name does not exist in the postponed list, consider the following state:

$$s = \forall v.(((Hyp(v) \wedge Post(v)) \Rightarrow Obl(v)) \wedge (Hyp(v) \Rightarrow Post(v)))$$
$$\text{where} \quad Obl(v) = (o_1(v) \wedge o_2(v) \wedge \cdots o_i(v) \cdots \wedge o_n(v)), \text{ and}$$
$$Post(v) = (p_1(v) \wedge p_2(v) \wedge \cdots \wedge p_j(v) \wedge \cdots \wedge p_m(v)).$$

Applying this rule produces the state:

$$s' = \forall v.(((Hyp(v) \wedge Post'(v)) \Rightarrow Obl'(v)) \wedge (Hyp(v) \Rightarrow Post'(v)))$$
$$\text{where} \quad Obl'(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_{i-1}(v) \wedge o_{i+1}(v) \wedge \cdots \wedge o_n(v), \text{ and}$$
$$Post'(v) = o_i(v) \wedge p_1(v) \wedge p_2(v) \wedge \cdots \wedge p_m(v).$$

If $p_j(v)$ is an object on the postponed list with the proposed name and $o_i(v) \Rightarrow p_j(v)$, then

$$s = \forall v.(((Hyp(v) \wedge Post(v)) \Rightarrow Obl(v)) \wedge (Hyp(v) \Rightarrow Post(v)))$$
$$\text{where} \quad Obl(v) = (o_1(v) \wedge o_2(v) \wedge \cdots o_i(v) \cdots \wedge o_n(v)), \text{ and}$$
$$Post(v) = (p_1(v) \wedge p_2(v) \wedge \cdots \wedge p_j(v) \wedge \cdots \wedge p_m(v)).$$

Applying this rule produces the state:

$$s' = \forall v.(((Hyp(v) \wedge Post'(v)) \Rightarrow Obl'(v)) \wedge (Hyp(v) \Rightarrow Post'(v)))$$
$$\text{where} \quad Obl'(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_{i-1}(v) \wedge o_{i+1}(v) \wedge \cdots \wedge o_n(v), \text{ and}$$
$$Post'(v) = o_i(v) \wedge p_1(v) \wedge p_2(v) \wedge \cdots \wedge p_{j-1}(v) \wedge p_{j+1} \wedge \cdots \wedge p_m(v).$$

If $p_j(v)$ is an object on the postponed list with the proposed name and $p_j(v) \Rightarrow o_i(v)$, then

$$s = \forall v.(((Hyp(v) \wedge Post(v)) \Rightarrow Obl(v)) \wedge (Hyp(v) \Rightarrow Post(v)))$$
$$\text{where} \quad Obl(v) = (o_1(v) \wedge o_2(v) \wedge \cdots o_i(v) \cdots \wedge o_n(v)), \text{ and}$$
$$Post(v) = (p_1(v) \wedge p_2(v) \wedge \cdots \wedge p_j(v) \wedge \cdots \wedge p_m(v)).$$

Applying this rule produces the state:

$$s' = \forall v.(((Hyp(v) \wedge Post'(v)) \Rightarrow Obl'(v)) \wedge (Hyp(v) \Rightarrow Post'(v)))$$
$$\text{where} \quad Obl'(v) = o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_{i-1}(v) \wedge o_{i+1}(v) \wedge \cdots \wedge o_n(v), \text{ and}$$
$$Post'(v) = p_1(v) \wedge p_2(v) \wedge \cdots \wedge p_{j-1}(v) \wedge p_{j+1} \wedge \cdots \wedge p_m(v).$$

In all three cases, s and s' are logically equivalent and the replacement preserves the required state implication described in section 2.1.2.

Rule #2 retrieves a pending lemma from the postponed list. It takes a postponed object, P, from the postponed list and rewrites an obligation, O, as $P \Rightarrow O$. Consider state s:

$$s = \forall(((Hyp(v) \wedge Post(v)) \Rightarrow Obl(v)) \wedge ((Hyp(v) \Rightarrow Post(v))))$$
$$\text{where} \quad Obl(v) = (o_1(v) \wedge o_2(v) \wedge \cdots o_i(v) \cdots \wedge o_n(v)), \text{ and}$$
$$Post(v) = (p_1(v) \wedge p_2(v) \wedge \cdots \wedge p_j(v) \wedge \cdots \wedge p_m(v)).$$

Applying this rule produces the state:

$$s' = \forall(((Hyp(v) \wedge Post(v)) \Rightarrow Obl'(v)) \wedge ((Hyp(v) \Rightarrow Post(v))))$$
$$\text{where} \quad Obl'(v) = o_1(v) \wedge \cdots \wedge o_{i-1}(v) \wedge (p_j(v) \Rightarrow o_i(v)) \wedge o_{i+1}(v) \wedge \cdots \wedge o_n(v).$$

s and s' are equivalent by simple boolean manipulation.

Rule #3 moves a postponed object from the postponed list back onto the obligation list. Consider a state s:

$$s = \forall v.(((Hyp(v) \wedge Post(v)) \Rightarrow Obj(v)) \wedge (Hyp(v) \Rightarrow Post(v)))$$
$$\text{where} \quad Obl(v) = (o_1(v) \wedge o_2(v) \cdots \wedge o_n(v)), \text{ and}$$
$$Post(v) = (p_1(v) \wedge p_2(v) \wedge \cdots \wedge p_j(v) \wedge \cdots \wedge p_m(v)).$$

Applying this rule produces the state:

$$s' = \forall v.(((Hyp(v) \wedge Post'(v)) \Rightarrow Obl'(v)) \wedge (Hyp(v) \Rightarrow Post'(v)))$$
$$\text{where} \quad Obl'(v) = p_j(v) \wedge o_1(v) \wedge o_2(v) \wedge \cdots \wedge o_n(v), \text{ and}$$
$$Post'(v) = p_1(v) \wedge p_2(v) \wedge \cdots p_{j-1}(v) \wedge p_{j+1} \wedge p_m(v).$$

This is the inverse of rule#1.

It is essential that the context of an obligation does not change after being moved back and forth from the postponed list and the obligation list. The simple checker maintains a constant hypothesis list and does not introduce the concept of scoping (i.e. all expressions are in the same scope); thus, the proof checker can postpone and retrieve obligations without changing the meaning of these obligations.

2.2.8 Equality Rule

The `EQ_rule` allows two expressions to be used interchangeably in any expression, given that they represent the same value. This rule allows the user to interchange a's and b's in expressions like $(a \equiv b) \Rightarrow f(a, b)$. The replacement obligation is identical to the original obligation except that some a's are replaced by b's and *vice versa*. The replacement and the original obligation are equivalent by substitution.

Applying the `EQ_rule` to state s, where

$$s = \forall v.((Hyp(v) \wedge Post(v)) \Rightarrow Obl(v))$$
$$\text{where } Obl(v) = o_1(v) \wedge \cdots \wedge o_i(v) \wedge \cdots \wedge o_n$$

yields state s', where

$$s' = \forall v.((Hyp(v) \wedge Post(v)) \Rightarrow Obl'(v))$$
$$\text{where } Obl'(v) = o_1(v) \wedge \cdots \wedge o_i^{'}(v) \wedge \cdots \wedge o_n$$

s' is equivalent to s by the claim that $o_i(v)$ and $o_i^{'}(v)$ are logically equivalent.

2.2.9 If Rule

The `IF_rule` is a replacement rule. It rewrites expressions of the form (`if` *True* `then` a `else` b) to a, and expressions of the form (`if` *False* `then` a `else` b) to b. It simplifies boolean expressions once the conditions of the (`if ... then ... else ...`) constructs are evaluated to be a boolean constant (*True* or *False*). This rule applies simple replacement to logically equivalent obligations, therefore can be justified by the reasoning given in the previous section for the `EQ_rule`.

Note that expressions of the form (`if` P `then` x `else` x) can be rewritten into x. The `IF_rule` does not directly support simplification of this form. However, obligations of this form can be simplified by first performing case analysis using the `PC_rule` to rewrite the expression into two clauses: ($P \equiv$ *True*) $\Rightarrow$ (`if` P `then` x `else` x) and (P $\equiv$ *False*) $\Rightarrow$ (`if` P `then` x `else` x). Then, the `EQ_rule` can be used to simplify the two clauses into (`if` *True* `then` x `else` x) and (`if` *False* `then` x `else` x) respectively. These two clauses can then be rewritten into x by the `IF_rule`. Finally, the two identical obligations can be combined into one using the `PC_rule`.

2.2.10 Discrete Rule

The `Discrete_rule` is based on the discreteness of integers. It discharges obligations of the form $(x > y) \equiv (x \geq (y + 1))$ or $(x < y) \equiv (x \leq (y - 1))$ given that both x and y are integers. The justification of this rule is the same as the other discharge rules as was presented in section 2.2.1.

2.3 Conclusion

The chapter has presented the ten proof rules which form the core of the proof checker. As shown in chapter 5, this small set of proof rules is sufficient to verify significant real-time systems.

Chapter 3

Implementation of the Proof Checker

The previous chapter gave a specification for a proof checker. This chapter presents the functions and procedures that implement this specification and is structured to closely parallel the specification. Sections 3.1 and 3.2 in this chapter correspond to Sections 2.1.1 and 2.1.2 in the previous chapter; they describe the structures of the proof checker and proofs constructed by this checker. The proof checker is implemented in FL, the functional interface language of the Voss [25] hardware verification system. FL provides an efficient implementation of Ordered Binary Decision Diagrams [5] which makes boolean manipulation simple. To support reasoning about systems of linear relations, the author added an implementation of the simplex method for linear programming to FL. Section 3.3 gives a detailed explanation of the implementation of the simplex method, and how it is incorporated into Voss. Section 3.4 presents the implementation of the ten proof rules in the same order as that of the specifications in Section 2.1.2 of the previous chapter.

3.1 Abstract Data Type for Proof State

Proof states are encapsulated in an abstract data type, *_state*. States are quadruples built with the constructor `STE` (See Section 3.2). The constructor `STE` is only defined within *_state*, this ensures that states are only constructed by the proof rules presented in this chapter. The four fields in a state are: the obligation list (type `boolean list`), the postponed list (type `postpone list`), the hypothesis list (type `boolean list`), and the claim (type `boolean`). Type `postponed` is defined as the constructor, `post`, followed by the boolean expression to *postpone*, and an identifier to reference to it (i.e. `post` *boolean string*). The type boolean is distinct from the built-in FL-type `bool`. Constructors are included for creating variables and arrays, for the standard boolean operations (And, Or, Not, etc.) and for comparisons of integers and reals. The structure of the type `boolean` is described in detail in Appendix A.

Proof states cannot be constructed or modified outside the abstract data type; however, there are four functions to read the fields of the data type:

- *(*`getclaim` *state)* returns the claim of the proof from the given proof state.
- *(*`gethypothesislst` *state)* returns the list of hypotheses from the given proof state.
- *(*`getpostponelst` *state)* returns the list of postponed objects from the given proof state.
- *(*`getobligationlst` *state)* returns the list of obligations from the given proof state.

3.2 The Proof Rules and some Implementation Techniques

Every proof rule provided by the proof checker takes a list of old obligations and a list of new obligations together with some auxiliary information for the particular rule. Then it either makes the appropriate replacement or fails with an error message if the proposed replacement is not valid. In most cases, the old obligation list is a singleton. As described in the previous chapter, *discharge* rules have empty new obligation lists. In the case of a discharge rule, a singleton list is replaced by an empty list. *Replacement* rules, on the other hand, have one or more elements in the new obligation list. In this case, one or more old obligations are replaced by the new obligations.

Elements of the hypothesis and obligation lists are accessed by indexing. Given the index (an integer) of the element in the list, the desired element is retrieved. All rules, except `PC_rule`, take a singleton old obligation list. The function `apply` is used in the implementation of all these rules. (`apply` *f n lst*) looks up the $\mathtt{n}^{\mathrm{th}}$ element in the obligation list, `lst`, applies the function `f` to this obligation to verify if the suggested resulting list proposed by the user is a valid replacement, then replaces the $\mathtt{n}^{\mathrm{th}}$ obligation by this list. With this structure, there is one core function per proof rule and this function is called by `apply` to validate the replacement.

Several features of FL are used extensively in the checker. FL is a functional language; accordingly, many auxiliary functions are recursive. Pattern matching is often used to enumerate cases according to the type constructors. The next three sections describe some of the functions implemented using these techniques, what it means for a rule to *fail* and explain how a concrete type is defined on top of the core FL types.

3.2.1 Defining the Concrete Types

Concrete types are types defined on top of the three FL types (`int`, `string`, and `bool`). These types are defined by a set of constructors, which can be constants or functions. For example, an `integer` is declared as

```
lettype integer =  const int;
                   I string;
                   i_array string integer;
                   ++ integer integer;
                   -- integer integer;
                   ** integer integer;
                   i_if boolean integer integer;
```

`const`, `I`, `i_array`, ++, −−, ∗∗, and `i_if` are constructors of the type `integer`. These constructors take arguments of various types to produce objects of type `integer`. In the proof checker, integers are represented symbolically, and these constructors build the data structures that represent expressions. Other functions in the proof checker are used to perform operations on these expressions. See Figure A.12 for descriptions of other concrete types.

3.2.2 Pattern Matching

As concrete types are made up of various constructors followed by some defined types, pattern matching is frequently used when writing expressions. As an example, consider the function `eval` which converts an expression of type `boolean` into an FL `bool`. An FL `bool` is represented by a BDD; this representation supports efficient manipulation of boolean expressions, for example, to implement the `PC_rule`. The following shows a few lines from this function:

$$\begin{array}{llcll}
letrec & eval\ \ True & = & T & /\backslash \\
 & eval\ \ False & = & F & /\backslash \\
 & eval\ \ (bool\ \ s) & = & (variable\ \ s) & /\backslash \\
 & eval\ \ (Not\ \ b) & = & (NOT\ \ (eval\ \ b)) & /\backslash \\
 & eval\ \ (And\ b1\ \ b2) & = & ((eval\ \ b1)\ \ AND\ \ (eval\ \ b2)) & /\backslash \\
 & eval\ \ (b_array\ \ s\ \ n) & = & (eval\ \ (bool\ \ (prBool\ \ (b_array\ \ s\ \ n)))) & /\backslash \\
 & eval\ \ ('>\ \ r1\ \ r2) & = & (eval\ \ (bool\ \ (prBool\ \ (r1\ '>\ \ r2)))) & /\backslash \\
 & eval\ \ (forall\ \ n\ \ b) & = & (eval\ \ (bool\ \ (prBool\ \ (forall\ \ n\ \ b)))) & /\backslash \\
 & & \ldots & &
\end{array}$$

The function traverses an expression tree, converts variables, inequalities, and universally quantified expressed into BDD nodes, and creates a BDD corresponding to the expression. In this example, pattern matching is also used to define a recursive function; terminal and non terminal calls are distinguished by the type constructor associated with the argument.

Many other functions in the checker are implemented with the same technique. For example, the functions `replaceBool`, `replaceInt`, and `replaceReal` replace all occurrences of a boolean, integer, or real valued subexpression respectively by another expression of the same type. Implementations of these functions traverse an expression tree by pattern matching, compare each leaf with the subexpression to be replaced, and apply the replacement to the matching subexpressions.

3.2.3 Failures

A proof rule fails when it cannot perform the requested discharge or replacement. Instead of returning the result, the core function for the proof rule generates an FL failure, (`error` *msg*), where *msg* is the error message for the failure. An FL failure can be trapped by the function `catch`: (*e1* `catch` *e2*) evaluates to e1 unless e1 causes a failure, in which case the expression is evaluated to *e2*. For example, the expression

```
let   s = (apply_rule state) /\
      s' = (apply_rule state') in
(apply_rule s) catch (apply_rule s')
```

evaluates to *s* if `apply_rule` successfully performed the request with the input *state*, and evaluates to *s'* if it failed.

3.3 Linear Programming

Simplex is used in the proof checker as a decision procedure for linear programs, i.e. systems of linear relations. This implementation uses simplex to determine the feasibility of a given set of relations rather than generating an optimal solution to some cost function. If a problem is infeasible, this procedure simply returns "infeasible", whereas, if the problem is feasible, a feasible solution can be exhibited as a counter example to the `LP_rule`.

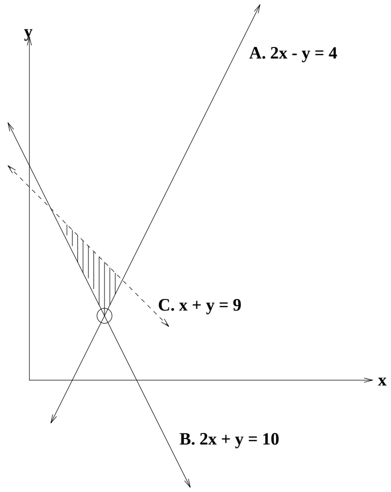


Figure 3.2: A system of linear relations.

3.3.1 Simplex Method

The simplex method, described by Papadimitriou and Steiglitz [23], was implemented to determine the feasibility of a given linear program. The simplex method takes a tableau in standard form and returns an example feasible solution for each feasible set and simply returns "infeasible" for infeasible sets. As an example, consider the following set of linear equations (See figure 3.2):

$$\begin{array}{ll} A. & 2x - y \leq 4 \\ B. & 2x + y \geq 10 \\ C. & x + y < 9 \end{array}$$

Standard Form

A system of the following form

$$\begin{array}{c} \min\ c'x \\ Ax = b \\ x \geq 0 \end{array}$$

is said to be in standard form. Programs with arbitrary inequalities ($<$, $\leq$, $\neq$, $=$, $\geq$, $>$) can be transformed into standard form. First, consider the general case with $\geq$ and $\leq$ relations and unconstrained variables.

A $\geq$ relation can be rewritten into standard form by introducing a *surplus variable.* For example,

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$$

can be rewritten as

$$\sum_{j=1}^{n} a_{ij} x_j - s_i = b_i$$
$$s_i \geq 0$$

where s_i is called a surplus variable.

A $\leq$ relation can be rewritten into standard form in a similar way by introducing a *slack variable.* For example,

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

can be rewritten as

$$\sum_{j=1}^{n} a_{ij} x_j + s_i = b_i$$
$$s_i \geq 0$$

where s_i is called a slack variable.

