Basic Research in Computer Science

BRICS DS-96-1 U. Engberg: Reasoning in the Temporal Logic of Actions

Reasoning in the Temporal Logic of Actions

The design and implementation of an interactive computer system

Urban Engberg

BRICS Dissertation Series **DS-96-1**

ISSN 1396-7002 **August 1996**

See back inner page for a list of recent publications in the BRICS Dissertation Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Reasoning in the Temporal Logic of Actions

The design and implementation of an interactive computer system

Urban Engberg

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Reasoning in the Temporal Logic of Actions

The design and implementation of an interactive computer system

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
Ph.D. Degree

by
Urban Engberg
September 1995

Abstract

Reasoning about algorithms stands out as an essential challenge of computer science. Much work has been put into the development of formal methods, within recent years focusing especially on concurrent algorithms. The Temporal Logic of Actions (TLA) is one of the formal frameworks that has been the result of such research. In TLA a system and its properties may be specified as logical formulas, allowing the application of reasoning without any intermediate translation. Being a linear-time temporal logic, it easily expresses liveness as well as safety properties.

TLP, the TLA Prover, is a system developed by the author for doing mechanical reasoning in TLA. TLP is a complex system that combines different reasoning techniques to be able to handle verification of real systems. It uses the Larch Prover as its main verification back-end, but as well makes it possible to utilize results provided by other back-ends, such as an automatic linear-time temporal logic checker for finite-state systems. Specifications to be verified within the TLP system are written in a dedicated language, an extension of pure TLA. The language as well contains syntactical constructs for representing natural deduction proofs, and a separate language for defining methods to be applied during verification. With the TLP translators and interactive front-end, it is possible to write specifications and incrementally develop mechanically verifiable proofs in a fashion that has not been seen before.

Dansk sammenfatning
(résumé in Danish)

Én af de største udfordringer indenfor det datalogiske forskningområde er, hvordan man kan gøre det praktisk muligt at ræsonnere om algoritmer – at vise at et program udført i de rigtige omgivelser opfører sig korrekt. Meget arbejde er blevet lagt i udviklingen af formelle metoder til at danne grundlag for sådanne ræsonnementer, og i de senere år har arbejdet specielt været koncentreret omkring parallelle systemer; systemer bestående af flere, parallelt udførte og interagerende programmer. Det sidste skyldes dels, at kompleksiteten af sådanne systemer hurtigt bliver langt mere omfattende end den af sekventielle systemer, og dels, at de fleste systemer vi i dag omgiver os med og gerne skulle kunne stole på er baserede på parallelt kørende processer.

Det store problem ved at ræsonnere omkring parallelle systemer er, at selv for små systemer vokser kompleksiteten i ræsonnementerne hurtigt ud over, hvad det er muligt at behandle med manuelle midler. For at kunne arbejde med virkelige og avancerede computer-systemer er det klart at man er nødt til at tage mekaniske – computer-baserede – midler i brug. Selv med mekaniske hjælpemidler er det imidlertid svært at nå særligt langt. Forsøg på at lade computere udføre *automatiske*, udtømmende analyser af parallelle systemer mislykkes ofte på grund af kompleksitetens eksplosions-artede vækst ved forstørrelsen af det analyserede system. Computer-*assisteret* bevisførelse skulle i teorien kunne hjælpe på nogle af de mere svære eksempler gennem en bedre opdeling af problemerne, men udviklingen har endnu ikke givet os et værktøj, der kan gøre ikke-trivielle ræsonnementer praktisk mulige.

Den såkaldte temporale aktions-logik, TLA, er én af de formelle grundflader for ræsonnering om komplekse systemer, der er blevet udviklet under de senere år. Med hjælp af TLA er det muligt at beskrive et system og de egenskaber, man ønsker det skal opfylde, ved hjælp af ganske simple logiske udtryk. Dermed er det muligt at ræsonnere om systemet og dets egenskaber på en meget direkte måde. Denne afhandling beskriver i et enkelt bidrag til området, hvordan et værktøj til at assistere i ræsonnering omkring systemer beskrevet i TLA kan konstrueres. Afhandlingen går især tæt på den arkitektoniske side af dette værktøj med en beskrivelse af, hvordan det er muligt at gøre mekanisk assisteret bevisførelse praktisk anvendelig.

Som sin hoved-konklusion viser afhandlingen, at vi *kan* konstruere værktøjer, der muliggør udtømmende, verificerbare ræsonnementer om ikke-trivielle systemer. I det præsenterede værktøj, TLP, er problemer relaterede til de meget forskelligartede sider af ethvert ræsonnement blevet løst ved at integrere flere forskellige metoder og til gengæld forsyne værktøjet med en ensartet, interaktiv overflade (front-end).

TLP's styrke vises i afhandlingen ved tre eksempler, der er blevet valgt således, at

de tilsammen bedst muligt beskriver de forskellige aspekter ved bevisførelsen. Det første er et simpelt, traditionelt 'forfinelses'-bevis – et bevis af hvordan man korrekt kan gå fra en overordnet system-beskrivelse til en mere detaljeret implementation – og giver en god indføring i de to grundlæggende ræsonnements-typer, der benyttes i den temporale logik, sikkerheds- og livligheds-ræsonnementer. Det andet behandler korrektheden af en distribueret algoritme til beregning af det kortest udspændende træ i en graf, og repræsenterer et mere komplekst, ikke-endeligt system, hvor livligheds-beviset kræver brug af induktive argumenter. Det tredje er endelig et mere realistisk eksempel, beviset af korrektheden af et $k \cdot 2^m$-bits multiplikations-kredsløb. TLP blev her brugt sammen med et automatisk værktøj, en såkaldt model-checker, og assisterede i at finde kritiske fejl i de indledende specifikationer.

Contents

List of Tables

List of Figures

1

Introduction

Reasoning about algorithms stands out as an essential challenge of computer science. There is a clear need for being able to prove that a program executed in a certain environment performs correctly. Much work has been put into the development of formal methods for handling such reasoning, within recent years focusing especially on concurrent algorithms. TLA, the Temporal Logic of Actions [20], is one of the formal frameworks that has been the result of such research. In TLA, a system and its properties may be specified as logical formulas, allowing application of reasoning without any intermediate translation.

The main problem with formal reasoning about concurrent systems, is that even for small systems the amount of reasoning to be performed is unmanageable without the help of mechanical tools. Even with mechanical tools, we are faced with great problems. Methods for automated reasoning keep getting better, but are still oppressed by the state-space explosion problem encountered as larger systems are examined. Mechanically assisted theorem proving could be the solution to some problems, but is feasible only if more advanced methods for dealing with proofs are constructed; proofs of any non-trivial facts about concurrent systems tend to get out of hand very quickly.

This thesis describes a contribution to the area of mechanically assisted theorem proving. It describes a tool for reasoning about systems specified within TLA, focusing especially on the engineering aspects of creating such a tool, making mechanical reasoning practical for the systems designer.

