**ABSTRACT**

**AN AUTOMATA-BASED AUTOMATIC VERIFICATION ENVIRONMENT**

**by**
**Yi Meng**

With the continuing growth of computer systems including safety-critical computer control systems, the need for reliable tools to help construct, analyze, and verify such systems also continues to grow. The basic motivation of this work is to build such a formal verification environment for computer-based systems.

An example of such a tool is the Design Oriented Verification and Evaluation (DOVE) created by Australian Defense Science and Technology Organization. One of the advantages of DOVE is that it combines ease of use provided by a graphical user interface for describing specifications in the form of extended state machines with the rigor of proving linear temporal logic properties in a robust theorem prover, Isabelle which was developed at Cambridge University, UK, and TU Munich, Germany. A different class of examples is that of model checkers, such as SPIN and SMV. In this work, we describe our technique to increase the utility of DOVE by extending it with the capability to build systems by specifying components. This added utility is demonstrated with a concrete example from a real project to study aspects of the control unit for an infusion pump being built at the Walter Reid Army Institute of Research. Secondly, we provide a formulation of linear temporal logic (LTL) in the theorem prover Isabelle. Next, we present a formalization of a variation of the algorithm for translating LTL into Büchi automata. The original translation algorithm is presented in Gerth *et al* and is the basis of model checkers such as SPIN. We also provide a formal proof of the termination and correctness of this algorithm. All definitions and proofs have been done fully formally within the generic theorem prover Isabelle, which guarantees the rigor of our work and the reliability of the results obtained. Finally, we introduce the automata theoretic framework for automatic verification as our future works.

# AN AUTOMATA-BASED AUTOMATIC VERIFICATION ENVIRONMENT

by
Yi Meng

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2005

**APPROVAL PAGE**

**AN AUTOMATA-BASED AUTOMATIC VERIFICATION ENVIRONMENT**

**Yi Meng**

Elsa L. Gunter, Dissertation Advisor — Date
Associate Professor of Computer Science Department, University of Illinois, Urbana - Champaign

Narain Gehani, Committee Member — Date
Professor of Computer Science Department, New Jersey Institute of Technology

~~Ali Mili, Committee Member~~ — Date
~~Professor of~~ Computer Science Department, New Jersey Institute of Technology

Marvin K. Nakayama, Committee Member — Date
Associate Professor of Computer Science Department, New Jersey Institute of Technology

Konrad Slind, Committee Member — Date
Assistant Professor of School of Computing, University of Utah

## BIOGRAPHICAL SKETCH

**Author:** Yi Meng

**Degree:** Doctor of Philosophy

**Date:** August 2005

**Undergraduate and Graduate Education:**

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2005

- Bachelor of Science in Information Management and Information System,
  Beijing Institute of Machinery, Beijing, China, 2001

**Major:** Computer Science

**Presentations and Publications:**

Elsa Gunter and Yi Meng, "Extending DOVE with Product Automata" *in Proceedings of Victor A. Carreño and César A. Muñoz and Sofiène Tahar's Theorem Proving in Higher Order Logics, 15th International Conference, Hampton, VA*, Supplemental Proceedings TPHOLs 2002, August 2002.

Yi Meng, "Design Oriented Verification and Evaluation Extension", *Presentation at Theorem Proving in Higher Order Logics (TPHOLs), 15th International Conference, Hampton, VA, August 2002.*

*To my parents, Lina and Xianchen*

## ACKNOWLEDGMENT

I would like to take this opportunity to express my gratitude to a number of people without whom I could never have completed this dissertation.

First of all, I sincerely thank my supervisor, Prof. Elsa L. Gunter. Without her encouragement, advice, and support, I would have neither begun nor finished this dissertation. I am deeply impressed with the thoroughness with which she read through the submitted version of this dissertation in a matter of days, providing valuable comments at a very detailed level.

I am also grateful to Prof. Narain Gehani, Prof. Ali Mili, Prof. Marc Q. Ma, Prof. Marvin K. Nakayama, Prof. Konrad Slind, Prof. Mengchu Zhou for being in my PhD proposal or dissertation committee and their contribution to this dissertation.

I also thank Dr. Ronald Kane, the Dean of Graduate Studies and Clarisa Gonzalez-Lenahan, the Associate Director of Graduate Studies of New Jersey Institute of Technology for helping me review this dissertation.

My thanks also goes to all the members of the Computer Science Department at the New Jersey Institute of Technology, for making my study in NJIT very enjoyable.

# TABLE OF CONTENTS

## TABLE OF CONTENTS
## (Continued)

**Chapter** **Page**

# LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

This chapter provides an overview of this dissertation. We start with the motivation of our work and proceed with presenting the main goal and desired results. It is followed by a description of the outline of subsequent chapters.

## 1.1 Motivation

During the past two decades, the importance of computer-based systems has been growing enormously. Computer-based systems are everywhere; airplanes, medical equipment, banks, and so on, are all computerized. The reliability of such systems has become a big issue in computer science. With the growth of their scale and functionality, the probability to introduce design faults increases. Design faults can lead to expensive system errors. Design faults of computerized systems can cause loss of time, money, or sometimes human life. Thus, there is a clear need for reliable tools which can analyze the design of the complicated computer systems for logical errors.

A major goal of software engineering is to enable developers to create high quality systems. There are many approaches which aim to remove mistakes from software development; one of the most promising one is formal methods [1,2,3,4]. Formal methods offer rigorous ways to model, design, and analyze systems by using specification and verification techniques based on mathematical formalisms, such as logic [5], automata [6] and graph theory [7]. By applying formal methods, we could reduce the number of errors and hence be more confident that our systems do what they are supposed to do. However, formal methods are not widely used mainly because of lack of user-friendly and powerful tools. Such tools should be able to increase system quality and reliability and simultaneously raising productivity. With these tools, all persons involved in a software project should be able to do operations like developing and entering specifications, debugging, checking consistency, refinement, verification and validation, simulation and testing.

Two well-developed approaches to formal verification are model checking [8, 9, 10] and theorem proving. Model checking is a model-based verification method. That means, it's a technique to build a finite model of a system and check some desired properties hold in that model. Proving the correctness of the system is thus performed as an exhaustive state space search. Model checking is guaranteed to terminate since the model is finite.

There are two major paradigms to model checking. The first one is to give system specifications in a temporal logic and describe the system as an extended state transition system. Model checking is performed as a check of whether the given extended state transition system is a model for the specification. The second way is to use automata to describe both the system itself and system specification. Model checking is performed by comparing the two automata to determine whether or not the system conforms to its specification.

The advantage of model checking [11] over interactive theorem proving is that it is largely automatic. It only require the user's effort in modeling the system, stating the specification, and deciding what abstraction is needed, if any. Compared to other verification methods, the user's part is rather small. Model checking provides useful counterexamples when certain properties fail to hold. These counterexamples can be used for system debugging. The main limitations of model checking are the state explosion problems and the limited expressive power of the various temporal logics used in model checking. Usually, model checking tools are restricted to finite-state systems with relatively small state spaces. There are several strategies that attempt to reduce this problem, such as use of Binary Decision Diagrams(BDD) [12, 13], Partial Order Reduction [14, 15], Symmetry [16], Abstraction [17], and so on. In Chapter 3, we will present our attempt [18] to reduce the state explosion problem via the introduction of modular reasoning. Prominent model checking systems are, SMV [19], SPIN [20], STeP [21], Maude [22], and Murphi [23], etc.

Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. It is the process of finding a proof of a property from the axioms or rules of the logic. Although proofs can be constructed by hand, we will only focus on machine-assisted theorem proving.

In contrast to model checking, theorem proving can deal directly with infinite domains by using techniques like structural induction. Theorem proving can be done either automatically or interactively with users. Recently, interactive theorem provers based on higher-order logic have become more mature. The most popular theorem proving verification tools are HOL [24], Isabelle [25, 26], PVS [27, 28], and ACL2 [29]. In our works, we choose Isabelle as our platform because Isabelle is more generic, flexible and more highly developed automation than HOL and PVS. However, theorem proving is a highly time consuming process and usually requires a great deal of expertise. Theorem proving is a much slower process than model checking.

Model checkers and theorem provers can be used to classify different sources of failure and perform the checks for logical faults in the system design, where the design fails to guaranteee the user requirements. Both model checking and theorem proving have their advantages and their weaknesses [30, 31]. Therefore, we propose as a long-term project to combine the complementary technologies of model checking and theorem proving methods in some degree to benefit from the advantage of both techniques. This thesis presents the first major steps in this project.

Our work is mainly motivated by the paucity of high quality, user friendly tools for the formal verification of computer-based systems. The main goal of our work is to improve the quality of certain tools used to preform the checks for design errors. We improved the functionallity of DOVE by adding the ability to compose Extended State Machines as the product of constituent ESMs, after having preformed the theoretical work to assure that it was a logically sound extension. We extended the class of problems that can be handled by model checkers to include properties that distinguish between finite and infinte behaviors. We improved the level of confidence that can be placed in LTL-based model checkers using the LTL to Büchi automata translation algorithm, by having given a rigorous proof of the algorithm underlying them. We use the theorem prover Isabelle, which is a state-of-the-art interactive theorem prover for higher-order logic. Higher-order logic theorem provers incorperate much automation, but at their core must be interactive, because of the undecidability of higher-order logic.

## 1.2 Overview of the Dissertation

Chapter 2 presents some preliminary background from mathematics and some tools used in our work. We start with an introduction to set theory, relations and functions. We interpret linear temporal logic (LTL) [32] on both finite and infinite sequences. *Behavior*, which is a disjoint sum of non-empty lists over an arbitrary type $\alpha$ and mapping functions from the natural numbers to $\alpha$, is defined to contain both finite and infinite sequences. A variant of Büchi automata [33] that is slightly different from traditional Büchi automata is introduced with the ability to accept both finite and infinite words. The new Büchi automata have separate accepting conditions for finite and infinite words. Two verification tools we used in this work are also briefly described. Design Oriented Verification and Evaluation (DOVE) [34] is a modeling and verification tool based on state machines. Isabelle [25] is a generic theorem proving environment developed at Cambridge University and TU Munich. It allows us to express mathematical formulae in a formal language and prove these formulae in a logical calculus.

Chapter 3 is a concise description of our approaches. We start with system modeling and verification using DOVE and a method to address the state explosion problem in DOVE. Then we introduce the formulation of LTL into Isabelle. A variant of a widely used model checking algorithm [35] for translating LTL formulae into Büchi automata is also formulated in Isabelle. The termination and correctness proofs of the algorithm are formally presented.

Chapter 4 concludes with a summary and pointers to further work and gives the evaluation of our work; the advantages and weaknesses of our work are also given. We consider that our approach in this work is a potentially practical method for software verification.

# CHAPTER 2

# FOUNDATIONS

Software verification methods are based on mathematical principles [36, 37]. Thus, it is necessary to introduce some mathematical material before we start our techniques. In this chapter, we focused on the mathematical foundations of our work. We present the basic concepts and theories that are used later in this thesis. Two modeling and verification tools, DOVE and Isabelle, will be introduced briefly.

## 2.1 Preliminaries

### 2.1.1 Sets, Functions and Relations

Set theory [38, 39, 40] is one of the most important and fundamental concepts in modern mathematics. It provides the basic language in which much other mathematics is expressed. Set theory also plays a principle role in formal methods.

A set is a finite or infinite well-defined collection of objects. Sets in our work are typed [41, 42]. Every element in a set has the same type. Traditionally, finite sets can be defined by explicitly listing its elements between curly braces, e.g. $\{1, 3, 5, 6\}$. Another notation for sets is to give some restriction on the possible values of its elements, e.g. $\{x| x \leq 8\}$. Two sets are equal if they have same elements.

A finite set is a set containing a finite number of elements. The cardinality of a finite set $A$ is the number of elements it contains, denoted by $|A|$. A infinite set is a set containing an infinite number of elements, e.g. the set of all natural numbers. One special set is the empty set, denoted $\emptyset$, that does not contain any element. The cardinality of the empty set is 0. The empty set seems trivial, but it is a very important element in set theory.

If a set $A$ contains an element $x$, we say $x$ is belong to the set $A$, i.e. $x \in A$. If $x$ is not an element of the set $A$, then $x$ does not belong to set $A$, i.e. $x \notin A$. So, for example, if $x$=5, $y$=4 and $A = \{1,3,5,6\}$, then $x \in A$ and $y \notin A$.

If every element in the set $A$ it is also an element of the set $B$, then $A$ is said to be a subset of $B$, written $A \subseteq B$. From the definition of the subset, we know that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$. If $A$ is a subset of $B$ and $A \neq B$, then $A$ is a proper subset of $B$, written $A \subset B$. For example, if $A = \{1,3,5,6\}$ and $B = \{1,5,6\}$, then we have $B \subseteq A$. In fact, we also have $B \subset A$ because $B \neq A$. Notice that, for all sets $A$, $\emptyset \subseteq A$ and $A \subseteq A$.

Several operations to construct new sets can be performed on existing sets. The intersection of two sets $A$ and $B$, written as $A \cap B$, is the set that consists all elements occurring in both sets. For example, if $A = \{1,3,5,6\}$ and $B = \{1,5,6\}$, then $A \cap B = \{1,5,6\}$. If two sets do not share any elements, then their intersection is empty and $A$ and $B$ are said to be disjoint. Some basic properties of intersections are, $A \cap B = B \cap A$, $A \cap B \subseteq A$, $A \cap A = A$, $A \cap \emptyset = \emptyset$.

The union of two sets $A$ and $B$, denoted by $A \cup B$, is the set that contains all elements occurring in either set. For example, if $A = \{1,3,5,6\}$ and $B = \{4,7,8\}$, then $A \cup B = \{1,3,4,5,6,7,8\}$. Some basic properties of union are, $A \cup B = B \cup A$, $A \subseteq A \cup B$, $A \cup A = A$, $A \cup \emptyset = A$.

The difference of two sets $A$ and $B$, denoted by $A - B$, is the set that contains all elements occurring in set $A$ but not in set $B$. For example, if $A = \{1,3,5,6\}$ and $B = \{4,5,6\}$, then $A - B = \{1,3\}$. The power set of a set $A$, denoted Power($A$), is the set of all subsets of $A$, including $A$ itself. For example, Power({1,2})={$\emptyset$,{1},{2},{1,2}}.

An ordered pair is a collection of two elements such that one can be distinguished as the first element and the other as the second element. Two ordered pairs are equal if and only if their first elements are equal and their second elements are also equal. The Cartesian product of two sets $A$ and $B$, denoted by $A \times B$, is the set of ordered pairs whose first element is an member of A and whose second element is an member of B. For example, if $A = \{a_0, a_1\}$ and $B = \{b_0, b_1\}$, then $A \times B = \{(a_0, b_0), (a_0, b_1), (a_1, b_0), (a_1, b_1)\}$. We can extend the definition of Cartesian product more generally to sets of ordered n-tuples for any positive integer n by repeatly apply Cartesian product for two sets.

A relation of arity $n$ is a set of $n$-tuples over a collection of domains. Each $n$-tuple contains exactly $n$ ordered elements. A binary relation is a special case of relation where $n$ is set to be 2. A binary relation is a set of ordered pairs. The well-known relation "<" is an example of a binary relation.

The converse of a relation $R$, denoted by $R^{-1}$, is defined as $\{(x, y) \mid (y, x) \in R\}$. The composition operator $o$ of two relations is defined as: $R \ o \ S = \{(x, z) \mid \exists y.\ (x, y) \in S \ \wedge \ (y, z) \in R\}$. The transitive closure of a binary relation $R$, written as $R^*$, is defined as follows: if there exists a sequence $z_0$,...,$z_n$ such that $(z_i, z_{i+1}) \in R$ for $0 \leq i < n$, with $z_0 = x$ and $z_n = y$, then we say $(x, y) \in R^*$.

A function of arity $n$ can be defined as a relation of arity $n + 1$, where the first $n$ elements uniquely determines the value of the $(n+1)$st elements. The terms "function" and "mapping" are usually used synonymously. The set of input values of a function $f$ is called the domain of $f$, and the set of possible output values, is called the codomain. The image of $f$ is the set of all actual outputs. Notice that the codomain and image are distinguished by possible and actual values.

A function can be injective, surjective and bijective. A function $f$ is said to be injective (one-to-one) if and only if for two members $x_1$ and $x_2$ in the domain of $f$, $f(x_1) = f(x_2)$ only if $x_1$=$x_2$. A function $f$ is surjective (onto) if and only if for each element $y$ in the codomain of $f$, there exists an element $x$ in the domain of $f$ such that $f(x) = y$. A function is said to be bijective if and only if it is both injective and surjective.

### 2.1.2 Behavior

In this section, we introduce an approach behavior for presenting both finite and infinite sequences. A similar data structure is mentioned previously by Chou and Peled [14]. Behavior will later be used as a sequence on which to interpret LTL.

Behavior is the theory of a new type ($\alpha$)behavior, which is defined as the disjoint sum of finite non-empty lists (($\alpha$)nelist) over an arbitrary type $\alpha$ and functions of type ($\mathsf{nat} \Rightarrow \alpha$), where nat is the domain of natural numbers and $\alpha$ is a codomain of arbitrary type, FinBe and InfBe are constructors for the disjoint union:

$$(\alpha)\,\mathsf{behavior} \equiv \mathsf{FinBe}\,((\alpha)\,\mathsf{nelist}) \mid \mathsf{InfBe}\,(\mathsf{nat} \longrightarrow \alpha)$$

The reason for having a unified type of both finite and infinite sequences is that some system behaviors can be either finite or infinite, depending on the context, and some system operations are more easily defined on behavior than they could be on other types.

The type of non-empty lists over a given type has already been defined in Isabelle and used by the DOVE system to interpret LTL [34,43]. Elements of the type of non-empty lists are either singleton elements from the underlying type, or sequences formed by adjoining a new element to the head of an existing non-empty list.

$$(\alpha)\,\mathsf{nelist} \equiv \mathsf{singleton}\,(\alpha) \mid \mathsf{NECons}\,(\alpha)\,(\alpha)\,\mathsf{nelist}$$

Some basic operations to manipulate nelist are listed in Table

We also need some basic operations to manipulate behavior. The types and definitions of the basic operations on behavior are described in Table 2.2 and Table 2.3:

**Table 2.1** Function Type and Description on nelist.

| *symbol* | *type* | *description* |
|---|---|---|
| \|#\| | $[\alpha, \alpha\ nelist] \Rightarrow \alpha\ nelist$ | nelist constructor |
| @ne | $[\alpha\ nelist, \alpha\ nelist] \Rightarrow \alpha\ nelist$ | append |
| NEis_singleton | $\alpha\ nelist \Rightarrow bool$ | singleton test |
| nehd | $\alpha\ nelist \Rightarrow \alpha$ | head |
| netl | $\alpha\ nelist \Rightarrow \alpha\ nelist$ | tail |
| nemem | $[\alpha, \alpha\ nelist] \Rightarrow bool$ | membership |
| neconcat | $\alpha\ nelist\ nelist \Rightarrow \alpha\ nelist$ | concatenation |
| neset | $\alpha\ nelist \Rightarrow \alpha\ set$ | nelist to set |
| nemap | $(\alpha \Rightarrow \beta) \Rightarrow \alpha\ nelist \Rightarrow \beta\ nelist$ | apply to all |
| nelength | $\alpha\ nelist \Rightarrow nat$ | length of nelist |
| nerev | $\alpha\ nelist \Rightarrow \alpha\ nelist$ | reverse |
| nezip | $[\alpha\ nelist, \beta\ nelist] \Rightarrow (\alpha * \beta) nelist$ | zip |
| nenodups | $\alpha\ nelist \Rightarrow bool$ | duplication test |
| netake | $[nat, \alpha\ nelist] \Rightarrow \alpha\ nelist$ | take a prefix |
| nedroporlast | $[nat, \alpha\ nelist] \Rightarrow \alpha\ nelist$ | drop a prefix or last |
| netakeTill | $(\alpha \Rightarrow bool) \Rightarrow \alpha\ nelist \Rightarrow \alpha\ nelist$ | take suffix |
| nedropTill | $(\alpha \Rightarrow bool) \Rightarrow \alpha\ nelist \Rightarrow \alpha\ nelist$ | drop suffix |
| nenthsuffix | $[nat, \alpha\ nelist] \Rightarrow \alpha\ nelist$ | nth suffix |

Function type and description on nelist

**Table 2.2** Function Type and Description on behavior.

| *symbol* | *type* | *description* |
|---|---|---|
| \|##\| | $[\alpha, \alpha\ behavior] \Rightarrow \alpha\ behavior$ | behavior constructor |
| @be | $[\alpha\ nelist, \alpha\ behavior] \Rightarrow \alpha\ behavior$ | append |
| BEis_singleton | $\alpha\ behavior \Rightarrow bool$ | singleton test |
| behd | $\alpha\ behavior \Rightarrow \alpha$ | head |
| betl | $\alpha\ behavior \Rightarrow \alpha\ behavior$ | tail |
| bemem | $[\alpha, \alpha\ behavior] \Rightarrow bool$ | membership |
| beconcat | $\alpha\ nelist\ behavior \Rightarrow \alpha\ behavior$ | concatenation |
| beset | $\alpha\ behavior \Rightarrow \alpha\ set$ | behavior to set |
| bemap | $(\alpha \Rightarrow \beta) \Rightarrow \alpha\ behavior \Rightarrow \beta\ behavior$ | apply to all |
| belength | $\alpha\ behavior \Rightarrow nat\ option$ | length of behavior |
| berev | $\alpha\ behavior \Rightarrow \alpha\ behavior$ | reverse |
| bezip | $[\alpha\ behavior, \beta\ behavior] \Rightarrow (\alpha * \beta) behavior$ | zip |
| benodups | $\alpha\ behavior \Rightarrow bool$ | duplication test |
| betake | $[nat, \alpha\ behavior] \Rightarrow \alpha\ nelist$ | take a prefix |
| bedroporlast | $[nat, \alpha\ behavior] \Rightarrow \alpha\ nelist$ | drop a prefix or last |
| betakeTill | $(\alpha \Rightarrow bool) \Rightarrow \alpha\ behavior \Rightarrow \alpha\ behavior$ | take suffix |
| bedropTill | $(\alpha \Rightarrow bool) \Rightarrow \alpha\ behavior \Rightarrow \alpha\ behavior\ option$ | drop suffix |
| benthsuffix | $[nat, \alpha\ behavior] \Rightarrow \alpha\ behavior$ | nth suffix |

Function type and description on behavior

**Table 2.3** Some Function Definitions on behavior.

(x |##| (FinBe y)) ≡ FinBe(x |#| y)

(x |##| (InfBe f)) ≡ (InfBe(%n. (case n of 0⇒ x | Suc(m)⇒(f m))))

((singleton a) @be b) ≡ (a|##|b)

((x|#|xs) @be ys) ≡ (x|##|(xs @be ys))

behd(FinBe x) ≡ nehd(x)

behd(InfBe f) ≡ (f 0)

betl(FinBe x) ≡ FinBe (netl(x))

betl(InfBe f) ≡ InfBe (%n. (f (Suc n)))

(x bemem (FinBe y)) ≡ (x nemem y)

(x bemem (InfBe f)) ≡ (∃ n . (x=(f n)))

belength(FinBe x) ≡ Some (nelength x)

belength(InfBe f) ≡ None

(bezip (FinBe x) y) ≡ FinBe(nezip_fin x y)

(bezip (InfBe f) g) ≡ InfBe(nezip_inf f g)

betake_aux 0 f ≡ ne[f 0]

betake_aux (Suc n) f ≡ ((betake_aux n f) @ne ne[f (Suc n)])

(benthsuffix n (FinBe x)) ≡ FinBe (nedroporlast n x)

(benthsuffix n (InfBe f)) ≡ (case n of 0 ⇒ (InfBe f) | (Suc m) ⇒ (InfBe (%k. f(n+k))))

The successor function *Suc* takes a natural number $n$ and returns the natural number $n+1$. The function *the* takes a variable of *option* type and returns the value of the variable, if there is one, and returns an unknown element of the correct type otherwise. In total, about 25 functions are defined and 68 theorems are proved in the theorem prover Isabelle on behavior. We do not list them all here because of the space constraints. Some examples of principle rules about behavior are given as follow in Table 2.4. We do not provide the proof for these theorems also because of the space constraints.