An unconstrained variable x_j can be split into x_j^+ and x_j^- where $x_j = x_j^+ - x_j^-$. Representing x_j in terms of x_j^+ and x_j^- replaces one unconstrained variable by two constraint variables.

$$unconstrained(x_j) \Rightarrow x_j = x_j^+ - x_j^-$$

$$x_j^+ \geq 0$$
$$x_j^- \geq 0$$

After translating a tableau from a general form into standard form as above, the simplex method can solve the system with $\geq$, $\leq$, and $=$.

For example, the system

$$\begin{array}{ll} A. & 2x - y \leq 4 \\ B. & 2x + y \geq 10 \\ C. & x + y \leq 9 \\ & (x \geq 0) \end{array}$$

can be transform into a standard tableau by introducing two split variables, y^+ and y^- to replace the unconstrained y, two slack variables, and one surplus variable. The resulting tableau has nine constraints and six variables.

$$\begin{array}{rrrrrrcr} 2x & -y^+ & +y^- & & & +s_3 & = & 4 \\ 2x & +y^+ & -y^- & & -s_2 & & = & 10 \\ x & +y^+ & -y^- & +s_1 & & & = & 9 \\ & & & & & & & \\ & & & & & x & \geq & 0 \\ & & & & & y^+ & \geq & 0 \\ & & & & & y^- & \geq & 0 \\ & & & & & s_1 & \geq & 0 \\ & & & & & s_2 & \geq & 0 \\ & & & & & s_3 & \geq & 0 \end{array}$$

Artificial Variables and Basic Feasible Solutions

Consider a linear program with n variables and m constraints. Typically, $n > m$, and if the linear program is feasible, the feasible region is an n dimensional convex polytope. It can be shown that at each vertex of the polytope, at least $n-m$ variables have value zero. In the simplex algorithm, vertices are identified by the choice of the other m variables. The values of these variables can be determined by solving the system of linear equations. This is called a *basic feasible solution* (or a BFS). If more than $n-m$ variables are zero at some vertex, that vertex is said to be degenerate, and it has more than one representation in the simplex algorithm.

For an optimization problem, a linear cost function assigns a cost to each point in the feasible region. It is straightforward to show that the minimum cost is achieved at some vertex of the polytope. The simplex method starts from one vertex of the polytope and moves from one vertex to another until it finds an optimal solution. These moves are called *pivots*. To start the pivoting process, a BFS must be identified.

To find an initial BFS, *artificial variables* are introduced. One new variable is introduced for each equality of the original standard form problem. Each of the equalities can be satisfied by setting the corresponding artificial variable to the appropriate value and setting all of the original variables to zero. This constructs a BFS for the linear program with artificial variables. Using the sum of the artificial variables as a cost function, the simplex algorithm searches for a vertex where all of the artificial variables are zero. If such a vertex is found, it corresponds to a solution to the original program. If no such vertex exists, the original problem is infeasible.

In the implementation used in the proof checker, the *steepest descent policy* is used to select the pivot. The pivot column is selected by

$$j = \min\{j : c_j < 0\}$$

where c_j corresponds to the marginal cost of bringing variable j into the tableau. The pivot column is selected by

$$B(i) = \min\{B(i) : x_{ij} > 0 \text{ and } \frac{x_{i0}}{x_{ij}} \leq \frac{x_{k0}}{x_{kj}} \text{ for every k with } x_{kj} > 0\}$$

Pivoting corresponds to moving along an edge of the polytope. The end of the edge is identified by one of the constraints (on variables being ≥ 0) becoming tight. Moving from vertex v to vertex u, a variable that was in the basis at v is zero at u. This variable is identified by the choice of i.

In the case of degeneracy, the cost, z, may not decrease, even though a column j with $(c_j - z_j) < 0$ is selected. Furthermore, it is possible for the algorithm to return to a previous BFS and loop indefinitely. To avoid cycling, the Bland's anticycling algorithm is used after every zero-improvement pivot. The column to enter the basis is selected by

$$j = \min\{j : c_j - z_j < 0\}$$

and the row by the same formula as that of the steepest descent algorithm,

$$B(i) = \min\{B(i) : x_{ij} > 0 \text{ and } \frac{x_{i0}}{x_{ij}} \leq \frac{x_{k0}}{x_{kj}} \text{ for every k with } x_{kj} > 0\}$$

Since the number of vertices is finite, and the cost is monotonically decreasing without cycling, this algorithm will terminate.

There are three possible cases after this cost function is minimized:

- case 1:

  the cost, z, is zero, and all artificial variables, x_i^a, are driven out of the basis

  $\Rightarrow$ a BFS to the original problem is found.

- case 2:

  at optimality the cost, $z > 0$

  $\Rightarrow$ the original problem is infeasible.

- case 3:

  z is reduced to zero, but some artificial variables remain in the basis at zero level.

In Case 3, one additional pivot is required for each artificial variable remaining in the basis to produce a basis consisting only of variables from the original problem. After driving out all zero-level artificial variables, there is a basic feasible solution for the original problem. This solution will be referred to as BFS B hereafter. The only way this can fail is that a row is zero in all the columns corresponding to non-artificial variables. This means the original problem is not of full rank (i.e. the row is implied by other rows in the system). In this case, this row can be removed from the system.

3.3.2 Strict Inequalities ($>$ and $<$)

The simplex method, described in Section 3.3.1, solves linear programs with relations $\geq$, $\leq$, and $=$. Although strict and non-strict inequalities are indistinguishable in typical numerical programming, considering the application of this implementation, theorems may be stated with tight bounds, in which case the difference is significant. In the proof checker, simplex is implemented using exact rational arithmetic which allows strict inequalities to be distinguished. Given inequalities with $>$ or $<$ relations, the program must be converted to standard form before applying the simplex algorithm.

To handle $>$ and $<$, we introduce a variable ϵ and write

$$\sum a_i > b$$

as

$$\begin{gathered} \sum a_i - \epsilon = b \\ \epsilon > 0 \end{gathered}$$

and

$$\sum a_i < b$$

as

$$\begin{gathered} \sum a_i + \epsilon = b \\ \epsilon > 0. \end{gathered}$$

The simplex algorithm is used to find a feasible point that minimizes $-\epsilon$. If $\epsilon > 0$ at this point, then the original program with a strict inequality was feasible; otherwise, the original program was infeasible.

If there is more than one strict inequality, the same ϵ is introduced to all inequalities to transform these inequalities to equalities. Then an attempt is made to minimize $-\epsilon$ and conclude feasibility as soon as ϵ becomes greater than zero. The feasible solution resulting from this stage is referred to as B' in later references.

A geometric interpretation of ϵ is the distance moving towards the interior of the polytope from the boundary. If there is a feasible solution with $\epsilon > 0$, that means there is a point satisfying all constraints but the point does not lie on the >- or <-constraints.

Consider the linear system

$$\begin{array}{ll} A. & 2x - y \leq 4 \\ B. & 2x + y \geq 10 \\ C. & x + y < 9 \\ & (x \geq 0) \end{array}$$

With the introduction of slack and surplus variables together with ϵ, the resulting system is:

$$\begin{array}{rrrrrrrcr} 2x & -y^+ & +y^- & & & +s_3 & & = & 4 \\ 2x & +y^+ & -y^- & & -s_2 & & & = & 10 \\ x & +y^+ & -y^- & +s_1 & & & +\epsilon & = & 9 \\ \\ & & & & & & x & \geq & 0 \\ & & & & & & y^+ & \geq & 0 \\ & & & & & & y^- & \geq & 0 \\ & & & & & & s_1 & \geq & 0 \\ & & & & & & s_2 & \geq & 0 \\ & & & & & & s_3 & \geq & 0 \\ & & & & & & \epsilon & \geq & 0 \end{array}$$

3.3.3 Not-equal-to Relations ($\neq$)

Let P be the feasible polytope for the program when not-equals-to relations are ignored. A not-equals-to relation excludes points that lie on the hyperplane defined by the corresponding equals-to relation. If this hyperplane does not intersect P, then all points in P satisfy the not-equals-to relation. If this hyperplane contains P, then the original program is infeasible. Finally, if the hyperplane intersects P but does not contain P, then the intersection of the hyperplane with P is of dimension one less than the dimension of P. In this case, almost all points in P satisfy the not-equals-to relation, and any remaining not-equals-to relations can be considered independently (because the number of not-equals-to relations is finite and therefore countable).

In the proof checker, the feasible polytope is never explicitly constructed. Instead, a BFS is found for the program when not-equals-to relations are ignored. Let B be such a BFS. Now, the not-equals-to relations can be considered one at a time. If B satisfies the relation, then the infeasible hyperplane of the relation does not contain B and therefore it does not contain all of P. On the other hand, if B is in the infeasible hyperplane, then the implementation pivots to find a BFS that is above or below this hyperplane. If no such BFS is found, then the original program is infeasible. If a suitable BFS is found for every not-equals-to relation, then the original program is feasible.

By examining one not-equals-to relation at a time, an exponential problem is avoided. An exponential number of linear programming problems would be generated, if both the *below* and *above* cases for each not-equals-to relation is considered at the same time. The implementation described above solves at most one linear programming problem per not-equal-to relation.

3.3.4 Special Cases

There are a few special cases which are not resolved by the methods described above.

- All zeros row:
  - If a linear program has a constraint of the form $0 \neq 0x$, $0 > 0x$, or $0 < 0x$, then the program is infeasible.
  - A constraint of the form $0 \leq 0x$, $0 = 0x$, or $0 \geq 0x$ is trivially satisfied everywhere and can be deleted from the linear program.
- A linear program with only not-equals-to relations is feasible as long as none of these are of the form $0 \neq 0x$.

Figure 3.3 shows the pseudocode for the implementation of linear programming.

standardize tableau (check for special cases)
rewrite $>$ and $<$ constraints, introduce ϵ
move $\neq$ constraints to the unresolved list
introduce artificial basis x_i^a
call simplex with cost $z = \sum x_i^a$ (BFS: B)
(without pivoting on ϵ column)
if $z_{opt} > 0$ **then** return **infeasible**
if an artificial variable is in the basis and cannot be driven out
then omit corresponding row
for each element on the unresolved list {
   if B satisfies this constraint
   then remove the constraint from the unresolved list
   update ϵ column for $>$ and $<$ constraints
} **call** simplex with cost $-\epsilon$ (BFS: B')
if cannot find feasible solution B' **then** return **infeasible**

for each unresolved element on the unresolved list{
/* pivot to find point satisfying the relation */
   add element to system as $<$
   if feasible **then** continue
   add element to system as $>$
   if feasible **then** continue
   return **infeasible**
}
return BFS.

Figure 3.3: Pseudocode for Linear Programming.

3.4 Implementation of Proof Rules

This section describes the implementation of the ten proof rules in a similar format as in Section 2.1.2 from the previous chapter. This section emphasizes implementation issues. For detailed usage of the proof rules, refer to the User Manual in Appendix A.4.

Each rule is summarized by a table. The field **Syntax** describes how to apply a rule to a proof state. It lists the arguments in the order in which the function is called. **Type** indicates whether the rule removes an obligation from the obligation list (*discharge* rule) or replaces the obligation with an expression that implies the old obligation (*replacement* rule). The **Expected Structure** is the general form of the obligation to be discharged or replaced. The **Arguments** section provides an explanation for each argument required by the function. The **Functionality** section describes the typical use of the function.

While reading this section, note the distinction between the conceptual proof rules, which are referred to as *x*`_rule`, and the implementation of these theories which are denoted by the names of the core functions which implement them (usually of the form `apply_`*x*).

Syntax:	(`apply_lp` n $state$)
Type:	*discharge*
Expected Structure:	$(a_1$ *And* a_2 *And* $\cdots$ *And* a_n *And* (*Not* c)) *Equal False*
Arguments:	n is the index of the target obligation. $state$ is the source state.
Functionality:	decision procedure for systems of linear inequalities.

Table 3.1: Linear Programming Rule

3.4.1 Linear Programming Rule

The function `apply_lp` is built on top of the FL function, `LP`, whose implementation was described in the previous section. `LP` takes as its argument a `string` representing a linear program as a tableau and returns an FL `bool`, `T` to indicate a feasible solution and `F` for an infeasible solution. Such tableau should be of the following form:

$$\begin{array}{lll} \{ & m, n; & \\ & r_1 & b_1, x_{11}, x_{12}, \cdots, x_{1m}; \\ & r_2 & b_2, x_{21}, x_{22}, \cdots, x_{2m}; \\ & \vdots & \\ & r_n & b_n, x_{n1}, x_{n2}, \cdots, x_{nm}; \\ \} & & \end{array}$$

where m is the number of variables; n is the number of (in)equalities; r_i is the relation of the i^{th} inequalities; b_i is the constant value on the ith row; and x_{ij} is the coefficient of the jth variable in the ith row.

The function `prnTableau` takes a clause of the form

$$((a_1 \textit{ And } a_2 \textit{ And } \cdots \textit{And } a_n \textit{ And } (\textit{Not } c)) \textit{ Equal False})$$

and transforms its negation to a string representing the corresponding tableau. This tableau has linear inequalities a_1, a_2, ..., a_n and $\neg c$ (i.e. c with relation inverted). The

Syntax:	`(apply_PredicateCalc` *index_list predicate_list state*`)`
Type:	*replacement/discharge*
Expected Structure:	none
Arguments:	*index_list* is the list of indices to the old obligation list. *predicate_list* is the list of replacements. *state* is the source state.
Functionality:	decision procedure for boolean manipulations.

Table 3.2: Predicate Calculus Rule

output from `prnTableau` is the input to `LP`. If `LP` returns `F`, indicating infeasibility of the system, the obligation is a tautology, and `apply_lp` discharges it. If `LP` returns `T`, indicating feasibility of the system, the rule fails.

3.4.2 Predicate Calculus Rule

The `PC_rule` is the only rule whose old obligation list varies in size. It takes an old obligation list of arbitrary size and replaces it with a new obligation list of arbitrary size. The new obligation list can be empty in which case `PC_rule` acts as a discharge rule.

FL represents boolean expressions (of type `bool`) using ordered binary decision diagrams (OBDDs) [5] and this allows symbolic manipulation of expressions. The function `apply_PredicateCalc` uses this feature to do tautology checking. As shown in section 3.2.2, the function `eval` uses pattern matching to translate expressions of type `boolean` into FL `bools`. It treats inequalities and `forall` expressions as single BDD nodes. After the list of old obligations and the list of new obligations are each rewritten as conjunctions of boolean values, the function `eval` is used to determine whether the new list implies the old list. If this holds, then the old list is removed from the obligation list and the new list is inserted in place of the old obligation with the lowest index.

Syntax:	`(apply_skolem` *n skolemized_expr subexpr i skolem_const state*`)`
Type:	*replacement*
Expected Structure:	any boolean expression with a universally quantified subexpression.
Arguments:	*n* is the index of the target obligation. *skolemized_expr* is the desired replacement. *subexpr* is the universally quantified subexpression to be skolemized. *i* is the quantifier to be replaced with a skolem constant. *skolem_const* is the proposed skolem constant. *state* is the source state.
Functionality:	skolemize universally quantified expressions.

Table 3.3: Skolemization Rule

Otherwise, the rule fails with an error message.

3.4.3 Skolemization Rule

The `Skolem_rule` retrieves the indexed obligation from the obligation list and skolemizes the specified subexpression of the obligation with the proposed skolem constant. The subexpression can be the entire obligation if the obligation is universally quantified.

The function `apply_skolem` examines the old obligation and the hypotheses in the hypothesis list to check if the proposed skolem constant is a free variable in any of these expressions. If the skolem constant already exists as a free variable, the rule fails. Otherwise, it is a valid skolem constant, and the function `replaceInt` is used to replace all occurrences of the identifier in the given subexpression by this constant. After the subexpression is skolemized, it is substituted into the old obligation in place of the old subexpression, and the rule checks if it matches the desired replacement given by the

Syntax:	`(instantiate` n k *state*`)`
Type:	*discharge*
Expected Structure:	$(\forall i.P(i)) \Rightarrow P'(j)$
Arguments:	n is the index of the target obligation. k is the value with which the quantifier is to be instantiated. *state* is the source state.
Functionality:	instantiate universally quantified expressions.

Table 3.4: Instantiation Rule

user. If the subexpression to be skolemized occurs more than once in the obligation, the implementation tries each instance individually to determine if the replacement produces the proposed result. If no replacement matches the result, the rule fails. Skolemization can only be applied to one universally quantified expression at each application, because a unique skolem constant is needed for each skolemization. If two identical subexpressions in the same obligation are to be skolemized, the rule must be applied twice.

3.4.4 Instantiation Rule

The `Instantiate_rule` is a *discharge* rule. It retrieves the indexed obligation from the obligation list and pattern matches its structure. The obligation is expected to be of the following structure:

$$(\forall i.P(i)) \Rightarrow P'(j),$$

where j is the instance.
If it does not match the required form, the rule fails with an error message indicating the expected structure of the obligation.

Once the structure is matched, the function `instantiate` uses `replaceInt` to replace

Syntax:	`(induct` *n k base state*`)`
Type:	*replacement*
Expected Structure:	$\forall i.P(i)$
Arguments:	n is the index of the target obligation. k is the proposed quantifier for the resulting universally quantified expression. *base* is the proposed base case. *state* is the source state.
Functionality:	provide reasonings with mathematical induction.

Table 3.5: Induction Rule

all occurrences of i, the identifier, by j, the instance, in $P(i)$. If this result is identical to $P'(j)$, then $P'(j)$ is a proper instantiation of $\forall i.P(i)$ and the obligation can be discharged as a tautology. Otherwise, the rule fails.

3.4.5 Induction Rule

The `Induction_rule` retrieves the indexed obligation from the obligation list and replaces it with three new obligations.

As described in Section 2.2.5, this rule writes an obligation of the form $\forall i.P(i)$ into

$P(base)$,

$\forall k.(k > base)\ AND\ (\forall i \in \{base, k-1\}.P(i)) \Rightarrow P(k)$, and

$\forall k.(k < base)\ AND\ (\forall i \in \{k+1, base\}.P(i)) \Rightarrow P(k)$.

For the base case, the function `replaceInt` is used to replace the identifier by the base case, *base*. For the induction steps, the implementation ensures that the identifier k is not a free variable within the predicate, P. Then it uses `replaceInt` on the predicate and constructs the forms for the two induction steps.