Reasoning in the Temporal Logic of Actions

TLA is a linear-time temporal logic [19]. This means that it easily expresses liveness (eventuality) as well as safety (invariance) properties. Systems and their properties may be described by logical formulas, making the expression of conjectures very straightforward. The TLA formula $\Pi \Rightarrow \Phi$ asserts that the system represented by Π satisfies the property, or implements the system, represented by Φ. As any formal logic, TLA allows completely rigorous reasoning, so it is clear that proofs can be checked mechanically.

To perform reasoning about a TLA conjecture, it is necessary to represent the logical system in a machine comprehensible form. For TLA, this has been done in a number of different ways. Joakim von Wright and Thomas Långbacka [34, 35], and Ching-Tsun Chou [7, 8] have in individual attempts implemented the semantics of TLA as theories in

the HOL theorem prover [15, 16], a system that guarantees soundness of any proofs based on the defined semantics. Peter Grønning, Leslie Lamport, and the author went in another direction when we chose to represent TLA by a set of axioms and proof rules, basing our implementation on the logically capable verification system known as the Larch Prover (LP) [13, 14]. Sara Kalvala has later [17] taken a third approach in an embedding of TLA in the Isabelle Theorem Prover [28, 29] in which the temporal operators are defined in terms of their semantics (similar to the HOL implementations), but where their use is governed by an axiomatic proof system.

One of the advantages of the LP implementation is that it quickly lets you perform small-sized proofs of invariant properties using the automated rewriting mechanisms of LP, and not having to spend time at the level of semantics. The use of the simple encoding of TLA in LP however suffers from the same problems as any encoding: pure technical aspects start to become distracting as soon as specifications and proofs start to grow, making reasoning unmanageable. This is even more the case when you begin to find profitable ways of enriching the encoding or, as we did, start to work with different encodings for temporal and pure predicate based reasoning. Furthermore, the LP system itself is not very helpful when it comes to proof management; its simple scripting system breaks very easily when specifications change and proofs should be change accordingly.

A possible solution to the mentioned problems is to create a dedicated system, letting specifications be written directly in the logic, rather than any specific encoding, while letting proofs be written in a more structured, manageable fashion. Fortunately, it is not necessary to build a new system from scratch, as LP (or any other verification system) may still serve as a reasoning platform. What that we need is a front-end that can take care of the TLA specific aspects, replacing conjectures and proofs by suitable encodings. The construction and analysis of such a system has been the main aim of the author's work through the last years, as described here.

The design and implementation of an interactive computer system

The system described in this thesis has been named the TLA Prover, TLP. TLP is a complex system that tries to combine different reasoning techniques to provide a tool able to handle verification of realistic systems. It uses the Larch Prover as its main verification back-end, but makes it possible to use different encodings of the logic, as well as other back-ends, whenever this seems profitable. Currently, we refer to two main areas of reasoning using the Larch Prover with two different encodings. *Action reasoning* is reasoning involving only constants, predicates, and actions. This is handled mainly by the term rewriting capabilities of LP, supplied with axioms and rules for boolean reasoning and extended with declarations and definitions of certain operators used in the TLP language. *Temporal reasoning* is reasoning that as well involves temporal operators. This is handled in a different setup, where we are able to represent temporal axioms and rules, and among these instances of the TLA proof rules. Temporal reasoning on finite-state systems and not involving any of the TLA rules, is furthermore supported by an implementation by David Long of decision procedures for linear-time temporal logic (LTL), based on Boolean Decision Diagrams. The LTL checker verifies the appropriate lemmas automatically, where the Larch Prover would need an explicit proof in terms of

temporal logic reasoning.

Specifications and properties to be verified within the TLP system are written in a dedicated language. The *TLP language* is an extension of pure TLA, based on the usual TLA syntax extended with constructs for expressing tuples, conditionals, and sets. As a more exceptional feature, the TLP language also contains syntactical constructs for representing natural deduction proofs with conjectures written in TLA, and a separate language for defining methods to be applied in the verification of proofs.

The *TLP translator* parses any file containing declarations, specifications, and proofs written in the TLP language. In addition to ordinary syntax checking, it is able to determine the wellformedness of any TLA expressions, performing sort checking to ascertain that e.g. the prime operator is only applied to predicates and state functions. The translator generates output for each individual encoding and back-end being used, thus ensuring that the different encodings correspond to the same specification. It uses information from sort checking together with user-supplied guidance to produce conjectures and verification directions, splitting up proofs to let each separate part be handled by the most favorable encoding and back-end.

With the *TLP interactive front-end* it is possible to write specifications and incrementally develop mechanically verifiable proofs of properties in a bottom-up or top-down fashion. The front-end links the separate back-ends, hiding technical details of the encodings, and ensures that each part of the proofs is verified correctly. Being built on top of the customizable text-editor GNU Emacs, it provides syntax directed features for editing of TLP scripts along with a fully interactive graphical interface to the translator and the verification back-ends.

TLP in a wider context

In the area of computer-assisted formal reasoning it is common to speak about *theorem proving* as opposed to *proof checking*. As stated by Peter Lindsay in a survey article [26], the term *theorem provers* generally refer to highly automated systems, such as resolution provers and the Boyer-Moore prover [5, 6]. As well model checkers should be added to this category. As *proof checkers* we usually refer to language-based systems that support some kind of computer-assisted development of proofs. Modern reasoning tools such as HOL [15, 16], Isabelle [28, 29], and as well LP, contain elements of both categories, in that they let the user create verifiable proofs based on a logical foundation, aided by theorem proving techniques such as decision procedures, resolution, and term rewriting.

TLP is not a new theorem prover, nor a proof checker. It does not intend to provide new methods for automatic theorem proving nor a new foundation for proof checking. Instead it tries to combine some of the existing techniques with a tool for managing the development of specifications and proofs – dedicated especially to the Temporal Logic of Actions. It provides a complete *reasoning environment*, in which proofs may be developed incrementally with the aid of different theorem provers, and with different encodings of the logic. Although dedicated for another reasoning area, TLP in many ways resembles NuPRL, a combined reasoning environment and theorem prover for constructive type theory [11]. TLP and NuPRL both provide a user interface which intends to make the tool available "at the workstation". A proof is a syntactic entity of the system's language, with an evident tree structure. From the user interface it is possible to edit and simultaneously

verify proofs, navigating quite freely between the different parts. This is an important feature, which is not always recognized. Writing a proof is most often a process of trial and error, and the number of changes and refinements that have to be done before the proof is finally verified can get very large. TLP, like NuPRL, assists the user by controlling the proof structure, letting him concentrate on the distinct parts while always providing him with information about the context.

TLP distinguishes itself by being based on existing verification tools; back-ends that it may apply for different verification purposes. We have chosen to use the Larch Prover as our main back-end. This has shown to have some advantages, in that the term rewriting mechanisms of this tool provides some very quick and smooth simplification of equations. The notion of deduction rules is adequate for representing the proof rules of TLA, and by splitting action and temporal reasoning we get a very efficient reasoning engine. The down side has been that the handling of some constructs, especially quantification, has been very difficult, where a tool such as HOL clearly would have provided a better platform. Still, the interesting aspect of TLP is not so much the choice of back-ends, as the way it combines the use of them, and presents the results to the user. With additional work, the TLP system might be able to use HOL or Isabelle as major back-ends, or even to combine these with the current LP system. Although we may not be able to solve all problems by adding new back-ends, certain problems, like the encoding and reasoning about *Enabled*-predicates, could clearly be solved this way.