**Table 2.4** Theorems About behavior.

Theorem 2.1. benthsuffix (Suc n) (x |##| xs) = benthsuffix n xs

Theorem 2.2. ∀ s. benthsuffix n (benthsuffix m s) = benthsuffix (n+m)

Theorem 2.3. n < the(belength s) → the (belength (benthsuffix n s) = Some k) = (the (belength s) = Some (k+n))

Theorem 2.4. the(belength (benthsuffix n s) = None) = (the(belength s) = None)

Theorem 2.5. (xs @be (FinBe ys) = xs @be (FinBe zs)) = ((FinBe ys)=(FinBe zs))

Theorem 2.6. betl(xs @be ys) = (if (∃ a . (xs = ne[a])) then ys else ((netl xs) @be ys))

### 2.1.3 Linear Temporal Logic

Linear Temporal Logic(LTL), introduced by Pnueli in 1977 [32,44,45], is one of the most popular specification formalisms for reasoning about reactive and concurrent systems. LTL is now commonly used in the area of formal verification, particularly in conjunction with model checking. It is often used to specify properties of interleaving sequences, and model the executions of a program. In this work, LTL is defined on top of propositional logic.

Given a propositional logic $P$, the syntax of LTL is as follows:

- Every formula of $P$ is a formula of LTL,

- If $\varphi$ and $\psi$ are LTL formulae, then so are $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\bigcirc\varphi)$, $(\bar{\bigcirc}\varphi)$, $(\Diamond\varphi)$, $(\Box\varphi)$, $(\varphi \bigcup \psi)$ and $(\varphi \bigvee \psi)$.

Here, we interpret LTL formulae over behavior, i.e., both finite and infinite sequences. Given $\sigma$ is a behavior, the semantics of LTL is defined as follows:

- $\sigma \models \eta$, where $\eta \in P$, iff (behd $\sigma) \models \eta$
- $\sigma \models (\neg\varphi)$ iff $\sigma \not\models \varphi$
- $\sigma \models (\varphi \wedge \psi)$ iff $\sigma \models \varphi$ and $\sigma \models \psi$
- $\sigma \models (\varphi \vee \psi)$ iff $\sigma \models \varphi$ or $\sigma \models \psi$
- $\sigma \models (\bigcirc\varphi)$ iff $\sigma$ is not a singleton and (benthsuffix $1\ \sigma) \models \varphi$
- $\sigma \models (\bar{\bigcirc}\varphi)$ iff $\sigma$ is a singleton or (benthsuffix $1\ \sigma) \models \varphi$
- $\sigma \models (\Diamond\varphi)$ iff there is an $n$ such that (benthsuffix $n\ \sigma) \models \varphi$
- $\sigma \models (\Box\varphi)$ iff for all $n$ (benthsuffix $n\ \sigma) \models \varphi$
- $\sigma \models (\varphi \bigcup \psi)$ iff there is an $i$ such that (benthsuffix $i\ \sigma) \models \psi$ and for all $j$, where $j < i$, (benthsuffix $j\ \sigma) \models \varphi$
- $\sigma \models (\varphi \bigvee \psi)$ iff either, for every $i$, (benthsuffix $i\ \sigma) \models \psi$, or for some $j$, (benthsuffix $j\ \sigma) \models \varphi$, and for every $i$, where $i \leq j$, (benthsuffix $i\ \sigma) \models \psi$

The first line in the semantics definition states that a formula from the propositional logic is interpreted in the first state of the behavior. The next three lines are the interpretations of Boolean operators negation, conjunction, and disjunction.

The operator $\bigcirc$ is called next. The formula $\bigcirc\varphi$ holds in a behavior $\sigma$ when $\sigma$ is not a singleton and the suffix of $\sigma$ starting from the second member satisfies $\varphi$. The operator $\bar{\bigcirc}\varphi$, called weaknext, is a weak version of $\bigcirc\varphi$. Where $\bigcirc\varphi$ means there is a next state and the suffix starting from the next state satisfies $\varphi$, $\bar{\bigcirc}\varphi$ means either there is no next state, or the suffix starting from the next state satisfies $\varphi$. Notice we have that

$$(\bigcirc\varphi) \wedge (\bar{\bigcirc}\psi) \equiv \bigcirc(\varphi \wedge \psi)$$

The operator $\bigcup$ is called until. The formula $\varphi \bigcup \psi$ holds when $\varphi$ holds until some point where $\psi$ holds. The operator eventually is a special case of until, i.e., $\Diamond\varphi = true \bigcup \varphi$. The operator $\bigvee$ is called release. The formula $\varphi \bigvee \psi$ holds in a behavior $\sigma$ if either $\psi$ holds for all suffixes of $\sigma$, or $\psi$ holds until some suffix of $\sigma$ where both $\varphi$ and $\psi$ hold. always is a special case of release, i.e., $\Box\varphi = false \bigvee \varphi$. Also, $\phi \equiv \psi$ if and only if for all $\sigma$, $\sigma \models \phi$ if and only if $\sigma \models \psi$. A recursive equations about $\bigcup$ is useful for the work presented in Chapter 3, $\varphi \bigcup \psi = \psi \vee (\varphi \wedge \bigcirc\varphi \bigcup \psi)$.

The formula $\Diamond\varphi$ eventually holds in a behavior $\sigma$ if there is a suffix of $\sigma$ where $\varphi$ holds. The formula $\Box\varphi$ always holds in a behavior $\sigma$ when all the suffixes of $\sigma$ satisfy $\varphi$. The operators eventually and always may be treated as syntactic sugar, or equally well as derived constructs, using the *until* ($\bigcup$) and *release* ($\bigvee$) operators and the equivalences $\Diamond\varphi = true \bigcup \varphi$ and $\Box\varphi = false \bigvee \varphi$.

In a similar fashion, we may eliminate all applications of negation except to the base propositions. That is, we may consider all formulae to be in *negation normal form*, and negation of general formulae to be a derived construct, defined using the LTL equivalences $\neg \bigcirc \varphi = \bigcirc\neg\varphi$, $\neg(\varphi \vee \psi) = (\neg\varphi) \wedge (\neg\psi)$, $\neg(\varphi \wedge \psi) = (\neg\varphi) \vee (\neg\psi)$, $\neg\neg\varphi = \varphi$, $\neg(\Box\varphi) = \Diamond\neg\varphi$, and $\neg(\Diamond\varphi) = \Box\neg\varphi$.

### 2.1.4 Büchi Automata

Automata theory [6] is widely used in many fields in computer science. It has been successfully applied into the domain of specification and verification of computer systems.

Finite automata are basically state machines over finite transition systems. Finite automata over infinite words, i.e. $\omega$-automata, can be used to describe the behavior of a system. Also, the system properties can be described using $\omega$-automata or translated into $\omega$-automata from other formalisms. Automatic verification can be performed using some graph algorithms if both the checked systems and their properties are described using the same graph representations.

One of the simplest classes of $\omega$-automata over infinite words is that of Büchi automata [33]. Usually, automata have labels on their transitions rather than on their states and have only one set of accepting states. In this work, we describe a variant, where labels are defined on states and two sets of accepting states are given. A Büchi automaton is a septuple $A = (\Sigma, S, \Delta, I, L, Fset, F)$ such that

- $\Sigma$ is a finite *alphabet*.
- $S$ is a finite set of *states*.
- $\Delta \subseteq S \times S$ is the *transition relation*.
- $I \subseteq S$ are the *start states*.
- $L : S \rightarrow \Sigma$ is a *labeling* of the *states*.
- $Fset$ is a set of sets of *accepting states* $f$ where $f \in Fset \rightarrow f \subseteq S$
- $F \subseteq S$ is the set of *finite accepting states* where $F \subseteq \bigcap Fset$.

An *execution* $\rho$ of $A$ is a finite or infinite behavior over S, $\rho$: behavior $\Rightarrow S$ such that

- (behd $\rho$) $\in I$. The first state is an initial state.
- For all $i \geq 0$, moving from the $i$th state in the *execution* to the $i$+1st state is consistent with the transition relation $\Delta$, i.e., (behd(benthsuffix i $\rho$), behd(benthsuffix (i +1) $\rho$))$\in\Delta$.

Let $inf(\rho)$ be the set of states that appear infinitely often in the *execution* $\rho$, where $inf(\rho)$ is finite if $S$ is finite. An infinite *execution* $\rho$ of a Büchi automaton $A$ is *accepting* if $inf(\rho) \cap f \neq \emptyset$, for all $f \in Fset$. That is, for all subsets of $Fset$, there is some accepting state that appears in $\rho$ infinitely often. A finite *execution* $\rho$ of a Büchi automaton $A$ is *accepting* when belast $\rho \in F$. A finite word of $A$, $v = (v_0, v_1, v_2, ..., v_n)$, is accepted by $A$ if and only of there exists a finite accepting execution $\rho = (s_0, s_1 \ldots, s_n)$ and $v_i \in L(s_i)$ for all $0 \leq i \leq n$. A infinite word, $v = (v_0, v_1, v_2, ...) \in \Sigma^\omega$, is accepted by $A$ if and only if there exists an infinite accepting execution $\rho$ such that $v_i \in L(s_i)$ for all $i \geq 0$ where $s_i$ is the $i^{th}$ element in $\rho$. The *language* $L(A) \subseteq \Sigma^\omega$ of a Büchi automaton $A$ consists of all the words accepted by $A$. For the automaton in Figure 2.1 over $\Sigma = \{\alpha, \beta, \gamma\}$, we have $S = \{s_0, s_1, s_2\}$, $I = \{s_0\}$, $\Delta = \{\{s_0, s_2\}, \{s_0, s_1\}, \{s_1, s_2\}, \{s_2, s_2\}\}$. An execution must start with state $s_0$ since it is the only initial node. An transition from a state to another must follow the transition relation $\Delta$. A word $\alpha\alpha\beta\gamma$ is accepted by the automaton. This is because there exists an execution $s_0 s_0 s_1 s_2$ that accepts the word, $s_0 \in I$ and $s_2 \in F$. The language of the automaton in Figure 2.1 can be denoted using the regular language expression $\alpha^+\beta\gamma^+$ when extended to denote both finite and infinite words.

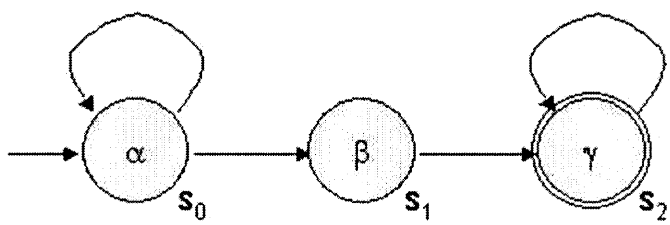


**Figure 2.1** A Büchi Automaton.

## 2.2 DOVE

Design Oriented Verification and Evaluation(DOVE) [34] was designed by the Australian Defense Science and Technology Organization under the direction of Tony Cant. It is primarily a tool for producing high-assurance system designs. It provides tools for constructing, presenting and reasoning about formal design models. DOVE is built in layers with a graphical user interface that is used for constructing and examing the design-models, and an underlying layer using the theorem prover Isabelle. The graphical interface of DOVE is written using Tcl/Tk [46] script language.

Design assurance in DOVE consists of three components: *modeling, animation*, and *verification*. The modeling component allow users to describe real-world system in DOVE. Animation is the activity of simulating a design model and checking its behavior. Verification is the process of proving the design model meets its requirements. Verification is a very effective way to provide design assurance and discover design errors.

DOVE uses a state-machine mechanism to model the specification of system behavior. A state machine in DOVE introduces the notion of memory at each state, which is updated by each consecutive transition which describes how to evolve the memory between states. The state machine graph consists of nodes and edges which represent states and transitions. There must be at least one node in the state machine and exactly one node defined as the initial state. Each transition has three parts: Let, Guard and Act. The Let part is used to simplify the other two parts of the transition definition. The transition is only performed if the guard is satisfied in the current memory. The Act, referring to action, defines how the memory is changed by the transition.

Three components are used for state machine designs. The *editor* provides a graphical interface for constructing state machine designs. The transition graph of a state machine is built by laying nodes and edges on a grid. Nodes and edges can also be moved, modified, or deleted by user. Relations between transition edges and state nodes are also created during the state machine design. The graph layout provided by the editor is very useful for the user to comprehend and analyze the system design. In the *animator*, the user can do certain simulations and experiments about the system. Animation in DOVE begins by setting initial values for the heap variables, and then is carried out by clicking edges of the state machine graph and calculating new values for the heap variables in accordance with the corresponding transition definitions. This symbolic feature provides a useful way to check whether all variables are updated as expected and whether the transition, which is protected by the guard definition, is performed correctly. Thus, animation can be used as a system validation tool. By using it, we can increase our confidence for the system design. However, the animation only gives a simple assurance of correctness of the design of the state machine. A higher level of assurance can be gained by proving whether the design satisfies given requirements. The *prover* is able to formally verify the properties of state machine designs. Requirements of the system are expressed in a formal language, which is designed to support the description of system behaviors. The prover checks these properties against the system state machines. The state machine graph is used to give the user visual feedback about the current proof state.

Verification in DOVE provides powerful facilities to express properties and to prove the system satisfies system requirements. The system requirements must be translated from informal natural language into a particular version of temporal logic supported by DOVE. DOVE then provides a collection of proof rules and tactics specialized for proving these temporal logic properties.

One of the advantages of DOVE is that it combines the ease of use provided by a graphical user interface for describing specifications in the form of extended state machines with the rigor of proving temporal logic properties in a robust theorem prover. We will provide more details about DOVE in Chapter 3.

## 2.3 A Introduction to Isabelle

Isabelle [36, 47, 48] is an interactive theorem prover being developed at Cambridge University, UK, and TU Munich, Germany. It allows mathematical formulas to be expressed in a formal language and provides tools for proving formulas in a logical calculus. Isabelle is used in a broad range of applications: proof of the correctness of computer hardware and software, properties proof of computer languages and protocols, formalising mathematics, program development.

Isabelle is a *generic* theorem prover. That means it is more flexible than other similar tools. Most other proof assistants are built around a single formal calculus. Isabelle's family embraces various logics. It represents rules as propositions and builds proofs by combining rules. These operations constitute a meta-logic in which the object-logics are formalized. It provides useful proof procedures for Constructive Type Theory [49], various first-order logics [50], Zermelo-Fraenkel set theory [51], and higher-order logic of computable functions [52]. Some logics are constructive, and some are classical. Some are based on sets, some are on types and functions and domains. This big family is not static. Some logics are added in, some become more mature, some are disappearing. In this work, we use Isabelle/HOL, which is the specialization of Isabelle for Higher-Order Logic (HOL) [53].

### 2.3.1 Higher Order Logic in Isabelle

Isabelle has a meta-logic, which is a part of higher order logic. Formulae in the meta logic are built using only implication $\Longrightarrow$, universal quantification $\bigwedge$ and equality $\equiv$. Other object-logics, such as first-order logics, Zermelo-Fraenkel set theory, and higher-order logic, are all formalized within Isabelle's meta-logic.

Here we will concentrate on higher-order logic (HOL) [54]. HOL uses the typed $\lambda$-calculus [5] and functional programming [55, 56] as bases. Functions are curried by default. The symbol $\%$ is used to represent $\lambda$-abstraction. To apply the function $f$ of type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ to two arguments $a$ and $b$, we write $f\ a\ b$. Therefore, for example, a function *equiv* to test if two natural numbers are equal can be declared as $equiv$ :: "$[nat, nat] \Rightarrow$ *bool*" and defined as $equiv\_def$ : "$equiv\ a\ b == (a = b)$".

Isabelle logics are hierarchies of theories. The root is the Pure theory, which implements the meta-logic. It provides all concepts and operations used in all object-logics.

Working with Isabelle is a procedure of defining theories. Each theory is like a module that contains types, terms, formulae, theorems, tactics, proof commands, etc. A new theory can be defined on existing theories along with its new declarations, definitions and proofs.

The *types* include basic types, function types and types built using type constructors. The type of truth values *bool* and the type of natural numbers *nat* are examples of basic types. Function types can be presented using $\Rightarrow$, e.g. $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$. Note that the $\Rightarrow$ associates to the right. A postfix type constructor can be used to build a new type using existing types. For example, we can build a list of natural number by *(nat)list*. A new datatype can be defined using the form:

$$\textsf{datatype}\ (\alpha_1, ..., \alpha_1)\ t \ = \ C_1\ \tau_{11}\ ...\ \tau_{1k_1}\ |\ ...\ |\ C_m\ \tau_{m1}\ ...\ \tau_{mk_m}$$

where $C_i$ are distinct constructor names, $t$ is the type constructor, $\alpha_i$ are distinct type variables and $\tau_{ij}$ are types.

The *terms* are those terms from the typed $\lambda$-calculus. They are embedded in the syntax of object-logics. If $f$ is a function of type $\tau_1 \Rightarrow \tau_2$ and $x$ is a term of type $\tau_1$ then $f\ x$ is a term of type $\tau_2$. Terms in Isabelle/HOL are strongly typed. If a type mismatch is found, Isabelle will print an error message.

The *formulae* are terms of type *bool*. Formulae can be constructed from basic constants *True* and *False* using logical connectives: $\neg$, $\wedge$, $\vee$, and $\longrightarrow$. Note that $\wedge$, $\vee$, and $\longrightarrow$ all associate to the right. Equality can be expressed by the infix function = of type $\alpha \Rightarrow \alpha \Rightarrow bool$. In formulae $x = y$, $x$ and $y$ have to be terms of the same type. Quantifiers are written as $\forall x.\ P$ and $\exists x.\ P$. Nested quantifications are written as $\forall x\ y\ z.\ P$. The syntax and grammar of HOL are presented in Table 2.5 and Table 2.6. Isabelle's HOL combines aspects of all the other object-logics. It is too large for us to present the whole detail of HOL here. More details about HOL in Isabelle can be found in [24].

**Table 2.5** Syntax of HOL.

| *name* | *meta-type* | *description* |
|---|---|---|
| Trueprop | $bool \Rightarrow prop$ | coercion to *prop* |
| Not | $bool \Rightarrow bool$ | negation($\neg$) |
| True | $bool$ | tautology |
| False | $bool$ | absurdity |
| If | $[\text{bool},\alpha,\alpha] \Rightarrow \alpha$ | conditional |
| Let | $[\alpha,\alpha \Rightarrow \beta] \Rightarrow \beta$ | let |
| SOME or @ | $(\alpha \Rightarrow bool) \Rightarrow \alpha$ | Hilbert description |
| ALL or ! | $(\alpha \Rightarrow bool) \Rightarrow bool$ | universal quantifier |
| EX or ? | $(\alpha \Rightarrow bool) \Rightarrow bool$ | existential quantifier |
| EX! or ?! | $(\alpha \Rightarrow bool) \Rightarrow bool$ | unique existence |
| LEAST | $(\alpha :: ord \Rightarrow bool) \Rightarrow \alpha$ | lease element |
| o | $[\beta \Rightarrow \gamma, \alpha \Rightarrow \beta] \Rightarrow (\alpha \Rightarrow \gamma)$ | composition |
| $=$ | $[\alpha, \alpha] \Rightarrow bool$ | equality |
| $<$ | $[\alpha :: ord, \alpha] \Rightarrow bool$ | less than |
| $\leq$ | $[\alpha :: ord, \alpha] \Rightarrow bool$ | less than or equals |
| & | $[bool, bool] \Rightarrow bool$ | conjunction |
| $\mid$ | $[bool, bool] \Rightarrow bool$ | disjunction |
| $\longrightarrow$ | $[bool, bool] \Rightarrow bool$ | implication |

Syntax of Higher Order Logic

**Table 2.6** Grammar of HOL.

*term* = expression of class term
| SOME *id. formula* | @ *id. formula*
| Let *id* = *term;...;id=term in term*
| if *formula then term else term*
| LEAST *id. formula*

*formula* = expression of type *bool*
| *term* = *term*
| *term* $\neq$ *term*
| *term* $<$ *term*
| *term* $\leq$ *term*
| *neg formula*
| *formula* & *formula*
| *formula* | *formula*
| *formula* $\longrightarrow$ *formula*
| ALL *id id* . formula* | *! id id* . formula*
| EX *id id* . formula* | *? id id* . formula*
| EX! *id id* . formula* | *?! id id* . formula*

---

Grammar of Higher Order Logic

### 2.3.2 Reasoning in Isabelle

Isabelle's proof mechanism is based on natural deduction [57, 58]. Every goal consists of a list of assumptions and one conclusion, i.e., $A_1 \Longrightarrow (A_2 \Longrightarrow ...(A_n \Longrightarrow B))$, where $\Longrightarrow$ is the implication of the meta-logic. It also can be abbreviated as $[|A_1; A_2; ...; A_n|] \Longrightarrow B$. Now we introduce some basic methods that Isabelle uses to work on the above goal $g$. In Isabelle, theorems and inference rules all have the same syntax. The method *rule* unifies $B$ with the current subgoal, replacing it by $n$ new subgoals, i.e., $A_1; A_2; ...; A_n$. The method *erule* unifies $B$ with current subgoal and unifies the first assumption $A_1$ with some assumption. The method *erule* deletes an assumption and replaces the subgoal with $n - 1$ new subgoals. The method *erule* is often used for elimination rules. Method *drule* unifies the first assumption $A_1$ with some assumption and deletes it. The subgoal is replaced by the $n - 1$ subgoals of $A_2; ...; A_n$ and a $n$th subgoal with an instantiation of $B$. The method *drule* is usually used for destruction rules. The method *frule* is like *drule* but it will keep the matching assumption $A_1$ in the assumption list. Proofs are contructed using introduction, elimination and other inference rules.