Syntax:	`(by_hypothesis` *n i state*`)`
Type:	*replacement*
Expected Structure:	none
Arguments:	n is the index of the target obligation. i is the index of the hypothesis on the hypothesis list. *state* is the source state.
Functionality:	retrieve information from hypotheses of the proof.

Table 3.6: Definition Rule

3.4.6 Definition Rule

The implementation of `Definition_rule` retrieves the indexed obligation, o, from the obligation list, retrieves the indexed hypothesis, h, from the hypothesis list, and replaces the old obligation by $h ==> o$.

3.4.7 Postponement Rules

There are three rules in this set: `postpone`, `by_postponement`, and `retrieve`.

The arguments of `postpone` are the index of the obligation to be postponed and a name with which to tag it. `Postpone` traverses the postponed list scanning for the name. If the name does not exist in the list, the obligation is simply removed from the obligation list and added to the beginning of the postponed list. Otherwise, the rule checks to see if this obligation is logically related to the postponed object with the same name. The object from the postponed list and the obligation are translated into their BDD representations with the function, `eval`. If the obligation implies the object, the obligation is removed from the obligation list and replaces the postponed object in the postponed list. If the implication is true in the other direction, the obligation is removed from the obligation list and the postponed list remains the same. When neither relation

Syntax:	(`postpone` *n name state*)
Type:	*discharge*
Expected Structure:	none
Arguments:	*n* is the index of the target obligation. *name* is the name with which to tag the postponed object. *state* is the source state.
Functionality:	postpone verification of an obligation.
Syntax:	(`by_postponement` *n name state*)
Type:	*replacement*
Expected Structure:	none
Arguments:	*n* is the index of the target obligation. *name* is the name of the postponed object to be retrieved. *state* is the source state.
Functionality:	retrieve information from postponed list.
Syntax:	(`retrieve` *name state*)
Type:	*replacement*
Expected Structure:	none
Arguments:	*name* is the name of the target postponed object. *state* is the source state.
Functionality:	move a postponed object back to the list of proof obligation to be verified.

Table 3.7: Postponement Rules

Syntax:	`(apply_equality` *n result state*`)`
Type:	*replacement*
Expected Structure:	$(x1 \equiv x2) \Rightarrow P$, where $x1$ and $x2$ are of the same type, `boolean`, `integer`, or `real`.
Arguments:	n is the index of the target obligation. *result* is the desired replacement. *state* is the source state.
Functionality:	rewrite an obligation given equality of two variables.

Table 3.8: Equality Rule

holds, the rule fails.

`By_postponement` is similar to the `Definition_rule`. It retrieves the indexed obligation, o, from the obligation list, retrieves the postponed object, p, with the matching name from the postponed list, and replaces the old obligation by p ==> o. It matches the name by traversing the postponed list as is done in `postpone`.

Unlike the other replacement rules, `retrieve` adds an obligation to the obligation list. The rule looks up the named postponed object by traversing the postponed list, removes it from the postponed list, and inserts it at the beginning of the obligation list.

3.4.8 Equality Rule

The `EQ_rule` retrieves the indexed obligation of the form $(x1 \equiv x2)$ ==> P. It replaces all occurrence of $x1$ by $x2$ in P using functions `replaceInt`, `replaceReal`, and `replaceBool`. The same replacement is done with the proposed new obligation. If the results from the two replacements match structurally, then the rule replaces the old obligation with the new obligation. If the two results do not match, the rule fails.

Syntax:	(`rewrite_if` *n result state*)
Type:	*replacement*
Expected Structure:	*x*`_if` *True a* `else` *b* or *x*`_if` *False a* `else` *b*, where *x*`_if` = `b_if`, `i_if`, or `r_if`.
Arguments:	*n* is the index of the target obligation. *result* is the desired replacement. *state* is the source state.
Functionality:	simplify conditional expressions.

Table 3.9: If Rule

3.4.9 If Rule

The function `rewrite_if` uses pattern matching to simplify (`if` ... `then` ... `else` ...) constructs once the conditions are evaluated to be `True` or `False`. It traverses the expression tree of the indexed obligation, matches the conditions with `True` or `False` and replaces the obligation with the `then` or `else` clauses accordingly. If the proposed replacement matches this resulting expression, the replacement is made. Otherwise, the rule fails.

3.4.10 Discrete Rule

The function `apply_discrete` uses pattern matching to match the obligation with the expected structures. If the retrieved obligation does not match any of these forms, the rule fails. Otherwise, the obligation is discharged from the obligation list.

Syntax:	`(apply_discrete` *n state*`)`
Type:	*discharge*
Expected Structure:	$(x > y)\ Equal\ (x \geq (y+1))$ or $(x < y)\ Equal\ (x \leq (y-1))$, where x and y are of type `integer`.
Arguments:	n is the index of the target obligation. *state* is the source state.
Functionality:	provide discreteness property of integers.

Table 3.10: Discrete Rule

3.5 User Interface

Interface functions can be built on top of the core functions described in the previous section to ease state manipulations. Because the proof state is protected by an abstract data type and the implementation of these user interface functions is outside the data type, the set of user interfaces does not affect the soundness of the resulting proof. This section describes the implementation of some user interface functions. It describes two types of Case Analysis: one over booleans and the other over integers, explains how an instance of a hypothesis can be discharged with one proof step, and how to use abbreviations while printing large expressions. General information for each function is tabulated in the same format as in the previous section. This set of functions can be extended by the users to suit the application.

3.5.1 Case Analysis over booleans

Case analysis "over booleans" uses the `PC_rule` to split obligation, o, into

$$(case\ Equal\ True) \Rightarrow o$$

Syntax:	`(CaseAnalysis` *n case state*`)`
Type:	*replacement*
Expected Structure:	none
Arguments:	n is the index of the target obligation. *case* is the case to apply case analysis on. *state* is the source state.
Functionality:	boolean case analysis.

Table 3.11: Case Analysis over Booleans

and

$$(case\ Equal\ False) \Rightarrow o.$$

Unlike the proof rules in the proof checker, users do not provide the form of the new obligations for this interface function.

3.5.2 Case Analysis over integers

Case Analysis "over integers" uses the `PC_rule` to break an obligation into multiple obligations with different ranges. The `LP_rule` is used to ensure that the subranges cover the integers. Like CaseAnalysis over booleans, this interface function does not require the form of the new obligations from the user.

3.5.3 Discharged by Unchanged

Unchanged handles three most common ways an obligation is discharged given information from the hypothesis list.

1. The target hypothesis and indexed obligation are structurally equivalent. The argument *value* is not needed for this scenario. In this case, the user provides a

Syntax:	(`CaseAnalysis2` *n expr lst state*)
Type:	*replacement*
Expected Structure:	none
Arguments:	*n* is the index of the target obligation. *expr* is any integer valued expression to apply case analysis on. *lst* is the list of integers (in increasing order) making up the subranges for the cases. *state* is the source state.
Functionality:	integer case analysis.

Table 3.12: Case Analysis over Integers

dummy variable as *value* which will be ignored by the function.

2. The obligation is a strict instantiation of the hypothesis, i.e. it structurally matches the hypothesis once all occurrences of the hypothesis's quantifier are replaced by the proposed instance, *expr*.

3. The obligation is an instantiation of the hypothesis, but the quantifier of the hypothesis does not match structurally with the instance, *expr*.

The first case is discharged by simple `PC_rule`, together with the `Definition_rule` which extracts the indexed hypothesis.

The next case is discharged by calling the `Definition_rule` to extract the related information, calling the `PC_rule` to rewrite the obligation into the form which can be handled by the `Instantiate_rule`, then calling the `Instantiate_rule` to verify the instantiation.

The last case is very similar to the second case, but it can handle cases where the bounds on the quantifier are not structurally identical to those of the hypothesis. The

Syntax:	(`Unchanged` *n hyp value state*)
Type:	*discharge*
Expected Structure:	none
Arguments:	*n* is the index of the target obligation. *hyp* is the index of the hypothesis which is used to discharge obligation *n*. *value* is the proposed value with which to instantiate the target hypothesis. *state* is the source state.
Functionality:	discharge instances of hypotheses as obligations.

Table 3.13: Discharged by Unchanged

`LP_rule` is used to validate the obligation's quantifier.

The interface function determines which case to apply by examining the structure of the obligation.

3.5.4 Printing a State

The function `print_State` prints all fields in the given state. It uses functions `getclaim`, `gethypothesislst`, `getpostponelst`, and `getobligationlst` to retrieve different fields from the proof state. Then it maps the printing functions to each of these lists. This function is useful in proof construction and debugging.

3.5.5 Print Abbreviation

The *print abbreviation* functions allow large expressions to be printed in a more compact and comprehensible form. The functions `abbrevBool`, `abbrevInt`, and `abbrevReal` introduce abbreviation-expression pair and append it to an abbreviation list. The abbreviation list is stored as an FL variable by the user. The function `print_abbrev` takes the abbreviation list and the state, retrieves fields from the state and substitutes expressions with abbreviations, then prints the resulting string.

3.6 Conclusion

This chapter has presented the implementation of the proof checker: the decision procedure for linear programming incorporated into the Voss System and the ten proof rules on top. The user interface functions are examples of how the system can be extended to ease proof development. Users can build similar functions according to their needs without compromising the soundness of the proof checker.

Chapter 4

Verification of Real-time Properties

The proof checker was implemented to verify timing issues of real-time systems. As a basis for formal verification, real-time systems are modeled in the Synchronized Transitions language. Real-time properties are stated as safety properties which can be captured by invariants of the programs. These invariants are manually translated into logic predicates as inputs to the proof checker. This chapter describes this approach to real-time verification and compares it with other existing approaches. Much of the material in this chapter is drawn from [15, 16].

Throughout this chapter, a simple, synchronous communication circuit is used as an example to illustrate how timing properties of circuits can be represented as real-time properties of programs, and how these real-time properties can be verified. Consider a transmitter-receiver pair operating at the same frequency as given by a global clock as shown in Figure 4.4. The transmitter outputs a sequence of values at a fixed period set by a global clock. Consecutive values are assumed to be distinct (for example, by using a self-timed encoding [31]); which is modeled by an alternation between the boolean values true and false. The receiver inputs one value for each period of the global clock. The transmitter and receiver operate at the same rate, but the relative timing of the two is not specified. To verify that this interface operates correctly, it must be shown that no values are dropped or duplicated. This is expressed by the two requirements below:

Requirement 1: When the transmitter is enabled to output a value, the receiver must have already acquired the current value.

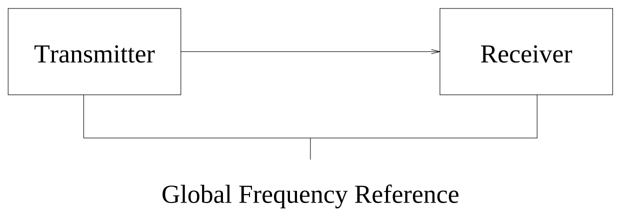


Figure 4.4: A synchronous communication circuit.

Requirement 2: When the receiver is enabled to input a value, the transmitter must have already sent a new value.

These requirements are *real-time* properties of the synchronous communication circuit. To verify that an implementation of the circuit satisfies these requirements, the circuit can be modeled as a concurrent program, and the requirements can be formalized as safety properties of the program.

This chapter shows how the essence of this protocol is captured in Synchronized Transitions and how the proof checker is used to show that no value is dropped or duplicated during the process.

4.1 Synchronized Transitions: a hardware description language

Synchronized Transitions (ST) is a hardware description language in which digital circuits are modeled as concurrent programs. Programs written in ST describe both the computation and the structure of digital circuits. It can be used to specify designs from very high level of abstraction down to gate level descriptions. ST is based on a few, simple concepts of concurrent programming such as guarded multiassignments called *transitions* and asynchronous composition of these transitions. For the purpose of the proof checker, only a subset of the language is used and described in this section. See [21, 28] for a more detailed description of ST.

ST programs are composed of *transitions*, guarded multi-assignments that can be composed asynchronously. Syntactically, transitions are written in the form:

$$\ll \texttt{precondition} \rightarrow \texttt{action} \gg$$

The precondition is a boolean valued expression and the action is a multiassignment. To avoid conflicting assignments, the variables appearing on the left side of the multiassignment must be distinct. For example,

$$\ll \texttt{a} = \texttt{b} \rightarrow \texttt{x, y := x+1, a} \gg$$

is a transition that when enabled can increment `x` and set `y` to the value of `a`. It is enabled whenever `a = b` holds. Two or more transitions may be combined with the asynchronous operator, $\|$. Such transitions are performed atomically (i.e. one at a time) and independently. There is no global thread of control – the order in which transitions are executed is independent of where they appear in the program. As an example, the following program sorts a, b, and c into descending order.

$$\begin{array}{rl} & \ll \texttt{a < b} \rightarrow \texttt{a, b := b, a} \gg \\ \| & \ll \texttt{b < c} \rightarrow \texttt{b, c := c, b} \gg \end{array}$$

Each of the two transitions can be executed independently whenever its precondition holds and the transition is enabled.

4.2 Safety Properties and Invariants

An ST program denotes a state transition relation that is the basis for verifying properties of programs. Given a program, P, V_P denotes the state variables of P, and T_P denotes the transitions of P. A state of P is an assignment of values to the elements of V_P. Let S_P denote the set of all such assignments. Thus, a state variable is a function from S_P to values of the underlying type of the variable. If x is a state variable and s is a state, let $x(s)$ (also written as $x.s$) denote the value of x in state s. If E is an expression of state variables, then $E(s)$ (or $E.s$) has the obvious meaning.

A transition is composed of a precondition and a multiassignment. Let $t = \ll g \rightarrow l := r \gg$ be a transition. The precondition, g is a function from states (i.e. S_P) to booleans: $g(s)$ is true if and only if t is enabled in state s. The multiassignment, $l := r$ is a function from S_P to S_P. Let m denote this function. $s_2 = m(s_1)$ if and only if state s_2 is obtained by performing the multiassignment $l := r$ in state s_1. Let $R_P \subseteq S_P \times S_P$ denote the state transition relation of P. Given two states, s_1 and s_2, a program can make a transition from s_1 to s_2 if and only if there is a transition that is enabled in state s_1 such that performing that transition leads to state s_2. More formally,

$$(s_1, s_2) \in R_P \quad \equiv \quad \exists \ll g \rightarrow m \gg \in T_P . g(s_1) \wedge (s_2 = m(s_1))$$

A system satisfies a safety property Q, if Q holds in the initial state, and in all states reachable from the initial state. A state, s is reachable from s_0 if and only if there exists a sequence of transitions which leads to state s when started at state s_0. A standard approach to verifying such a safety property is to find an invariant, I, such that $Q_0 \Rightarrow I$ and $I \Rightarrow Q$, where Q_0 is the initial state predicate (a condition which holds in the initial state). A predicate I is an invariant of the program P (written as $inv(I, P)$), if I holding in one state guarantees that I will hold in all successive states. Two properties are used in proving a predicate to be an invariant:

Property 1 *Let (T_1, V) and (T_2, V) be programs where T_1 and T_2 are sets of transitions and V is a set of variables. A predicate I is an invariant of $(T_1 \| T_2, V)$ if and only if I is an invariant of both (T_1, V) and (T_2, V).*

Given a program P and a predicate I on states of P, property 1 shows that each transition of P can be considered separately in showing that I is an invariant of P. The next property shows how to establish that I is an invariant of a single transition.

Property 2 *Let $P = (\ll C \rightarrow l := r\gg, V)$. Let I be a predicate. I is an invariant of P if and only if:*

$$\begin{array}{ll} \forall s_1, s_2 \in S_p. & I(s_1) \wedge C(s_1) \wedge (l(s_2) = r(s_1)) \wedge (\forall v \in V - L : v(s_2) = v(s_1)) \\ \quad\Rightarrow & I(s_2) \end{array}$$

where L is the set of variables appearing in l.

Given these two properties, to determine whether the predicate I is an invariant of the program $P = (t_1 \| t_2 \| \dots \| t_n, V)$, where $t_i = \ll C_i \rightarrow l_i := r_i \gg$, verification of the following clause is required:

$$\begin{array}{ll} \forall s_1, s_2 \in S_P(& \\ & (I(s_1) \wedge C_1(s_1) \wedge (l_1(s_2) = r_1(s_1)) \wedge (\forall v \in V - L_1 : v(s_2) = v(s_1))) \Rightarrow I(s_2) \\ \bigwedge & (I(s_1) \wedge C_2(s_1) \wedge (l_2(s_2) = r_2(s_1)) \wedge (\forall v \in V - L_2 : v(s_2) = v(s_1))) \Rightarrow I(s_2) \\ \bigwedge & \dots \\ \bigwedge & (I(s_1) \wedge C_n(s_1) \wedge (l_n(s_2) = r_n(s_1)) \wedge (\forall v \in V - L_n : v(s_2) = v(s_1))) \Rightarrow I(s_2) \\ ) & \end{array}$$

To simplify the expression, state s_2 is written as $M(s_1)$, where M, the *multiassignment* $(l := r)$, is a function over states of P. The conditions $(l(s_2) = r(s_1))$ and $(\forall v \in V - L : v(s_2) = v(s_1))$ are dropped, since it is implied by the definition of M. The simplified condition

$$\begin{array}{ll} \forall s_1, s_2 \in S_P(& \\ & (I(s_1) \wedge C_1(s_1)) \Rightarrow I(M_1(s_1)) \\ \bigwedge & (I(s_1) \wedge C_2(s_1)) \Rightarrow I(M_2(s_1)) \\ \bigwedge & \dots \\ \bigwedge & (I(s_1) \wedge C_n(s_1)) \Rightarrow I(M_n(s_1)) \\ ) & \end{array}$$

is the input to the proof checker for verifying $inv(I, P)$. As the input to the proof checker, $I(M_i(s_1))$ is expanded into $I(s_1)$ with all occurrences of l_i replaced by r_i, where $M_i = (l_i := r_i)$.

When the focus is on proving that a transition preserves an invariant, we sometimes use the notion of a *pre* state (the state before a transition occurs) and a *post* state (the state after a transition has occurred). We write $x.pre$, equivalent to $x(pre)$, to denote the value of x before an execution of a transition and $x.post$, equivalent to $x(post)$, to denote the value of x after the transition is executed.