When providing a formal framework to enable reasoning about a system inside a theorem prover or proof checker, there are different approaches to how the system is embedded in the tool's logic. It is here common to speak of *deep embeddings* versus *shallow encodings* as explained among others by Boulton [4].

A *deep embedding* is one in which the syntax of the embedded system is represented by terms in the tool's logic, where semantic functions are added to assign meaning to the terms. A *shallow embedding* is one in which just the semantic operators are defined in the tool's logic, while a user-interface serves to translate the syntax of the embedded system into semantic structures.

Deep encodings are in some way the most simple and pure. The language and its semantics is handled completely by the embedding logic, which ensures correctness of any proof. Deep encodings also make it possible to quantify over syntactic structures, something that could make the encoding of TLA's *Enabled* predicates trivial (a task which is still unsolved in the TLP system). A deep encoding of a complex system is however not very easy to obtain, as the complete mapping between the syntax and semantics of the language has to be expressed in the logic. And the embedding logic may not always be very well suited for modeling the embedded language. In some of our initial experiments with embedding TLA in LP, we thus found the built-in type system to be insufficient for representing the different TLA sorts. Had a deep encoding been a major issue, we might have been better off using a system such as HOL with an encoding like the ones presented by von Wright and Långbacka and by Chou (see chapter 4).

With a shallow encoding, the interface handles the mapping between syntactic specifications and their semantics. This simplifies the work that has to be done to get a working system. It also makes the resulting system less secure, in that the mapping is removed from the logical system.

In the TLP system we have chosen to use a shallow encoding. As mentioned earlier,

we found that one general encoding of TLA in the logic of a proof checker was very hard to find, which would still make reasoning feasible at all levels. Using a translator made the interface language much simpler, and yet more flexible – new syntactic constructs could be added without changing the logical basis. Much more important however, the finding that we might profit from using more than one encoding of the same specification made a shallow encoding the only possible choice.

You might say that TLP uses an even shallower encoding than what we usually refer to by the term. In TLP, some of the reasoning itself has been moved from the logical framework to the interface. The system has been built so that the translator may chose to generate conjectures for certain facts in one specific encoding, while asserting the validness of these conjectures in another. This is the way it links the results from the different back-ends. It obviously constitutes yet another security risk, as the translator easily could introduce circular arguments or any kind of contradicting statements if developed without care. But the gained flexibility is also what makes TLP unique: it may combine techniques as different as natural deduction, term rewriting, resolution, and model checking in the verification of a single proof, something that no other tool is able to do by itself.

Some notes

The original work on finding suitable ways of mechanizing reasoning in TLA, on which the design of the TLP system has been based, was done in cooperation with Peter Grønning and Leslie Lamport. Many of the basic ideas with respect to the logical framework and the Larch Prover encodings are due to this joint work, of which the author contributed with a substantial part. With Peter Grønning, the author performed the initial experiments with different encodings, writing some of the first mechanically verified TLA proofs, which lead to the decision of using different encodings for temporal reasoning and action reasoning. The author wrote the first versions of the translator for handling the different encodings, and in the process designed the first version of the TLP language with separate declaration and definition constructs. The idea of extending this language to be able to express proofs was due to joint discussions, while the design was created by the author, inspired by the formalized language by Leslie Lamport for writing hand proofs and ideas provided by Peter Grønning through his experimenting with the system. The way of handling case and contradiction proofs was thus invented by the author, as well as the basic idea of having typed methods, applicable on the same level as subproofs, etc. The basic idea and the design of the TLP front-end are due to the author alone, as well as many of the internal proof techniques that was developed concurrently with it (as e.g. the automatically applied quantification functions and pre-order verification). The implementation of the TLP system, including the translator, the make-system, and the front-end, has been written by the author, as most of examples of the use of TLP including the three main examples presented in this thesis. The encoding for, and use of the LTL checker provided by David Long, is as well due to the author.

The TLP system itself has grown to become slightly more than a first prototype through the later years. The translator source which currently contains generators for three different encodings, consists of just under 10,000 lines of Standard ML, lex, and yacc code. The front-end, which relies heavily on the high-level functionality of built-in

Emacs constructs, is almost 5,000 lines of Emacs Lisp. And along with the system itself, there is a growing number of practical examples on the use of TLP, of which are also examples written by Peter Grønning and Stephan Merz. It is possible to obtain a version of the TLP system running on Digital DECstations and Sun SPARCs together with the examples by following the instructions in appendix E.

Although the thesis is meant partly to be serving as a guide and manual to the TLP system, the description of TLP herein is not complete. There are lots of details that the author would have liked to comment on and describe. The current state of TLP makes this rather hard, however: it is a complicated tool which is still very much on a prototype level; many things should improve, and much may change. Using the system does mean that the user has to be interested in learning about problems and shortcomings partly by himself and in putting some hard work into understanding how the tools work to solve his problems. The description in this thesis thus mostly gives an overview, listing as comprehensively as possible the features and ideas.

Organization of the thesis

Before turning to the formal discussion of mechanical reasoning, we have devoted chapter 2 to an introduction to the practical use of TLP, a *guided tour* that will take you through the steps of writing a TLA specification and verifying a simple proof of some of its properties.

The rest of the thesis has been split into three parts. Part I describes some of the logical issues of reasoning in TLA and discusses the problems involved in providing a system for mechanical verification. In chapter 3 we give a brief introduction to TLA, its syntax and semantics, with some simple examples. Chapter 4 continues with a discussion of how TLA may be encoded for handling by a mechanical verification tool, as well as giving a brief introduction to the Larch Prover. Chapter 5 finally describes the problems and proposed solutions concerning the design of a human-computer interface.

Part II contains an outline of the TLP system. Chapter 6 gives a full description of the TLP language with examples, while chapter 7 gives a more brief description of the translator, how it works and may be used. Finally, chapter 8 presents the interactive front-end, describing its multiple features at different levels.

Part III concludes with three detailed examples on the use of TLP. The examples have been chosen so that they together form an extensive illustration of the different aspects of TLP reasoning. The first describes a traditional refinement proof, based on a simple example from Lamport's TLA report [20], which nicely illustrates some trivial safety proofs as well as a more complicated liveness analysis in connection with the refinement of the separate actions. The second, concerned with the correctness of a distributed spanning-tree algorithm, presents a more complicated, non-finite state example with quantified formulas. This example contains a non-trivial liveness proof, which constitutes one of the more interesting studies of the use of TLP, including the application of the TLA Lattice rule, based on well-founded induction. The third example finally represents a more realistic example of refinement in a real-world perspective, the verification of a $k \cdot 2^m$-bit multiplier circuit constructed by recursive composition. TLP was here used together with a model-checker, and assisted in finding critical bugs in the original specifications.

The three parts are followed by conclusions and pointers to future work in chapter 12.

Acknowledgments

The work presented in this thesis is to a very high degree due to initiatives, ideas, and personal help from Leslie Lamport. I would like to thank him for his encouragement and deep involvement in the work, the many good discussions we've had, and for our friendship through the years. Likewise I would like to thank Peter Grønning for his participation and contributions; many details of the TLP system should be attributed to him, where this is not always explicitly stated in the text.