Introduction rules. An introduction rule can be used to introduce a logical connective in a formula containing a specific logical symbol. For example, the disjunction introduction rule says that if we have $P$ or we have $Q$ then we have $P \vee Q$. As inference rules:

$$\frac{P}{P \vee Q} \quad \frac{Q}{P \vee Q}$$

The rule introduces the disjunction symbol($\vee$) in its conclusion. We are mainly dealing with backwards proof in Isabelle. So when we apply this rule, the subgoal already has the form of a disjunction; the proof step will make the disjunction disappear. We only need to prove $P$ or $Q$ in next step. To apply an introduction rule, we simple need to use the command *rule* or *rule_tac*, e.g., *apply(rule disjI1)*. Two disjunction introduction rules are defined in Isabelle:

$$P \Longrightarrow P \mid Q \qquad \textit{(disjI1)}$$

$$Q \Longrightarrow P \mid Q \qquad \textit{(disjI2)}$$

**Elimination rules.** Elimination rules work in the opposite way from introduction rules. They describe how to destruct logical symbols in a formula. For example, the conjunction elimination rule says if we have $P \wedge Q$ and from $P$ and $Q$ we can conclude $R$, then we have $R$. The rule is as follows:

$$\frac{P \wedge Q \quad \dfrac{\begin{matrix} P & Q \\ \vdots & \vdots \end{matrix}}{R}}{R}$$

The rule eliminates the conjunction symbol($\wedge$) in its conclusion replacing it with the two new hypotheses of $P$ and $Q$ separately. To apply elimination rules, we use the command *erule* or *erule_tac*. The conjunction elimination rule is defined in Isabelle as:

$$[P \wedge Q; [[P; Q] \Longrightarrow R]] \Longrightarrow R \quad (conjE)$$

In Isabelle, there are also some other kinds of rules: destruction, unification and substitution, quantifiers, etc. Some basic Inference rules in HOL are listed in Table 2.7, Table 2.8 and Table 2.9

**Table 2.7** The HOL Rules.

| | |
|---|---|
| refl | t = (t :: '$\alpha$) |
| subst | [\| s = t; P s \|] $\Longrightarrow$ P (t :: '$\alpha$) |
| ext | (!!x :: '$\alpha$. (f x :: '$\beta$) = g x) $\Longrightarrow$ (% x. f x) = (% x. g x) |
| impI | (P $\Longrightarrow$ Q) $\Longrightarrow$ P $\longrightarrow$ Q |
| mp | [\| P $\longrightarrow$ Q; P \|] $\Longrightarrow$ Q |
| iff | (P $\longrightarrow$ Q) $\longrightarrow$ (Q $\longrightarrow$ P) $\longrightarrow$ (P = Q) |
| someI | P(x :: '$\alpha$) $\Longrightarrow$ P(@ x . P x) |
| True_or_False | (P = True) \| (P = False) |

**Table 2.8** Derived Rules for HOL.

| | |
|---|---|
| sym | $s = t \Longrightarrow t = s$ |
| trans | $[\| r = s; s = t \|] \Longrightarrow r = t$ |
| ssubst | $[\| t = s; P\ s \|] \Longrightarrow P\ t$ |
| box_equals | $[\| a = b; a = c; b = d \|] \Longrightarrow c = d$ |
| arg_cong | $x = y \Longrightarrow f\ x = f\ y$ |
| fun_cong | $f = g \Longrightarrow f\ x = g\ x$ |
| cong | $[\| f = g; x = y \|] \Longrightarrow f\ x = g\ y$ |
| not_sym | $t\ \tilde{=}\ s \Longrightarrow s\ \tilde{=}\ t$ |

Equality

| | |
|---|---|
| TrueT | True |
| FalseE | $\text{False} \Longrightarrow P$ |
| conjI | $[\| P; Q \|] \Longrightarrow P \,\&\, Q$ |
| conjunct1 | $[\| P \,\&\, Q \|] \Longrightarrow P$ |
| conjunct2 | $[\| P \,\&\, Q \|] \Longrightarrow Q$ |
| conjE | $[\| P \,\&\, Q; [\| P; Q \|] \Longrightarrow R \|] \Longrightarrow R$ |
| disjI1 | $P \Longrightarrow P \mid Q$ |
| disjI2 | $Q \Longrightarrow P \mid Q$ |
| disjE | $[\| P \mid Q; P \Longrightarrow Q; Q \Longrightarrow R \|] \Longrightarrow R$ |
| notI | $(P \Longrightarrow \text{False}) \Longrightarrow \tilde{}P$ |
| notE | $[\| \tilde{}P; P \|] \Longrightarrow R$ |
| impE | $[\| P \longrightarrow Q; P; Q \Longrightarrow R \|] \Longrightarrow R$ |

Propositional logic

| | |
|---|---|
| iffI | $[\| P \Longrightarrow Q; Q \Longrightarrow P \|] \Longrightarrow P = Q$ |
| iffD1 | $[\| P = Q; P \|] \Longrightarrow Q$ |
| iffD2 | $[\| P = Q; Q \|] \Longrightarrow P$ |
| iffE | $[\| P = Q; [\| P \longrightarrow Q; P \longrightarrow Q \|] \Longrightarrow R \|] \Longrightarrow R$ |

Logical equivalence

**Table 2.9** More Derived Rules for HOL.

| | |
|---|---|
| allI | (!! x . P x) $\Longrightarrow$ ! x . P x |
| spec | ! x . P x $\Longrightarrow$ P x |
| allE | [\| ! x . P x; P x $\Longrightarrow$ R \|] $\Longrightarrow$ R |
| all_dupE | [\| ! x . P x; [\| P x; ! x . P x \|] $\Longrightarrow$ R \|] $\Longrightarrow$ R |
| exI | P x $\Longrightarrow$ ? x . P x |
| exE | [\| ? x . P x; !! x . P x $\Longrightarrow$ Q \|] $\Longrightarrow$ Q |
| ex1E | [\| P a; !! x . P x $\Longrightarrow$ x = a \|] $\Longrightarrow$ ?! x . P x |
| ex2E | [\| ?! x . P x; !! x . [\| P x; ! y . P y $\longrightarrow$ y = x \|] $\Longrightarrow$ R \|] $\Longrightarrow$ R |
| some_equality | [\| P a; !! x . P x $\Longrightarrow$ x = a \|] $\Longrightarrow$ (@ x . P x) = a |
| | Quantifiers and descriptions |
| ccontr | (~P $\Longrightarrow$ False) $\Longrightarrow$ P |
| classical | (~P $\Longrightarrow$ P) $\Longrightarrow$ P |
| excluded_middle | ~P \| P |
| disjCI | (~Q $\Longrightarrow$ P) $\Longrightarrow$ P \| Q |
| exCI | (! x . ~P x $\Longrightarrow$ P a) $\Longrightarrow$ ? x . P x |
| impCE | [\| P $\longrightarrow$ Q; ~P $\Longrightarrow$ R; Q $\Longrightarrow$ R \|] $\Longrightarrow$ R |
| iffCE | [\| P = Q; [\| P; Q \|] $\Longrightarrow$ R; [\| ~P; ~Q \|] $\Longrightarrow$ R \|] $\Longrightarrow$ R |
| notnotD | ~~P $\Longrightarrow$ P |
| swap | ~P $\Longrightarrow$ (~Q $\Longrightarrow$ P) $\Longrightarrow$ Q |
| | Classical logic |
| if_P | P $\Longrightarrow$ (if P then x else y) = x |
| if_not_P | ~P $\Longrightarrow$ (if P then x else y) = y |
| split_if | P(if Q then x else y) = (Q $\longrightarrow$ P x) & (~Q $\longrightarrow$ P y)) |
| | Conditionals |

### 2.3.3 Isabelle System and Interface

Isabelle is implemented in ML [55]. The standard user interface is shell-based. But Isabelle also provide a friendly Emacs-based Proof General [59] interface. We used the Proof General interface in this work.

Isabelle is an *interactive* theorem prover. Thus, unlike automatic theorem provers, Isabelle is directed by the user during a proof. After starting a goal, the user directs Isabelle by some operations on the goal, called tactics at each step. Isabelle provides various kinds of tactics for rewriting, simplification, resolution, assumption, induction and so on. By using the tactics, the user tries to solve the goal. Tactics may lead to subgoals. After solving all the subgoals, the user has a formal proof of the goal. Once a theorem has been proved it becomes a derived rule of inference for use with tactics in proving new theorems.

Isabelle provides good notational support. New notations can be introduced using normal mathematical symbols. Proofs can be written in a structured notation based upon traditional proof style, or more straightforwardly as sequences of commands. Definitions and proofs may include TeX source, from which Isabelle can automatically generate typeset documents.

Isabelle has also proven useful for doing large proofs, having many tools that allow the automation of difficult and tedious details. Thus it is particularly suitable for embedding other formalisms and developing verification systems.

# CHAPTER 3

# TECHNIQUES

This chapter presents our main approaches. We start with a description of system modeling and verification in DOVE and the method to address the state explosion problem. In the second part, we will talk about a formulation of Linear Temporal Logic in Isabelle [60]. Finally, we describe the automatic verification framework and an application using our approaches.

## 3.1 System Modeling and Verification in DOVE

The DOVE tool is used to provide support for high-level system modeling, design, and formal reasoning about state machine design for computer-based systems. We will introduce safety properties [61] verification using DOVE and discuss how DOVE is extended with product automata [18].

### 3.1.1 Safety Properties Verification using DOVE

DOVE comprises three main components: the graphical editor for drawing state machines as specifications of systems, the animator for exploring various execution paths, and a prover, built on Isabelle, for verifying temporal logic properties of state machines.

State machine definitions have two parts: a topology or state transition diagram part and a transition definition part. The presence of a transition between two states in the diagram indicates the possibility that the state machine may undergo a transition between them. The definition of the transition determines if, and how, such a transition can occur. The Edit Mode is used to specify state machine designs by providing the means for laying out the state transition graph of a machine; declaring types, constants, variables and inputs; defining the associated transitions; and checking occurrences of variables, e.g. variables declared and not used, or identifiers used and not declared. The ability to model a system using the graphical editor substantially speeds up the process and increases the confidence level, when compared to describing the system as expressions in a language.

The Animation Mode is used to observe how variables and terms evolve during execution of the state machine. The basis of animation is the animation path, which is a path in the transition graph of the machine. Animations are carried out using the graph by selecting a final or initial node, proceeding through intermediate edges via back substitution or forward animation and finishing at some initial or final state. The ability to explore sample executions through animation helps the user to deepen his understanding of the state machine and to do a limited degree of testing. The highest degree of assurance is provided by stating and proving the needed properties of the system using the prover.

Proof Mode provides the means for defining, editing and browsing machine properties, including a check of the consistency of properties with different versions of machine specifications, and interactively proving a property. Once a state machine definition has been saved, all its transitions are translated into definitions in Isabelle automatically and a proof can be commenced.

In DOVE, only safety properties can be checked. A safety property asserts the absence of undesirable states, i.e., no bad things happened so far. For this reason, the behaviour of the state machine is interpreted finite sequences of configurations. A *past-fashion* temporal logic is used as the language of system properties. The syntax of the temporal logic is defined datatypes of temporal formulae as follows:

$$
\begin{array}{lcll}
\text{datatype } \mathtt{Temp} & = & \text{true} & \mid \ \text{false} \\
 & \mid & \text{first} & \mid \ \text{pred (configTY} \Rightarrow \text{Boolean)} \\
 & \mid & \text{Not } \mathtt{Temp} & \mid \ \text{init } \mathtt{Temp} \\
 & \mid & \text{Previously } \mathtt{Temp} & \mid \ \text{Always } \mathtt{Temp} \\
 & \mid & \text{Sometime } \mathtt{Temp} & \mid \ \text{PreviouslyS } \mathtt{Temp} \\
 & \mid & \mathtt{Temp} \wedge \mathtt{Temp} & \mid \ \mathtt{Temp} \vee \mathtt{Temp} \\
 & \mid & \mathtt{Temp} \Rightarrow \mathtt{Temp} & \mid \ \mathtt{Temp} \Leftrightarrow \mathtt{Temp} \\
 & \mid & \mathtt{Temp} \text{ FromThenOn } \mathtt{Temp} & \mid \ \mathtt{Temp} \text{ FromThenOnS } \mathtt{Temp} \\
 & \mid & \text{MostRecently } \mathtt{Temp}\ \mathtt{Temp} & \mid \ \text{MostRecentlyS } \mathtt{Temp}\ \mathtt{Temp} \\
 & \mid & \forall \beta \Rightarrow \mathtt{Temp} & \mid \ \exists \beta \Rightarrow \mathtt{Temp} \\
 & \mid & \text{At stateDT} & \mid \ \text{By transitionDT}
\end{array}
$$

The stateDT and transitionDT are the Isabelle types of states and transitions. The inputTY and heapTY are the types of various input variables and heap variables. Then the overall configuration type is then defined to be the cross product of the four component types.

$$\text{configTY} = \text{stateDT} \times \text{transitionDT} \times \text{inputTY} \times \text{heapTY}$$

The semantics of the temporal logic is defined using the modeling function "$\models$". If history models $\xi$ a temporal logic formula $\varphi$, we write $\xi \models \varphi$. The boolean function $\Gamma \vdash \varphi$ is true if and only if for every history $\xi$ such that $\xi \models \varphi$. The temporal logic is interpreted on the non-empty list and inductively, the semantics is defined as following:

- $\sigma \models$ true, iff True;
- $\sigma \models$ pred $b$ iff the current history models $b$;
- $\sigma \models$ At $S$ iff for some t, m, i, $\sigma_0 = (S, t, i, m)$;
- $\sigma \models$ By $T$ iff for some s, m, i, $\sigma_0 = (s, T, i, m)$;
- $\sigma \models$ Not $q$ if not $\sigma \models q$;
- $\sigma \models \varphi \wedge \psi$ iff $\sigma \models \varphi$ and $\sigma \models \psi$;
- $\sigma \models \forall x.(fx)$ iff for all x, $\sigma \models (fx)$;
- $\sigma \models$ Previously $q$ iff there is no previous history or the previous history models $q$;
- $\sigma \models q$ FromThenOn $r$ iff either for all histories $\sigma'$ where $\sigma'$ is a prefix of $\sigma$, $\sigma' \models r$, or there exists a history $\sigma''$ where $\sigma''$ is a prefix of $\sigma$ such that $\sigma'' \models q$ and for all histories $\sigma'$ where $\sigma''$ is a prefix of $\sigma'$ and $\sigma'$ is a prefix of $\sigma$, $\sigma' \models r$.

Other temporal operators are defined as syntactic sugar. The proof system is represented in the sequent calculus style. The sequence consists of a list of temporal formulae as *hypotheses* $h_i$, and a target *goal* $g$. The sequence can be expressed by terms of the form $h_1, ..., h_n \vdash g$. The infix function turnstile has type: Temp list $\Rightarrow$ boolean.

### 3.1.2 Formal Definitions of Automata and Products

Before introducing the method to extend DOVE, we will give a formal definition in higher-order logic of the type of the extended state automata used in DOVE, their semantics of execution, and how we extend this with product automata.

**Extended State Machines** Informally, an extended state machine (or automaton) is a tuple of a set of states, a set of labeled transitions, and an initial state. In DOVE, the states are augmented with memory when executed. A transition is a directed edge between a pair of states coupled with a guarded action to be committed when that transition is executed. The transition may be executed only in the case that the guard holds in the memory of the originating state of the transition, and in which case the action yields the memory of the terminating state. Memory is an association of values to variables. The guards are expressed as propositions over the variables in the memory, and the actions are expressed as assignments of values to those variables.

This notion for state machine is similar to those discussed in the literature, and a typical example can be found in Chapter 4 of [62]. One way in which DOVE extends this notion is by separating the variables into two categories, which in DOVE are referred to as input variables and heap variables. Input variables are read-only in that no transition may alter their values. Their values are considered to be supplied by the environment. As such, when defining an execution, we must assume that their values may change at any point during a sequence of transitions. While this is manifest in the proof rules in Isabelle for proving temporal formulas for state machines defined in DOVE, it is a subtle point which complicates the definition of an execution and warrants highlighting.

When users define a state machine in DOVE, they do so using a graphical user interface. This is used to generate a description in Isabelle of the extended state machine and properties that the user wishes to prove. This description of the extended state machines in Isabelle is a shallow embedding in the sense that the variables of the extended state machine are modeled as variables of Isabelle, as opposed to introducing a separate syntax for variables. Such a light-weight embedding is advantageous when the goal is exclusively proving properties in the model. However, it limits the ability to express meta-properties in the logic, such as stating what an extended state machine is, or what the product of two extended state machines is. Therefore, in this section, we will adopt a deeper embedding. The definition we will give has been rendered in higher-order logic. However, as in the informal description above, it is desirable to express things using set-theoretic notation. In all formal definitions below, such set-theoretic notations should be interpreted as using a standard rendering of naive set theory in higher-order logic, such as one given by sets as predicates.

In attempting to formally define what an extended state machine is, we have to decide how to represent the writable variables versus the read-only variables. Our ultimate goal is to define a product for composing automata, and in such a composition variables which may be read-only in one component may need to be writable in some other. Therefore, we will represent these two classes of variables as disjoint subsets of a single type of variables. For our purposes, the precise type used for representing variables does not matter, so we will use a type variable for this, allowing it later to be specialized to integers or strings or perhaps some other complex structures. Having made this choice, we will need to be able to express the requirement on transitions that they only involve the variables associated with the particular extended state machine. We will capture this notion of restricted dependence by the following definitions:

$$\mathsf{same\_on}\ \mathit{dom}\ f\ g\ = \forall\, x.\ x \in \mathit{dom} \Rightarrow (f\ x = g\ x)$$

That is, two functions are the same on given domain if they have the same values on all elements of that domain.

$$f\ \mathsf{only\_depends\_on}\ s = \forall\, m_1\ m_2.\ \mathsf{same\_on}\ s\ m_1\ m_2\ \Rightarrow (f\ m_1 = f\ m_2)$$

A function on functions only depends on a subset $s$ if it always returns the same value when applied to functions that are the same on $s$. The motivation for this definition is that our memories are functions assigning values to variables, but the guards and actions are only allowed to depend on that part of the memory that corresponds to the writable and read-only variables.

A transition is well-formed with respect to a set of writable variables and a set of read-only variables provided that the guard depends only on the union of the writable and the read-only variables, the action depends only on the writable variables, and the action does not assign any new values to the non-writable variables.

$$\begin{array}{l}
\textsf{is_transition}\ (state_1, state_2, guard, action)\ writable_vars\ read_only_vars = \\
\quad guard\ \textsf{only_depends_on}\ (writable_vars \cup read_only_vars) \wedge \\
\quad action\ \textsf{only_depends_on}\ writable_vars \wedge \\
\quad \forall\ memory\ var.\ (\neg(var \in writable_vars)) \Rightarrow \\
\qquad (action\ memory\ var = memory\ var)
\end{array}$$

We are now in a position to give a formal definition of an extended state machine:

$$\begin{array}{l}
\textsf{is_esm}\ (states, labeled_transitions, writable_vars, read_only_vars, \\
\qquad initial_state, initial_condition) = \\
\quad (writable_vars \cap read_only_vars = \phi) \wedge \\
\quad (\forall\ ((s_1, s_2, g, a), l) \in labeled_transitions. \\
\qquad \textsf{is_transition}(s_1, s_2, g, a)\ writable_vars\ read_only_vars \wedge \\
\qquad s_1 \in states\ \wedge\ s_2 \in states) \wedge \\
\quad (\forall\ ((s_1', s_2', g', a'), l')\ \in labeled_transitions. \\
\qquad (l = l') \Rightarrow ((g = g') \wedge (a = a'))) \\
\quad initial_state \in states \wedge \\
\quad initial_condition\ \textsf{only_depends_on}\ writable_vars
\end{array}$$

A tuple of states, transitions, writable variables, read-only variables, initial state, and initial condition is a state machine if

- the writable variables and the read-only variables are disjoint,
- the transitions are well-formed with respect to the writable and read-only variables,
- the start and end states of each transition are among the states of the machine,
- transitions with the same label have the same guarded actions,
- the initial state is one of the states of the machine,
- the initial condition only depends on the writable variables.

**Execution** Up to now we have defined what it means to *be* an extended state machine; we have in effect described its syntax. We are still left with describing how to *execute* an extended state machine; that is we are left with describing its semantics. The semantics of an extended state machine is the set of all it executions. So what is an execution? Informally, it is a sequence of moves through the state machine starting from a memory that satisfies the initial condition of the state machine, and then follows consecutive transitions. More formally, an execution is a pair of an initial memory and a sequence of pairs of transitions and resulting memories, where the start state of each transition is the end state of the previous transition. However, this is not a complete description. We need to be more precise about what we mean by resulting memories and enabled by the previous memory.

DOVE is only capable of dealing with properties that are provable in finite time (safety properties), so we will use lists for sequences. It would not be fundamentally different if we extended to both finite and infinite sequences.

For the sake of readability, we shall make a couple of short definitions.

$$(\mathsf{last\_state}\ \mathit{initial\_state}\ [] = \mathit{initial\_state}) \wedge$$
$$(\mathsf{last\_state}\ \mathit{initial\_state}\ (\mathsf{CONS}\ (((s_1, s_2, g, a), l), \mathit{memory})\ ::\ \mathit{seq})\ =\ s_2)$$

The last state in a list of pairs of labeled transitions and memories is the initial state if the list is empty, and otherwise is the end state of the transition at the head of the list.

$$(\mathsf{last\_memory}\ \mathit{initial\_memory}\ []= \mathit{initial\_memory}) \wedge$$
$$(\mathsf{last\_memory}\ \mathit{initial\_memory}\ (\mathsf{CONS}\ (((s_1, s_2, g, a), l), \mathit{memory})\ ::\ \mathit{seq})\ =\ \mathit{memory})$$

The last memory in a list of pairs of labeled transitions and memories is the initial memory (for the intended execution) if the list is empty, and otherwise is the memory at the head of the list.