4.3 Expressing Real-time Properties

Returning to the synchronous circuit example, the physical circuit can be described by two simple ST transitions.

$$\begin{array}{l} \ll\ T.v := NOT\ \ T.v\ \gg \\ \|\ \ll\ R.v := T.v\ \gg \end{array}$$

$T.v$ represents the logical value of the signal output by the transmitter, and $R.v$ represents the logical value of the signal input by the receiver. There is no precondition for either transition; the multiassignments can be performed at any time.

The first transition models the transmitter. It states that the value output by the transmitter alternates between empty and non-empty values ($T.v := NOT\ \ T.v$). The second transition, which models the receiver, is similar to the transmitter transition. The transition models the receiver retrieving the signal from the transmitter ($R.v := T.v$).

In the interleaved model of concurrency provided by ST, the physical structure of the synchronous circuit can be expressed in a clean and simple manner. However, this simple untimed program does not satisfy the two requirements stated above. In this model, statements in a program are executed atomically, but the order of execution is unspecified. Consider the case where the first transition is executed twice consecutively. This scenario, corresponding to the case where two signals are output by the transmitter without the first being retrieved by the receiver, violates *requirement 1* stated in the introduction of this chapter. Conversely, the case where the second transition is executed twice consecutively violates *requirement 2*. Timing properties of the system must be captured in the model in order to verify that the system satisfies the stated requirements.

To reason about timing properties, additional constraints must be included in the model of the system. Using the notation and properties of Synchronized Transition programs, these constraints can be expressed by adding auxiliary variables to the program These variables are called *auxiliary* variables, because they are introduced for timing verification and do not correspond to signals of the physical circuit. A real valued variable, τ is introduced to represent the current time, and other variables are introduced for time related bookkeeping. In general, there are two kinds of timing properties:

Timing lower bounds: a transition is not performed until after a specified time.

Timing upper bounds: a transition is guaranteed to be performed by a specified time.

Timing lower bounds can be expressed by strengthening the transition's preconditions. In particular, systems of inequalities describing timing relationships can be introduced as preconditions to transitions. Likewise, the invariant to be proven includes systems of linear inequalities in addition to boolean relationships. This motivates adding a decision procedure for systems of linear inequalities to the proof checker. Given this decision procedure, the proof checker can be used to reason about timing issues in real-time systems.

In the synchronous circuit program, auxiliary variables are introduced for time related bookkeeping and preconditions are added to the two transitions.

$$\begin{array}{l} \ll\ \tau \geq T.\tau + \pi\ \rightarrow\ T.v, T.\tau := NOT\ \ T.v, \tau\ \gg \\ \|\ \ll\ \tau \geq R.\tau + \pi\ \rightarrow\ R.v, R.\tau := T.v, \tau\ \gg \end{array}$$

The variables τ, $T.\tau$ and $R.\tau$ are introduced to the program. The real valued variable, τ, is introduced to represent the current time, while $T.\tau$ and $R.\tau$ are introduced to denote the time at which the transmitter outputs a signal and the time at which the receiver inputs a signal respectively. The precondition of the first transition $\tau \geq T.\tau + \pi$ enforces a delay of at least π time units between the output of successive values by the transmitter. Likewise, the precondition of the second transition $\tau \geq R.\tau + \pi$ enforces a delay of at least π time units between the retrieval of successive values by the receiver.

Timing upper bounds can be expressed as safety properties of the program's environment. Assertions are added to the program to state that the current time cannot exceed a certain value until after some enabled transition is performed. These assertions are written as a protocol describing the environment [29]. In addition to deriving separate lemmas for each transition of the program, a separate lemma shows that this protocol maintains the invariant.

The protocol for the synchronous circuit environment can be described by four clauses:

- $P_1 \triangleq \tau.post \leq T.\tau + \pi$

  is the timing upper bound for the transmitter. It ensures that signals are generated at most π time units apart.

- $P_2 \triangleq \tau.post \leq R.\tau + \pi$

  is the timing upper bound for the receiver.

- $P_3 \triangleq unchanged(T) \wedge unchanged(R)$

  is the abbreviation for $(T.post = T.pre) \wedge (R.post = R.pre)$ which means that if the environment takes an action, $T.\tau$, $R.\tau$, $T.v$, and $R.v$ remain unchanged.

- $P_4 \triangleq \tau.post \geq \tau.pre$

  states the current time after an action by the environment, $\tau.post$ must be greater than or equal to the time before the action, $\tau.pre$. In other words, time increases monotonically.

The two requirements stated in the introduction can be formalized as safety property of the program.

$$\begin{aligned} Q \quad \triangleq \quad & (\tau \geq T.\tau + \pi) \Rightarrow (R.v = T.v) \\ \wedge \quad & (\tau \geq R.\tau + \pi) \Rightarrow (R.v \neq T.v) \end{aligned}$$

The first clause of the safety property states that the transmitter can not output a new value until the receiver has picked up the old one. This clause is equivalent to *requirement 1*. The second clause states that when the receiver picks up a value, it is a new one. This clause corresponds to *requirement 2*.

Given the ST program and the protocol, the following invariant is constructed for the synchronous circuit.

$$\begin{aligned} I \quad \triangleq \quad & \tau \leq T.\tau + \pi \\ \wedge \quad & \tau \leq R.\tau + \pi \\ \wedge \quad & (T.\tau \leq \tau) \\ \wedge \quad & (R.\tau \leq \tau) \\ \wedge \quad & (R.\tau > T.\tau) \Rightarrow (R.v = T.v) \\ \wedge \quad & (R.\tau < T.\tau) \Rightarrow (R.v \neq T.v) \\ \wedge \quad & (R.\tau \neq T.\tau) \wedge (|R.\tau - T.\tau| < \pi) \end{aligned}$$

The first two clauses are P_1 and P_2 from the protocol as described above. $(T.\tau \leq \tau) \wedge (R.\tau \leq \tau)$ states that the circuit must appear to be causal as indicated by the auxiliary variables, $(R.\tau > T.\tau) \Rightarrow (R.v = T.v)$ ensures that the transmitter does not output a new value until the receiver has obtained the old one. Similarly, $(R.\tau < T.\tau) \Rightarrow (R.v \neq T.v)$ states that when the receiver picks up a value, it is a new value. These two clauses imply the safety property that no value is duplicated during the process. The last clause states that the transmitter and receiver events must occur at different times. In hardware terminology, coincident transmitter and receiver events would constitute a timing hazard, and in practice some minimum separation must be guaranteed. These issues are explored further in the next chapter.

An initial state, Q_0, can be selected to be

$$\begin{array}{ll} & R.\tau = \tau \\ \wedge & T.\tau = \tau - \pi/2 \\ \wedge & R.v = T.v \end{array}$$

Given such Q_0, it is easy to see that $(Q_0 \Rightarrow I) \wedge (I \Rightarrow Q)$ holds.

4.4 Summary

To summarize the approach described above, a real-time system is modeled as a concurrent program in ST and its environment is described using a protocol. Then, invariants are formulated for the system and translated into proof goals for the checker. Through human interaction to the proof checker using the inference rules, the proof goal is simplified to a conjunction of tautologies.

The approach of capturing real-time properties by introducing auxiliary variables is employed in [1], and the approach of describing the environment of the program using protocols is described in [29].

Chapter 5

Verifying STARI

STARI (Self-Timed At Receiver's Input) is a signaling technique that combines synchronous and asynchronous design methods to achieve a higher bandwidth communication than either alone. STARI uses a synchronous transmitter, a synchronous receiver, and a self-timed FIFO. This chapter demonstrates an application of the proof checker by applying it to verify the timing properties of STARI. Section 5.1 provides an overview of the STARI interface and addresses some of the timing criteria for the system. Section 5.2 models the system as an ST program and formulates an invariant for the program which implies the safety properties of the system. Section 5.3 summarizes results from the proof. Experience from using the proof checker is discussed in section 5.4, and section 5.5 evaluates the effectiveness of the proof checker on the STARI proof.

5.1 STARI Interfaces

The implementation of the synchronous transmitter-receiver pair described in Chapter 4 is only a model to demonstrate an approach to verifying safety properties in real-time systems; it can fail if the value output by the transmitter changes at almost the same time as the value is input by the receiver. This is because with real hardware, operations take some amount of time and are not instantaneous as would be suggested by the atomic semantics of Synchronized Transitions. In a traditional synchronous system, a global clock is used to ensure that the changing and sampling of data are separated in time. This separation must be larger than the uncertainty in the timing of the clock signal. This

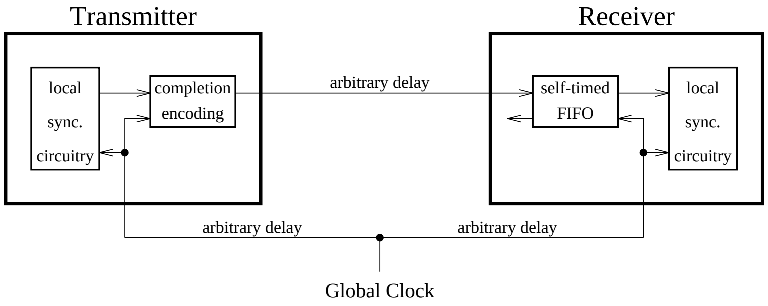


Figure 5.5: STARI communication

uncertainty is called skew. Skew often limits the performance of synchronous systems. To show that an interface operates correctly, it must be shown that new data arrives at the synchronous section of the receiver at a time that is well defined relative to the receiver's clock.

STARI is motivated by the observation that it is a relatively simple matter to distribute a frequency reference signal throughout a large system. On the other hand, it is difficult to control the exact phase of high frequency signals. As mentioned above, this skew limits the performance of purely synchronous systems. Self-timed designs avoid clock skew by using handshake protocols. If no assumptions are made about the delays of components and wires, then each transmitted bit must be acknowledged before the next one is sent. In self-timed circuits, these handshakes determine the rate of data transmission, and the round-trip delay incurred on each transmission-acknowledge cycle can limit performance. In a STARI interface, a global clock determines the rate of data transmission and the receiver's self-timed FIFO can compensate for skews exceeding several clock periods. In this way, STARI overcomes both the clock-skew limitations of purely synchronous designs and the round-trip delays of purely self-timed interfaces.

Figure 5.5 shows a STARI interface. The key component is a self-timed FIFO that receives data from a synchronous transmitter and delivers data to a synchronous receiver.

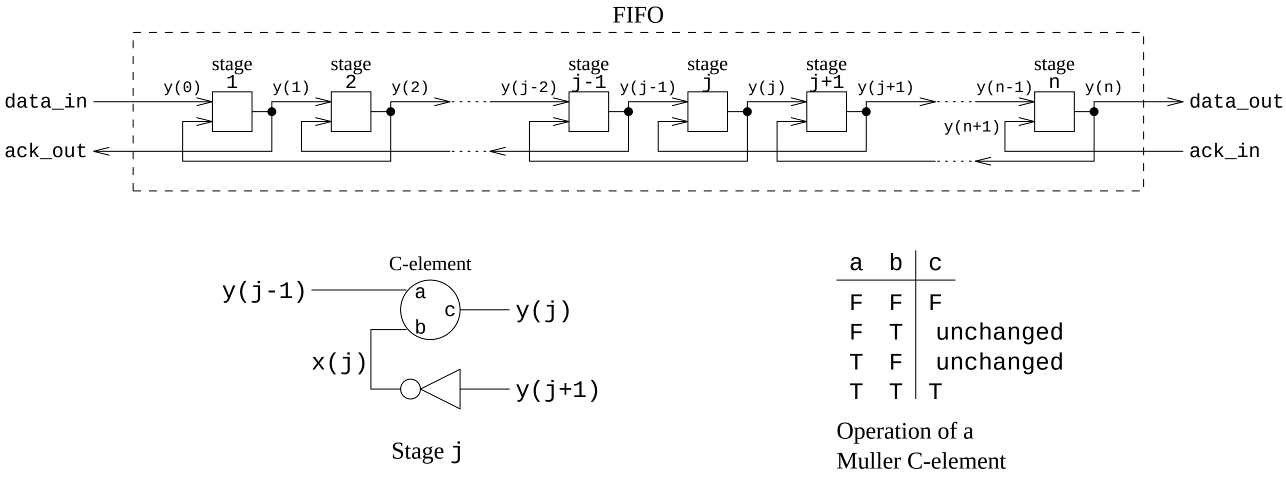
During each cycle (period of the global frequency reference), the transmitter sends one datum that is inserted into the FIFO upon arrival. Successive values are distinguished by using a self-timed data encoding [31]. Likewise, the receiver removes one item from the FIFO each cycle. Once properly initialized, the FIFO never overflows or underflows.

For correct operation, the FIFO must complete each insert and remove operation within one cycle. When this requirement is met, the FIFO appears as a synchronous component to both the transmitter and the receiver. Furthermore, both the transmitter and the receiver appear to the FIFO as well-behaved self-timed systems. The transmitter produces a new data value each clock cycle, just as if the FIFO were another component synchronous to its own clock. As will be shown, the FIFO performs each insert operation within one cycle, which means the FIFO acknowledges the previous data value before the next value arrives. Thus, the synchronous transmitter satisfies the self-timed signaling conventions of the self-timed FIFO. Likewise, a prompt response of the FIFO to acknowledgements from the receiver guarantees that the receiver does not issue an acknowledgement until the corresponding data value is present. Therefore, the interface between the FIFO and the receiver is correctly timed.

5.1.1 Self-timed FIFOs for STARI

To verify STARI, a particular FIFO implementation must be chosen. Consider an implementation that uses a ripple FIFO where successive data values in the FIFO are distinguished according to some self-timed encoding. For verification purposes, the analysis of the interface does not depend on the specific data encoding. The transmitter output can be modeled as alternating between two values: "full" (represented by true), and "empty" (represented by false).

A self-timed FIFO can be implemented using a linear array of stages with outputs `y(1)` ... `y(n)` which operate according to the following rule: stage `j` may copy its input,



$$\ll \texttt{y(j-1) = x(j)} \rightarrow \texttt{y(j).v := y(j-1).v} \gg$$
$$\| \ll \texttt{x(j) :=} \neg\ \texttt{y(j+1)} \gg$$

ST code for stage j (without auxiliary variables for timing).

Figure 5.6: A self-timed FIFO

`y(j-1)`, to its output, `y(j)`, when its successor stage has acquired its current value (i.e. `y(j)` = y(j+1)). Thus, when a stage and its successor are both empty, the stage can acquire a full value from its predecessor. Conversely, when a stage and its predecessor are both full, the stage can acquire an empty value from its predecessor. This protocol has a simple implementation consisting of a Muller C-element and an inverter as shown in Figure 5.6.

This design is delay insensitive [30] and will function correctly regardless of the delays in the C-elements, inverters, and wires as long as the transmitter and the receiver observe the self-timed protocol. In a self-timed design, these conditions for the transmitter and receiver are enforced by handshakes using the data and acknowledge signals. In a STARI interface, the time between when a stage is enabled to perform an action and when that action is taken must be bounded. For the schedules described below, the transmitter and receiver can be guaranteed to operate according to the self-timed protocol when each performs one operation during each cycle of the global clock. Because the transmitter does not require acknowledgements from the FIFO to send successive values, the performance of STARI is not limited by round-trip delays.

5.1.2 A schedule for STARI

To verify STARI, it is necessary to show that after each data value arrives from the transmitter, an `ack_out` event is generated by the FIFO before the next value from the transmitter arrives. Similarly, after each `ack_in` event from the receiver, the FIFO is required to output a new data value before the next `ack_in` event occurs. To perform a new operation, each FIFO stage must wait for data from its predecessor (or the transmitter) and an acknowledgement from its successor (or the receiver). These dependencies are transitive; therefore, the timing of each stage depends on the times of the operations of all stages and the transmitter and receiver. Accordingly, a global schedule for

FIFO operations is required to establish the correct operation of STARI. The schedules presented in this section are from [15]. These schedules are presented in an informal, intuitive fashion. In section 5.2, the version with bounded delays is formalized using the ST notation, and the verification of this version using the proof checker is described in sections 5.3 and 5.4.

The schedule of STARI depends on the model of the timing for the operation of the transmitter, receiver, and FIFO. The model used here assumes that the clock skew between the transmitter and receiver has some arbitrary, constant value. On the other hand, stage delays are only bounded from above. The actual delay of a stage can be anywhere from zero to this bound, and the stage can exhibit different delays for different operations. This model uses the quantities defined below:

`n`: The number of stages in the FIFO. Assume `n` > 0.

δ: The stage delay. The delay from when a stage has received both a new data value at its input and an acknowledgement for its current output until the stage outputs the new value is at most δ. In this model, δ is an upper bound, and the actual delay may differ for different stages or for successive operations of the same stage.

π: The period of the global clock. New data values arrive at `data_in` separated by exactly π time units, and successive acknowledgements arrive at `ack_in` separated by exactly π time units.

λ: The time from a transmitter event until the corresponding receiver event. In a correctly operating interface, the transmitter will output a value on `data_in` at some time, τ_t, and the receiver will assert an `ack_in` for this value at some later time τ_r. In this case, $\lambda = \tau_r - \tau_t$. Since successive transmitter events and successive receiver events occur with the same period, λ is a constant. For $(n+1)\delta < \lambda < (n+1)(\pi-\delta)$

it will be shown that the STARI interface operates correctly, in which case λ can be understood as the FIFO latency.

To motivate the schedule for FIFO operations, a simplistic scenario with fixed delays is considered first. With completely deterministic timing, the analysis for this case is straightforward, and many of the ideas from this simplified version can be applied directly to the bounded delay model and appear in the proof. In the fixed delay case, each stage performs an operation once every π time units, and the details of the schedule are determined by the relative phases of these operations. A convenient way to describe these relative phases is to derive the delays between when a stage receives a new value at its data input and when it propagates this value to its output.

The delay for stage $\mathtt{j}$ is written as $\Delta(\mathtt{j})$. The sum of these delays is the latency of the FIFO:

$$\sum_{\mathtt{j}=1}^{\mathtt{n}+1} \Delta(\mathtt{j}) \quad = \quad \lambda \tag{5.1}$$

Note that $\Delta(\mathtt{n}+1)$ corresponds to the delay from when a value is output by the FIFO until the subsequent acknowledgement is output by the receiver.