As sponsors for my work, I would like to thank Digital Equipment Corporation's Systems Research Center and Bob Taylor for a very nice year in Palo Alto, and Digital's External Research Projects for donating hardware, without which my extended work would not have been possible.

Lots of people have been helpful and willing to discuss the problems I have had. From the Systems Research Center I would like to thank especially Martín Abadi, Jim Saxe and Jim Horning. John Guttag and Steve Garland contributed extensively to our work by providing the LP theorem prover, but were also very helpful whenever we met, and were even willing to change their tool to suit our needs.

After returning to the Computer Science Department in Århus I have had profitable discussions with a number of people, of which I would like to mention Henrik Andersen, Olivier Danvy, Uffe Engberg, Klaus Havelund, Peter Herrmann, Sara Kalvala, Heiko Krumm, Kim Guldstrand Larsen, Karoline Malmkjær, Tom Melham, and Arnulf Mester.

I owe much to Stephan Merz for his extensive experimenting with TLP, finding a number of bugs and encouraging the further development of the tools. Other testers that contributed with questions and comments were John Donaldson, Marc Foissotte, and Denis Roegel.

For letting me use his LTL checker, and instructing me in its use, I thank David Long. I also would like to thank the people behind New Jersey SML and related tools, among all Dave MacQueen, David Tarditi, and John H. Reppy for their excellent products and nice user support. Likewise I would like to give my thanks and full support to Richard Stallman and the GNU community for the much-more-than-text-editor, Emacs, and the instantaneous help I have got whenever in trouble. For implementing the first version of the Emacs/TLP front-end, I thank Christian Lynbech.

My examiners, Hans Henrik Løvengreen, Tom Melham, and Mogens Nielsen have been exceptionally thorough when reading and commenting on the submitted version, giving me many useful hints for the final paper. The errors and shortcomings that remain are not due to their ignorance, but solely to my own limited efforts.

Glynn Winskel has been my always encouraging supervisor, helping me getting it all finished when it seemed most difficult. And last but not least I want to thank my wife Solveig for all her support and encouragement, enduring my inability to ever finish my work.

Århus, March 1996

Urban Engberg

2

A guided tour of TLP

We begin by a simple example showing what the TLP system is all about. We will illustrate how you may write a specification of a simple system, supposed to be correct with respect to a single safety property, an invariant that we want to prove. We give a proof, and have it verified within the TLP system, giving an overview of the system and some of the most important concepts.

2.1 A simple example

The example we are going to look at is a small parallel system. We assume two processes *P0* and *P1* who may perform a number of non-critical operations, but once in a while need to perform an operation in a critical region of the system, in which each must exclude access by the other – writing a shared memory cell, for instance. Mutual exclusion is assured with the help of a flag *critical* that for each process tells whether it is in the critical region. Each process examines the value of the other process' flag before entering the region. The operation of examining the flag and setting the process' own flag is considered atomic (implemented by some kernel procedure).

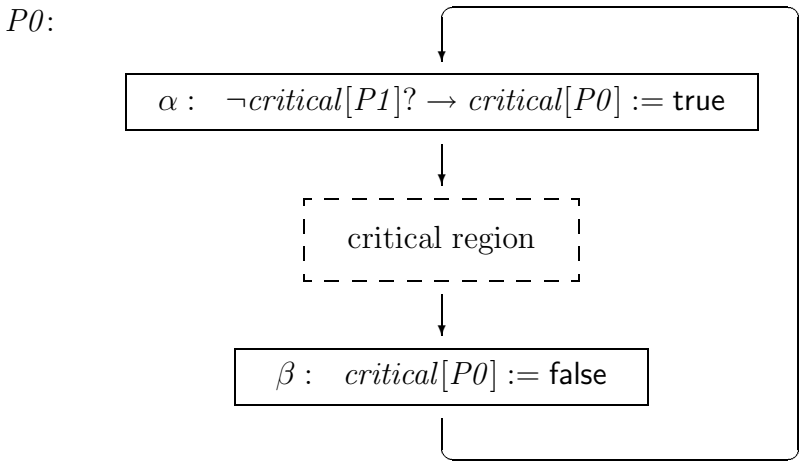


Figure 2.1: Process *P0* of our simple example.

We are going to describe this system on a very general level, in which we ignore the

non-critical operations and represent only the actions where a process enters or leaves the critical region. Figure 2.1 contains a diagram illustrating one of the processes.

2.2 The TLA specification

In figure 2.2 on the next page we present the TLA representation of the system, as it may be specified as a script in the TLP system.

The script begins with two directives, giving it a name and indicating that it depends on basic definitions given in the *frame*-files. Then follows declarations of the variables *critical* and *pc*.

critical is the flag described above that tells whether a process is in the critical region. Rather than using two variables *critical0* and *critical1* we use a parameterized declaration taking a process identifier; this saves us some work as the processes are identical. The other parameterized variable, *pc*, represents the program counter of each process.

We then declare some values used in the example, *P0* and *P1* being the names of the two processes used as parameters to the variables, and *alpha* and *beta* indicating the two possible values of the program counters.

The operator *other* takes a process identity and returns the identity of the other process. It is declared in TLP as a unary operator and defined by two LP rewrite rules.

We are then ready to make the essential definitions that let us specify the algorithm itself. The state function v is simply a tuple containing all the variables used in the specification.

Init is a predicate specifying the initial conditions of the two processes; for each process p, *critical*`(`p`)` is false and the program counter equals *alpha*. (A list bulleted with `/\`'s denotes the conjunction of the items.)

N is the disjunction of actions that may be performed, which are for each process either *Na* or *Nb*. *Na* is the action where a process enters the critical region, indicated by the fact that the program counter *pc* when the action is performed has the value *alpha*, and after it has finished, the value *beta* (the latter value is referred to through the `'` (prime) operator as *pc*`(`p`)'`). *Na* has as a pre-condition that *critical* of the other process must be false (it won't be performed unless this is the case), and sets *critical* of the process itself to true. As we are dealing with logics and not giving an operational description of the algorithm, we explicitly state that neither *critical* nor the program counter of the other process are changed. *Nb* is the action where a process simply leaves the region again, which means that the program counter changes back to *alpha*, and *critical* of the process is set back to false.

With these definitions, the formula *Pi* representing the two processes may be given. It states that the predicate *Init* is initially satisfied and that each step thereafter (indicated by the `[]` (always) operator) is *either* an N step or a step in which none of the variables v of the system are changed.