An execution in an extended state machine starting from an initial memory is a list of pairs of labeled transitions from the extended state machine and memories such that either the list is empty or

- the tail of the list is an execution
- the last state of the tail of the execution is the start state of the next transition
- the guard is enabled in some memory that is the same as the previous end memory on the writable variables (we allow the read-only variables to change) and in that memory we execute the action to acquire the new memory.

$$\begin{aligned}
&\textsf{is_execution}\ (states, transitions, writable_vars, read_only_vars,\\
&\qquad\qquad initial_state, initial_condition)\ initial_memory\ config_list =\\
&\quad \textsf{is_esm}(states, transitions, writable_vars, read_only_vars,\\
&\qquad initial_states, initial_condition) \wedge\\
&\quad initial_condition\ initial_memory \wedge\\
&\quad ((config_list = []) \vee\\
&\quad (\exists\ s_1\ s_2\ guard\ action\ l\ memory\ tail_seq.\\
&\qquad (config_list = (\textsf{CONS}\ (((s_1, s_2, guard, action), l), memory)\ tail_seq)) \wedge\\
&\qquad \textsf{is_execution}\ tail_seq \wedge\\
&\qquad ((s_1, s_2, guard, action), l)\ \in transitions \wedge\\
&\qquad (\textsf{last_state}\ initial_state\ tail_seq = s_2) \wedge\\
&\qquad (\exists\ mem.\ \textsf{same_on}\ writable_vars\ mem\\
&\qquad\qquad\qquad (\textsf{last_memory}\ initial_state\ initial_memory\ tail_seq) \wedge\\
&\qquad guard\ mem\ \wedge\ action\ mem = memory)))
\end{aligned}$$

We do not intend to go into the details of the particular temporal logic used in DOVE in this work, but briefly a state machine is said to satisfy a given temporal logic formula provided every sequence of memories derived from the executions of the state machine satisfies the formula.

**Product Automata** Having defined the syntax and semantics of extended state machines, we are in a position to give the definition of the product of two state machines. Using the labels on the transitions, our product will allow synchronization of transitions having the same label. The states of the product is the subset of the product of the states that occurs in the set of transitions of the product (together with the product of the two initial states, if it is not already there). The transitions are effectively the merging of those transitions from the two automata that have the same label, unioned with the remaining transitions lifted to the product states. The writable variables are just the union of each set of writable variables. The readable variables are the union of each set of readable variables, minus any that are in the union of the writable variables. The variables that are in the intersection of the union of the writable variables and the union of the readable variables are those that are communicating values between the automata. The initial state is just the product of the two original initial states, and the initial condition is the intersection of the original initial conditions.

Let the state of a transitions be its start state and its ending state.

$$\textsf{stateof}\ ((state_1, state_2, guard, action), label) = \{state_1, state_2\}$$

The product is defined as

$$
\begin{array}{l}
\textsf{esm\_prod}\ (states_1, transitions_1, wvars_1, rvars_1, init\_state_1, init\_cond_1) \\
\qquad\qquad (states_2, transitions_2, wvars_2, rvars_2, init\_state_2, init\_cond_2) \ = \\
\textsf{let}\ prod\_trans \ = \\
\quad \{(((s_1, s_2), (s_1', s_2'), (\lambda m.\ g_1\ m\ \wedge\ g_2\ m),\ a_1 \circ a_2), l,)\ \mid \\
\qquad\quad ((s_1, s_1', g_1, a_1), l)\ \in\ transitions_1 \wedge \\
\qquad\quad ((s_2, s_2', g_2, a_2), l)\ \in\ transitions_2\} \cup \\
\ \ \{(((s_1, s_2), (s_1', s_2), g, a), l)\ \mid \\
\qquad\quad (s_1, s_1', g, a)\ \in\ transitions_1\ \wedge\ \neg\exists\, t.\ (t, l)\ \in transitions_2\} \cup \\
\ \ \{(((s_1, s_2), (s_1, s_2'), g, a), l)\ \mid \\
\qquad\quad (s_2, s_2', g, a)\ \in\ transitions_2\ \wedge\ \neg\exists\, t.\ (t, l)\ \in transitions_1\} \\
\textsf{and} \\
\ \ prod\_states\ = \{(init\_state_1, init\_state_2)\}\ \cup\ \bigcup_{t \in prod\_trans} \textsf{statesof}\ t \\
\textsf{in} \\
\ \ (prod\_states, prod\_trans, wvars_1\ \cup\ wvars_2, \\
\ \ (rvars_1\ \cup\ rvars_2)\ \cap\ \overline{(wvars_1\ \cup\ wvars_2)},\ (init\_state_1, init\_state_2), \\
\ \ \lambda\, m.\ init\_cond_1\ m\ \wedge\ init\_cond_2\ m)
\end{array}
$$

It follows from this definition that the product of two extended state machines is again an extended state machine, provided their writable variables are disjoint. Note that if the writable variables of the first automaton are disjoint from the second automaton, then $a_1 \; o \; a_2 = a_2 \; o \; a_1$(for all $a_1$ and $a_2$ in the definition of the transitions in the product automaton above). Therefore, the product of two automata in one order is isomorphic to the product in the other order.

Given an execution sequence, we can project that execution sequence to an execution sequences of each of the component automata.

$$
\begin{array}{l}
(proj_1 \; (states_1, trans_1, wvars_1, rvars_1, init\_state_1, init\_cond_1) \; [] = []) \wedge \\
(proj_1 \; (states_1, trans_1, wvars_1, rvars_1, init\_state_1, init\_cond_1) \\
\qquad \qquad (\mathsf{CONS} \; ((t, l), mem) \; tail\_seq) = \\
\quad \mathsf{if} \; \exists \, t'. \; (t', l) \; \in \; trans_1 \\
\quad \mathsf{then} \; \mathsf{CONS} \; (((\mathsf{SOME} \; t'. \; (t', l) \; \in \; trans_1), l), mem) \; (proj_1 \; tail\_seq) \\
\quad \mathsf{else} \; proj_1 \; tail\_seq
\end{array}
$$

We can prove that if a given initial memory and sequence of transition-memory pairs is an execution of the product automaton, then the same initial memory together with the projection of that sequence is an execution of the corresponding component automaton. Therefore, for every sequence of memories derived from an execution in the product automaton, there exists an almost identical sequence of memories derivable from a sequence in the component automaton. (The original sequence may have additional memories that are the same as their immediate predecessors in the sequence on the writable variables of the component automaton.) Therefore, for an appropriate class of temporal logic formulae (those that only involve the writable variables of the component automaton, and are stuttering invariant), if a formula holds of the component automaton, it automatically also holds of the product automaton. It is our hope in future work on this system to be able to incorporate into DOVE an ability to automatically transfer appropriate theorems from component automata to the corresponding product automata.

### 3.1.3 Extending DOVE with Products

In the previous section we described the mathematics of the product of two automata. In this section we will discuss our method of implementing the construction of product automata as an extension to DOVE. Our current approach is to add an external tool that can parse files produced by DOVE, analyze the contents of those files to determine the details of the component automata to be composed, construct the product automaton, determine layout information for it, and finally output all this information into a new file that can be input into DOVE.

In the course of a DOVE session, various local files are created, such as an smg file, a thy file, an nw file, etc. The smg file, which stands for state machine graph file (for example, plugin.smg), contains all of the information required to describe the extended state machine. This file includes not only the construction and layout information about the state machine graph, but also the information to define variables, state conditions and transitions between states.

An smg file is a sequence of lines, each beginning with a keyword, followed by data relevant to the item being added. Firstly, the smg file gives some preferences for the display of the state machine. The global variable gridOn tells us the canvas is gridded by being set to 1, and not gridded by being set to 0. The variable SetGridSize says the size of the grid.

The nodes in the smg file are defined using the keyword file RestoreNode followed by the node number, node coordinates and node name. For example, in the plugin state machine graph file, we define theWait state by

file_RestoreNode 0 {20.0 10.0} Wait

The node number of Wait is 0 and it is located at (20.0, 10.0). The edges in the plugin smg file are created by the keyword RestoreEdge followed by the edge number, the number of the starting node, the number of the ending node, their directions, some coordinates it travels through, and the location of the label and its name. For example, the edge Plugin in the plugin smg file is defined as follows:

file_RestoreEdge 0 0 north 1 south {{20.0 13.0}} {{20.0 11.0} {20.0 12.0} {20.0 13.0} {20.0 14.0} {20.0 15.0}} {20.0 13.0} Plugin

In this example, its edge number is 0, it comes out from the north of the node 0 and goes into the south of node 1, its label, Plugin, is at (20.0, 13.0), and it travels through the path of [(20.0, 11.0), (20.0, 12.0), (20.0, 13.0), (20.0, 14.0), (20.0, 15.0)].

The smg file gives two kinds of variables: heap variables and input variables. The heap variables are defined using the keyword dvd_def. It is followed by information about their names, types, status and some comments on them. Also we define input variables by div_defs followed by the same information as the heap variables.

As for the definition of the transitions, the smg file use dtr_defs. It gives a list of all the transitions followed by details of individual transitions. These details include the comments, status and the content of the transitions. The content of a transition has guard and act definitions in it.

The smg file also should have an initial state which is defined by the variable di_startState. The initial condition is given by setting the variable di_predicate. Moreover, we can add some comments on the initial state by di_description.

In addition, the smg file contains some optional information about the extended state machine. For example, if the state machine has been checked and there are no syntax errors, the variable dchksmgChecked is set to be 0, otherwise it equals 1.

From all the information above, we already know enough information to construct the state machine. Any modifications of the smg file will directly change the state machine in DOVE. By creating a new smg file, we can generate a new extended state machine without starting up the DOVE. We can construct the extended state machine which is the composition of more than one component in one model without the need to interact with DOVE.

Using the above information, we parse the smg files of component automata to extract information to reconstruct the automata. From this, we build the product automaton. For this, we follow quite closely the mathematical description given in the previous section. The code was written in SML [55], a functional programming language similar to the typed lambda calculus. SML data types and functions are used to compute the constructions previously given as mathematical formulas. After constructing the product, we still need to generate layout information before we can generate a smg file to add the product automaton to DOVE.

In DOVE, layout information is generated from interactions with the user. The user places nodes at various locations on the drawing canvas and draws edges between the various nodes, indicating curvature by the path of the mouse. The layouts may be altered by clicking and dragging the various entities to be changed. DOVE does some work to generate a decent presentation of the graph, but the basic layout information comes from the user. When we automatically generate the product automata, we must also automatically generate some positioning for the components; to make the user generate this information would be almost tantamount to making the user create the product in the first place. To generate this information, we make use of the graph visualization tool dot [63]. Dot is applied to a file that lists the nodes and edges of a directed graph, together with any desired labeling of the nodes and edges, and the desired shape (and color) of the nodes. For each node, dot adds the size (height and width) of the circle and the position of its center. Each edge is extended with path information, consisting of the position and direction of the terminating arrowhead follow by a sequence of coordinates that the edge will pass through, and the coordinates of the left edge of the label.

We must parse the information returned from dot and combine it with the non-graphical information for the product automaton. Also, the graphical information produced by dot is not completely suitable for directly inputting into an smg file. We need to perform scaling, and better layouts seem to be given by thinning the points for layout of the transitions. Once we adjust the information from dot and combine it with the non-graphical information, we can finally produce an smg file that describes the product automaton to DOVE. Once this file exists, the user can start up DOVE with it, and proceed to state and prove properties about it.

We began this project because we were attempting to use DOVE to reason about a medium-sized real-world safety-critical system. This system could be naturally decomposed into a hierarchy of subsystems communicating through limited intefaces of input and output variables. In attempting to use DOVE, we found ourselves attempting to compose these subsystems by hand. The work described above outlines a way to build the interactive components into one extended state machine by extending DOVE with product automata. By using the information we get from parsing the smg file in DOVE, we can create a new state machine graph externally without have to use DOVE to create it interactively.

With future extensions of this tool, we should be able to reason about the various components and then have those results automatically carried over to the product when the product is formed or its theory is subsequently updated.

### 3.1.4 Applications

The example given below is intended to monitor the behavior of another device. This example consists of two components: a component for monitoring, whether the device is plugged in and receiving adequate power, and a component for monitoring when the device is adequately powered, whether it is producing values within an acceptable range.

Figure 3.1 shows a screen snapshot of the DOVE canvas for the PlugIn Monitor component of the system. The gridded canvas is the DOVE state machine window which is used for designing the machine. The three nodes representing the three states in the PlugIn Monitor model are Wait, CheckPlugin and CheckUnplug. The edges with appropriate labels are transitions between these states. Several variables are needed. The heap variable PluggedIn represents whether the machine is plugged in. The input variable Volt is supplied by the environment and is monitored to trace when the device is properly plugged in. Finally, an initial state Wait should be defined in which the machine is unplugged.

The system checks whether the device is plugged in before going from the Wait to the CheckPlugIn mode. We have the variable Volt as the guard for the three transitions: PlugIn, Unplug and RePlugin. At each transition, if the guard conditions are meet, the corresponding transition will be taken, and the variables will be updated. In the initial state, if the device is plugged in and receiving a voltage greater than 10 volts, the transition Plugin will be taken and PluggedIn will be set to true. The plug monitor will stay in the CheckPlugin state unless the voltage drops below 10 volts. In that case, it will enter the CheckUnplug state and PluggedIn will be updated to false. Once the device is replugged in and receiving more than 10 volts, it will reenter the CheckPlugin state. The monitor will keep running in this loop infinitely. Here, the PlugIn Monitor is correctly and clearly modeled in DOVE.

Now we can formally prove some safety properties of the PlugIn Monitor using the DOVE Property Manager. One important requirement of the PlugIn Monitor is that if the value of Volt is dropped under 10 then the variable PluggedIn is set to be false. Verification in DOVE corresponds to proving that all executions of a state machine satisfy a certain property. This property can be represented using the turnstile ("|-") operator. For example, the above property can be written as,

$$\vdash \text{(Previously (Volt = 5))} \longrightarrow \text{(PluggedIn = False)}$$

The basic idea of proof in DOVE is to use induction on the execution of the state machine. Suppose the initial state satisfies the property and every transition of the state machine also preserves the property, then the property holds for all executions on the state machine.

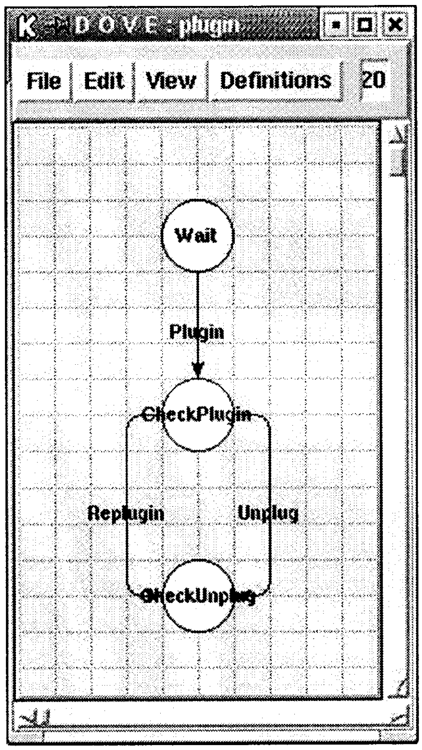


**Figure 3.1** A simple plug monitor in DOVE.

Figure 3.2 shows the DOVE Prover. We proved the above property by three steps, Topology, BackSubstitute and MasterBlast.

However, the PlugIn Monitor is just a simple example of modeling a system. Life is not always so easy. When dealing with a bigger project in which some models interact with each other, some problems come up. The Value monitor is a component in which the variable ValueOk shows the status of the value variable. The state machine of Value Monitor is showed as Figure 3.3. The three states Wait, CheckValueOk and CheckValueFault are defined in the Value Monitor state machine. Six transitions connect these states and update variables if the guard of the transition is satisfied.

In the initial state of Wait, once the variable PluggedIn becomes true, the variable ValueOk will be set to true. The device will enter the CheckValueOk state. This can happen in one of two ways. When the system being monitored first starts up, the PlugIn Monitor and the Value Monitor synchronize on beginning to monitor it's state. Thereafter, if the power drops below a certain threshold, then the Value Monitor returns to its Wait state, and reenters CheckValueOk when it detects that the PlugIn Monitor has determined that the power has returned to an acceptable level. Once in the CheckValueOk state, if the input variable Test is shown to be below 5, ValueOk is set to false, the device will enter the CheckValueFault state. If variable Test is set back to greater than 5, ValueOk is set back to true, and the CheckValueOk state will be reentered. In both CheckValueOk and CheckValueFault states, if the device is unplugged, the device will go back to initial Wait state.

Between these two models, the Value Monitor uses the PluggedIn variable, which is written by the PlugIn Monitor, as an input variable. Unfortunately, with the current DOVE tools, these two interactive components could not be composed into one single model. In order to conquer this, we need to extend DOVE with product automata.

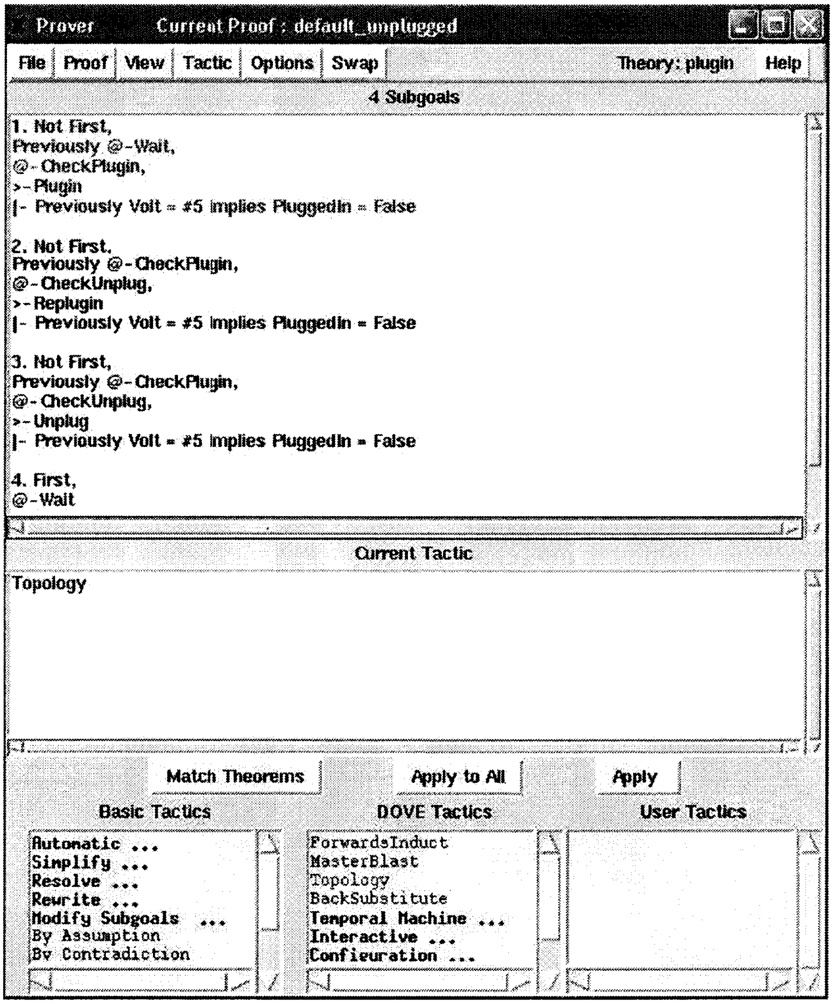


**Figure 3.2** Property Proof in DOVE.

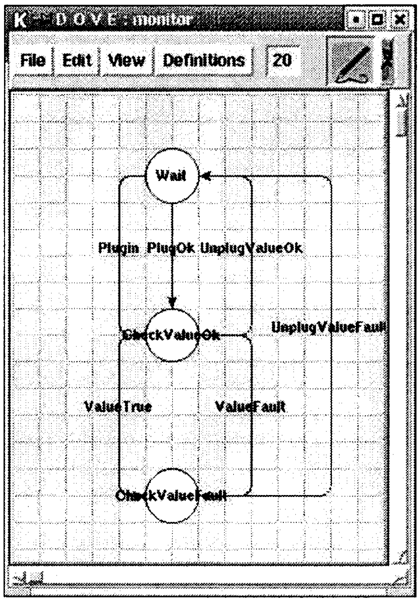


**Figure 3.3** A Monitor for Checking Values in DOVE.

We have started the DOVE with the product state machine graph file produced from the PlugIn and Value Monitor components. Figure 3.4 shows a screen snapshot of DOVE with the product in editing mode.

Now we can prove some properties concerning both the PlugIn and Value monitors in the product automaton. In the Value monitor, the variable ValueOk is set to be true if and only if the value of PluggedIn is true and the value of Value is greater than or equal to 5 in the previous state. The value of PluggedIn is set to be true if the value of Volt is greater than or equal to 10 in the previous state. For example, we have the following property:

$$\vdash \text{Previously ((Previously Volt=12) And Value=6))} \longrightarrow \text{ValueOk = True}$$

We state the temporal property in the DOVE proof manager and graphic prover as in Figure 4.1. By using *topology*, the property is split into cases uniquely determined by the graph information from the graph. Figure 3.6 shows the result after using *topology* tactic.