In steady state operation, the stages of the FIFO can be partitioned according to the order in which their data and acknowledge inputs arrive. The stages closest to the transmitter receive new data values after they have received an acknowledgement for the previous value from their successors. When a data value arrives at the input of such a stage, it is copied to the output δ time units later. For the stages closest to the receiver, data values arrive before acknowledgments. If a stage and its predecessor both wait for acknowledgements, then $\pi - \delta$ time unit elapses between the arrival and departure of a data value at the stage (see [16]). The remaining stage waits for acknowledgement but its predecessor waits for data. The time for this stage to forward a data value is bounded by the times for the other two cases.

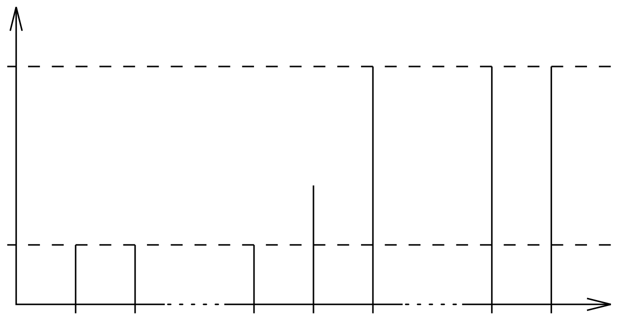


Figure 5.7: Stage-to-stage transfer times

Let stage $\mathtt{k}$ be the first stage that waits for acknowledgements. To satisfy equation 5.1 and the relationships $\delta < \Delta(\mathtt{j}) \leq \pi - \delta$, a simple pigeon-hole argument yields:

$$\Delta(\mathtt{j}) \quad = \quad \left\{ \begin{array}{ll} \delta & \text{, if } \mathtt{j} < \mathtt{k} \\ \alpha & \text{, if } \mathtt{j} = \mathtt{k} \\ \pi - \delta & \text{, if } \mathtt{k} < \mathtt{j} \end{array} \right\} \tag{5.2}$$

where

$$\begin{aligned} \mathtt{k} &= \mathtt{n} + 1 - \left\lfloor \frac{\lambda - (\mathtt{n}+1)\delta}{\pi - 2\delta} \right\rfloor \\ \alpha &= \lambda + (\mathtt{k} - 1) * (\pi - 2\delta) - \mathtt{n}(\pi - \delta) \end{aligned} \tag{5.3}$$

To ensure that $\mathtt{k}$ is between 1 and $\mathtt{n}$, it is required that

$$\begin{array}{rclcll} 0 & < & \delta & & & \text{, FIFO stages are causal} \\ 2\delta & < & \pi & & & \text{, minimum ``clock'' period} \\ (\mathtt{n}+1)\delta & < & \lambda & < & (\mathtt{n}+1)(\pi - \delta) & \text{, bounds on skew} \end{array}$$

Figure 5.7 shows $\Delta(\mathtt{j})$ for a typical STARI interface with fixed delays.

Now consider a FIFO with bounded delays. The delay between when a stage receives a new data value and when it outputs the value may be lower than in the fixed delay case. As the total latency of the FIFO remains fixed, there must also be stages which have delays greater than those in the fixed delay case. This happens when a stage receives a new data value earlier than it would have in the fixed delay version and must wait longer for an acknowledgement. It can be shown that in the bounded delay model no stage performs an operation later than the corresponding action is performed in the fixed delay version. This observation leads to the schedule for the bounded delay model.

The schedule for STARI with bounded delays is a schedule for the *total* delay from the time that a data token arrives at the input of the FIFO until it is output by stage $\mathtt{j}$ as given by $\Psi(\mathtt{j})$ defined below:

$$\Psi(\mathtt{j}) \;=\; \sum_{\mathtt{j}=1}^{\mathtt{i}} \Delta(\mathtt{i}) \tag{5.4}$$

Because Ψ is derived from Δ, this schedule identifies a "waiting for data" region ($\mathtt{j} < \mathtt{k}$ with $\mathtt{k}$ defined by equation 5.3) and a "waiting for acknowledgement" region ($\mathtt{j} \geq \mathtt{k}$). In operation, a stage in the "waiting for data" region may end up waiting for an acknowledgement because of a data value arriving early; however, it will not wait longer than the time allowed by the "waiting for data" schedule above. Likewise, a stage in the "waiting for acknowledge" region may wait for a data token, but not so long as to violate the schedule.

5.2 An ST Program for STARI

To verify the timing properties of STARI, the interface is modeled as an ST program. In this program, τ represents the current time and `y(0)` through `y(n+1)` represent signal values. For $1 \leq \mathtt{i} \leq \mathtt{n}$, `y(i)` is the output of the $\mathtt{i}^{\mathrm{th}}$ FIFO stage. The output of the transmitter is the signal `y(0)`, and `y(n+1)` is the "acknowledge" signal from the receiver. Three attributes are associated with each signal:

`y(i).v` The value of the logical datum output by FIFO stage `i`, true (full) or false (empty).

$\mathtt{y(i)}.\tau$ The time at which y(i).v was assigned its current value.

$\mathtt{y(i)}.\iota$ The time at which the value currently held by y(i).v was output by the transmitter.

Given this framework, the descriptions of the transmitter and receiver are straightforward. The transmitter changes the value of y(0).v once every π time units. The transition

$$\ll\ \tau \geq \mathtt{y(0)}.\tau + \pi \rightarrow \mathtt{y(0).v},\ \mathtt{y(0)}.\tau,\ \mathtt{y(0)}.\iota := \mathtt{NOT\ y(0).v},\ \tau,\ \tau\ \gg$$

states that changes of the transmitter's output, y(0), occur at least π time units apart. Likewise, the protocol,

$$\tau.post \ \leq\ \mathtt{y(0)}.\tau + \pi$$

ensures that changes of y(0) are at most π time units apart. Thus y(0) changes once every π time units as required. The description of the receiver is equivalent (see Figure 5.8).

To describe the FIFO, note that stage j can change its output when stage j-1 has provided a new input value and stage j+1 has acknowledged the current output. Thus, the transition for stage j of the FIFO is

$$\begin{array}{l} \ll\ (\mathtt{y(j\text{-}1).v} \neq \mathtt{y(j).v})\ \mathtt{AND}\ (\mathtt{y(j).v} = \mathtt{y(j\text{+}1).v}) \\ \quad \rightarrow \mathtt{y(j).v},\ \mathtt{y(j)}.\tau,\ \mathtt{y(j)}.\iota := \mathtt{y(j\text{-}1).v},\ \tau,\ \mathtt{y(j\text{-}1)}.\iota \\ \gg \end{array}$$

The entire FIFO is described by the asynchronous composition

$$\mathop{\Big\|}_{\mathtt{j}=1}^{\mathtt{n}} \begin{array}{l} \ll\ (\mathtt{y(j\text{-}1).v} \neq \mathtt{y(j)})\ \mathtt{AND}\ (\mathtt{y(j).v} = \mathtt{y(j\text{+}1).v}) \\ \quad \rightarrow \mathtt{y(j).v},\ \mathtt{y(j)}.\tau,\ \mathtt{y(j)}.\iota := \mathtt{y(j\text{-}1).v},\ \tau,\ \mathtt{y(j\text{-}1)}.\iota \\ \gg \end{array}$$

No timing constraints are included in the guard because the FIFO stages only have an upper bound on their delays. A stage is allowed to perform its operation immediately after receiving new data and acknowledge inputs.

The following protocol asserts that the transition for FIFO stages can be enabled for at most δ time units before being executed:

$$\begin{array}{l} \forall \mathtt{j} \in \{1 \ldots \mathtt{n}\}\,. \\ \quad ((\mathtt{y(j-1).v} \neq \mathtt{y(j).v}) \wedge (\mathtt{y(j).v} = \mathtt{y(j+1).v})) \\ \Rightarrow \quad (\tau.post < \max(\mathtt{y(j-1)}.\tau,\ \mathtt{y(j+1)}.\tau) + \delta) \end{array}$$

The complete program for STARI is given in figure 5.8.

5.2.1 The invariant

To verify STARI, it is necessary to show that the self-timed protocol of the FIFO is satisfied by the real-time behavior of the transmitter, FIFO, and receiver. The first criterion is that each value output by the transmitter is inserted into the FIFO before the transmitter outputs another value. Formally, let

$$R_1 \triangleq (\tau \geq \mathtt{y(0)}.\tau + \pi) \Rightarrow (\mathtt{y(0).v} = \mathtt{y(1).v})$$

R_1 is a safety property of the program. The second criterion is that the corresponding condition for the receiver,

$$R_2 \triangleq (\tau \geq \mathtt{y(n+1)}.\tau + \pi) \Rightarrow (\mathtt{y(n).v} \neq \mathtt{y(n+1).v})$$

R_2 is also a safety property.

To verify properties R_1 and R_2, an invariant of the program is established, I such that $I \Rightarrow (R_1 \wedge R_2)$. The key clause of this invariant is a schedule for the internal operations of the FIFO. In particular, the invariant includes the conjunct

$$\forall \mathtt{i} \in \{1 \ldots \mathtt{n}\}.\ \ \mathtt{y(i)}.\tau < \mathtt{y(i)}.\iota + \Psi(\mathtt{i})$$

Intuitively, $\Psi(\mathtt{i})$ is the maximum time allowed for a value to propagate from the transmitter to the output of stage $\mathtt{i}$. The key property of the schedule Ψ that will be used in

Constraints on program parameters:

$$\begin{array}{rcccll} 0 & < & \mathtt{n} & & & \text{, there is a FIFO} \\ 0 & < & \delta & & & \text{, FIFO stages are causal} \\ 2\delta & < & \pi & & & \text{, minimum "clock" period} \\ (\mathtt{n}+1)\delta & < & \lambda & < & (\mathtt{n}+1)(\pi-\delta) & \text{, bounds on skew} \end{array}$$

Transitions for the transmitter, FIFO, and receiver:

$$\begin{array}{ll} & \ll \quad \tau \geq \mathtt{y(0)}.\tau \;\mathtt{+}\; \pi \\ & \quad \rightarrow \mathtt{y(0).v},\ \mathtt{y(0)}.\tau,\ \mathtt{y(0)}.\iota \;\mathtt{:=}\; \mathtt{NOT\ y(0).v},\ \tau,\ \tau \\ & \gg \\ \| & \mathop{\|}_{\mathtt{j=1}}^{\mathtt{n}} \ll \quad (\mathtt{y(j-1).v} \neq \mathtt{y(j).v}) \wedge (\mathtt{y(j).v} = \mathtt{y(j+1).v}) \\ & \quad\quad \rightarrow \mathtt{y(j).v},\ \mathtt{y(j)}.\tau,\ \mathtt{y(j)}.\iota \;\mathtt{:=}\; \mathtt{y(j-1).v},\ \tau,\ \mathtt{y(j-1)}.\iota \\ & \quad\quad \gg \\ \| & \ll \quad \tau \geq \mathtt{y(n+1)}.\tau \;\mathtt{+}\; \pi \\ & \quad \rightarrow \mathtt{y(n+1).v},\ \mathtt{y(n+1)}.\tau,\ \mathtt{y(n+1)}.\iota \;\mathtt{:=}\; \mathtt{NOT\ y(n+1).v},\ \tau,\ \mathtt{y(n)}.\iota \\ & \gg \end{array}$$

Protocol for the environment (i.e. assumptions about time):

$$\begin{array}{l} \mathit{unchanged}(\mathtt{y}) \\ \tau.\mathit{post} \;\geq\; \tau.\mathit{pre} \\ \tau.\mathit{post} \;\leq\; \mathtt{y(0)}.\tau + \pi \\ \;\forall \mathtt{i} \in \{1 \ldots \mathtt{n}\}\mathtt{.} \\ \quad\quad (\mathtt{y(i-1).v} \neq \mathtt{y(i).v}) \wedge (\mathtt{y(i).v} = \mathtt{y(i+1).v}) \\ \quad \Rightarrow \; \tau.\mathit{post} < \max(\mathtt{y(i-1)}.\tau,\, \mathtt{y(i+1)}.\tau) + \delta \\ \tau.\mathit{post} \;\leq\; \mathtt{y(n+1)}.\tau + \pi \end{array}$$

Figure 5.8: A Synchronized Transitions program for STARI

$$
\begin{array}{rcl}
\mathtt{k} & = & \mathtt{n}+1-\left\lfloor\dfrac{\lambda-(\mathtt{n}+1)\delta}{\pi-2\delta}\right\rfloor \\[2ex]
\Delta(\mathtt{i}) & = & \begin{cases}\delta & \text{, if } \mathtt{i}<\mathtt{k} \\ \lambda+(\mathtt{k}-1)*(\pi-2\delta)-\mathtt{n}(\pi-\delta) & \text{, if } \mathtt{i}=\mathtt{k} \\ \pi-\delta & \text{, if } \mathtt{i}>\mathtt{k}\end{cases} \\[2ex]
\Psi(\mathtt{i}) & = & \sum_{\mathtt{j}=1}^{\mathtt{i}}\Delta(\mathtt{j}) \\[2ex]
I_{\text{sched}} & = & \forall \mathtt{i}\in\{1\ldots\mathtt{n}\}.\ \ \mathtt{y(i)}.\tau<\mathtt{y(i)}.\iota+\Psi(\mathtt{i}) \\
I_{\text{causal}} & = & \forall \mathtt{i}\in\{0\ldots\mathtt{n}+1\}.\ \ \mathtt{y(i)}.\tau\leq\tau \\
I_{\lambda} & = & (\mathtt{y(0)}.\tau=\mathtt{y(0)}.\iota)\wedge(\mathtt{y(n+1)}.\tau=\mathtt{y(n+1)}.\iota+\Psi(\mathtt{n}+1)) \\
I_{t} & = & \tau\leq\mathtt{y(0)}.\tau+\pi \\
I_{f} & = & \forall \mathtt{i}\in\{1\ldots\mathtt{n}\}. \\
 & & \qquad (\mathtt{y(i-1)}.\mathtt{v}\neq\mathtt{y(i)}.\mathtt{v})\wedge(\mathtt{y(i)}.\mathtt{v}=\mathtt{y(i+1)}.\mathtt{v}) \\
 & & \quad\Rightarrow\ (\tau<\max(\mathtt{y(i-1)}.\tau,\mathtt{y(i+1)}.\tau)+\delta) \\
I_{r} & = & \tau\leq\mathtt{y(n+1)}.\tau+\pi \\
I_{\text{insert}} & = & \forall \mathtt{i}\in\{0\ldots\mathtt{n}\}. \\
 & & \qquad (\mathtt{y(i)}.\mathtt{v}=\mathtt{y(i+1)}.\mathtt{v})\Rightarrow(\mathtt{y(i)}.\iota=\mathtt{y(i+1)}.\iota) \\
 & & \quad\wedge\ (\mathtt{y(i)}.\mathtt{v}\neq\mathtt{y(i+1)}.\mathtt{v})\Rightarrow(\mathtt{y(i)}.\iota=\mathtt{y(i+1)}.\iota+\pi) \\[2ex]
I & = & I_{\text{sched}}\wedge I_{\lambda}\wedge I_{\text{causal}}\wedge I_{t}\wedge I_{f}\wedge I_{r}\wedge I_{\text{insert}}
\end{array}
$$

Figure 5.9: The invariant for STARI

the remainder of this chapter is

$$\forall \mathtt{i}\in\{1\ldots\mathtt{n}+1\}.\ \ \delta\leq\Psi(\mathtt{i})-\Psi(\mathtt{i}-1)\leq\pi-\delta.$$

Note that $\Psi(\mathtt{i})-\Psi(\mathtt{i}-1)$ is denoted $\Delta(\mathtt{i})$

The complete invariant is shown in figure 5.9. Each of the clauses has a simple, intuitive interpretation. As is often the case with invariant based verification, several "bookkeeping" clauses are needed to describe the set of states that the system can reach. As described in the previous paragraph, the clause I_{sched} gives a schedule for the internal operations of the FIFO. The clause I_{causal} states that the FIFO is causal as described by the auxiliary variables: no signal may have an assignment time that is in the future. The clause I_λ asserts that the schedule is tight at the transmitter and receiver, which implies that the FIFO latency is λ, matching λ's intuitive interpretation. The clauses I_t, I_f, and I_r state that the transmitter, FIFO, and receiver respectively have completed all operations that should have happened in the past. The clause I_{insert} can be understood by assuming that no data values are dropped by the FIFO, in which case this clause implies that the values of the $.\iota$ variables are the times at which these values were output by the transmitter.

The clause I_{insert} is implied by the other clauses of the invariant and can be proven by induction over the stages of the FIFO. This approach was taken in the manual proof in [16]. However, when verified by the proof checker, many implicit induction arguments were discovered in the hand-written proof, most of them simple lemmas about unchanged variables. By adding the clause I_{insert} to the invariant, the arguments by induction over the structure of the FIFO become induction arguments over the sequences of states that the system can traverse. The proof was more easily verified by the latter approach.

5.3 The STARI Proof

With the described technique, STARI was modeled as a concurrent program, safety properties of the system were identified, and a predicate, I, which implies these safety properties was proven to be the invariant of the program. To stimulate an appreciation of the process, this section discusses one segment of the proof in detail, highlights some techniques used in constructing the proof, and presents some of the flaws uncovered in the manual proof.

5.3.1 A snapshot from the proof

As mentioned in chapter 2, a proof can be viewed as a tree with the claim as the root, proof-rules labeling the edges, and simple tautologies at the leaves. The STARI proof is mapped into such structure. The root is the claim that I is an invariant of the program shown in figure 5.8. It can be written as

$$
\begin{aligned}
& (I(s_1) \wedge C_t(s_1)) \Rightarrow I(M_t(s_1)) \\
\bigwedge \quad & (I(s_1) \wedge C_f(s_1)) \Rightarrow I(M_f(s_1)) \\
\bigwedge \quad & (I(s_1) \wedge C_r(s_1)) \Rightarrow I(M_r(s_1)) \\
\bigwedge \quad & (I(s_1) \wedge C_p(s_1)) \Rightarrow I(M_p(s_1))
\end{aligned}
$$

The `PC_rule` splits the claim into four separate clauses. Viewed as a tree, there is the claim at the root with four edges, labeled `PC_rule`, splitting it into its four children:

$$
\begin{aligned}
& (I(s_1) \wedge C_t(s_1)) \Rightarrow I(M_t(s_1)), \\
& (I(s_1) \wedge C_f(s_1)) \Rightarrow I(M_f(s_1)), \\
& (I(s_1) \wedge C_r(s_1)) \Rightarrow I(M_r(s_1)), \\
and \quad & (I(s_1) \wedge C_p(s_1)) \Rightarrow I(M_p(s_1)).
\end{aligned}
$$

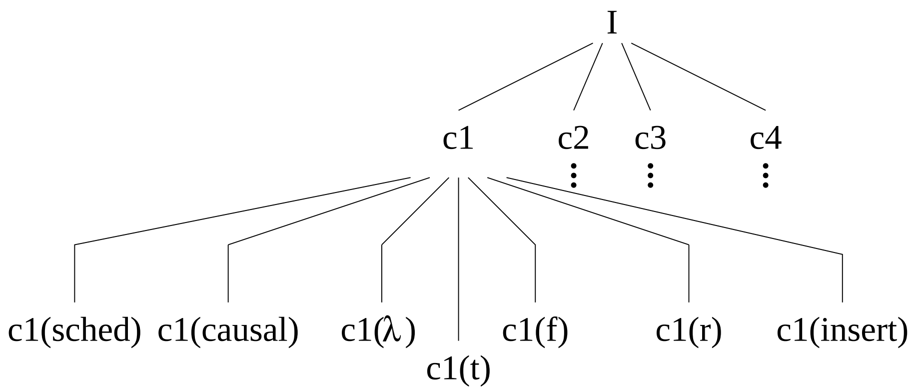


Figure 5.10: A branch from the STARI proof tree.