2.3 Reasoning about the system

We are interested in showing that the given system assures mutual exclusion of the two processes inside the critical region. What we want to show is thus that *Pi* implies that

Name MutualExclusion

%-Use frame

Variables

critical(*p*) : **Bool**
pc(*p*) : **Val**

Values

P0, *P1*
alpha, *beta*

Operators

other : **Val -> Val**

Act

Do "assert *other*(*P0*) -> *P1*"
Do "assert *other*(*P1*) -> *P0*"

End

Statefunctions

vp(*p*) == (*critical*(*p*), *pc*(*p*))
v == (*vp*(*P0*), *vp*(*P1*))

Predicates

Initp(*p*) == /\ ~*critical*(*p*)
/\ *pc*(*p*) = *alpha*
Init == *Initp*(*P0*) /\ *Initp*(*P1*)

Actions

Na(*p*) == /\ *pc*(*p*) = *alpha* /\ *pc*(*p*)' = *beta*
/\ ~*critical*(*other*(*p*))
/\ *critical*(*p*)'
/\ **Unchanged**(*critical*(*other*(*p*)), *pc*(*other*(*p*)))
Nb(*p*) == /\ *pc*(*p*) = *beta* /\ *pc*(*p*)' = *alpha*
/\ ~*critical*(*p*)'
/\ **Unchanged**(*critical*(*other*(*p*)), *pc*(*other*(*p*)))
Np(*p*) == *Na*(*p*) \/ *Nb*(*p*)
N == *Np*(*P0*) \/ *Np*(*P1*)

Temporal

Pi == *Init* /\ [][*N*]_*v*

Figure 2.2: The TLP specification of the system.

at most one of the program counters at any time is equal to *beta*. In TLP we can define the predicate *MutExcl* for mutual exclusion as

Predicates
MutExcl == ~(*pc*(*P0*) = *beta* /\ *pc*(*P1*) = *beta*)

and then show the theorem *Pi* => []*MutExcl*.

The experienced will quickly see that it is not possible to show *MutExcl* directly as an invariant of the actions of *Pi*. To show []*MutExcl* we need first to show a stronger invariant, *Inv*, including the statement that for each process *p* either *critical*(*p*) is true or the program counter *pc*(*p*) is equal to *alpha*. This may thus be defined as

Predicates
Invp(*p*) == *critical*(*p*) \/ *pc*(*p*) = *alpha*
Inv == /\ *Invp*(*P0*) /\ *Invp*(*P1*)
/\ *MutExcl*

Now we are ready to begin the proof. A proof session in TLP is started by the user writing a proof outline, which is just a proof in a low level of detail. The initial outline of the proof of *Pi* => []*MutExcl* might look like

Theorem MutExcl
Pi => []*MutExcl*
Proof
<1>1 *Pi* => []*Inv*
Qed
Qed

The part between **Proof** and the last **Qed** is the proof of the theorem; any substep of the proof is opened by a *label* <*level*>*item* and closed by the matching **Qed**.

Before starting the verification session we might as well fill in some more steps that we know will be needed. To show *Pi* => []*Inv* we have to use the basic TLA induction rule INV1:

$$\frac{I \;/\backslash\; [N]\_f \Rightarrow I'}{I \;/\backslash\; [][N]\_f \Rightarrow [] I}$$

This states that if we can prove the fact that the predicate I is maintained by the action $[N]\_f$, then we may deduce that I is always valid, given that it is initially valid and we always perform $[N]\_f$ steps. To show *Pi* => []*Inv* we should thus first show that *Inv* is initially satisfied, i.e. by showing *Init* => *Inv*, then show that $[N]\_v$ maintains it, *Inv* /\ $[N]\_v$ => *Inv*', and finally apply the INV1 rule to get *Inv* /\ [][N]_f => []*Inv*. We insert this in the proof so that step <1>1 now looks as

<1>1 *Pi* => []*Inv*

<2>1 **Assume** *Init* **Prove** *Inv*
Qed

<2>2 **Assume** *Inv*, [*N*]_*v* **Prove** *Inv'*
Qed

<2>3 *Inv* /\ [][*N*]_*v* => []*Inv*
INV1 with pred_I <- *Inv*, act_N <- *N*, sf_f <- *v*
Qed

Qed

In the steps <2>1 and <2>2 we have used a new notation, writing e.g. **Assume** *Init* **Prove** *Inv* instead of *Init* => *Inv*. The conjecture that we prove is really the same, but by the **Assume**-construct we indicate that we want it proved in a natural deduction style, by assuming that *Init* is true for a given state of the system and showing that *Inv* then must be satisfied as well.

We would usually write the definitions of the theorem and the proof in a separate file. Like in the case of the specification we begin by some directives

Name Invariant

%-**Use** def
%-**Include** ../../base/methods

giving all local definitions the name 'Invariant', declaring that they depend on definitions in the specification file (`def.tlp`), and finally expressing that we may use verification methods defined in our basic methods file.

2.4 Starting the verification session

The interactive front-end to TLP is built on top of the real-time display editor GNU Emacs [32]. This is convenient, as Emacs also lets you do all the editing of specifications and proofs in a standardized way, whether in the middle of a verification session or not. You start the verification session by switching to the Emacs buffer containing the proof and typing 'control-C control-C a return' (see chapter 8 for an extensive description of the front-end commands). Emacs then splits the current frame into three windows, a large window showing the proof, and two smaller windows displaying what is done by TLP (labeled "`*TLP Make*`") and the output of the verification back-end ("`*Larch Prover*`" or "`*LTL Checker*`") respectively. In the "`*TLP Make*`" window you will now see that the specification file and the proof file are translated, and the output then executed through the Larch Prover (LP). After a minute or so, execution is stopped, with "Verification problem: step could not be verified" displayed in the Emacs echo area. The Emacs frame will then look like in figure 2.3 on the next page.

The line containing the **Qed** of step <2>1 has been highlighted to indicate the point where the problem appeared, the so-called *verification point*. A **Qed** indicates that the current goal should have been proved by the verification back-end, which has not been possible. The Make buffer has disappeared, instead allowing the "`TLP proof correction buffer`" to take its place. This is where we will now do all our work on the proof, writing additional steps and instructions and editing the original ones.