The tactic *topology* produces 16 subgoals. They clearly describe what are the previous states, what must be true in each previous state in order to get the current state,

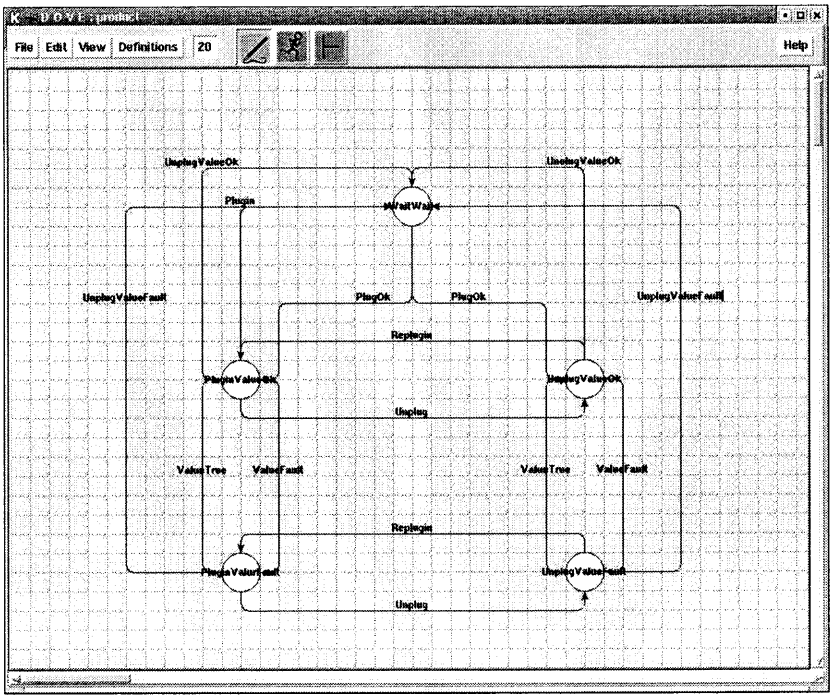


**Figure 3.4** Product of PlugIn and Value Monitor.

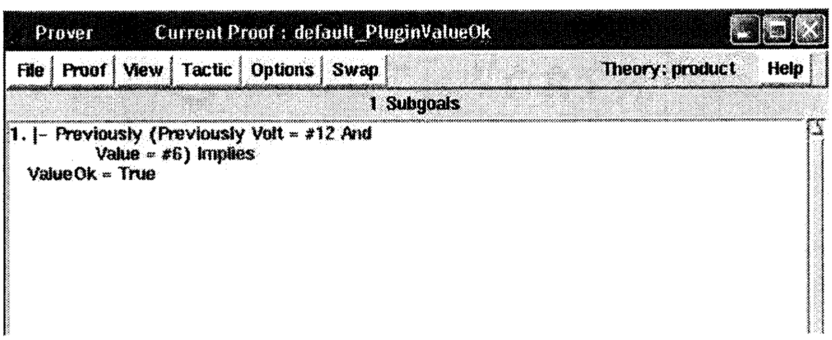


**Figure 3.5** Property Proof in DOVE Prover.

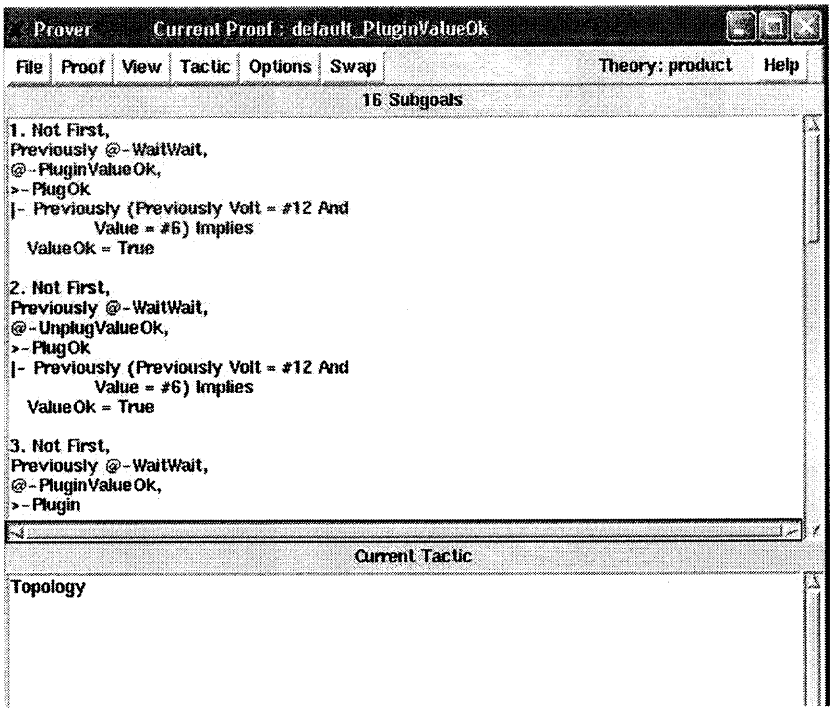


**Figure 3.6** Result of Topology in DOVE Prover.

and what is needed to be proved in current state. We use DOVE's *back-substitute* to replace occurences of variable names in the corresponding temporal sequent with the values assigned to them by the last transition. Finally, tactic *MasterBlast* can be used to prove the subgoal for the *initial* state. Then the temporal property is proved as shown in Figure 3.7.

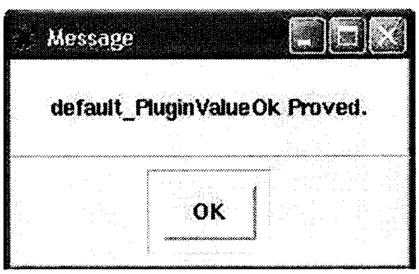


**Figure 3.7** Finished Proof.

## 3.2 Formulating LTL in Isabelle

Work on embedding temporal logics has been done by Agerholm and Skjødt [64], Clarke and Emerson [65], and Schneider and Hoffmann [66]. In this chapter, we present a formal formulation of linear temporal logic (LTL) in the Isabelle theorem prover. The syntax and semantics of LTL are formally defined. Also, the axioms and proof rules are provided for the complete axiomatization of LTL. Later in this chapter, we introduces how LTL is used for system specifications [67, 68, 69].

### 3.2.1 Embedding LTL in Isabelle

In this work, LTL is built on propositional logic [70]. We chose Isabelle's Higher-Order Logic as the object logic to build the embedding of LTL since HOL is a well developed logic with many tools and extensions built on it.

We use the facilities in the Isabelle system for embedding different logics to present a formulation of LTL. Isabelle has also proved useful for doing large proofs, having many tools that allow the automation of difficult and tedious details. Thus it is particularly suitable for both implementing LTL as well as actually using it to develop proofs in LTL.

In Chapter 2, we already gave the syntax and semantics definition of LTL. This embedding of LTL in Isabelle is very close to the syntax and semantics we gave in Chapter 2 with one bit of expansion. We add the logical atoms true and false. These are logically equivalent to the propositions True and False. However, by separating them out, the algorithm is capable of producing a smaller automaton when these LTL versions of true and false are used instead of the propositions True and False. As a concession to efficiency, we have added these two atoms in to the Isabelle definition of LTL formulae. To capture normal form LTL formulae, we have defined in Isabelle the following datatype:

$$
\begin{array}{lcllcl}
\text{datatype } \alpha \text{ ltl} & = & \text{ltl\_True} & & | & \text{ltl\_False} \\
& | & \text{Base "}\alpha\text{ propsi"} & & | & \text{Neg "}\alpha\text{ propsi"} \\
& | & \text{Until "}\alpha\text{ ltl" "}\alpha\text{ ltl"} & & | & \text{Release "}\alpha\text{ ltl" "}\alpha\text{ ltl"} \\
& | & \text{Or "}\alpha\text{ ltl" "}\alpha\text{ ltl"} & & | & \text{And "}\alpha\text{ ltl" "}\alpha\text{ ltl"} \\
& | & \text{Next "}\alpha\text{ ltl"} & & | & \text{Weak\_Next "}\alpha\text{ ltl"}
\end{array}
$$

where $\alpha$ propsi $= \alpha \Rightarrow$ bool.

The modalities *Eventually*::($\alpha$ ltl$\Rightarrow\alpha$ ltl), *Always*::($\alpha$ ltl$\Rightarrow\alpha$ ltl), *ltl_Not*::($\alpha$ ltl$\Rightarrow\alpha$ ltl), and *Imply*($\rightarrow$)::([$\alpha$ ltl, $\alpha$ ltl] $\Rightarrow\alpha$ ltl) are defined as syntactic sugar.

$$
\begin{array}{lcl}
\text{Eventually } \varphi & = & \text{Until ltl\_True } \varphi \\
\text{Always } \varphi & = & \text{Relase ltl\_False } \varphi \\
\text{ltl\_Not ltl\_True} & = & \text{ltl\_False} \\
\text{ltl\_Not ltl\_False} & = & \text{ltl\_True} \\
\text{ltl\_Not (Base } \varphi) & = & \text{Neg } \varphi \\
\text{ltl\_Not (Neg } \varphi) & = & \text{Base } \varphi \\
\varphi \rightarrow \psi & = & \text{Or (ltl\_Not } \varphi) \ \psi)
\end{array}
$$

The semantics of LTL formula are given using the satisfiability predicate:

$$\_ \models \_ :: (\alpha)\ behavior \Rightarrow \alpha\ ltl \Rightarrow bool$$

The notation $\xi \models \varphi$ means that a behavior $\xi$ satisfies the LTL formula $\varphi$. Thus, for each behavior $\xi$, we have $\xi \models ltl\_True$. Also, we observe that $\varphi \rightarrow \psi$ if and only if $\xi \models \varphi \rightarrow \xi \models \psi$, and $\xi \models \varphi \leftrightarrow \xi \models \psi$ if and only if $\xi \models \varphi \rightarrow \psi$ and $\xi \models \psi \rightarrow \varphi$. In addition to the equations we give in Chapter 2, we also have $(\varphi \vee \psi) = \neg((\neg\varphi) \wedge (\neg\psi))$, $(\varphi \rightarrow \psi) = (\neg\varphi) \vee \psi$, $\Box\varphi = \neg(\Diamond(\neg\varphi))$. Parentheses can often be omitted by defining priority on logical connectives. Priority for operators in LTL are, by descending order, $\bar{\bigcirc}, \bigcirc, \Diamond, \Box, \bigcup, \bigvee$. The operator $\bigcup$ is associate to right, e.g., $\varphi \bigcup \psi \bigcup \phi = (\varphi \bigcup (\psi \bigcup \phi))$. The operator $\bigvee$ is associate to left, e.g., $\varphi \bigvee \psi \bigvee \phi = ((\varphi \bigvee \psi) \bigvee \phi)$. Some examples of LTL formulae and their meaning are listed below:

| | |
|---|---|
| $\varphi \rightarrow \bigcirc\psi$ : | "If $\varphi$ then $\psi$ in the next state". |
| $\varphi \rightarrow \Box\psi$ : | "If $\varphi$ then always $\psi$ in all states". |
| $\varphi \rightarrow \Diamond\psi$ : | "If $\varphi$ then in some later state $\psi$". |
| $\Box(\varphi \rightarrow \psi)$ : | "In whatever state, if $\varphi$ then $\psi$ in that state". |
| $\Diamond\Box\varphi$ | "Starting from some state, $\varphi$ will hold permanently". |
| $\Box\Diamond\varphi$ | "For all states, $\varphi$ will hold in some later state". |

Our embedding of temporal logic in Isabelle is quite different from the embedding of temporal logic in DOVE. We interpret LTL on both infinite and finite sequences. Our logic is in a *future-fashion* instead of *past-fashion* used in DOVE. This enables us to handle both safety and liveness properties. While in DOVE, only safety properties can be checked.

### 3.2.2 Axiomatization of LTL

After giving the formal definitions, we also provide an axiomatization for propositional LTL [71]. Following theorems can form an axiomatization of LTL:

$$
\begin{array}{ll}
A_1 & \xi \models \neg\Diamond\varphi \leftrightarrow \xi \models \Box\neg\varphi \\
A_2 & \xi \models \Box(\varphi \rightarrow \psi) \rightarrow \xi \models (\Box\varphi \rightarrow \Box\psi) \\
A_3 & \xi \models \Box\varphi \rightarrow \xi \models (\varphi \wedge \bar{\bigcirc}\Box\varphi) \\
A_4 & \xi \models \neg\bar{\bigcirc}\varphi \leftrightarrow \xi \models \bigcirc\neg\varphi \\
A_5 & \xi \models \bigcirc(\varphi \rightarrow \psi) \rightarrow \xi \models (\bigcirc\varphi \rightarrow \bigcirc\psi) \\
A_6 & \xi \models \bar{\bigcirc}(\varphi \rightarrow \psi) \rightarrow \xi \models (\bar{\bigcirc}\varphi \rightarrow \bar{\bigcirc}\psi) \\
A_7 & \xi \models \Box(\varphi \rightarrow \bigcirc\psi) \rightarrow \xi \models (\varphi \rightarrow \Box\psi) \\
A_8 & \xi \models (\varphi \bigcup \psi) \leftrightarrow \xi \models (\psi \vee (\varphi \wedge (\bigcirc(\varphi \bigcup \psi)))) \\
A_9 & \xi \models \bigcirc\varphi \rightarrow \xi \models \Diamond \bigcirc \mathit{ltl\_True} \wedge \bar{\bigcirc}\varphi
\end{array}
$$

These theorems are the basis of a *sound* and *complete* relative to a complete system for propositional logic. The *soundness* of a system assure that only correct assertions can be proved. It is proved by showing all the theorems can be proved from our definitions. A system is called *complete* if it is capable to prove all correct formula that can be expressed using the system. These theorems can be used as the basis of a proof system. They are not proved here because we are not providing a formal proof system here. However, the following is a list of theorems derivable from A1 – A9 and the rule for implication without resorting back to the definition of $\models$:

R1: $\neg\Box\varphi \leftrightarrow \Diamond\neg\varphi$
R2: $\neg\Diamond\varphi \leftrightarrow \Box\neg\varphi$
R3: $\Box\varphi \rightarrow \varphi$
R4: $\varphi \rightarrow \Diamond\varphi$
R5: $\Box\varphi \rightarrow \bigcirc\varphi$
R6: $\bigcirc\varphi \rightarrow \Diamond\varphi$
R7: $\Box\varphi \rightarrow \Diamond\varphi$
R8: $\Diamond\Box\varphi \rightarrow \Box\Diamond\varphi$
R9: $\Box\Box\varphi \leftrightarrow \Box\varphi$
R10: $\Diamond\Diamond\varphi \leftrightarrow \Diamond\varphi$
R11: $\bigcirc(\varphi \wedge \psi) \leftrightarrow \bigcirc\varphi \wedge \bigcirc\psi$
R12: $\bigcirc(\varphi \vee \psi) \leftrightarrow \bigcirc\varphi \vee \bigcirc\psi$
R13: $\Box(\varphi \wedge \psi) \leftrightarrow \Box\varphi \wedge \Box\psi$
R14: $\Diamond(\varphi \vee \psi) \leftrightarrow \Diamond\varphi \vee \Diamond\psi$
R15: $\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$
R16: $\Box\varphi \vee \Box\psi \rightarrow \Box(\varphi \vee \psi)$
R17: $(\Diamond\varphi \rightarrow \Diamond\psi) \rightarrow \Diamond(\varphi \vee \psi)$
R18: $\Diamond(\varphi \wedge \psi) \rightarrow \Diamond\varphi \wedge \Diamond\psi$
R20: $\Box\varphi \rightarrow \varphi \wedge \bigcirc\Box\varphi$
R21: $\Diamond\varphi \rightarrow \varphi \wedge \bigcirc\Diamond\varphi$
R22: $\Box(\varphi \rightarrow \psi) \rightarrow (\Diamond\varphi \rightarrow \Diamond\psi)$
R23: $\Box\varphi \rightarrow (\Box\psi \rightarrow \Box(\varphi \wedge \psi))$
R24: $\Box\varphi \rightarrow (\Diamond\psi \rightarrow \Diamond(\varphi \wedge \psi))$
R25: $\Box(\Box\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$
R26: $\Box(\varphi \rightarrow \Diamond\psi) \rightarrow (\Diamond\varphi \rightarrow \Diamond\psi)$

### 3.2.3 System Properties Specification using LTL

Specifying the system itself and specifying its properties are different activities. Although, the system and its properties can be given with same formalism, automata, for example. In many cases, they are expressed by different formalisms. The system can be described using transition systems or automata. The system modeling was addressed in part in DOVE and is not closely related to our work here. The system properties, on the other hand, can be given in a logical formalism. In our work, we choose LTL as such a formalism because the simple formalism of LTL is surprisingly powerful when specifying properties of inter leaving sequences and modeling the execution of a program. As we mentioned in Chapter 2, Linear Temporal Logic has been deemed expressive enough for most purposes [72], while retaining a relatively simple syntax and semantics.

Let $P$ be a system that admits multiple executions. Such a system can be described using a transition system or an automaton. Each execution of $P$ is represented by a *behavior*, which is a finite or infinite sequence of states. Let $\Gamma$ be a set of *behaviors* generated for the system $P$ and $\varphi$ be any LTL formulae. If all the *behaviors* of system $P$ satisfies $\varphi$, we write $P \models \varphi$. If not all the *behaviors* of system $P$ satisfies $\varphi$, then we write $P \not\models \varphi$. Notice that $P \not\models \varphi$ does not mean $P \models \neg\varphi$; sometimes when $P \not\models \varphi$, some of *behaviors* do satisfy $\varphi$.

We are particularly interested in two kinds of system properties: *safety properties* [73] and *liveness properties* [74, 75]. Safety properties asserts the absence of undesirable states during a certain time. In another words, nothing bad will happen, e.g. a television system will not shut off itself without a user pressing the power off button. Liveness properties assert some desirable state will eventually be reached. Unlike safety properties, liveness properties require something good will happen, e.g. the television will change the channel if a user pushes the change channel button.

In our work, one of the reason we interpret LTL on *behaviors* instead of only finite sequences is that both safety properties and liveness properties can be easily expressed.

### 3.3 Formalizing the Translation of LTL Formulae to Büchi Automata

The algorithm for translating LTL formulae into Büchi Automata is widely used in model checking field. In this work, we present a formulation of the translation algorithm from Gerth *et al.* [35]. The translation algorithm has been improved by Daniele and Giunchiglia and Vardi [76], Schneider and Hoffmann [66], Couvreur [77], Gastin and Oddoux [78], Giannakopoulou and Lerda [79], Somenzi and Bloem [80], and Thirioux [81].

### 3.3.1 Translating LTL into Büchi Automata

In this section, we present our algorithm for translating an LTL formula $\mu$ into an automaton that accepts exactly all words satisfying $\mu$. We modify the algorithm presented in [35] by Gerth *et al.* and [82] by Gunter and Peled. The algorithm in [35] can only produce automata that accept infinite words. The algorithm in [82] can only output automata that accept finite words. We merger these two algorithms to a new version that can produce automata that accept both finite and infinite words. We also use a variation of Büchi Automata we introduced in Chapter 2, $B = (\Sigma, S, \Delta, I, L, Fset, F)$. The new version of automata is defined by us to accept both finite and infinite words.

Before applying the translation algorithm, we need convert the formula $\mu$ into *negation normal form*, where negation can only be applied to the propositional variables. It is done using the LTL equivalences $\neg\bar{\bigcirc}\varphi = \bigcirc\neg\varphi, \neg(\varphi \vee \psi) = (\neg\varphi) \wedge (\neg\psi), \neg(\varphi \wedge \psi) = (\neg\varphi) \vee (\neg\psi), \neg\neg\varphi = \varphi, \neg(\Box\varphi) = \Diamond\neg\varphi, \neg(\Diamond\varphi) = \Box\neg\varphi$. Next, we eliminate all the occurrences of *eventually* ($\Diamond$) and *always* ($\Box$) operators, using the *until* ($\bigcup$) and *release* ($\bigvee$) operators and the equivalences $\Diamond\varphi = true \bigcup \varphi$ and $\Box\varphi = false \bigvee \varphi$.

The algorithm takes an LTL formula $\varphi$ as input and constructs a graph with states and transitions as the output automaton. The algorithm decomposes the formula $\varphi$ according to its boolean structure and temporal operators. The following data structure is used by the algorithm as a graph node for the generated automaton $B$:

*Name.* A unique identifier of the node.

*Incoming.* A set of the identifiers of nodes with edges that point to the current node.

*New.* A set of subformulae of $\varphi$ that must hold at the current node and have not been processed yet.

*Old.* A set of subformulae of $\varphi$ that must hold at the current node and have already been processed.

*Next.* A set of subformulae of $\varphi$ that must hold at every immediate successors of the current state.

*Strong*. A flag showing whether the current node must not be the last one in the sequence.

The *Strong* field is originally introduced in [82] by Gunter and Peled to indicate when the current state cannot be the last one in the sequence. We also keep a set *Nodes_Set* of nodes, each having the same fields above. The set *Nodes_Set* is initially empty and will contain all the nodes we need to build the automaton once the algorithm terminates.

The main idea of the algorithm is to separate the LTL formulas into two parts: one that holds in the current state, and the other that holds in the next state, using:

$$\varphi \mathsf{U} \psi = \psi \vee (\varphi \wedge \bigcirc \varphi \mathsf{U} \psi)$$
$$\varphi \mathsf{V} \psi = (\varphi \wedge \psi) \vee (\psi \wedge \bar{\bigcirc} \varphi \mathsf{V} \psi)$$

Several small functions used by the algorithm are defined. The function new_name() generates a new unique name for each new created node. The functions *New1*, *New2*, *Next2* are defined in Table 3.1:

**Table 3.1** Functions for Splitting LTL Formulae.

| Formula | New1 | Next1 | New2 | Strong |
|---|---|---|---|---|
| $\varphi \mathsf{U} \psi$ | $\{\varphi\}$ | $\{\varphi \mathsf{U} \psi\}$ | $\{\psi\}$ | $\surd$ |
| $\varphi \mathsf{V} \psi$ | $\{\psi\}$ | $\{\varphi \mathsf{V} \psi\}$ | $\{\varphi, \psi\}$ | |
| $\varphi \wedge \psi$ | $\{\varphi, \psi\}$ | $\emptyset$ | - | |
| $\varphi \vee \psi$ | $\{\varphi\}$ | $\emptyset$ | $\{\psi\}$ | |
| $\bigcirc \varphi$ | $\emptyset$ | $\{\varphi\}$ | - | $\surd$ |
| $\bar{\bigcirc} \varphi$ | $\emptyset$ | $\{\varphi\}$ | - | |

A new function SF takes an LTL formula as input and calculates the total subformulae we can get by decomposing it. Thus we have

$$
\begin{aligned}
\mathsf{SF}\ \mathsf{ltl\_True} &= \{\mathsf{ltl\_True}\} \\
\mathsf{SF}\ \mathsf{ltl\_False} &= \{\mathsf{ltl\_False}\} \\
\mathsf{SF}\ \mathsf{Base}\,\beta &= \{\mathsf{Base}\,\beta\} \\
\mathsf{SF}\ \neg\beta &= \{\neg\beta\} \\
\mathsf{SF}\ \varphi \wedge \psi &= \{\varphi \wedge \psi\} \cup (\mathsf{SF}\ \varphi) \cup (\mathsf{SF}\ \psi) \\
\mathsf{SF}\ \varphi \vee \psi &= \{\varphi \vee \psi\} \cup (\mathsf{SF}\ \varphi) \cup (\mathsf{SF}\ \psi) \\
\mathsf{SF}\ \bigcirc\varphi &= \{\bigcirc\varphi\} \cup (\mathsf{SF}\ \varphi) \\
\mathsf{SF}\ \bar{\bigcirc}\varphi &= \{\bar{\bigcirc}\varphi\} \cup (\mathsf{SF}\ \varphi) \\
\mathsf{SF}\ \varphi \bigcup \psi &= \{\varphi \bigcup \psi\} \cup (\mathsf{SF}\ \varphi) \cup (\mathsf{SF}\ \psi) \\
\mathsf{SF}\ \varphi \bigvee \psi &= \{\varphi \bigvee \psi\} \cup (\mathsf{SF}\ \varphi) \cup (\mathsf{SF}\ \psi)
\end{aligned}
$$

where $\beta$ ranges over basic propositions and $\varphi$ and $\psi$ range over LTL formulae. Another function max_SF is defined based on the function SF. Given a set of LTL formulae $A$, max_SF returns a single formula $\varphi$ from $A$ such that for all $\psi$ in $A$, the cardinality of SF $\varphi$ is less than or equal to the cardinality of SF $\psi$.

The algorithm for translating an LTL formula into a generalized Büchi automata [83] is presented in Figure 3.8. To translate the LTL formula $\mu$, the algorithm starts with a single node (line 42-45) that has a single incoming edge from a dummy special node *init*. The *new* field of the node contains the formula $\mu$ and has empty *old* and *next* fields. And the result set *Nodes_Set* is initialized to be empty.