Since the invariant is a conjunction of several clauses, each of the above obligations can be further broken down into

$$\begin{array}{rll}
 & (I(s_1) \wedge C_i(s_1)) \Rightarrow & I_{\text{sched}}(M_i(s_1)), \\
 & (I(s_1) \wedge C_i(s_1)) \Rightarrow & I_{\text{causal}}(M_i(s_1)), \\
 & (I(s_1) \wedge C_i(s_1)) \Rightarrow & I_{\lambda}(M_i(s_1)), \\
 & (I(s_1) \wedge C_i(s_1)) \Rightarrow & I_{t}(M_i(s_1)), \\
 & (I(s_1) \wedge C_i(s_1)) \Rightarrow & I_{f}(M_i(s_1)), \\
 & (I(s_1) \wedge C_i(s_1)) \Rightarrow & I_{r}(M_i(s_1)), \\
and & (I(s_1) \wedge C_i(s_1)) \Rightarrow & I_{\text{insert}}(M_i(s_1)). \\
 & & where\ i \in t, f, r, p.
\end{array}$$

See figure 5.10.

This section examines the branch of the proof which verifies that the transitions for the FIFO maintain I_{sched}, $(I(s_1) \wedge C_f(s_1)) \Rightarrow I_{\text{sched}}(M_f(s_1))$, the clause of the invariant that asserts the real-time schedule for STARI. This example is chosen as it emphasizes the real-time aspects of the verification.

Recall that a proof state consists of the claim, the hypothesis list, the obligation list, and the postponed list. (See section 2.1.1). For simplicity, only the obligations and the hypotheses are shown in this presentation. Starting with the obligation given above, $I_{\text{sched}}(M_f(s_1))$ is skolemized with the skolem constant *sk_i* so we can apply case analysis on the term. Using `PC_rule` to apply case analysis on (sk_i=j), we split our obligation into 2 terms.

$$
\begin{aligned}
&\text{obligations :}\\
&\quad (sk\_i = j) \Rightarrow (\\
&\qquad\qquad\qquad (\text{if}(sk\_i = j) \text{ then } \tau \text{ else } y(sk\_i).\tau)\\
&\qquad\qquad\quad < (\text{if}(sk\_i = j) \text{ then } y(sk\_i - 1).\iota \text{ else } y(sk\_i).\iota) + \Psi(sk\_i)),\\
&\quad (sk\_i \neq j) \Rightarrow (\\
&\qquad\qquad\qquad (\text{if}(sk\_i = j) \text{ then } \tau \text{ else } y(sk\_i).\tau)\\
&\qquad\qquad\quad < (\text{if}(sk\_i = j) \text{ then } y(sk\_i - 1).\iota \text{ else } y(sk\_i).\iota) + \Psi(sk\_i))\\
&\text{hypotheses :}\\
&\quad [\mathrm{C_f},\ \mathrm{I},\ \text{constraint}]
\end{aligned}
$$

The second obligation follows directly from the fact that no variables appearing in the obligation were modified by the transition. In the checker, this takes six steps: two steps (using `EQ_rule` and `IF_rule`) replace each *if* expression with the corresponding *else* clause; two steps (using `Definition_rule` and `Instantiate_rule`) instantiate the corresponding clause of the invariant on the hypothesis list with j; and the remaining two steps (using `PC_rule`) perform rewrites to put the obligations into forms suitable for the other rules. Having discharged the simpler of the two obligations, the state of the proof becomes

$$
\begin{aligned}
&\text{obligations :}\\
&\quad (sk\_i = j) \Rightarrow (\\
&\qquad\qquad (\text{if}(sk\_i = j) \text{ then } \tau \text{ else } y(sk\_i).\tau)\\
&\qquad\qquad < (\text{if}(sk\_i = j) \text{ then } y(sk\_i - 1).\iota \text{ else } y(sk\_i).\iota) + \Psi(sk\_i))\\
&\text{hypotheses :}\\
&\quad [\mathrm{C_f},\ \mathrm{I},\ \text{constraint}]
\end{aligned}
$$

Using `EQ_rule` and `IF_rule`, the obligation is rewritten to $\tau < y(j-1).\iota + \Psi(j)$. To obtain an upper bound for τ, the clause I_f of the invariant is used. Since the precondition C_f holds, instantiating I_f with $i = j$ yields $\tau < \max(y(j-1).\tau,\ y(j+1).\tau) + \delta$. This leads to the essence of the real-time verification of STARI. The current time is bounded from above according to the greater of $y(j-1).\tau$ and $y(j+1).\tau$, that is according to whether the data or acknowledge input of stage j arrived last. In either case, the schedule, holding for stages $j-1$ and $j+1$ before performing the transition for stage j, is used to show that it holds for stage j after the transition is performed.

Case analysis is performed according to which of $y(j-1).\tau$ and $y(j+1).\tau$ is greater. The case for $y(j-1).\tau > y(j+1).\tau$ is presented here; the other case is similar. For the case with $y(j-1).\tau > y(j+1).\tau$, rewriting the max function yields the proof obligation

$$\tau < y(j-1).\tau + \delta \quad \Rightarrow \quad \tau < y(j-1).\iota + \Psi(j) \qquad (o_1)$$

Using linear programming, it can be shown that:

$$
\begin{array}{rcll}
\tau & < & y(j-1).\tau + \delta & (L_0)\\
y(j-1).\tau & \leq & y(j-1).\iota + \Psi(j-1) & (L_1)\\
\Psi(j) & = & \Psi(j-1) + \Delta(j) & (L_2)\\
\Delta(j) & \geq & \delta & (L_3)\\
\hline
\tau & < & y(j-1).\iota + \Psi(j) & (L_4)
\end{array}
$$

In the checker, `PC_rule` is used to replace proof obligation o_1 with L_1, L_2, L_3, and $L_1 \wedge L_2 \wedge L_3 \Rightarrow L_4$. (Note that L_0 is the antecedent of o_1 and L_4 is the consequent.) This implication is discharged immediately by `LP_rule`, demonstrating the utility of a decision procedure for linear programming when reasoning about real-time systems. Of the remaining obligations, the first two can be discharged by instantiating the appropriate hypotheses. The fourth obligation, however, reveals a limitation of this checker. Instantiating the definition of Δ it is straightforward to verify that $\Delta(j) \geq \delta$ for the cases $j < k$ and $j > k$. On the other hand, the case $j = k$ produces the obligation

$$\delta \;\; \leq \;\; \lambda + (k-1) * (\pi - 2\delta) - n(\pi - \delta)$$

To verify this obligation, it is necessary to instantiate the definition of k, and then reason about the inequalities involving non-linear operations such as floor. Instead, the `Postponement_rules` are used to transfer this obligation to the suppose list. At the end of the proof, there are two such obligations on the suppose list, the one just described and the closely related one:

$$\lambda + (k-1) * (\pi - 2\delta) - n(\pi - \delta) \;\; < \;\; \pi - \delta$$

Both can be verified in a few minutes using pencil, paper, and a little bit of high-school algebra.

5.3.2 Some Proof Techniques

The `LP_rule` and the `PC_rule` are the core of the proof checker. The `LP_rule` is intended to reason about linear inequalities within the proof, and the `PC_rule` is intended to support boolean manipulation. While developing the STARI proof script, it was observed that in addition to the originally intended functions, these two proof rules are used extensively to restructure expressions to the forms required by other proof rules. Fixed sequences of proof rules are applied to achieve certain subgoals. By examining the proof scripts, common patterns were recognized and interface functions were built to capture these proof sequences. The following paragraphs present some proof techniques involved in developing the STARI proof script, and discuss the interface functions implemented to support such techniques.

One frequently used proof technique is *case analysis*: to divide an obligation into different cases and reason about each case with an appropriate method. See sections 3.5.1 and 3.5.2 for a description of the interface functions `CaseAnalysis1` and `CaseAnalysis2`.

Because many of the proof rules are implemented using structural matching of expressions; proof states can include cumbersome obligations that require simplification. For example, the `Instantiate_rule` traverses an expression tree and blindly replaces every occurrence of the quantifier by the specified instant without recognizing what the expression truly represents. The expression $\forall i.f(i+1)$ instantiated by $j-1$ yields $f(j-1+1)$, and the `LP_rule` is required to verify $(j-1+1)=j$ before the replacement can be applied. In such cases, the `PC_rule` is used to formulate the simplified expression, the `LP_rule` to verify such replacement, and the `EQ_rule` to write the expression into the simpler form. This process could be extremely tedious when large expressions are involved. Such tedious steps can be avoided by adding an arithmetic decision procedure to the proof checker. This approach is discussed in section 6.1.

An invariant proof verifies that a predicate continues to hold after a state change in which value to some $v \in V$ is modified by a transition (given that V is the set of variables in the ST program). Such a predicate is often a conjunction of clauses, and the predicate is proven to be an invariant by analyzing each clause and the effect of each transition on these clauses. Frequently no variable in a clause is changed after a particular transition is performed, and this case is readily discharged by the `PC_rule`. Many times, the obligation which needs to be proven is a simple instantiation of a clause from the hypotheses. The interface function `Unchanged` discharges such obligations (see section 3.5).

5.3.3 Flaws from Manual Proof

Having attempted to verify a hand-written proof, various technical flaws were uncovered. Most of the errors uncovered were typographical mistakes and inadequate justifications for proof steps, and most of these flaws were found in the process of translating the manual proof steps into inputs to the proof checker. All of these errors were in the proof, not in the theorem statement. A more serious error was revealed when attempting to verify

$$\delta \;\; \leq \;\; \lambda + (k-1) * (\pi - 2\delta) - n(\pi - \delta) \;\; < \;\; \pi - \delta$$

In the original formulation of the claim, there was an "off-by-one" error in the definition of k. Although this was not detected by the proof checker, the checker brought it to our attention by reducing the correctness of the claim to the correctness of this simple formula, upon the examination of which the error was discovered.

5.4 Observations and Experiences

From verifying STARI a few observations are noted in relation of the proof checker and the proof itself. Two of the major issues are: the similarity in the overall structure of the verified proof and the manual proof, and how the chosen language, FL, helps the proof development process.

5.4.1 Verified Proof versus Manual Proof

The overall structure of the STARI proof verified by the proof checker is the same as that of the manual proof. Both version verify the invariant by considering pairs of states, s and s' where s' is produced by performing some transition from state s. The invariant is assumed to hold in state s, and the proofs show that it continues to hold in state s'. In both proofs, the invariant consists of four clauses: three corresponding to the three transitions representing the transmitter, receiver and FIFO; and one clause corresponding to the protocol of the ST program. The central argument of each proof shows that the FIFO maintains the schedule, and both proofs do this by considering whether the data or the acknowledgement arrives last at each C-element. Because the invariant holds in state s, the schedule holds for this last arriving input. The proofs then show that the invariant holds in state s'.

The major difference between the manual proof and the verified proof is the change in representation of the insert time. As mentioned earlier, the manual proof involves implicit inductions which correspond to many proof steps in their counterparts within the verified proof. In the verified proof, these induction arguments were eliminated by adding the clause I_{insert} to the invariant. Although this clause is redundant (it is implied by the other clauses), the overall proof is simplified by its inclusion.

The number of steps required for the same argument differs in the manual proof and the verified proof. The verified version often involves more steps because of the rigorous nature of a verified proof. However, this is not necessarily the case where reasoning about linear inequalities and boolean manipulation is involved. The two built-in decision procedures for systems of linear inequalities and boolean manipulation discharge such proof obligations in a single step, whereas the manual proof requires multiple steps to simplify the expressions to a manageable size before they can be discharged.

5.4.2 FL as a meta-language

FL was the natural choice for the meta-language of the proof checker, because it provides built-in support for BDDs and supports abstract data types. It is easy to define a next-state function in a functional language. Using FL as the meta-language, it is very natural to pass a proof state to a function, have the inference rule create a new state with respect to the context of the proof rule and the input state, then return this new state. The language also allows a proof to be viewed as a program, a more structured entity which helps organize the proof. Related sections of a proof script can be combined to become a function. Branches from a proof tree which employ the same proof technique, can be written as a function with input variables to adapt to slight variations in similar cases. When similar proof sequences are required, the function is simply called with appropriate values.

Using a functional language to implement the proof checker also has disadvantages. It is harder to build and modify complex data structures, and hence long lists have to be traversed linearly during a lookup for an item. This is a performance problem which can be solved by more sophisticated programming.

5.5 Evaluating the Proof and the Proof Checker

The motivation for this research is that hand written proofs often contain implicit assumptions and unstated arguments, both of which can lead to errors and unsoundness. Theorem provers can be used to verify such proofs, however, existing theorem provers are often extremely tedious and/or require mathematically sophisticated users. Theorem provers which reduce all claims to a small set of fundamental axioms are often tedious [7]. Those that use sophisticated tactics that may allow for shorter proofs can be baffling to a naive user [10, 22, 26, 27, 3, 4].

The hypothesis of this research is that correctness proofs for real-time systems can be machine checked using a small set of decision procedures, and these procedures can be used at a level of detail comparable to typical hand proofs. A simple proof checker was implemented specifically for the verification of real-time systems. A real-time system

(STARI) was verified with the small set of inference rules. The structure of the verified proof is at a level of detail comparable to typical hand proofs.

From this exercise, the STARI proof was made more sound. As described in section 5.3, an off-by-one error in the statement, and various technical flaws were uncovered in the process as mentioned in section 5.3.

Chapter 6

Conclusion

This thesis has presented a technique for verifying timing properties for real-time systems. The system is modeled as an ST program; real-time requirements are formulated as safety properties; and a simple proof checker with a small set of inference rules is used to verify manually generated proofs of these properties. This chapter compares the results from this thesis with its initial conjectures, presents some of the unanticipated findings of this investigation, and highlights a few of the most significant findings.

6.1 The Simple Approach to Proof Checking

A simple proof checker was implemented. With its ten inference rules and small type system, the checker is powerful enough to verify real-time properties of concurrent designs such as STARI.

The initial design of the proof checker had nine proof rules: the rules currently existing in the checker except the `Induction_rule`. While translating the hand-written STARI proof to input to the proof checker, many implicit induction steps were discovered. It appeared that these could be eliminated by modifying the invariant. However, two induction arguments remained for proving two critical lemmas. Therefore, the `Induction_rule` was implemented and incorporated into the system. No further extension to the proof checker was needed to verify the STARI proof. The author believes that many proofs of real-time properties are based on similar arguments using predicate calculus, systems of linear inequalities, and simple quantifications over the integers.

Identity properties	:	$(a + 0 = a) \wedge (a * 1 = a)$
Cancellation law	:	$(a * c = b * c) \wedge (c \neq 0)$
		$\Rightarrow a = b.$

Figure 6.11: Identity Properties and Cancellation Law of reals

Unlike typical theorem provers, the type system in this checker is small. It includes `boolean`, `integer`, and `real` types as well as arrays of these three types. This type system is sufficient for the STARI proof, since the model for STARI does not require sophisticated data structures. In the manual proof of STARI, the FIFO is represented as an array of records: the output of a C-element is represented by $\{y(i).\tau,\ y(i).v,\ y(i).\iota\}$. In the proof checker, this is translated into three arrays. To verify systems with more elaborate types, the set of data types provided by the checker may be insufficient and require extension.

As mentioned in previous chapters, the `LP_rule` was used extensively for algebraic manipulation while verifying the STARI proof. Multiple proof steps are required to arrange an obligation into a structure that can be discharged by the `LP_rule` and to rewrite other obligations with the appropriate substitutions. (See section 5.3.2). This approach can become extremely tedious as the expressions grow. One possible enhancement to the proof checker is to implement a decision procedure for polynomial arithmetic. Such a decision procedure, given an expression, would simplify the expression and rewrite it into a canonical form of sums of products. The design of this decision procedure needs to be carefully considered to avoid introducing 'surprises'. (See section 2.1.2). Inference rules which capture the *identity properties* and the *cancellation law* of reals could also prove useful to the checker. Figure 6.11 states these two properties of reals.

6.2 Proofs as Programs

Using the proof checker, it was observed that a proof can be viewed as a program. For many years, people have written long programs where syntactic and type correctness is verified by a compiler. This allows programmers to concentrate on the algorithms and not tedious typing and syntactic issues. In the case of proofs, the proof checker allows users to concentrate on developing the proofs with the checker flagging unsound arguments. This approach allows users to focus on the high level structure of the proofs.

Commonly used proof sequences can often be encapsulated in a function which is called with different arguments to provide similar arguments within a proof. This approach is similar to implementing interface functions to the checker except that it is intended to be more problem specific. It avoids repetition, reduces the amount of code involved, and increases the readability of the proof script.

Many existing theorem provers maintain libraries of verified lemmas which can be reused in different proofs. HOL [7] is an example of such a theorem prover. A large amount of extra work is often required to identify a suitable set of hypotheses when creating such a lemma, and when the lemma is applied, more work may be required to show that these hypotheses are satisfied. As an alternative to instantiating lemmas, the proof checker presented here allows an interface function to be executed every time a similar argument is needed. If the function provides a correct proof, the obligation is discharged. Although there is some lemma corresponding to the class of predicates discharged by the function, the statement of this lemma is implicit, sparing the user the tedium of deriving and justifying a formal statement of the lemma. Re-executing the interface function increases the execution time for a proof; however, the built-in decision procedures make the checker fast enough that this trade-off is justifiable.