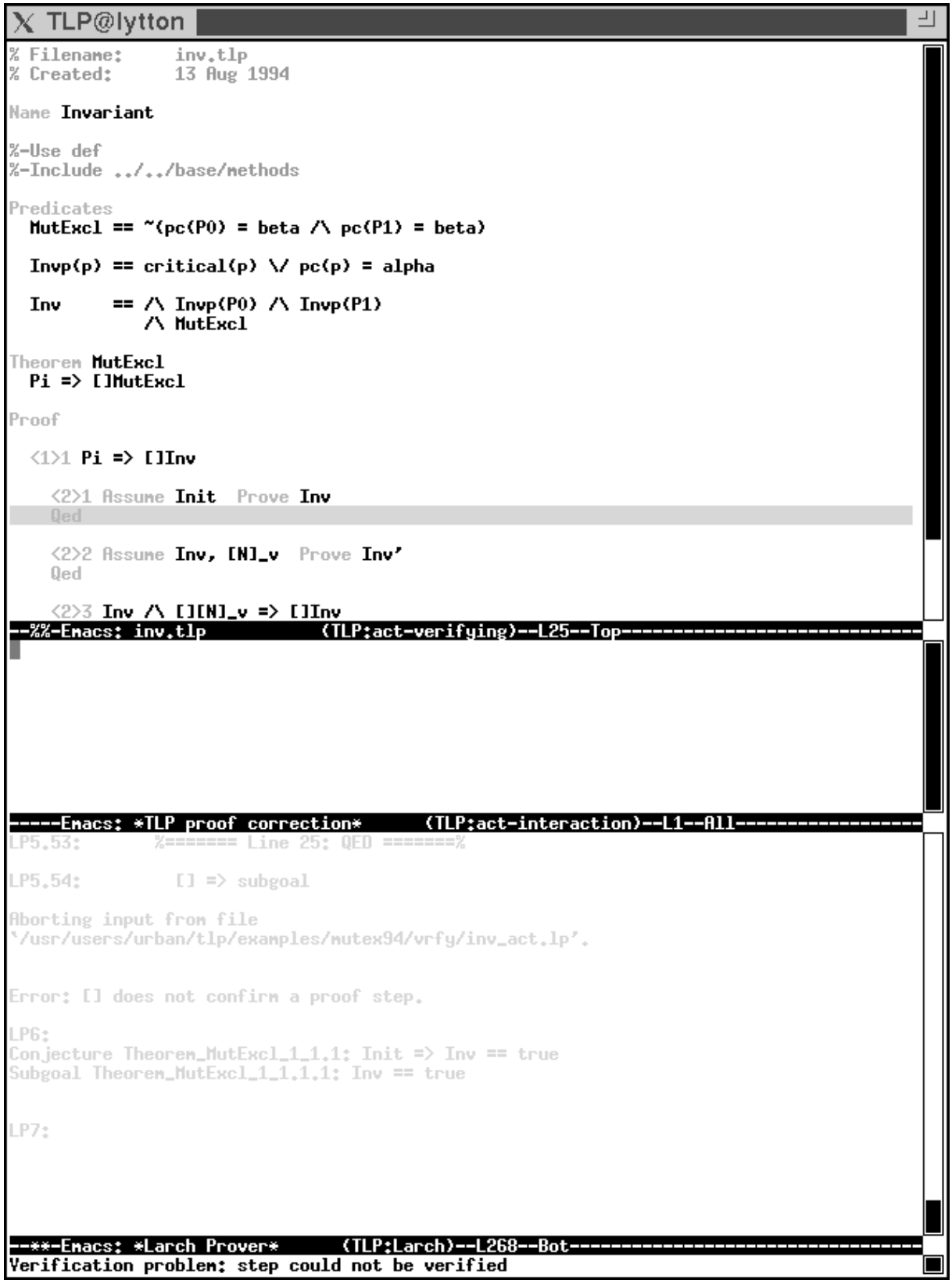


Figure 2.3: The Emacs frame at the first interaction point.

To find out what to do next we might first try to find out what the status of the proof is. Typing 'control-C control-C space' opens a "`*Completions*`" buffer in the bottom window, as shown in figure 2.4, containing some different command options. Typing

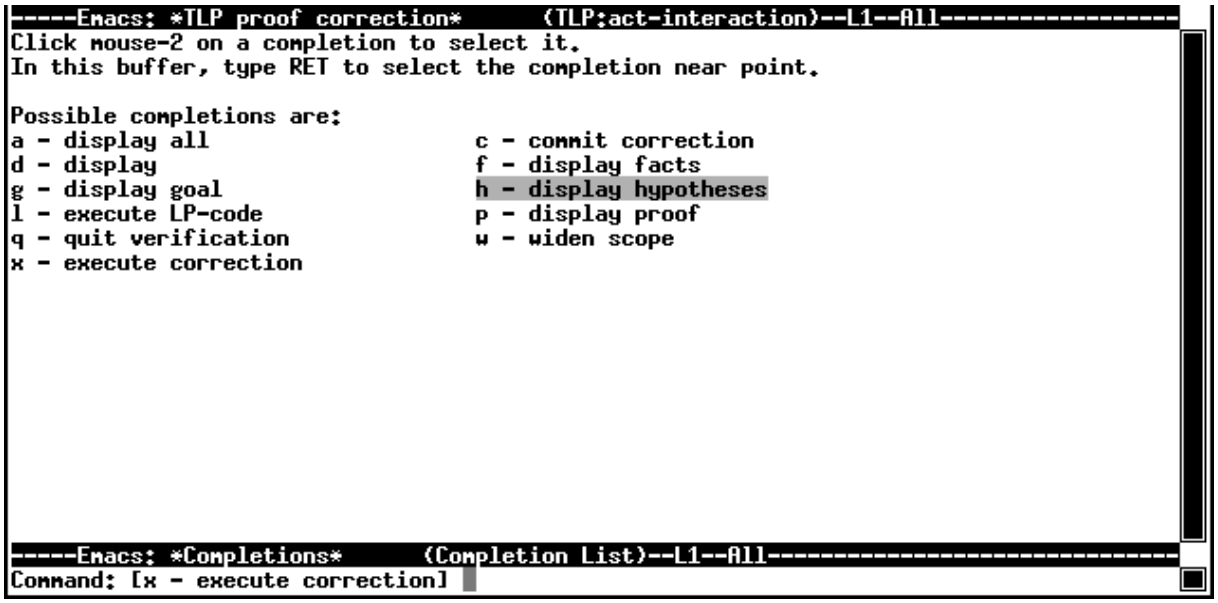


Figure 2.4: The command completion buffer while verifying.

'h return' or simply clicking the mouse on top of "`display hypotheses`" makes the completions buffer disappear, returning to the LP buffer in which you will now see the currently assumed hypotheses of the proof. There is only one hypothesis, namely *Init*, indicated by an LP rewrite rule

```
Rewrite rules:
Theorem_MutExcl_1_1ImpliesHyp.1: Init -> true
```

You may similarly have the goal displayed, which will tell you that this is just *Inv*, as expected. Now, to be able to deduce *Init* from *Inv*, it is clear that we need to know what the definitions of these are; rewriting the predicates with their definitions should make the proof trivial. However, no definitions are used in a TLP proof until explicitly asked for by the user. What we are going to do is therefore to tell TLP to *expand Init* and *Inv* by their definitions in the hypotheses and the goal respectively. We use the so-called verification method 'Expand' that is defined in the basic methods file that come as part of the TLP system, and edit the proof correction buffer so that it now looks as in figure 2.5.

Then we ask TLP to execute the correction and continue verification by typing 'control-C control-C return'. The new code is then automatically inserted and indented at the right point in the proof, and the translated output executed through LP. After a few seconds, however, execution is stopped again as the step still cannot be verified. We may again have the goal and hypotheses displayed (actually, the goal is already displayed in the LP buffer), which will now tell us that the goal has been rewritten to *Invp*`(`*P0*`)` & *Invp*`(`*P1*`)` & *MutExcl*

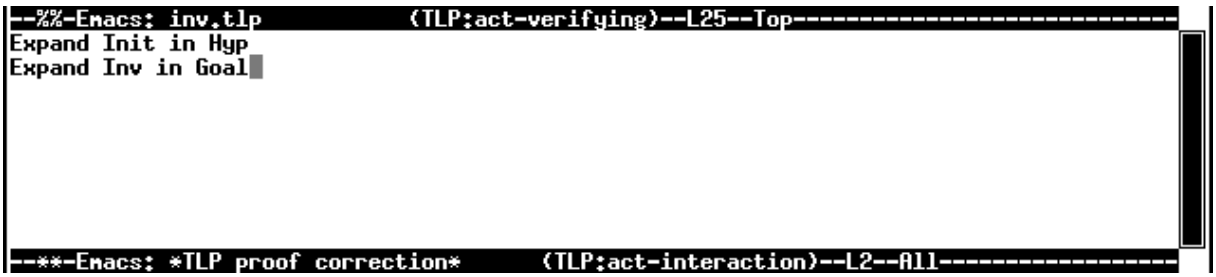


Figure 2.5: The first two methods inserted into the proof correction buffer.

and that the hypotheses are *Initp*(*P0*)and *Initp*(*P1*). Obviously, by using the definitions of *Init* and *Inv*, we only got the first level of the definitions expanded.