The algorithm works recursively. For the current node $s$, the algorithm checks if there are subformulae to be processed in the field *New* in $s$ (line 4). If the field *New* is empty, then the node is *completely-processed*. We need to check if it should be adds to the *Nodes_Set*. If there is a node $r$ in *Nodes_Set* what has the same subformulae as $s$ in its *Old* and *Next* fields and has the same *Strong* field (lines 5-6), then we do not need to add $s$ into *Nodes_Set*. Instead, the set of *Incoming* of $s$ are added to the *Incoming* of $r$ (line 7). If there

```
record graph_node = [Name : string, Incoming : set of string,
  New : set of formula, Old : set of formula, Next : set of formula, Strong : bool];
function expand(s, Nodes_Set)
  if New(s) = ∅ then
    if exists node r in Nodes_Set with
        Old(r) = Old(s) and Next(r) = Next(s)
    Incoming(r); = Incoming(r) ∪ Incoming(s);
        return(Nodes_Set);
    else return(expand([Name ⇐ new_name(),
        Incoming ⇐ {Name(s)}, New ⇐ Next(s),
        Old ⇐ ∅, Next ⇐ ∅, Strong(s)], Nodes_Set ∪ {s}))
  else
    let η = max_SF(New(s));
    New(s) := New(s) \ {η}; Old(s) := Old(s) ∪ {η};
    case η of
    η = A, or ¬A, where A is a proposition, or η = true, or η = false ⇒
        if η = false or ¬η ∈ Old(s) then return(Nodes_Set)
        else return (expand([Name ⇐ Name(s), Incoming ⇐ Incoming(s),
        New ⇐ New(s), Old ⇐ Old(s), Next ⇐ Next(s), Strong(s)], Nodes_Set));
    η = φ⋃ψ
        s1 := [Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s) ∪ ({New1(η)} \ Old(s))
            Old ⇐ Old(s), Next = Next(s) ∪ {Next1(η)}, true];
        s2 := [Name ⇐ Name(s),
            Incoming ⇐ Incoming(s),
            New ⇐ New(s) ∪ ({New2(η)} \ Old(s)),
            Old ⇐ Old(s), Next ⇐ Next(s), Strong(s)]);
        return(expand(s2, expand(s1, Nodes_Set)));
    η = φ⋁ψ, or φ ∨ ψ ⇒
        s1 := [Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s) ∪ ({New1(η)} \ Old(s))
            Old ⇐ Old(s), Next = Next(s) ∪ {Next1(η)}, Strong(s)];
        s2 := [Name ⇐ Name(s), Incoming ⇐ Incoming(s)
            New ⇐ New(s) ∪ ({New2(η)} \ Old(s)),
            Old ⇐ Old(s), Next ⇐ Next(s), Strong(s)]);
        return(expand(s2, expand(s1, Nodes_Set)));
    η = φ ∧ ψ ⇒
        return(expand([Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s) ∪ ({φ, ψ} \ Old(s)),
            Old ⇐ Old(s), Next ⇐ Next(s), Strong(s)], Nodes_Set))
    η = ◯φ ⇒
        return(expand([Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s), Old ⇐ Old(s),
            Next ⇐ Next(s) ∪ {φ}, true], Nodes_Set))
    η = ◯̄φ ⇒
        return(expand([Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s), Old ⇐ Old(s),
            Next ⇐ Next(s) ∪ {φ}, Strong(s)], Nodes_Set))
end expand;
function creat_graph(μ)
  return(expand([Name ⇐ new_name(), Incoming ⇐ {init},
    New ⇐ {μ}, Old ⇐ ∅, Next ⇐ ∅, false], ∅))
create_graph;
```

**Figure 3.8** The LTL Translation Algorithm.

is no such node $r$ in *Nodes_Set*, then $s$ is added to *Nodes_Set* and a new node $s'$ is created (lines 9-11). A fresh name is given to $s'$. The *Incoming* field of $s'$ contains the name of $s$. The *Next* field of $s$ is the *New* field of $s'$. Also the *Old* and *Next* fields of $s'$ are initialized to be empty.

On the other hand, if the field *New* is not empty, we use the function max_SF to select a formula $\eta$ in *New* and remove it from *New*. In the original Gerth algorithm, and in the Gunter-Peled algorithm, the choice of the formula $\eta$ is non-deterministic. In our algorithm, we use the function max_SF to eliminate the non-determinism by choosing the maximal formula $\eta$ instead of choosing an arbitrary formula because this is helpful for us to prove the termination of the algorithm later on.

If $\eta$ is a literal and $\neg\eta$ in *Old*, then the current node is discarded since it contains a contradiction(lines 16-17). Otherwise, $\eta$ is added to *Old*, if it is not already there.

If $\eta$ is not a literal, $s$ is processed according to the outmost operator of $\eta$ as follows:

$\eta = \varphi \bigcup \psi$: The node $s$ is split into $s_1$ and $s_2$ (lines 20-28). For the first copy $s_1$, $\varphi$ is added to *New* and $\varphi \bigcup \psi$ is added to *Next*. For the second copy $s_2$, $\psi$ is added to *New*. The *Strong* field of $s_1$ is set to be true. The *Strong* field of $s_2$ is set to be *Strong*(s). The fact that $\varphi \bigcup \psi = \psi \vee (\varphi \wedge \bigcirc(\varphi \bigcup \psi))$ is used in the splitting.

$\eta = \varphi \bigvee \psi$: The node $s$ is split into $s_1$ and $s_2$ (lines 29-38). For the first copy $s_1$, $\psi$ is added to *New* and $\varphi \bigvee \psi$ is added to *Next*. For the second copy $s_2$, both $\varphi$ and $\psi$ are added to *New*. Both *Strong* fields of $s_1$ and $s_2$ are set to be *Strong*(s). The fact that $\varphi \bigvee \psi = \psi \wedge (\varphi \vee \bigcirc(\varphi \bigvee \psi))$ is used in the splitting.

$\eta = \varphi \vee \psi$: The node $s$ is split into $s_1$ and $s_2$ (lines 29-38). $\varphi$ is added to the *New* of $s_1$ and $\psi$ is added to the *New* of $s_2$. Both *Strong* fields of $s_1$ and $s_2$ are set to be *Strong*(s).

$\eta = \varphi \wedge \psi$: A replacement node $s'$ of $s$ is created (lines 39-42). Both $\varphi$ and $\psi$ are added to the *New* of $s'$. Both *Strong* fields of $s_1$ and $s_2$ are set to be *Strong*(s).

$\eta = \bigcirc\varphi$: A replacement node $s'$ of $s$ is created (lines 43-46). $\varphi$ is added to the *Next* of $s'$. The *Strong* field of $s'$ is set to be true.

$\eta = \bar{\bigcirc}\varphi$: A replacement node $s'$ of $s$ is created (lines 47-50). $\varphi$ is added to the *Next* of $s'$. The *Strong* field of $s'$ is set to be *Strong*(s).

The function create_graph is the start of the whole translation algorithm. create_graph takes an LTL formula $\mu$ as input and calls the expand function. The first argument of the expand is a node with $\mu$ in its *New* field, *init* in its *Incoming* field. The *Old* and *Next* fields of the node are set to be empty. The second argument of expand is an empty node set.

The above description of the algorithm for translating an LTL formula into a Büchi automaton is imperative, in keeping with the spirit of the algorithm presented by Gerth *et al.* in [35]. In order to reason about this algorithm in Isabelle it was necessary to functionalize it. In place of updates to fields of existing nodes, we have to create new elements. In place of updating the *Incoming* filed of existing nodes in *Nodes_Set*, we must create a new *Nodes_Set* with the node to be "updated" removed and a new node with increased *Incoming* field added. Similarly, the functions such as new_name upon which expand depends must also be functionalized.

Once the algorithm terminates, we can convert the set of nodes *Nodes_Set* into a generalized Büchi automaton $B = (\Sigma, S, \Delta, I, L, Fset, F)$ as follows:

- The alphabet $\Sigma$ consists of sets of sets of negated and non-negated propositions that appear in the translated formula $\varphi$.
- The set of states $S$ consists of the nodes in *Nodes_Set*.
- $(s, s') \in \Delta$ when $Name(s) \in$ *Incoming*$(s')$.

- The set of initial states $I \subset S$ is the set of nodes that contain the special incoming edge *init*.

- $L(s)$ is the set of all sets $X$ of propositions such that every proposition that has a positive occurrence in a formula of *Old* of $s$ is in $X$, while no proposition that has a negative occurrence is in $X$.

- The accepting states set $Fset$ satisfies that for each $f \in Fest$, there is a formula of the form $\varphi \bigcup \psi$ such that the *Old* field of every node in $f$ either contains $\psi$, or does not contain $\varphi \bigcup \psi$.

- The finite accepting states set $F$ contains all the states $s$ that $s \in \bigcap Fset$ and $(\textit{Strong } s) = \mathsf{false}$.

### 3.3.2 Termination Proof of the Algorithm

The algorithm of Gerth *et al.* [35] has been used for many tools in practice, e.g., the model checker SPIN [84]. However, a formal proof the termination of the algorithm does not exist in the literature. It's critical that a verification algorithm itself to be proved to be correct. The termination is a fundamental requirement for the correctness of the algorithm. Thus, here we propose a method to define the algorithm in a generic theorem prover, Isabelle and give the formal proof of the termination.

The translation algorithm works recursively. Proving termination of a recursive algorithm can be achieved by finding a *well-founded relation R* on the inputs to the function and showing that the recursive calls decrease under the relation $R$ [85, 86].

Each total recursive function defined in Isabelle must specify a well-founded relation to justify the termination of the function. Formally, the relation $\prec$ is *well-founded* if it admits no infinite descending chains

$$\cdots \prec a_2 \prec a_1 \prec a_0.$$

Isabelle provides theorems for constructing a well-founded relation. We use a way to specify a *measure function* $f$ into the natural numbers, where $x \prec y \iff f\ x < f\ y$. However, in the translation algorithm, there is no obvious single well-founded relation on the arguments to the algorithm such that the argument decreases for every recursive call. In each recursive call, not all the arguments are necessarily decreasing under the usual measures.

To find a useful well-founded relation, we observe that, at the outermost level, we do two different kinds of recursive calls. The first kind of recursive call will add a new node to *Nodes_Set* and start processing *Next* as *New*. The second kind of recursive call is when there are formulae in *New*, and the recursive call is made to a new node structure where one of the formulae in *New* has been broken up.

In the first case, the remaining nodes we can create from the original formula is decreasing, although not strictly. The nodes in *Nodes_Set* are uniquely determined by their *Old* and *Next* components. The *New* field must be empty. We never put two different nodes with the same *Old* and *Next* into *Nodes_Set*, but instead merge their *Incoming* fields, and throw away one of the names. So the calculation of the number of nodes that are already in *Nodes_Set* can be simplified to the calculation of the number of elements in the set:

$$\{(\mathit{Old}(n), \mathit{Next}(n)) \mid \forall\, n.\ (n \in \mathit{Nodes\_Set})\}$$

When a node is not ready to be inserted into *Nodes_Set*, it will either be split into two nodes or updated into a new version. When a node $q$ is split into nodes $q_1$ and $q_2$, the following holds:

$$\begin{aligned}\mathsf{SF\_set}(\mathit{New}(q) \cup \mathit{Old}(q) \cup \mathit{Next}(q)) &= \mathsf{SF\_set}(\mathit{New}(q_1) \cup \mathit{Old}(q_1) \cup \mathit{Next}(q_1)) \\ &\cup\ \mathsf{SF\_set}(\mathit{New}(q_2) \cup \mathit{Old}(q_2) \cup \mathit{Next}(q_2))\end{aligned}$$

The function $\mathsf{SF\_set}$ is used to calculate the set of subformulae that a set of formulae can create. When a node $q$ is updated into a new version $q'$, then the following holds:

$$\mathsf{SF\_set}(\mathit{New}(q) \cup \mathit{Old}(q) \cup \mathit{Next}(q)) = \mathsf{SF\_set}(\mathit{New}(q') \cup \mathit{Old}(q') \cup \mathit{Next}(q'))$$

These two equations can be proven directly from the algorithm and the definition of LTL. Thus, when we insert a node into *Nodes_Set*, the subformulae in the *Old* and *Next* are all from the original formula. The translation process does not create new formulae.

From the above we see that there is an upper bound for the number of nodes left to be created by the current node that can be inserted into *Nodes_Set*. Also, the number of nodes we are creating is increasing, although not strictly at each step. Moreover, the number of possible nodes to create is bounded, so every time we insert a node into *Nodes_Set*, the distance to the upper bound is decreasing. The following relation $\mathsf{remain\_nodes}$ can be used to calculate the distance:

$$\begin{aligned}
&\mathsf{remain_nodes}\ \mathit{Nodes_Set}\ q \ = \\
&\mathsf{card}(\{(x, y) \mid (x \in \mathsf{SF_set}(\mathit{New}(q) \cup \mathit{Old}(q) \cup \mathit{Next}(q))) \\
&\qquad \&(y \in \mathsf{SF_set}(\mathit{New}(q) \cup \mathit{Old}(q) \cup \mathit{Next}(q)))\} - \\
&\qquad \{(\mathit{Old}(n), \mathit{Next}(n)) \ \mid \ \forall\, n.\ (n \in \mathit{Nodes_Set})\})
\end{aligned}$$

The function $\mathsf{card}$ is the cardinality function for finite sets.

However, the $\mathsf{remain\_nodes}$ function does not decrease strictly during every recusive call. This makes it difficult to use it as a well-founded relation to prove termination. Now we consider what is happening when the distance to the bound stays constant. The second kind of recursive call will not insert a node into *Nodes_Set*. It will repeatedly break up *New* by selecting a formula in *New* using the function $\mathsf{max\_SF}$ until it is empty. In this case, the complexity of *New* is decreasing. For defining the complexity of *New*, we use the function $\mathsf{max\_SF}$ to select a maximal subformula, one that is not a subformula of any other. Then when we remove it from *New* and put it into *Old*. The total number of subformulae in *New* will go down by at least 1. The relation $\mathsf{complex\_new}$ is defined as the following:

$$\textsf{complex\_new}\ node\ =\ \textsf{card}(\textsf{SF\_set}\ (New\ node))$$

To combine these two relations to work together, we use the *lexicographic product* (<*lex*>) of two well-founded relations. Given relations $ra$ and $rb$, the lexicographic product is formally defined as follows:

$$ra < *lex* > rb\ =\ \{((a,b),(a',b')).\ (a,a') \in ra\ \vee\ a = a'\ \wedge\ (b,b') \in rb\}$$

The lexicographic product decreases if either its first component decreases or its first component stays the same and the second component decreases. It is also proved that if two given relations are well-founded, their lexicographic product is also well-founded.

The following relation is built to serve as the total well-founded relation to prove the termination of the translation algorithm:

$$\begin{aligned}&\textsf{inv\_image}\ (\textsf{less\_than}\ < * \ lex * > \ \textsf{less\_than})\\ &(\lambda(node, Nodes\_Set).\ (\textsf{remain\_nodes}\ (Nodes\_Set, node), \textsf{complex\_new}\ node))\end{aligned}$$

Isabelle/HOL *less_than* is defined as a relation, which is a set of pairs of natural numbers, i.e., $((x,y) \in less\_than) = (x < y)$. The *inv_image* is used to generalized the *inverse image* of a relation, where $inv\_image\ r\ f\ =\ \{(x,y).\ (fx, fy) \in r\}$.

This relation is defined in Isabelle and its well-foundedness is proved by Isabelle's classical reasoner. The termination of the algorithm is also proved in Isabelle by case analysis. We do the case analysis on whether the *New* field of the input node is empty first. If it is empty, we prove that after the algorithm inserts a node into *Nodes_Set*, the remaining nodes it can create that can be inserted into *Nodes_Set* has decreased. In the case that the input *New* field is not empty, we do a case analysis on what the maximal formula in the *New* field is. The complexity of the *New* field is proved to be decreasing. This termination proof is carried out in an *interactive-fashion* in Isabelle. In our case, the termination of the algorithm can not be proved by an automated theorem prover. This is because the measure function we derived here is not syntactically suggested from the function definition. There can not exist an algorithm that can prove or disprove the termination of a function by only given the recursive definition of a general recursive function. It is generally impossible to always compute the necessary induction principle to prove a theorem. Some attempts to do so are generally driven by syntax. This falls in the scope of what *inductive* theorem provers [87, 88]. In particular, it is beyond what rippling can do.

### 3.3.3 Correctness Proof of the Algorithm

In this section, we present a formal correctness proof of the LTL to Büchi Automata translation algorithm. An informal proof has essentially already been given by Gerth et al. in [35], but we feel that the level of proof discourse in that work is at a high enough level with enough details omitted that the proof would benefit from the intense scrutiny afforded by a formal proof within a theorem prover such as Isabelle. Formalizing their proof poses new challenges. Several lemmas about the expand function are claimed to be true "by construction". Particularly given that expand is a rather complex general recursive function, it is not clear what it means for a fact about it to be true "by construction". In our proof, these results follow as corollaries of results proved by induction, and that the inductive results have a general assume-guarantee nature to them: if a fact holds of the state (or that portion of the input representing the state) before the function is executed, then it will hold of the resulting state after the execution. Also, since we extended their algorithm for accepting finite words, we need to provide the proof for finite case.

In the algorithm for translating LTL formulae into Büchi Automata, we first use the function create_graph to get a set of nodes *Nodes_Set* from an LTL formula $\eta$. Then we translate the *Nodes_Set* into a Büchi Automata. The algorithm in Figure 3.8 works recursively. The function expand is actually doing two things. If the *New* field is not empty, expand splits the node into two or refines it into a new version. If the *New* field is empty, expand insert the node into *Nodes_Set* and start with a new node. Thus, to prove the correctness of the algorithm, we can break the function expand into two functions: splits and grow . The modified algorithm is shown in Figure 3.9. The recursive function splits takes a node $s$ as input and repeatly splits or refines node. The output of splits is a list of nodes that are split or refined (line 6-10) from the original node $s$. If the *New* field of the $s$ is empty, then splits returns an empty list(line 4). The input of the function grow is a list of nodes, which usually are the result of splits , and a set of nodes *Nodes_Set* . The function grow checks the node at the head of the node list against *Nodes_Set* (line 46-48). If there exists a node in *Nodes_Set* that has the same *Old* and *Next* fields, then grow only updates the *Incoming* field of that node and goes on process the tail of the node list(line 49). If there is no such node in *Nodes_Set*, then grow inserts the node into *Nodes_Set* and starts with a new node, where the new *New* field is set to be the old *Next* field(line 50-52). The function get_graph is simply the start of the algorithm with the LTL formula $\mu$. The function get_graph starts with a LTL formula $\mu$ and calls the splits and grow . The arguments of splits are initially set to be an node which contains $\mu$ in its *New* field and *init* in its *Incoming* field. The arguments of grow are set to be the result of grow and an empty *Nodes_Set* .

Theorem 3.1 guarantees the modification of the algorithm will not change the result. Thus, the correctness proof of create_graph can be reduced to the conrrectness proof of get_graph.

**Theorem 3.1** *Given the same LTL formula $\eta$, the two functions create_graph and get_graph will produce the same result.*

*Proof:* To prove two functions are the same, we prove that if they are given the same LTL formula $\eta$ as input, then they produce the same result. To show this, we show that if a node $s$ is in the output node set of create_graph, it is also in the output node set of get_graph, and vice versa. This can be proved by induction on both functions. ■

```
record graph_node = [Name : string, Incoming : set of string,
  New : set of formula, Old : set of formula, Next : set of formula, Strong : bool];
function splits s =
  if New(s) = ∅ then return[];
  else
    let η = max_SF(New(s)); New(s) := New(s) \ {η}; Old(s) := Old(s) ∪ {η};
    case η of
    η = A, or ¬A, where A is a proposition, or η = true, or η = false ⇒
        if η = false or ¬η ∈ Old(s) then return [];
        else return (splits([Name ⇐ Name(s), Incoming ⇐ Incoming(s),
        New ⇐ New(s), Old ⇐ Old(s), Next ⇐ Next(s), Strong(s)]));
    η = φ ⋃ ψ
        s1 := [Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s) ∪ ({New1(η)} \ Old(s))
            Old ⇐ Old(s), Next = Next(s) ∪ {Next1(η)}, true];
        s2 := [Name ⇐ Name(s),
            Incoming ⇐ Incoming(s),
            New ⇐ New(s) ∪ ({New2(η)} \ Old(s)),
            Old ⇐ Old(s), Next ⇐ Next(s), Strong(s)]);
        return(splits (s1) @ splits (s2));
    η = φ ⋁ ψ, or φ ∨ ψ ⇒
        s1 := [Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s) ∪ ({New1(η)} \ Old(s))
            Old ⇐ Old(s), Next = Next(s) ∪ {Next1(η)}, Strong(s)];
        s2 := [Name ⇐ Name(s), Incoming ⇐ Incoming(s)
            New ⇐ New(s) ∪ ({New2(η)} \ Old(s)),
            Old ⇐ Old(s), Next ⇐ Next(s), Strong(s)]);
        return(splits (s1) @ splits (s2));
    η = φ ∧ ψ ⇒
        return(splits([Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s) ∪ ({φ, ψ} \ Old(s)),
            Old ⇐ Old(s), Next ⇐ Next(s), Strong(s)]))
    η = ◯φ ⇒
        return(splits([Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s), Old ⇐ Old(s),
            Next ⇐ Next(s) ∪ {φ}, true]))
    η = ◯̄φ ⇒
        return(splits([Name ⇐ Name(s), Incoming ⇐ Incoming(s),
            New ⇐ New(s), Old ⇐ Old(s),
            Next ⇐ Next(s) ∪ {φ}, Strong(s)]))
end splits;
function grow(nl, Nodes_Set)
  if nl = [] then return(Nodes_Set);
  else
    let s = (hd nl);
    if exists node r in Nodes_Set with
        Old(r) = Old(s) and Next(r) = Next(s)
        Incoming(r); = Incoming(r) ∪ Incoming(s);
        return(grow((tl nl), Nodes_Set));
    else return(grow((tl nl), grow(splits([Name ⇐ new_name(),
        Incoming ⇐ {Name(s)}, New ⇐ Next(s),
        Old ⇐ ∅, Next ⇐ ∅, Strong(s)], Nodes_Set ∪ {s}))))
end grow;
function get_graph(μ)
  return(grow(splits([Name ⇐ new_name(), Incoming ⇐ {init},
    New ⇐ {μ}, Old ⇐ ∅, Next ⇐ ∅, false]), ∅))
get_graph;
```

**Figure 3.9** The Modified LTL Translation Algorithm.

Now, we can prove the correctness of the algorithm by proving the modified algorithm using get_graph. The theorem 3.2 is the main goal we want to prove:

**Theorem 3.2** *The automaton A constructed by the algorithm from an LTL formula $\mu$ accepts exactly the behaviors that satisfy $\mu$.*

*Proof:* Lemma 3.6 and Lemma 3.12 prove this theorem in two directions. ■

In what follows, let $\xi$ be a *behavior* consisting of propositions, and let $\sigma$ be a *behavior* consisting states of $A$, the automaton that is constructed from an LTL formula $\mu$. If the node $n$ is a member of node list returned by *splits*($s$), then we say $n$ is a terminal descendant of $s$ and $s$ is an ancestor of $n$. Also, let $\bigwedge\Xi$ denote the conjunction of a set of formulae $\Xi$, the conjunction of the empty set is set to be *True*. A function max_SF selects a maximal subformula from a set of formulae, which is not a subformula of any other formulae.