Using a traditional theorem prover, a small change at one step can cause a large change in the expressions produced by proof tactics or rewriting heuristics leading to a failure in another part of the proof. In other words, a small change can lead to divergence from the original proof. In our proof checker, the user provides the rewritten forms for obligations at each step, and this tends to prevent such divergence. Often, functions are written to compute these rewritten forms. Like a well-structured program, a well-structured proof has well defined interfaces between the different functions and modules, and these interfaces make proofs robust to incremental changes.

The observation that proofs can be viewed as programs suggests that a proof debugger could be implemented along the lines of a traditional program debugger: single stepping through functions, printing variables, and tracing back after a step is executed. Because the proof checker is implemented on top of a purely functional language, backward execution should also be possible. Tracing proof steps when a rule fails accounts for a large fraction of the time required to develop a proof. A debugger for the proof checker which allows users to single step an interface function and displays subexpressions within a proof state could benefit proof development.

6.3 The Postponement Rules

The `Postponement_rules` were introduced to the proof checker before the checker was fully developed. Proof obligations which could not be discharged by the incomplete proof checker were moved to the postponed list and retrieved back onto the obligation list after the appropriate rules were implemented. While experimenting with these three rules, it was discovered that they can be used to provide a lemma mechanism to the checker, to construct proofs with more structured layout, and to allow users to refer to obligations by name instead of by their index.

As mentioned in the previous section, a lemma can be specified as a function and the function can be executed whenever the lemma is needed. Alternatively, the corresponding obligation can be moved to the postponed list the first time the lemma is needed. This lemma can be applied from this list for each subsequent use. After the last use, the postponed lemma can be moved back to the obligation list to be discharged with one sequence of proof steps.

These rules also allow the user to postpone tedious steps in the proof, sketch out the structure of proof, then retrieve and verify one piece of proof at a time. As a result the proof becomes more structured and readable. Each postponed object in the list is tagged with a name. By postponing all obligations and retrieving only the obligation currently being worked on, users can work with names instead of indices.

6.4 Variable skew version of STARI proof

The STARI proof described in chapter 5 verifies a model of STARI which assumes that the clock skew between the transmitter and receiver has some arbitrary, constant value. A more ambitious proof verifying the variable skew model of STARI is under development. Functions are implemented to substitute similar proof sequences, and the `Postponement_rules` are used extensively in the proof.

6.5 Summary

A simple proof checker was implemented on top of the functional language FL. With a small set of inference rules and a simple type system, it is powerful enough to verify real-time properties of a communication protocol, STARI. An "off-by-one" error was discovered in the hand-written proof. The design decision that requires the user to provide replacements for obligations and the `Postponement_rules` distinguish this checker from traditional theorem provers. They allow users to view proofs generated from the checker as programs. By providing decision procedures for predicate calculus and systems of linear inequalities, the checker allows the verified proof to closely follow the structure of a manual proof. The simplicity of the checker maintains the overall structure of a manual proof in its certified version.

Bibliography

[1] Martín Abadi and Leslie Lamport. Composing Specifications. In J.W. de Bakker et al., editors, *Proceedings of the REX Workshop, "Stepwise Refinement of Distributed Systems"*. Springer-Verlag, 1989. LNCS 430.

[2] Flemming Andersen, Kim Dam Petersen, and Jimmi S. Pettersson. Program Verification using HOL-UNITY (Progress Report). In *HUG '93: HOL User's Group Workshop*, pages 1–17, UBC, Vancouver, 1993.

[3] Robert S. Boyer and J. Strother Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. Technical Report ICSCA-CMP-44, Institute for Computing Science and Computer Applications, University of Texas, January 1985.

[4] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[5] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] R.M. Burstall. Research in Interactive Theorem Proving at Edinburgh University. *LFCS-Department of Computer Science, University of Edinburgh*, October 1986.

[7] Cardell-Oliver, Herbert, and Joyce. UBC HOL Course, June 1990. Lecture Notes from UBC HOL Course, 4-8 June 1990.

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

[9] Colin Clark. *Elementary Mathematical Analysis*. Wadsworth Publishers, California, 1982.

[10] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, pages 287–305, Bad Herrenalb (Blackforest), Germany, September 1994. Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication 4/94.

[11] David L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Proceedings of the International Workshop on Verification of Finite State Systems (LNCS)*, Berlin, 1989. Springer-Verlag.

[12] Karen A. Frenkel. An interview with Robin Milner. *Communications of the ACM*, 36(1):90–95, January 1993.

[13] S.J. Garland and J.V. Guttag. An Overview of LP: the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications.* Springer-Verlag, 1989.

[14] David M. Goldschlag. Mechanically Verifying Safety and Liveness Properties of Delay Insensitive Circuits. *Formal Methods in System Design*, 5:207–225, 1994.

[15] Mark R. Greenstreet. Using Synchronized Transitions for Simulation and Timing Verification. In Jørgen Staunstrup and Robin Sharp, editors, *1992 Workshop on Designing Correct Circuits*, pages 215–236, Lyngby, Denmark, January 1992. Elsevier. An earlier version published as Matsushita Information Technology Laboratory technical report MITL-TR-01-91.

[16] Mark R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication.* PhD thesis, Department of Computer Science, Princeton University, 1993.

[17] A.G. Hamilton. *Logic for Mathematicians.* Cambridge University Press, Cambrige, 1988.

[18] John Harrison. A HOL Decision Procedure for Elementary Real Algebra. In *HUG '93: HOL User's Group Workshop*, pages 428–440, UBC, Vancouver, 1993.

[19] Henrik Hulgaard, Steven M. Burns, et al. Practical applications of an efficient time separation of events algorithm. In *ICCAD93*, pages 146–151, November 1993.

[20] Henrik Hulgaard, Steven M. Burns, et al. An algorithm for exact bounds on the time separation of events in concurrent systems. Technical Report 94-02-02, Department of Computer Science, University of Washington, Seattle, 1994.

[21] Jørgen Staunstrup and Mark R. Greenstreet. *Formal Methods for VLSI Design*, chapter 2. Elsevier Science Publishers B.V. (North-Holland), 1990.

[22] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2), February 1995.

[23] Christos H. Papdimitriou and Kenneth Steiglitz. *Combinatorial Optimization - Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, New Jersey., 1982.

[24] Kenneth H. Rosen. *Elementary Number Theory*, page 21. Addison Wesley, 1988.

[25] Carl-Johan H. Seger. Voss — A Formal Hardware Verification System User's Guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, November 1993. Available by anonymous ftp as ftp://ftp.cs.ubc.ca/pub/local/techreports/1993/TR-93-45.ps.gz.

[26] J.U. Skakkebæk and N. Shankar. A Duration Calculus Proof Checker: Using PVS as a Semantic Framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, December 1993.

[27] J.U. Skakkebæk and N. Shankar. Towards a Duration Calculus Proof Assistant in PVS. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *FTRTFT*, volume 863 of *LNCS*, pages 660–679. S-V, 1994.

[28] Jørgen Staunstrup. *A Formal Approach to Hardware Design.* Kluwer, 1993.

[29] Jørgen Staunstrup, S. Garland, and J. Guttag. Localized Verification of Circuit Descriptions. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407.* Springer Verlag, 1989.

[30] Jan T. Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Eindhoven University of Technology, 1984.

[31] Tom Verhoeff. Delay-insensitive codes - an overview. *Distributed Computing*, 3:1–8, 1988.

Appendix A

User Manual

The proof checker mechanically verifies existing proofs. It enforces the use of sound proof rules thus increasing the rigor of the proof. The checker is implemented on top of Voss [25], a hardware verification system developed by Carl Seger of the University of British Columbia. Voss provides Ordered Binary Decision Diagrams which are used for boolean manipulation, linear programming which is used as a decision procedure for systems of linear inequalities, and a functional language, FL, which is used as an interface language to the proof checker.

This document serves as a user manual for the proof checker. It concentrates on how to use the proof checker and does not go into details of the theory behind it. Section A.1 gives an overview of the structure of the proof checker. Section A.2 explains how to start the system. Section A.3 describes the syntax used in the checker. Section A.4 lists all the proof rules with their functionalities. Section A.5 describes some interface functions which simplify the generation of proofs and some auxiliary functions which help to form new proof states. Section A.6 is a simple example of how the proof checker is used to verify a simple proof by induction.

A.1 Structure of Proof Checker

The proof checker represents a proof as a sequence of proof states. Each proof state includes the hypotheses and claim of the theorem as well as any pending proof obligations. Initially, a proof has a single obligation, the claim of the theorem. By applying proof rules, pending obligations are rewritten into simpler obligations or discharged. A completed proof has an empty obligation list.

A proof state in the checker is composed of a claim, a hypothesis list, an obligation list, and a postponed list.

- The claim is the main goal to be proven. This field associates the theorem to be proven with its proof.

- The hypothesis list contains the hypotheses of the proof. These are stated at the beginning of the proof. No element can be added to or removed from this list once the proof is stated.

- The obligation list is the list of pending proof obligations to be discharged before the claim is proven. Initially, this list contains exactly one element: the claim. The proof is complete when this list becomes empty.

- The postponed list contains all unverified assumptions made along the course of the proof. Initially, this list is empty. An obligation can be moved to or removed from this list with the Postpone rules described in Section A.4. Moving a proof obligation to the postponed list is the only way a proof obligation can be discharged without actually proving it. The postponed object can be moved back onto the obligation list to be discharged by a proof rule further on along the proof. When a proof is completed, all obligations on the postponed list are printed, and it is the user's responsibility to verify them.

A.2 How to Start/Exit the System

The proof checker is installed under `/isd/local/generic/bin/`. To invoke the system, either add this directory under your path, update the path by typing `source .cshrc`, and type `checker` or simply type `/isd/local/generic/bin/checker`. This command executes Voss and loads the FL files for the proof checker, loads the file with interface functions to the proof checker and returns with the FL prompt. To get an earlier version of the proof checker, type `checker [version#]`.

To load the system without the interface functions, the user will have to execute and load the core system manually by typing

```
load /isd/local/generic/lib/checker/state.fl.
```

The core system only includes the core functions of the checker and does not include interface functions which ease state manipulation. In most cases, the whole system is desired.

Although the system is interactive, keeping a proof script is handy since regenerating a proof state can be time consuming. Once a proof script is generated, it can be loaded onto the system with the command `load "script.fl";` assuming that the proof script is named `script.fl`.

To exit the system, simply type `quit;` after the prompt `:` as follows:

```
:  quit;
```

A.3 Syntax Used in the Checker

Before a user can initialize and/or manipulate a proof state, an understanding of the data structures of different data types defined in this proof checker is needed. FL provides primitive types `bool`, `int`, and `string`. For the purpose of this proof checker, more complicated types are needed. There are three main types defined in the proof checker: `integer`, `real` and `boolean`. See Figure A.12 for the definitions of these types. Note that the type constructors will be infix operators in the near future. This section describes the structures of these types and gives examples on how to declare variables and define the problem before starting a proof.

The type *int* is the FL integer type described in Section 2.2 of [25], and the type *string* is the FL string type described in Section 2.8 of [25].

The type `boolean` has the constants `True` and `False`. A boolean variable can be declared with the constructor `bool` as follows:

```
bool "a";
```

A reference to this variable can be defined as follows:

```
:  let x = bool "a";
```

In this case, `x` is a reference to the boolean variable `bool ''a''`. Consider the following three statements:

```
:  let a = bool "a";
:  let b = bool "a";
:  let c = bool "c";
```

The pointers `a` and `b` refer to the same variable `bool ''a''`, and `c` refers to the variable `bool ''c''`. The name of the pointer is not required to be the same as the string assigned to the boolean variable, although it is usually convenient.

`Integer` constants are constructed with the constructor `const` followed by its integer value. For example, the constant 3 is written as `(const 3)`. An integer variable, `x`, can be declared as:

```
:  let x = I "x";
```

```
<boolean> ::=  True |
               False |
               bool   <string> |                   // declare a boolean variable
               Not    <boolean> |
               Equal <boolean> <boolean>    |
               ==>   <boolean> <boolean>    |      // implication
               Or     <boolean> <boolean>   |
               And    <boolean> <boolean>   |
               b_array <string> <integer>   |      // declare a boolean array
               '>=    <real> <real> |              // real comparisons
               '>     <real> <real> |
               '<=    <real> <real> |
               '<     <real> <real> |
               '=     <real> <real> |
               '<>    <real> <real> |
               $>=    <integer> <integer> |        // integer comparisons
               $>     <integer> <integer> |
               $<=    <integer> <integer> |
               $<     <integer> <integer> |
               $=     <integer> <integer> |
               $<>    <integer> <integer> |
               forall <integer> <boolean>

<integer> ::= const  <int> |                       // declare an integer constant
              I      <string> |                    // declare an integer valued variable
              i_array <string> <integer> |         // declare an integer array
              ++ <integer> <integer> |             // addition
              -- <integer> <integer> |             // subtraction
              **  <integer> <integer> |            // multiplication
              i_if <boolean> <integer> <integer>

<real> ::=    rconst <int> <int> |                 // declare a real constant
              R      <string> |                    // declare a real valued variable
              r_array <string> <integer> |         // declare a real array
              '+ <real> <real> |                   // addition
              '-  <real> <real> |                  // subtraction
              '*  <integer> <real> |               // multiplication
              r_if <boolean> <real> <real>
```

Note: All binary operators are infix operators.

Figure A.12: Definition of `Boolean` type, `Integer` type, and `Real` type.

`Real` is the type for rational numbers. `Real` constants are declared as `(rconst n d)` where n is the integer value for the numerator and d is the integer value for the denominator. `Real` variables are declared the same way as `integer` variables except that they use the constructor `R`.

With these constants and variables, more complicated expressions can be constructed using the operations depicted in figure A.12

A.4 Proof Rules

Proof rules are the only way to manipulate a proof state. This section describes these rules. Section A.4.1 describes how to start and end a proof. Section A.4.2 lists the ten proof rules provided by the proof checker. Each function is explained with three fields: **syntax**, the function name and how it can be called, **description**, a brief explanation of the rule, and **error message**, a list of possible error messages resulting from the rule and the meaning of each of these messages. Error messages of the form `** <error message> **` are specific to a proof rule and other error messages, without the `**`, are generated from general subroutines used in different proof rules.

A.4.1 To Start/End a proof: (`Start_proof/Done`)

To initialize a proof, the `Start_proof` function creates a proof state from the claim we want to prove and the hypotheses of the proof. This function takes a boolean and a boolean list and returns a state. For example, if we want to prove $(x > y)$ given that $(x > r)$ and $(r > y)$, then we would initialize the proof with

```
:  let state = (Start_proof(x $> y) [(x $> r), (r $> y)]);
```

assuming that x, y, and r are of type `integer`.

A proof state can be viewed with the function `(print_State state)`. This function takes a proof state and returns a string listing the four fields in the proof state.

After initializing the proof with `Start_proof`, we manipulate this proof state through other proof rules until the obligation list becomes empty. When the obligation list becomes empty, we can conclude the proof with the function `Done`. (`Done` *state)* takes the proof state, *state*, makes sure the obligation list is empty, and prints that the claim has been proven subject to any assumptions that were added to the postponed list in the course of the proof. For the above example, we would get the following message when the proof is done.

```
: Done state;
"(x $> y)
is proven with the unverified postponed objects:
"
```

In this case, there is no remaining postponed object in the proof.

1. **Start Proof:**

   syntax:

   `(Start_proofclaim hypothesis_lst)`

   description:

   The function `Start_proof`creates an initial proof state from the claim and the list of hypotheses.

   error message:

   This function does not generate any error messages.

2. **End Proof:**

   syntax:

```
(Donestate)
```

description:

The function `Done` confirms verification of the proof by checking the obligation list and the postponed list. It gives warning if debug mode was used in generating the proof. This mode is explained in Section A.4.3.

error messages:

`** done:  currently in debug mode **`

indicates an attempt to end proof in an invalid mode.

```
WARNING: Part of this proof was generated in DEBUG mode.
Soundness is not guaranteed.
```

indicates that the proof may not be sound because part of the proof was generated in a mode designed for proof debugging.

A.4.2 The Ten Proof Rules

This section describes the proof rules provided by the proof checker. Note that all proof rules are functions which take an index (or indices for the `PC_rule`) to the old obligation list together with other auxiliary information and return a state of type `_state`.

1. **Linear Programming Rule:**

   syntax:

   ```
   (apply_lp n state)
   ```

   description:

   The Linear Programming Rule is a discharge rule. It requires that the n^{th} obligation of the proof state is of the following form:

   ```
   ((Not a) And b And c And ...)  Equal False
   ```

where `a`, `b`, and `c` are linear inequalities or negations of linear inequalities.

If the obligation holds as a tautology, the rule simply discharges it as a pending proof obligation, resulting that the obligation list reduces its size by one. Otherwise, the rule fails.

error messages:

`** LP Rule Failed:  obligation not of form b Equal False **`

indicates that the structure of the proof obligation is of neither forms mentioned above.

`** LP Rule Failed:  system is feasible **`

indicates that the obligation cannot be discharged because it is not a tautology.

`Element not in the list`

indicates that there are less than `i` hypotheses on the hypothesis list. However, note any other rule can produce the same error message when the index of the obligation list is out-of-bounds.

2. **Predicate Calculus Rule:**

   syntax:

   (**apply_PredicateCalc** `index_list predicate_list state)`

   description:

   The Predicate Calculus Rule is a replacement rule. It takes as its arguments a list of indices of the obligation list, `index_list`, a list of boolean expressions, `predicate_list`, and the proof state, `state`. If the conjunction of the list of boolean expression of `predicate_list` implies the list of indexed obligations, then the indexed obligations are replaced by this list of boolean expressions. Then the indexed obligations are removed from the obligation list, and `predicate_list` is inserted into the obligation list where the first indexed obligation was before the removal of the old list.

error message:

`** Predicate Calculus Rule Failed: expressions do not imply obligations **`

indicates that the desired implication does not hold and the replacement cannot be done.