We might continue by inserting another pair of 'Expand's in the correction buffer and re-executing. But as we want the proof to be as nice looking and concise as possible, we could also edit the previous two commands, to let them do all the work. We ask TLP to let us edit the whole proof step instead of just inserting text at the verification point by the 'widen scope' command (as usual, type 'control-C control-C' followed by 'w return'). After calling this command, you will see that step <2>1 has been inserted into the correction buffer, and that the highlighted *verification region* that you are now working on has been expanded to cover the complete step in the proof buffer. You should then edit this so that the 'Expand' commands will also expand *Initp*, *Invp*, and *MutExcl*. The Emacs frame will then look as in figure 2.6.

2.4.1 The action step

With this correction, LP successfully verifies the step and goes on to <2>2, where it stops again. This time we want to prove *Inv'* from the hypotheses *Inv* and $(v = v') \mid N$. The second hypothesis is a disjunction, when expanded containing five disjuncts altogether. To use it, we must split the proof into cases, representing each of the disjuncts. We begin by widening the proof scope so that we now work on the complete step. We insert a few lines between the label and the **Qed**, and position the cursor in between. There we may now type 'control-C control-D space', to which Emacs will let us choose a special syntactic construct to insert into the correction buffer, as displayed in figure 2.7. Selecting a 'casestep' we get

<3>1 **Case**
Qed

inserted at the point of the cursor. After the **Case**, we add the formula *Na*(*P0*), being the first disjunct. Repeating the process we add case steps for *Nb*(*P0*), *Na*(*P1*), *Nb*(*P1*), and **Unchanged**(v). At the end of the proof we insert the built-in method call **By-Cases**, which makes LP apply the right tactic for case proofs, and make sure that N and Np are expanded by their definitions in the hypotheses.

We are a bit clever this time, and insert

Activate *Inv*, *Invp*, *MutExcl*

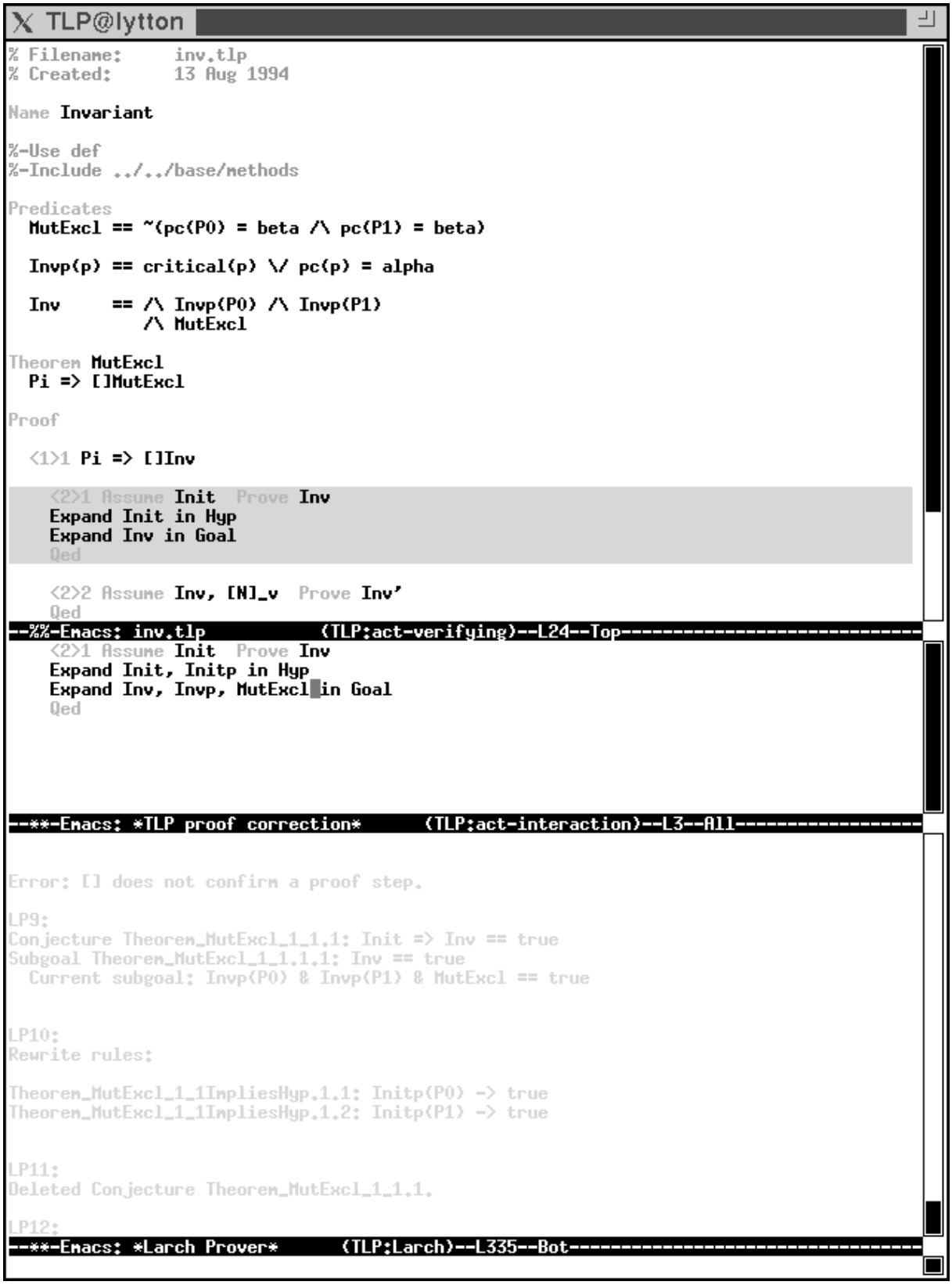


Figure 2.6: The Emacs frame after editing step <2>1.