**Lemma 3.1** *If New (s) is not empty, then for all nodes n, where n is a terminal descendant of s, we have max_SF(New (s))* $\in$ *Old (n).*

*Proof:* In the definition of the splits, every recursive call on a node $s$ will remove the max_SF*(New(s))* from *New* field and add it to *Old* field(line 6). When a node $s$ is split into $s_1$ and $s_1$(line 12-28), max_SF*(New(s))* is inserted into both *Old(*$s_1$*)* and *Old(*$s_2$*)*. When a node $s$ is refined to $s'$(line 8-11 and 29-40), max_SF*(New(s))* is also inserted into *Old(*$s'$*)*. Also, we notice that the *Old* field only grows. Once some formula is added into *Old* field, it will never get out. There is no operation to remove formula from the *Old* field in the function splits. If we start with splits(s), max_SF*(New(s))* goes to the *Old* field and stays there till the end of function splits. Thus, by induction we have for all nodes n in splits(s), max_SF*(New(s))* $\in$ *Old* (n). ■

**Lemma 3.2** *For all n, where n is a terminal descendant of s, if there are no two nodes with both the same Old and Next fields in Nodes_Set, then there exists one and only one node r in grow(splits(s), Nodes_Set)) such that Incoming(n)⊆Incoming(r), Old(n)=Old(r) and*

*Next(n)=Next(r). Moreover, for all* $r \in$ grow(splits(s), *Nodes_Set)) such that Incoming(s)* $\subseteq$ *Incoming(r), either r is in Nodes_Set, or there exists a terminal descendant n of s such that Incoming(n)*$\subseteq$*Incoming(r), Old(n)=Old(r) and Next(n)=Next(r).*

*Proof:* By induction on the function grow. If the input node list is not empty, the function grow checks the node $n$ at the head of the node list against the nodes in the *Nodes_Set*. If there is a node $r$ in *Nodes_Set* with the same *Old* and *Next* fields, then $r$ satisfies the conclusion. This is because once a node is added into *Node_Set*, we can only update its *Incoming* field. We can not change its *Old* and *Next* fields or remove it from *Nodes_Set*. If there is no such node in *Nodes_Set* with the same *Old* and *Next* fields, then $n$ is inserted into *Nodes_Set* and $n$ will be the node to satisfy the conclusion. The reason is same as the first case. Every node $n$ in the input node list will eventually be checked against the *Node_Set*. Thus, there exists one and only one node r in grow(splits(s), Nodes_Set)) such that *Incoming*(n)$\subseteq$*Incoming*(r), *Old*(n)=*Old*(r) and *Next*(n)=*Next*(r) for each $n$.

Also, there are two ways that a node $r$ can be in grow(splits(s), Nodes_Set)). First, it is already in *Node_Set*, then there will be a node with the same *Old* and *Next* fields in the final *Node_Set*. We already proved this above. Second, it is added into *Node_Set* in line 52. This is the only point where a node can be added into *Node_Set*. And once added into *Node_Set*, there will be a node in the final *Node_Set* with same *Old* and *Next* fields for the reason above. ■

Lemma 3.2 describes the relationship between the result of the function Splits and the node set *Node_Set*, which is very useful for connecting the original input LTL formulae and the final output *Node_Set*. Also, a very important property about *Node_Set* is given. If the input *Node_Set* does not have two nodes with the same *New* field and *Old* field, then there are no such pair of nodes in the output *Nodes_Set*. This is the basis of several lemmas we proved below.

**Lemma 3.3** *For every initial state* $q \in I$ *of an automaton constructed from the LTL formula* $\mu$*, we have* $\mu \in Old(q)$.

*Proof:* A node of an automaton can be an initial state if and only if it has *init* in its *Incoming* field. One fact about the splits function is that splits does not change the *Incoming* field when splitting and refining nodes, i.e., all terminal descendants have the same *Incoming* field with their ancestor. This is because from the definition of the function splits, we know that splits never add or remove to *Incoming* field of any nodes. From Lemma 3.1, if we start from the initial node $s = [Name \Leftarrow \text{new\_name}(), Incoming \Leftarrow \{\text{init}\}, New \Leftarrow \{\mu\}, Old \Leftarrow \emptyset, Next \Leftarrow \emptyset, \text{false}]$, then every nodes that is in the result node list of splits(s) has *init* in its *Incoming* field and $\mu$ in its *Old* field. Also, the result of splits(s) contains all the nodes that can have *init* in it since every call to grow will change the *Incoming* fields. From Lemma 3.2, for all terminal descendants $n$ of $s$, there is one and only one corresponding node in the grow(splits(s), $\{\}$) with the same *Old* and *Next* fields and *init* in its *Incoming* field. For all nodes in grow(splits(s), $\{\}$) with *init* in its *Incoming* field, there exists a terminal descendant $n$ of $s$ that has *init* in its *Incoming* field and has the same *Old* and *Next* fields since we started with an empty *Node_Set*. There is a one-to-one relationship between all terminal descendants $n$ of $s$ and all nodes $r$ in grow(splits(s), $\{\}$) for *Incoming(n)*$\subseteq$*Incoming(r)*, *Old(n)=Old(r)* and *Next(n)=Next(r)*. Thus, for every initial state $q \in I$ of an automaton constructed from the LTL formula $\mu$, we have $\mu \in Old(q)$. ■

**Lemma 3.4** *Let $\sigma$ be an execution that is accepted by A, which is constructed from an LTL formula $\mu$. Let $\sigma_i$ denote behd(benthsuffix i $\sigma$), the $i^{th}$ state in the execution, and let $\sigma_0$ be the initial state and $\eta$ be an LTL formula in Old($\sigma_0$). Then, by case analysis on $\eta$, one of the following holds:*

1. *Base* $\rho$*, where* $\rho$ *is a proposition: Base* $\rho \in Old(\sigma_0)$

2. $\varphi \bigcup \psi$: $\exists\, j \geq 0.\ \forall 0 \leq i < j.\ \varphi, \varphi \bigcup \psi \in Old(\sigma_i)$ *and* $\psi \in Old(\sigma_j)$

3. $\varphi \bigvee \psi$: $\forall i.\ \psi \in Old(\sigma_i)$ *or* $\exists\, j \geq 0.\ \forall 0 \leq i \leq j.\ \psi, \varphi \bigvee \psi \in Old(\sigma_i)$ *and* $\varphi \in Old(\sigma_j)$

4. $\varphi \wedge \psi$: $\varphi \in Old(\sigma_0)$ *and* $\psi \in Old(\sigma_0)$

5. $\varphi \vee \psi$: $\varphi \in Old(\sigma_0)$ *or* $\psi \in Old(\sigma_0)$

6. $\bigcirc\varphi$: $\neg$*BEis_singleton*$(\sigma)$ *and* $\varphi \in Old(\sigma_1)$

7. $\bar{\bigcirc}\varphi$: BEis_singleton$(\sigma)$ *or* $\varphi \in Old(\sigma_1)$

*Proof:* Note that cases *ltl_True* and *Neg* $\varphi$ are similar with the *Base* case. LTL formula *ltl_False* can not be in *Old* field since the algorithm will terminate once a formula *ltl_False* is met. This lemma can be proved by the induction. We only provide the proof for *Until* case. Other cases can be proved similarly and were done in the Isabelle proof.

When the algorithm is processing a formula $\eta$=$\varphi \bigcup \psi$(line 6), the node is split into two nodes. For the first copy, $\varphi$ is inserted into the *New* field and $\varphi \bigcup \psi$ is inserted into the *Next* field. For the second copy, $\psi$ is inserted into the *New* field. The formula $\varphi \bigcup \psi$ is inserted to the *Old* field of both copies. We also know that $\varphi$ and $\psi$ will eventually go to their own *Old* field. This can be proved by induction on the function *splits*. If our path goes to the second copy, then the conclusion $\exists\, j \geq 0.\ \forall 0 \leq i < j.\ \varphi, \varphi \bigcup \psi \in Old(\sigma_i)$ *and* $\psi \in Old(\sigma_j)$ will be satisfied by choosing j=0. If our path goes to the first copy, we have $\varphi$ and $\varphi \bigcup \psi$ in the *Old* field. When this node is fully processed, the algorithm starts with a new node. The *New* field of the new node is set to be the old *Next* field, which contains $\varphi \bigcup \psi$ in it. The algorithm repeats this procedure if we always choose the path to the *first copy* until the *second copy* is chosen sometime later. Since the execution $\sigma$ satisfies the acceptance conditions of the automaton $A$, there must exists some state that has $\psi$ in its *Old* field. Thus, we have the conclusion $\exists\, j \geq 0.\ \forall 0 \leq i < j.\ \varphi, \varphi \bigcup \psi \in Old(\sigma_i)$ *and* $\psi \in Old(\sigma_j)$.

■

The function BEis_singleton is used to test if a sequence is a singleton. If a sequence $\sigma$ is infinite, then BEis_singleton($\sigma$) is set to be false.

**Lemma 3.5** *Let $\sigma$ be an execution of A constructed from an LTL formula $\mu$ and $\sigma_i$ denote behd(benthsuffix $i$ $\sigma$), the $i^{th}$ state in the execution. Let $\eta$ be an LTL formula in Old($\sigma_i$). Then, for all $i$, (benthsuffix $i$ $\sigma$) $\models \eta$.*

*Proof:* By induction on $\psi$. The base case is for formulae of the form $\rho$, where $\rho$ is a proposition. The base case can be proved directly from the construction. We will only show the *Until* induction case. According to Lemma 3.4, we have $\exists\ j \geq 0.\ \forall 0 \leq i < j.\ \varphi, \varphi \bigcup \psi \in Old(\sigma_i)\ and\ \psi \in Old(\sigma_j)$. Also we have, (benthsuffix j $\xi$) $\models \psi$ and for each $0 \leq$ i < j, (benthsuffix i $\xi$) $\models \varphi$, along with the semantic definition of LTL, we have $\xi \models \psi$. Other cases are treated similarly. ■

**Corollary 3.1** *Let $\sigma$ be an execution of A constructed from an LTL formula $\mu$ and $\sigma_0$ be the initial state. Let $\xi$ be a word that is accepted by $\sigma$. Then for all LTL formula $\psi$ in Old($\sigma_0$), $\xi \models \psi$.*

*Proof:* By Lemma 3.5 and set $i$ to be 0. ■

**Lemma 3.6** *Let $\sigma$ be an execution of the automaton A constructed from the LTL formula $\mu$. Let $\xi$ be a word that is accepted by $\sigma$. Then we have $\xi \models \mu$.*

*Proof:* Let $q_0$ be an initial state of $\sigma$. From Lemma 3.3, we have $\mu \in Old(q)$. From Lemma Corollary 3.1 we have for all LTL formula $\psi$ in Old($q_0$), $\xi \models \psi$. Thus, we have $\xi \models \mu$. ■

**Lemma 3.7** *For all nodes $n$ in an automaton A constructed from an LTL formula $\mu$, Strong(n) is uniquely determined the its Old field.*

*Proof:* Initially, the construction starts with a node $s$ containing $\mu$ in its *New* field. The *Strong* field indicates the current node must not be the last state in a sequence, i.e., there is something that needs to happen and has not yet happened so far. Only two forms of LTL formulae force something to happen in the future states. One is *Until*, the other is *Next*.

The *Strong(s)* field is set to be *true* only for these two cases and the Strong(s) field is set to be false if and only if either there exists $\bigcirc\varphi \in Old(s)$ or there exists $\varphi \bigcup \psi \in Old(s)$, but $\psi \notin Old(s)$. If an LTL formula $\varphi \bigcup \psi$ is met during the construction(line 12), $\varphi \bigcup \psi$ is inserted into both *Old* fields and $\varphi$ is inserted into the *New* field. We already proved that $\varphi$ will eventually go to the *Old* fields. Note that there is no operation to change the *Strong* field back to *false*. Once the *Strong* field of a node is set to be *true*, it will always be *true*. If an LTL formula $\bigcirc\varphi$ is met during the construction(line 33), $\bigcirc\varphi$ is inserted into the *Old* field. Thus, there are only two cases for the *Strong* field of a node $n$ in an automaton A to be *true*. The first is that $n$ contains an LTL formula of form $\varphi \bigcup \psi$ in its *Old* fields. The second is that $n$ contains an LTL formula of form $\bigcirc\varphi$ in its *Old* field. ■

Lemma 3.7 guarantees there is no two nodes in an automaton A constructed from an LTL formula $\mu$ that have some *Old* and *Next* field but have different *Strong* field. Thus, we only need to check the *Old* and *Next* fields in line 47.

A function *VarNext* is defined to choose *Next* or *Weak_Next* according to the *Strong* field of a node. If the *Strong* field is true, we choose *Next*, otherwise we choose *Weak_Next*. The function *VarNext* is defined as follow:

VarNext :: "bool ⇒ 'a LTL ⇒ 'a LTL"

"VarNext s p = (if s then Next p else Weak_Next p)"

**Lemma 3.8** *When a node $q$ is split during the construction in line 12-28 into two nodes $q_1$ and $q_2$, the following holds:*

$$(\bigwedge Old(q) \wedge \bigwedge New(q) \wedge (VarNext\ (Strong(q))\ \bigwedge Next(q))) \leftrightarrow$$

$$((\bigwedge Old(q_1) \wedge \bigwedge New(q_1) \wedge (VarNext\ (Strong(q_1))\ \bigwedge Next(q_1)))\vee$$

$$(\bigwedge Old(q_2) \wedge \bigwedge New(q_2) \wedge (VarNext\ (Strong(q_2))\ \bigwedge Next(q_2))))$$

*Similarly, when a node $q$ is refined to a new version $q'$ in line 8-11 and 29-40, the following holds:*

$$(\bigwedge Old(q) \wedge \bigwedge New(q) \wedge (VarNext\ Strong(q)\ \bigwedge Next(q))) \leftrightarrow$$

$$(\bigwedge Old(q') \wedge \bigwedge New(q') \wedge (VarNext\ Strong(q')\ \bigwedge Next(q')))$$

*Proof:* Directly from the definition of function *splits* and LTL semantics. ■

Lemma 3.8 guarantees every recursive call preserve conjunction of the subformulae among *New*, *Old* and *Next* fields. No new formulae will be added in or removed from *New*, *Old* and *Next* fields.

**Lemma 3.9** *Let q be a node and* $q_1$, $q_2$, $q_3$,...,$q_n$ *be all its terminal descendants. So, at the end of the construction, we have:*

$$\xi \models \bigwedge New(q) \leftrightarrow \xi \models \bigvee_{1 \le i \le n} (\bigwedge Old(q_i) \wedge (VarNext\ Strong(q_i)\ \bigwedge Next(q_i)))$$

*Also, if* $\xi \models \bigvee_{1 \le i \le n} (\bigwedge Old(q_i) \wedge (VarNext\ Strong(q_i)\ \bigwedge Next(q_i)))$, *then there exists a node* $q_i$ *such that* $\xi \models \bigwedge Old(q_i) \wedge (VarNext\ Strong(q_i)\ \bigwedge Next(q_i))$ *and for each* $\varphi \bigcup \psi \in Old(q_i)$ *with* $\xi \models \psi$, $\psi$ *is also in Old(*$q_i$*).*

*Proof:* Let Nodes_Set = grow(splits(q),{}). From Lemma 3.2, we know that for each $q_i$, there will be a corresponding node in *Nodes_Set* with the same *Old* and *Next* field and all nodes in *Nodes_Set* are coming from the result of *splits*. The result of splits can be used as nodes in *Nodes_Set* if only *Old* and *Next* field are concerned. Thus, using Lemma 3.8, this lemma can be proved by induction on the construction. If a node $q_i$ contains $\varphi \bigcup \psi$ in its *Old* field, then there are two cases, either $q_i$ has $\varphi$ in its *Old* field, or $q_i$ has $\psi$ in its *Old* field. If $\varphi \bigcup \psi$ in the *Old* field, it is chosen as the maximal formula sometime before(line 6). When $\varphi \bigcup \psi$ was inserted into *Old* field, the node is split into two(line 12-20), one has $\varphi$ in its *New* field and the other has $\psi$ in its *New* field. We also know that formulae in *New* field will eventually go to the *Old* field. Thus, if a node $q_i$ contains $\varphi \bigcup \psi$ in its *Old* field, then there are two cases, either $q_i$ has $\varphi$ in its *Old* field, or $q_i$ has $\psi$ in its *Old* field. And if $\xi \models \psi$, we only have the second case, where $\psi$ in the *Old* field. ■

**Lemma 3.10** *Let A be an automaton constructed from the LTL formula* $\mu$. *Then*

$$\xi \models \mu \leftrightarrow \xi \models \bigvee_{q \in I} (\bigwedge Old(q) \wedge (VarNext\ Strong(q_i)\ \bigwedge Next(q_i))).$$

*Proof:* Using Lemma 3.9, where *New*(q) is initially set to $\{\mu\}$. ■

**Lemma 3.11** *Let A be an automaton constructed from an LTL formula* $\mu$ *and let* $\xi$ *be a word such that* $\xi \models \bigwedge Old(q) \wedge (VarNext\ Strong(q)\ \bigwedge Next(q))$. *If* $\xi$ *is the infinite, then there exists a node* $q'$ *in A such that (q,q') is a transtion of A and* $(benthsuffix\ 1\ \xi) \models \bigwedge Old(q') \wedge (VarNext\ Strong(q')\ \bigwedge Next(q'))$. *Moreover, let* $\Gamma = \{\psi \mid \varphi \bigcup \psi \in Old(q)$ *and* $\psi \notin Old(q)$ *and* $(benthsuffix\ 1\ \xi) \models \psi\}$. *Then there exits a transition (q,q') such that* $\Gamma \subseteq Old(q')$. *Similarly, if* $\xi$ *is the finite, then either* $\xi$ *is a singleton, or there exists a node* $q'$ *in A such that (q,q') is a transtion of A and* $(benthsuffix\ 1\ \xi) \models \bigwedge Old(q') \wedge (VarNext\ Strong(q')\ \bigwedge Next(q'))$. *Also, let* $\Gamma = \{\psi \mid \varphi \bigcup \psi \in Old(q)$ *and* $\psi \notin Old(q)$ *and* $(benthsuffix\ 1\ \xi) \models \psi\}$. *Then there exits a transition (q,q') such that* $\Gamma \subseteq Old(q')$.

*Proof:* In the construction, when a node $q$ is finished and inserted into *Node_Set*(line 50-52), a new node $q'$ is created with *New*($q'$) = *Next*($q$). We know that $(benthsuffix\ 1\ \xi) \models New(q')$. Once $q'$ is fully processed, *New*($q'$) goes to *Old*($q'$). This can be proved by induction. Using Lemma 3.9, we can get $(benthsuffix\ 1\ \xi) \models \bigwedge Old(q) \wedge (VarNext\ Strong(q)\ \bigwedge Next(q))$. Lemma 3.9 also guarantees there is a $q'$ that will satisfies the acceptance conditions. For the finite case, if $\xi$ is a singleton, then $q$ is the last node in the execution. If $\xi$ is not a singleton, this case can be proved similarly with the infinite case. And eventually, there will be node $q'$ that either there is not formula in its *Old* field has *Until* form or for each formula $\varphi \bigcup \psi \in Old(q')$ with $\xi \models \psi$, $\psi$ is also in Old($q'$).

■

Lemma 3.11 can be used to find the successor during the construction of an execution in lemma 3.12. In the infinite sequence case, it guarantees the existence of the successor. In finite case, it guarantees the existence of the successor or the completion of the construction.

**Lemma 3.12** *Let* $\xi \models \mu$. *There exists an execution* $\sigma$ *of automaton A constructed from the LTL formula* $\mu$ *such that* $\sigma$ *accepts* $\xi$.

*Proof:* By Lemma 3.10, there exists a node $q_0 \in I$ such that $\xi \models \bigvee_{q \in I}(\bigwedge Old(q) \wedge (VarNext\ Strong(q_i)\ \bigwedge Next(q_i)))$. Using Lemma 3.11, we can construct an execution step by step. If $\xi \models \bigwedge Old(q) \wedge (VarNext\ Strong(q)\ \bigwedge Next(q))$, then we choose $q'$, which is the successor of $q$ that satisfies $(benthsuffix\ 1\ \xi) \models \bigwedge Old(q) \wedge (VarNext\ Strong(q)\ \bigwedge Next(q))$. Also, Lemma 3.11 guarantees the execution $\sigma$ we constructed satisfies the acceptance conditions for both finite and infinite words. ■

Now we will give an example to illustrate how proof is formally done in Isabelle. We will only give one proof because there is no enough space to present all proofs for these lemmas. Lemma 3.5 indicate that if $\sigma$ is an execution of A constructed from an LTL formula $\mu$ and $\sigma_i$ denote behd(benthsuffix $i$ $\sigma$), the $i^{th}$ state in the execution. And let $\sigma_i$ be the $i^{th}$ state in $\sigma$ and $\eta$ be an LTL formula in Old($\sigma_i$). Then, for all $i$, (benthsuffix $i$ $\sigma$) $\models \eta$. This lemma is formally stated in Isabelle as follow:

```
lemma run_word_compatible:
"(accept_exec_beh (ns2ba(get_graph(l))) run word) ⟶
   (∀ m.  x∈(old_of (behd (benthsuffix m run))) ⟶
     ((benthsuffix m word) ⊨ x))"
```

The function *ns2ba* is defined to translate the result of *get_graph(l)* into a Büchi automaton. The function *accept_exec_beh* takes a Büchi automaton, an execution, and a word as arguments. If the execution is accepted by the Büchi automaton and accepts the word, then *accept_exec_beh* returns *true*. Otherwise it will return *false*. To prove lemma `run_word_compatible`, we do an induction on $\eta$. The base case is for formulae of form *ltl_True*, *ltl_False*, *Base* $\rho$, and *Neg* $\rho$. We give an example proof for case *Base* $\rho$ here. The proof script is shown in Table 3.2. The case of form $\varphi \bigcup \psi$ can be defined as:

```
lemma until_compatible:
(ALL m.  (Until p q)∈(old_of (behd (benthsuffix m r))) ⟶
   (∀ m.  x∈(old_of (behd (benthsuffix m run))) ⟶
     (EX j.  (q∈(old_of (behd(benthsuffix (m+j) r)))) ∧
       (ALL i<j.  p∈(old_of (behd(benthsuffix (m+i) r))) ∧
       (Until p q)∈(old_of (behd(benthsuffix (m+i) r))))))"
```

Lemma `until_compatible` is proved using the fact stated in Lemma 3.4. We are not providing the proof script because of the space constraint. All lemmas for the correctness proof of the algorithm have been defined and formally proved in Isabelle. Again, some intermediate lemmas are omitted because we don't have enough space here.