3. **Instantiation Rule:**

syntax:

```
(instantiate n k state)
```

description:

The Instantiation Rule is a discharge rule. It requires that the n^{th} obligation is of the form $\forall P \Rightarrow Q$. It discharges the obligation if Q is a proper instantiation of $\forall P$ with the instant `k`. Otherwise, the rule fails.

error messages:

`** Instantiate Rule Failed **`

indicates that Q is not a proper instantiation of $\forall P$.

`** Instantiate Rule Failed: obligation not in the required form forall P ==> Q **`

indicates that the structure of the proof obligation is not of the form $\forall P \Rightarrow Q$.

4. **Skolemization Rule:**

syntax:

```
(apply_skolem n skolemized_expr subexpr i skolem_const state)
```

description:

The Skolemization Rule is a replacement rule. It takes as its arguments the index of the targeted obligation, `n`, the desired resulting replacement, `skolemized_expr`, the universally quantified subexpression to be skolemized, `subexpr`, the quantifier to be skolemized over, `i`, the proposed skolem constant, `skolem_const`, and the proof state, `state`. Note that universal quantification, in the checker, is always over all integers. The rule skolemizes the obligation or a subexpression of the obligation. Valid choices for the skolem constant are variables that do not appear free in the targeted obligation or on the hypothesis list. If `skolemized_expr` is a valid replacement, then the n^{th} obligation is replaced by it. Then `skolemized_expr` becomes the n^{th} obligation in the new state. Otherwise, the rule fails.

error messages:

```
** Skolem Rule Failed:  skolem constant is a free variable in
hypothesis list **
```

indicating that the proposed skolem constant already exists as a free variable in the hypotheses on the hypothesis list.

```
** Skolem Rule Failed:  skolem constant is a free variable in
expression **
```

indicating that the proposed skolem constant already exists as a free variable in the targeted obligation.

```
** Skolem Rule Failed:  unable to do replacement **
```

indicates that the proposed replacement does not match the expected result structurally and that the replacement cannot be done.

`** Skolem Rule Failed: not universal quantified expression **`

indicates that `subexpr` is not a universally quantified expression and cannot be skolemized.

5. **Induction Rule:**

   syntax:

   (**induct** `n k base [b, up, down] state`)

   description:

   The Induction Rule is a replacement rule. It provides a mechanism to reason by mathematical induction over integers. It replaces the n^{th} obligation, which must be a universally quantified expression, by three new obligations: one for the base case, one to induct up, and one to induct down. This rule takes as its arguments the index of the targeted obligation, `n`, the quantifier of the resulting universally quantified expressions for the induction steps, `k`, the base case value, `base`, the list of expected results (the base case, `b`, the case inducting up, `up`, and the case inducting down, `down`), and finally the proof state, `state`. The rule rewrites the n^{th} obligation of the form $\forall i.P(i)$ into

   (a) $P(base)$

   (b) $\forall k.((k > base) And (\forall j \in \{base, k-1\}.P(j))) \rightarrow P(k)$

   (c) $\forall k.((k < base) And (\forall j \in \{k+1, base\}.P(j))) \rightarrow P(k)$

   After the replacement, the base case, the argument inducting up, and the argument inducting down become the n^{th}, the $(n+1)^{th}$, and the $(n+2)^{th}$ obligations respectively, and the old n^{th} obligation is removed from the list.

error messages:

`** Induct Rule Failed:  proposed index is a free variable in expression **`

indicates that `k`, the proposed quantifier for the universally quantified expression is a free variable in the original obligation and that it is an illegal choice for the quantifier.

`** Induct Rule Failed:  invalid rewrite for base case **`

indicates that `b` is not a valid rewrite for the induction base case.

`** Induct Rule Failed:  invalid rewrite for case inducting up **`

indicates that `up` is not a valid rewrite for the case inducting up.

`** Induct Rule Failed:  invalid rewrite for case inducting down **`

indicates that `down` is not a valid rewrite for the case inducting down.

`** Induct Rule Failed:  not universally quantified expression **`

indicates that the n^{th} obligation is not a universally quantified expression as required.

6. **Definition Rule:**

syntax:

(**by_hypothesis** `n i state`)

description:

The Definition Rule is a replacement rule. It allows users to retrieve information from the hypothesis list and apply it to a specific obligation. It takes as its arguments the index of the targeted obligation, `n`, the index of the hypothesis to use, `i`, and the proof state, `state`. This rule replaces the n^{th} obligation, obligation(`n`), by the new obligation, `hypothesis(i) ==> obligation(n)`, where hypothesis(`i`) represents the i^{th} hypothesis. If the i^{th} hypothesis exists, the old obligation is removed from the list and the new obligation becomes the n^{th} obligation in the new state. Otherwise, the rule fails.

error message:

`Element not in the list`

indicates that there are less than `i` hypotheses on the hypothesis list. However, note any other rule can produce the same error message when the index of the obligation list is out-of-bounds.

7. **Postpone Rules:**

   The Postpone Rules manipulate the postponed list. This set of rules allows the user to move ahead in a proof without actually proving an obligation. Another use of these rules is to postpone proving an obligation that appears more than once in the course of the proof. The user then discharges the obligation with one sequence of proof steps. There are three rules in this set: **postpone**, **by_postponement**, and **retrieve**. There is a name associated with each element in the postponed list, and these three rules refer to the postponed objects by their names.

   - **syntax:**

     (**postpone** `n name state`)

     description:

     The function, **postpone**, discharges the n^{th} obligation by moving it onto the postponed list and assigns it the name, `name`. If the name is already associated with a postponed object, then it checks to see if this new postponed object is logically related to the old one. If the new postponed object implies the old one, the old object is removed from the postponed list and the new one is added to the front of the list. If the old postponed object implies the new one, the postponed list is not changed. If neither case holds, the rule fails.

error message:

`** Postponed Rule Failed:  name already existed for unrelated assumption **`

indicates that the name is already used for a postponed object which neither implies the targeted obligation, nor is implied by the targeted obligation.

- **syntax:**

  (**by_postponement** `n name state`)

  description:

  The function, **by_postponement**, is similar to the **Definition Rule**. Instead of retrieving information from the hypothesis list, this rule uses an assertion whose justification has been postponed. It looks up the postponed object named `name` from the postponed list and replaces the n^{th} obligation, obligation(`n`), by `postpone(name) ==> obligation(n)`, where postpone(`name`) represents the postponed object tagged with the name `name`.

  error message:

  `** By Postponement Failed:  failed to match name **`

  indicates that the suggested name given is not a name for any postponed object on the postponed list.

- **syntax:**

  (**retrieve** `name state`)

description:

The function, **retrieve**, moves the postponed object named `name` from the postponed list back to the obligation list to be discharged by other proof rules. The retrieved postponed object is inserted to the beginning of the obligation list.

error message:

`** Retrieve Failed: failed to match name **`

indicates that the suggested name given is not a name for any postponed object on the postponed list.

8. **Equality Rule:**

syntax:

(**apply_equality** `n result state`)

description:

The Equality Rule is a replacement rule. It takes as its arguments the index of the targeted obligation, `n`, the expected resulting replacement, `result`, and the proof state, `state`. It rewrites an obligation of one of the following forms:

```
   (b1 Equal b2) ==> f(b1,b2),
   (i1 $=    i2) ==> f(i1,i2),
or (r1 '=    r2) ==> f(r1,r2),
```

where `f` represents an arbitrary expression,
into expression with the same structure while using (`b1`,`b2`), (`i1`,`i2`), and (`r1`,`r2`) as interchangeable pairs. Some of the valid rewrites for obligation `(b1 Equal b2) ==> f(b1,b2)` are:

(b1 Equal b2) ==> f(b1,b2),

(b1 Equal b2) ==> f(b2,b1),

(b1 Equal b2) ==> f(b1,b1),

(b1 Equal b2) ==> f(b2,b2),

$\vdots$

error messages:

`** Equality Rule Failed:  invalid rewritten form **`

indicates that `result` does not structurally match any proper replacement resulting from this rule.

`** Equality Rule Failed:  obligation not of form i=j ==> f(i,j) **`

indicates that the structure of the proof obligation matches none of the forms mentioned above.

9. **If Rule:**

syntax:

(**rewrite_if** `n result state`)

description:

The If Rule is a replacement rule. It rewrites the n^{th} obligation as follows:

(x_if True a else b) $\xrightarrow{becomes}$ a,

and (x_if False a else b) $\xrightarrow{becomes}$ b,

where `x_if` is any of `b_if`, `i_if`, or `r_if`.

Then the rule checks to see if the expected rewrite, `result`, is legal.

error message:

`** rewrite if's Failed: invalid rewritten form **`

indicates that the proposed replacement, `result`, is not a legal rewrite.

10. **Discrete Rule:**

    syntax:

    (**apply_discrete** `n state`)

    description:

    The Discrete Rule is a discharge rule. It is used to exploit the discreteness property of integers. It discharges the n^{th} obligation if it is of the form

    ```
       (x $> y) Equal (x $>= (y++one))
    or (x $< y) Equal (x $<= (y--one)).
    ```

    where *one* is defined to be the integer constant 1.

    error message:

    `** apply_discrete Failed: obligation not in correct form **`

    indicates that the obligation cannot be discharged because it is of neither of the forms mentioned above.

A.4.3 Proof Debugging: debug mode

Running a proof in the proof checker takes time. It would be time consuming if the user is required to run a proof from the start for every error in the proof script. The debug mode allows the user to manipulate proof states without verification. After correcting an error, the user can re-execute previously verified parts of the proof script where proof steps are not checked. Since the proof rules replace and discharge obligations as requested without verification, execution in this mode is very fast. Normal execution is resumed when the modified portion of the script is reached. Any proof states derived from proof rules executed in debug mode are marked as untrustworthy. Thus, when the entire proof is debugged, it must be executed again with every step checked for the theorem to be certified by the checker.

The functions **begin_debug** and **end_debug** set and reset debug mode. The function **begin_debug** takes a state and puts it in debug mode. The function **end_debug** takes a state and puts it in normal (default) mode. It is not required to have matching pairs of **begin_debug** and **end_debug** in the proof. Using **begin_debug** in debug mode does not alter the mode. Using **end_debug** in normal mode does not change the mode either. Attempting to end a proof in the debug mode is an error. The function `Done` requires a proof state to be in normal mode.

A.5 User Interface

On top of the core system, there are a few user interface functions and some auxiliary functions to ease the tedium in generating proof scripts for the proof checker. The first part of this section lists the interface functions, and the second part of the section describes the auxiliary functions accessible to the users.

A.5.1 Interface Functions

There are three sets of User Interface Functions: **Case Analysis**, **Unchanged**, and **Print Abbreviation**. There are certain proof techniques that utilize the same or similar sequence of proof rules, and manipulate proof obligations in similar ways. Each set of interface functions reduces the number of proof steps by encapsulating a specific sequence of proof rules and manipulations of the proof states in one function call. Since these functions, like simple proof rules, provide functionality to discharge or simplify proof obligations, we describe them in the same way as the ten proof rules in Section A.4.2. For each set of interface functions, we present its syntax, give a description of its functionality, and list possible error messages produced by the functions.

1. **Case Analysis:**

   There are two versions of Case Analysis: one over `booleans` called `CaseAnalysis`, and the other over `integers` called `CaseAnalysis2`.

   - **syntax:**

     ```
     (CaseAnalysis n case state)
     ```

     description:

     Case Analysis is like a replacement rule. It takes as its arguments the index of the targeted obligation, `n`, the case to apply case analysis on, `case`, and the proof state, `state`. It splits a proof obligation, into two separate obligations: one with `case` being `True`, and the other with `case` being `False`. A proof obligation, `P`, becomes:

     `(case Equal True) ==> P`

     and, `(case Equal False) ==> P`.

     In place of the n^{th} obligation, `P`, `(case Equal True) ==> P` becomes the n^{th} obligation, and `(case Equal False) ==> P` becomes the $(n+1)^{th}$ obligation in the new proof state.

     error messages:

     This function does not generate any error messages.

   - **syntax:**

     (**CaseAnalysis2** `n i lst state`)

description:

CaseAnalysis2 is like a replacement rule. Given a monotonically increasing list of integers, it enumerates all possible values for a variable and applies case analysis over that variable. This interface function takes as its arguments the index of the targeted obligation, `n`, the variable, `i`, over which to apply case analysis, the list specifying the desired integer ranges, `lst`, and the proof state, `state`. Given the integer ranges `[x(0), x(1), .., x(m)]`, the n^{th} obligation, `obligation(n)` is replaced by the following list:

```
                                (i < x(0))     ==> obligation(n)
(i < x(1))              And     (x(0) <= i)    ==> obligation(n)
                        ...
(x(m) <= i) And ...     And     (x(0) <= i)    ==> obligation(n)
```

In place of the old n^{th} obligation, this list becomes the n^{th}, $(n{+}1)^{th}$ ... $(n{+}m)^{th}$ obligations in the new proof state.

error messages:

This interface function calls `apply_PredicateCalc` and `apply_lp`; therefore, it can produce the same error messages these rules generate.

2. **Unchanged:**

syntax:

```
(Unchanged n hyp value state)
```

description:

In many cases, an obligation can be discharged by instantiating a hypothesis. This function takes the index of the targeted obligation, `n`, the index to the desired hypothesis, `hyp`, the value to instantiate the hypothesis by, `value`, and the proof state, `state`. Then it tries to discharge the obligation by instantiating the hypothesis by the given value.

error messages:

Unchanged calls the **Predicate Calculus Rule**, the **Definition Rule**, the **Instantiation Rule** and the **Linear Programming Rule**, therefore, all error messages generated from these proof rules are possible error messages for this interface function.

`** Unchanged Failed: instantiation and obligation match failed **`

is an error message generated exclusively by this function. It indicates that the hypothesis cannot be instantiated to structurally match the obligation.

3. **Print Abbreviation:**

   In many cases, expressions in a proof state can be large and difficult for the user to read. Print Abbreviation is a set of functions which allow users to introduce abbreviations for expressions.

   - **syntax:**

     (**abbrevBool** `abbrev expr abbrev_lst`)

     (**abbrevInt** `abbrev expr abbrev_lst`)

     (**abbrevReal** `abbrev expr abbrev_lst`)

     description:

     Functions **abbrevBool**, **abbrevInt**, and **abbrevReal** introduce abbreviations for boolean expressions, integer expressions, and real expressions respectively. They take the proposed abbreviation, `abbrev`, for an expression, `expr`, and add it to the abbreviation list, `abbrev_lst`.

WARNING: This set of functions does not check if the proposed abbreviation is already used for another expression. Since this is an interface function, it does not affect the soundness of the proof checker, however, it can create confusion if one name is used to represent two different expressions.

error messages:

There are no error messages for this set of functions.

- **syntax:**

  (**Bexpand** `b abbrev_lst`)

  (**Iexpand** `b abbrev_lst`)

  (**Rexpand** `b abbrev_lst`)

  description:

  Functions **Bexpand**, **Iexpand**, and **Rexpand** print abbreviations for boolean, integer, and real expressions respectively in their expanded form. The argument `b` is the abbreviation for an expression, and `abbrev_lst` is the abbreviation list where all the abbreviation-expression matches are stored.

  error messages:

  `"no such abbreviation"`

  indicates that `b` is not defined as an abbreviation for any expression.

- **syntax:**

  (**display_abbrev** `abbrev_lst`)

  description:

  The function **display_abbrev** shows all abbreviation-expression matches. This shows all possible abbreviations which can be used in an expression.

error messages:

This function does not generate any error messages.

- **syntax:**

  (**print_abbrev** `abbrev_lst state`)

  description:

  The function **print_abbrev** displays a proof state in its abbreviated form.

  error messages:

  This function does not generate any error messages.

A.5.2 Auxiliary Functions

The proof rules which rewrite proof obligations require proposed replacements from the user. Generating expressions can be a tedious job. The proof checker provides four functions to retrieve different elements from a proof state, and functions for minor modifications of expressions.

The four functions which retrieve elements from a proof states are:

1. (**getobligation** `n state`) which retrieves the n^{th} obligation from the obligation list in `state`.

2. (**getpostpone** `n state`) which retrieves the n^{th} postponed from the postponed list in `state`.

3. (**gethypothesis** `n state`) which retrieves the n^{th} hypothesis from the hypothesis list in `state`.

4. (**getclaim** `state`) which retrieves the claim from `state`.

The auxiliary functions **replaceInt**, **replaceReal**, and **replaceBool** are replacement functions. (**replaceInt** `expr i1 i2`) replaces all occurrences of the integer valued subexpression, `i1`, by the integer valued subexpression, `i2`, in the boolean expression, `expr`; (**replaceReal** `expr r1 r2`) replaces all occurrences of the real valued subexpression, `r1`, by the real valued subexpression, `r2`, in the boolean expression, `expr`; and (**replaceBool** `expr b1 b2`) replaces all occurrences of the boolean, `b1`, by the boolean, `b2`, in the boolean expression, `expr`.

The functions **lhs** and **rhs** take a boolean expression of the form `(a ==> b)`, and return the left hand side of the implication, `a`, and the right hand side of the implication, `b`, respectively.

A.6 Example

This section shows how the proof checker can be applied to an induction proof by proving the following:

$$\sum_{i=1}^{n} f_i = f_{n+2} - 1$$

where f_i is the i^{th} Fibonacci number, and

$n \geq 1.$

This example is from [24].

This claim can be proven by mathematical induction. The base case where $n = 1$ follows since $\sum_{i=1}^{1} f_i = 1$ and this is the same as $f_{1+2} - 1 = f_3 - 1 = 2 - 1 = 1$. The induction hypothesis is $\sum_{i=1}^{n} f_i = f_{n+2} - 1$. We show that under this assumption that $\sum_{i=1}^{n+1} f_i = f_{n+3} - 1$ as follows:

$$\begin{aligned}
\sum_{i=1}^{n+1} f_i &= (\sum_{i=1}^{n} f_i) + f_{n+1} \\
&= (f_{n+2} - 1) + f_{n+1} \\
&= (f_{n+1} + f_{n+2}) - 1 \\
&= f_{n+3} - 1.
\end{aligned}$$

We have followed this manual proof as a guideline to produce a checked version. The following is the script for the machine checked version. This file can be found under `/isd/local/generic/lib/checker/examples`

Appendix B

Proof Script for STARI

B.1 Proof Script for the Transmitter Transition

B.2 Proof Script for the FIFO Transition

B.3 Proof Script for the Receiver Transition

B.4 Proof Script for the Protocol