**Table 3.2** Proof Script for Base Case for Lemma 3.5.

```
lemma base_compatible:
"(accept_exec_beh (ns2ba(get_graph(l))) run word) ⟶
    (Base A)∈(old_of (behd (benthsuffix m run))) ⟶
       ((benthsuffix m word) ⊨ (Base A))"
apply (rule impI)+
apply (cases word)
apply (cases run)
apply (simp del: ns2ba_def)
apply (erule conjE)+
apply (case_tac nelista)
apply (simp del: ns2ba_def)
apply (case_tac nelist)
apply (simp del: ns2ba_def)
apply (drule_tac x="Base A" in bspec)
apply assumption
apply (simp del: ns2ba_def)
apply (simp del: ns2ba_def)
apply (simp del: ns2ba_def)
apply (rotate_tac 7)
apply (drule_tac x="m" in spec)
apply (rotate_tac 7)
apply (drule_tac x="Base A" in bspec)
apply assumption
apply simp
apply simp
apply (case_tac run)
apply simp
apply (simp del: ns2ba_def)
apply (erule conjE)+
apply (case_tac m)
apply (simp del: ns2ba_def)
apply (rotate_tac 6)
apply (drule_tac x="0" in spec)
apply (rotate_tac 7)
apply (drule_tac x="Base A" in bspec)
apply (simp del: ns2ba_def)
apply (simp del: ns2ba_def)
apply (simp del: ns2ba_def)
apply (rotate_tac 6)
apply (drule_tac x="Suc nat" in spec)
apply (rotate_tac 7)
apply (drule_tac x="Base A" in bspec)
apply (simp del: ns2ba_def)
apply (simp del: ns2ba_def)
done
```

# CHAPTER 4

# CONCLUSION AND OUTLOOK

## 4.1 Summary

In this work, we have enhanced verification techniques based on novel combinations of theorem proving and model checking. Our contribution includes an extension of DOVE with product automata and their application, formulation of LTL in Isabelle, and formal proof of correctness of the algorithm for translating LTL formulae into Büchi Automata.

The work extending of DOVE gained us a lot of experiences for doing the verification on real-life problems. We studied the formal verification tool DOVE, learned its strengths and weaknesses, and extended it with product automata to reduce the burden of the state explosion problem for the designer.

The formulation of the algorithm for translating LTL formulae into Büchi Automata in a formal logic earned us a chance to experience the formal proof of a non-trivial algorithm. During the proof, we learned the length of the proof, the mathematical theories needed, the level of expertise in the theorem prover required, and the time required to carry out such a proof. Our formulation of the algorithm and its correctness proof result more than 9,500 lines of Isabelle code. One lesson is that formal algorithm proof requires non-trivial human expertise and time. We also noticed the difference between the formal and informal proof, doing proof with the theorem prover Isabelle forces us to be honest in our arguments. Informal proof such as, "obvious", "directly from", and "immediately from" do not work. We learned that some trivial claims in the informal proof are actually non-trivial.

The main contribution of this dissertation is the improvement of the easy of use and reliability of tools for formal verification. We have increased the automation of an interactive tool while giving mathematical justification for it. We have increased the confidence level in a class of model checkers by formally verifying one of the core algorithms use by them. And we have increased automation in the domain of fully expansive interactive proof and we have increased the confidence level of fully automated tools by subjecting one of their central algorithms to the rigor of fully expansive proof.

## 4.2 Related Work

As is well known, verification techniques based on automata theory and temporal logic always draw a lot of attentions.

DOVE [34] is tool to provide support for the formal analysis of state machine designs. In DOVE, the modelling and reasoning activities can be driven directly from the state machine in a graphical framework. Verification in DOVE is carried out by doing inductive proofs over automata instead of model checking. DOVE can only deal with finite sequences and can only handle safety properties. The algorithm is embedded in the theorem prover as a family of tactics. In DOVE, the correctness is guaranteed one example at a time, by its embedding in Isabelle.

The translation algorithm we modified was presented in the Gerth *et al.* [35]. They described a tableau-based algorithm for obtaining an automaton from a linear temporal logic formula. The algorithm is to be used in model checking in an "on-the-fly" fashion. That means the automaton can be constructed simultaneously along with the generation of the model. The algorithm can be used to check the validity of the linear temporal logic properties by only constructing part of the model and part of the automaton. However, the algorithm can only be used to translate temporal logic interpreted on infinite sequences. In our work, the algorithm is enriched with the ability to work on both finite and infinite sequences by defining linear temporal logic on a special sequence *behavior*. Also, we provided a formal proof of the termination of the algorithm, which is a crucial part of the correctness of a recursive algorithm.

Combining mechanical theorem proving and model checking has been a hot topic in recent years. Several other related works draw attention. In the Chou [14], they formally verified a meta-theory of model checking using mechanical theorem proving. A case study is carried out using the mechanical theorem prover HOL to verify the correctness of a partial-order reduction technique for reducing the state search performed by model checkers. There is a lot of similar infrastructure in our work and their proof work. Moreover, their experience with verifying nontrivial algorithms in HOL helped us to employ our proof in Isabelle.

Model checking for temporal logic properties can give counter examples if the properties fail to hold for the checked system. The counter example will be used as a certificate of system failure. On the other hand, if the check succeeds, no such certificate will be given. In the Namjoshi [89], they gave a deductive proof of the reason why the model checking is successful. They created a deductive proof system for verifying branching time properties expressed in the $\mu$-calculus and showed how to generate a proof in the system from a successful model checking run. Basically, we are all aiming to prove that the algorithm is correct. While their work is side-stepping whether the algorithm is always correct by having it generate a proof that it is correct in each specific example.

In [82], Gunter and Peled suggested a new application for temporal logic, as a way of assisting the debugging of a concurrent or sequential program. They defined temporal logic over finite sequences as the specification formalism for the automatic verification of extended state systems. Also, they described a debugging tool based on the idea which can be used for finding paths to assisting in building test suites and hence be more confident about the correctness of the system. In that paper, they describe a variant of the algorithm in Gerth *et al.* that applies to LTL formulae interpreted over finite sequences. Our work in this paper merges the two algorithms. The algorithm here has ability for handling both infinite and finite sequences of program behaviors and has non-trivial proof about the termination of the algorithm while both of these features are absent in that work.

### 4.3 Future Work

So far, in previous chapters, we presented some techniques for formal specification and verification. Our ultimate goal is to create a tool for automatic verification. In this section, we present some possible future works. We will introduce the automata framework for building an environment in Isabelle for model checking LTL specifications, i.e., checking whether a modeled system presented as Büchi automata satisfies a given LTL specification. The automata theoretic framework was proposed by Kurshan [90], Vardi and Wolper [91], and Alpern and Scheider [92].

As we mentioned in Chapter 2, one of the advantages of using automata is that both a modeled system and its specification can be presented in the same way. We use Büchi automata to represent a system $A = (\Sigma, S, \Delta, I, L, S)$. It contains a set of state $S$. $\Delta \subseteq S \times S$ is the transition relation. $I \subseteq S$ is a set of intial states. The labeling function $L: S \rightarrow \Sigma$ associates each state with a set of propositions which hold in that state.

A specification of a system can be given as an automaton $B$ over the same alphabet as $A$. The system model $A$ satisfies the specification $B$ if there is an inclusion between the language of the system $A$ and the language of the specification $B$, i.e.,

$$L(A) \subseteq L(B)$$

Let $\overline{L(B)}$ be the complement of the language $L(B)$, i.e., the language $\Sigma^\omega \setminus L(B)$ of words not accepted by $B$. Then, the above inclusion can rewritten as

$$L(A) \cap \overline{L(B)} = \emptyset$$

This means all accepted word of $A$ are allowed by $B$. If the intersection is empty, the system model $A$ satisfies the specification $B$. If the intersection is not empty, elements in it are counterexamples [93]. Checking for the emptiness of the language obtained from two automata is simpler than checking for language inclusion.

However, if the specification automaton is translated from an LTL formula $\varphi$, we can translate the negation of the formula $\varphi$ into an automaton $\overline{B}$ directly rather than translate $\varphi$ into an automaton $B$ and then complement it.

An important property of Büchi automata is their closure under intersection, union and complementation [62]. This means that there exists an automaton that accepts exactly the intersection or the union of the language of two given automata, or the complementation language of a given automaton. These properties enable us to do some constructions on automata without lose any information.

We give the following formal description of the automatic verification method. Given the system automaton $A$ and specification expressed using LTL formula $\varphi$: First, we need to normalize the LTL formula $\neg\varphi$. Then we need to translate normal form $\neg\varphi$ into a generalized Büchi automaton $B$, and then convert the generalized Büchi automaton into a simple Büchi automaton $B$. Next, we also need to build the product automaton $A \times B$ and check the emptiness of $A \times B$. If the intersection is empty, the specification holds for $A$. If the intersection is not empty, any elements in it are counterexamples.

# APPENDIX A

# PROGRAMS

This appendix include programs in Isabelle and the SML programming language. We put out programs in: `http://www-faculty.cs.uiuc.edu/~egunter` instead of here because of the space constraint.

**REFERENCES**


[1] H. Saiedian, "An invitation to formal methods," *IEEE Computer*, vol. 29, no. 4, pp. 16–30, Apr. 1996, a "roundtable" of short articles by several authors.

[2] *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion*, NASA Office of Safety and Mission Assurance, Washington, DC, May 1997.

[3] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[4] R. E. Bloomfield, D. Craigen, F. Koob, M. Ullmann, and S. Wittmann, "Formal methods diffusion: Past lessons and future prospects." in *SAFECOMP*, 2000, pp. 211–226.

[5] H. P. Barendregt, "Lambda calculi with types," *Handbook of logic in computer science (vol. 2): background: computational structures*, pp. 117–309, 1992.

[6] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computability*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[8] E. M. Clarke and S. Berezin, "Model checking: Historical perspective and example," in *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX'98*, ser. Lecture Notes in Computer Science, H. de Swart, Ed., vol. 1397. Springer Verlag, May 1998, pp. 18–24.

[9] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA: MIT Press, 1999.

[10] W. Visser, K. Havelund, G. Brat, and S.-J. Park, "Model checking programs," in *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, 2000.

[11] E.M. Clarke, O. Grumberg, and K. Hamaguchi, "Another look at LTL model checking," in *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, David L. Dill, Ed., vol. 818. Standford, California, USA: Springer-Verlag, 1994, pp. 415–427.

[12] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: $10^{20}$ States and Beyond," in *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*. Washington, D.C.: IEEE Computer Society Press, 1990, pp. 1–33.

[13] E. L. Gunter and D. Peled, "Unit checking: Symbolic model checking for a unit of code." in *Verification: Theory and Practice*, 2003, pp. 548–567.

[14] C.-T. Chou and D. Peled, "Formal verification of a partial-order reduction technique for model checking." *Journal of Automated Reasoning*, vol. 23, no. 3-4, pp. 265–298, 1999.

[15] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-order reduction in symbolic state space exploration," in *Computer Aided Verification*, 1997, pp. 340–351.

[16] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," *Formal Methods System Design*, vol. 9, no. 1-2, pp. 105–131, 1996.

[17] C. Daws and S. Tripakis, "Model checking of real-time reachability properties using abstractions," in *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. London, UK: Springer-Verlag, 1998, pp. 313–329.

[18] E. Gunter and Y. Meng, "Extending DOVE with product automata," in *Theorem Proving in Higher Order Logics, 15th International Conference - Supplemental Proceedings, TPHOLs 2002*, V. A. Carreño, C. A. Muñoz, and S. Tahar, Eds., Hampton, VA, August 2002.

[19] K. L. McMillan, "Symbolic model checking: an approach to the state explosion problem," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, 1992.

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

[21] Z. Manna, N. Bjorner, A. Browne, E. Y. Chang, M. Colon, L. de Alfaro, H. Devarajan, A. Kapur, J. Lee, H. Sipma, and T. E. Uribe, "Step: The stanford temporal prover," in *TAPSOFT*, 1995, pp. 793–794.

[22] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker," in *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, ser. Electronic Notes in Theoretical Computer Science, F. Gadducci and U. Montanari, Eds., vol. 71. Elsevier, 2002.

[23] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[24] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[25] M. M. Wenzel, "Isabelle/Isar — a versatile environment for human-readable formal proof documents," Ph.D. dissertation, Institut fur Informatik, TU München, 2002.

[26] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Saratoga, NY: Springer-Verlag, June 1992, pp. 748–752.

[27] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas, "PVS: Combining specification, proof checking, and model checking," in *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, Rajeev Alur and Thomas A. Henzinger, Eds., vol. 1102. New Brunswick, NJ: Springer Verlag, / 1996, pp. 411–414.

[28] D. L. Dill, "The Murphi verification system," in *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification.* London, UK: Springer-Verlag, 1996, pp. 390–393.

[29] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach.* Norwell, MA: Kluwer Academic Publishers, 2000.

[30] S. Rajan, N. Shankar, and M. K. Srivas, "An integration of model checking with automated proof checking," in *Proceedings of the 7th International Conference on Computer Aided Verification.* London, UK: Springer-Verlag, 1995, pp. 84–97.

[31] K. Havelund and N. Shankar, "Experiments in theorem proving and model checking for protocol verification," in *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods.* London, UK: Springer-Verlag, 1996, pp. 662–681.

[32] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems - Specification.* Springer-Verlag, 1992.

[33] J. R. Büchi, "On a decision method in restricted second order arithmetic," in *Proceedings International Congress on Logic, Methodology and Philosophy Science.* Stanford University Press, 1960, pp. 1 – 11.

[34] A. Cant, K. Eastaughffe, C. Liu, B. Mahony, J. McCarthy, and M. Ozols., "State-machine modelling in the DOVE system," Research report, DSTO-RR-0255, 1999.

[35] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Protocol Specification, Testing, and Verification*, P. Dembiski and M. Sredniawa, Eds. Chapman & Hall, Aug. 1995, pp. 3–18.

[36] L. C. Paulson, "The foundation of a generic theorem prover," *Journal of Automated Reasoning*, vol. 5, no. 3, pp. 363–397, 1989.

[37] P. B. Andrews, *An introduction to mathematical logic and type theory: to truth through proof.* San Diego, CA, USA: Academic Press Professional, Inc., 1986.

[38] L. C. Paulson, "Set theory for verification: I. From foundations to functions," *Journal of Automated Reasoning*, vol. 11, no. 3, pp. 353–389, 1993.

[39] W. C. Powell, "A completeness theorem for zermelo-fraenkel set theory." *Journal of Symbolic Logic*, vol. 41, no. 2, pp. 323–327, 1976.

[40] L. C. Paulson, "Set theory for verification: II. Induction and recursion," Computer Laboratory, University of Cambridge, Tech. Rep. 312, 1993.

[41] E. Pasaali;, W. Taha, and T. Sheard, "Tagless staged interpreters for typed languages," in *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM Press, 2002, pp. 218–229.

[42] L. C. Paulson, "A formulation of the simple theory of types (for Isabelle)," in *COLOG-88: International Conference on Computer Logic*, ser. LNCS 417, P. Martin-Löf and G. Mints, Eds. Springer, 1990, pp. 246–274.

[43] A. Cant, B. Mahony, and J. McCarthy., "Design oriented verification and evaluation: The DOVE project," Research report, DSTOCTRC1349, 2002.

[44] A. Pnueli, "Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends," *Current trends in concurrency. Overviews and tutorials*, pp. 510–584, 1986.

[45] R. Alur and T. A. Henzinger, "A really temporal logic," *J. ACM*, vol. 41, no. 1, pp. 181–203, 1994.

[46] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[47] L. C. Paulson, "Isabelle: The next 700 theorem provers," in *Logic and Computer Science*, P. Odifreddi, Ed. Academic Press, 1990, pp. 361–386.

[48] L. C. Paulson and T. Nipkow, "Isabelle tutorial and user's manual," Computer Laboratory, University of Cambridge, Tech. Rep. 189, Jan. 1990.

[49] R. L. Constable, "ML programming in constructive type theory (abstract)," in *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*. London, UK: Springer-Verlag, 1997, p. 87.

[50] G. Hunter, *An Introduction to the Metatheory of Standard First Order Logic*. University of California Press, 1996.

[51] L. C. Paulson and K. Grabczewski, "Mechanizing set theory," *Journal of Automated Reasoning*, vol. 17, no. 3, pp. 291–323, 1996.

[52] F. Regensburger, "HOLCF: Higher order logic of computable functions," in *Higher Order Logic Theorem Proving and Its Applications*, E. T. Schubert, P. J. Windley, and J. Alves-Foss, Eds. Berlin,: Springer, 1995, pp. 293–307.

[53] L. C. Paulson, "Introduction to Isabelle," *Computer Laboratory, University of Cambridge*, no. 280, 1993.

[54] A. Church, "A formulation of the simple theory of types." *Journal Symbolic Logics*, vol. 5, no. 2, pp. 56–68, 1940.

[55] L. C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1991.

[56] L. C. Paulson and A. W. Smith, "Logic programming, functional programming, and inductive definitions," in *Extensions of Logic Programming*, ser. LNAI 475, P. Schroeder-Heister, Ed. Springer, 1991, pp. 283–310.

[57] F. J. Pelletier, "A brief history of natural deduction," *History and Philosophy of Logic*, vol. 20, no. 1, pp. 1–31, March 1999.

[58] D. Prawitz, *Natural Deduction: A Proof Theoretical Study*. Almquist and Wiksell, 1965.

[59] D. Aspinall, "Proof General: A generic tool for proof development," in *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. London, UK: Springer-Verlag, 2000, pp. 38–42.

[60] T. Nipkow, "Term rewriting and beyond — theorem proving in Isabelle," *Formal Aspects of Computing*, vol. 1, pp. 320–338, 1989.

[61] T. Latvala, "Efficient model checking of safety properties," in *Model Checking Software. 10th International SPIN Workshop*, T. Ball and S. Rajamani, Eds. Springer, 2003, pp. 74–88.

[62] D. A. Peled, *Software Reliability Methods*. New York: Springer-Verlag, 2001.

[63] E. Koutsofios and S. C. North, "Drawing graphs with dot," ATT Bell Laboratories Technical Report, Murray Hill, NJ, Tech. Rep., 1999.

[64] S. Agerholm and H. Skjødt, "Automating a model checker for recursive modal assertions in HOL," Department of Computer Science, University of Aarhus, Tech. Rep. DAIMI Report Number IR-92, January 1990.

[65] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*. London, UK: Springer-Verlag, 1982, pp. 52–71.

[66] K. Schneider and D. W. Hoffmann, "A HOL conversion for translating linear time temporal logic to omega-automata," in *Theorem Proving in Higher Order Logics*, 1999, pp. 255–272.

[67] A. Pnueli, "The temporal logic of programs," Weizmann Science Press of Israel, Jerusalem, Israel, Israel, Tech. Rep., 1997.

[68] D. Peled and L. Zuck, "From model checking to a temporal proof," in *SPIN '01: Proceedings of the 8th international SPIN workshop on model checking of software*. New York, NY: Springer-Verlag New York, Inc., 2001, pp. 1–14.

[69] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach," in *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1983, pp. 117–126.

[70] A. Hendriks, "Computations in propositional logic," Ph.D. dissertation, University of Amsterdam, 1996.

[71] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY: ACM Press, 1985, pp. 97–107.

[72] E. L. Gunter and D. Peled, "Tracing the executions of concurrent programs." *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 4, 2002.

[73] G. Goodson, J. Wylie, G. Ganger, and M. Reiter, "The safety and liveness properties of a protocol family for versatile survivable storage infrastructures," 2004.

[74] Z. Manna and A. Pnueli, "Adequate proof principles for invariance and liveness properties of concurrent programs," *Science of Computer Programming*, vol. 4, no. 3, pp. 257–289, 1984.

[75] A. P. Sistla, "Safety, liveness and fairness in temporal logic." *Formal Aspects of Computing Journal*, vol. 6, no. 5, pp. 495–512, 1994.

[76] M. Daniele, F. Giunchiglia, and M. Y. Vardi, "Improved automata generation for linear temporal logic," in *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1999, pp. 249–260.

[77] J.-M. Couvreur, "On-the-fly verification of linear temporal logic," in *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*. London, UK: Springer-Verlag, 1999, pp. 253–271.

[78] P. Gastin and D. Oddoux, "Fast LTL to Büchi automata translation," in *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2001, pp. 53–65.

[79] D. Giannakopoulou and F. Lerda, "From states to transitions: Improving translation of LTL formulae to Büchi automata," in *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*. London, UK: Springer-Verlag, 2002, pp. 308–326.

[80] F. Somenzi and R. Bloem, "Efficient Büchi automata from LTL formulae," in *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2000, pp. 248–263.

[81] X. Thirioux, "Simple and efficient translation from LTL formulas to Büchi automata," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, 2002.

[82] E. Gunter and D. Peled, "Temporal debugging for concurrent systems," in *Tools and Algorithms for Construction and Analysis of Systems, 8th International Conference, TACAS '02*, ser. LNCS, J.-P. Katoen and P. Stevens, Eds., vol. 2280. Grenoble, France: Springer, April 2002, pp. 431–444.

[83] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis, "Memory-efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, vol. 1, no. 2/3, pp. 275–288, 1992.

[84] G. Holzmann, *The SPIN Model Checker, Primer and Reference Manual.* Reading, Massachusetts: Addison-Wesley, 2004.

[85] K. Slind, "Another look at nested recursion," in *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs'00*, ser. Lecture Notes in Computer Science, M. Aagaard and J. Harrison, Eds., no. 1869. Portland, Oregon, USA: Springer-Verlag, August 2000, pp. 498–518.

[86] S. Krstic and J. Matthews, "Inductive invariants for nested recursion." in *TPHOLs*, ser. Lecture Notes in Computer Science, D. A. Basin and B. Wolff, Eds., vol. 2758. Springer, 2003, pp. 253–269.

[87] R. Boyer, M. Kaufmann, and J. Moore, "The Boyer-Moore theorem prover and its interactive enhancement," 1995.

[88] S. Autexier, D. Hutter, H. Mantel, and A. Schairer, "System description: inka 5.0 - a logic voyager," in *CADE-16: Proceedings of the 16th International Conference on Automated Deduction.* London, UK: Springer-Verlag, 1999, pp. 207–211.

[89] K. S. Namjoshi, "Certifying model checkers," in *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification.* London, UK: Springer-Verlag, 2001, pp. 2–13.

[90] S. Aggarwal, R. P. Kurshan, and K. K. Sabnani, "A calculus for protocol specification and validation." in *Protocol Specification, Testing, and Verification*, 1983, pp. 19–34.

[91] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS '86)*, D. Kozen, Ed. IEEE Computer Society Press, 1986, pp. 332–344.

[92] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987.

[93] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao, "Efficient generation of counterexamples and witnesses in symbolic model checking," in *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation.* New York, NY, USA: ACM Press, 1995, pp. 427–432.