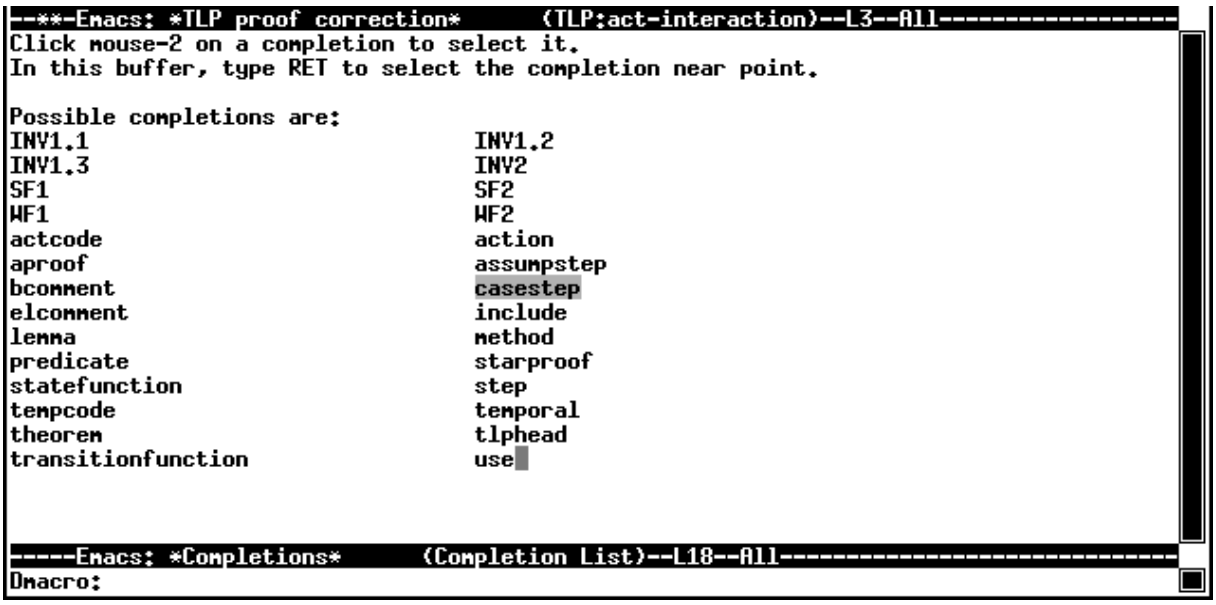


Figure 2.7: The dmacro completion buffer.

in the beginning of the step, so that these definitions automatically will be used in the proof. The only thing missing is then to let each case step make use of the definition of its disjunct; we may either activate these definitions as well or insert expand-commands in each step (as shown in figure 2.8 on page 20). Activating the definitions gives a more concise proof, but also slows down verification. The general rule in TLP is to keep definitions passive as much as possible, as this also makes it easier to understand what is going on.

2.4.2 The temporal steps

After executing the new parts, the action step is verified, and TLP moves on to work on the temporal parts of the proof. (You will notice that the mode-specification below the proof buffer changes to "`TLP:temp-verifying`".)

When writing the proof outline, we already stated that step <2>3 should be proved by application of the INV1 rule (with proper instantiations), and this does indeed go through without problems. Instead, TLP stops at the **Qed** of step <1>1, Pi => []Inv. It is easy to see that this step should be deduced from steps <2>1 and <2>3, expanding Pi to get the two hypotheses $Init$ and [][N]_v. To make this deduction, we need to insert three lines, so that the proof now looks as

<1>1 Pi => []Inv

⋮

By-Implication
Expand Pi in **Hyp**
UseTempFact **Step**1, **Step**3
Qed

The effects of the three method applications are as follows:

By-Implication – like **By-Cases**, tells TLP to do the proof by showing that `[]`*Inv* is satisfied by a given behaviour if *Pi* is;

Expand *Pi* in **Hyp** – expands *Pi* by its definition; and

UseTempFact **Step**1 **Step**3 – a standard method that lets TLP make a deduction based on the two mentioned steps.

With these additions, step `<1>1` is proved, and the top-level step is the only one missing. This just uses the fact of step `<1>1`, *Pi* `=>` `[]`*Inv*, to conclude the weaker result, *Pi* `=>` `[]`*MutExcl*, by expanding *Inv* and applying the temporal rule *AndBox*, which states that the 'Box' operator distributes over conjunctions.

TLP finally moves on to the last mode of verification, "`ltl-checking`", which applies decision procedures to automatically prove certain steps of pure propositional and linear time temporal logic. As the proof does not contain any steps to be proved this way, TLP quickly finishes verification and concludes with the message "`Complete verification ended`" in Emacs' echo area. The finished proof is shown in figure 2.8.

```
Theorem MutExcl
  Pi => []MutExcl
Proof
  <1>1 Pi => []Inv
    <2>1 Assume Init Prove Inv
    Expand Init, Initp in Hyp
    Expand Inv, Invp, MutExcl in Goal
    Qed
    <2>2 Assume Inv, [N]_v Prove Inv'
    Activate Inv, Invp, MutExcl
      <3>1 Case Na(P0)
      Expand Na in Hyp
      Qed
      <3>2 Case Nb(P0)
      Expand Nb in Hyp
      Qed
      <3>3 Case Na(P1)
      Expand Na in Hyp
      Qed
      <3>4 Case Nb(P1)
      Expand Nb in Hyp
      Qed
      <3>5 Case Unchanged(v)
      Expand v, vp in Hyp
      Qed
    By-Cases
    Expand N, Np in Hyp
    Qed
    <2>3 Inv /\ [][N]_v => []Inv
    INV1 with pred_I <- Inv, act_N <- N, sf_f <- v
    Qed
  By-Implication
  Expand Pi in Hyp
  UseTempFact Step1, Step3
  Qed
By-Implication
UseTempFact Step1
Expand Inv in Step1
Activate AndBox
Qed
```

Figure 2.8: The complete proof of the invariant.

Part I

Reasoning in the Temporal Logic of Actions

3

The Temporal Logic of Actions

This chapter contains a brief introduction to the Temporal Logic of Actions, the formalism invented by Leslie Lamport which forms the basis for the work presented in this thesis. The chapter is to a large degree based on Lamport's TLA Report [20], in which you will find a wider-ranging exposition of the formalism.

TLA is a logic for specifying and reasoning about concurrent systems. It consists of ordinary predicate logic, extended with two classes of variables called *rigid variables* and *flexible variables*, and the two operators $'$ (prime) and $\Box$. Rigid variables are parameters of our specifications, and are also sometimes referred to as constants. Flexible variables are the 'program' variables of our specifications; we often refer to these simply as variables. The prime operator is our next state operator, used for specifying *actions* (transitions) and $\Box$ is the usual 'always' operator for specifying temporal properties.

3.1 The logic of actions

We assume a set Val of values, denoting data items. The semantics of TLA is based on the concept of a *state*, which is an assignment of values to (flexible) variables. We write $[\![F]\!]$ to denote the semantic meaning of the syntactic object F. The meaning of a variable x is said to be a mapping from states to values, so that for a state s, we let $s[\![x]\!]$ denote the value $s(x)$ of x in s.

A *state function* is an ordinary expression built from variables and values, such as $x + y - 1$. The meaning $[\![f]\!]$ of a state function f is again a mapping from states to values. $s[\![f]\!]$ is defined as the value of f where we for all free variables x substitute $s[\![x]\!]$. $[\![x + y - 1]\!]$ thus denotes the mapping from a state s to the value $s[\![x]\!] + s[\![y]\!] - 1$ (for brevity we don't bother to differentiate between the syntactic symbols $+$, $-$, and 1, and the semantic values they represent, and write both the same way).

A *predicate* is a boolean-valued state function, such as $x = y + 1$. The meaning $[\![P]\!]$ of a predicate P is thus a mapping from states to the values *true* and *false*. We say that s *satisfies* P iff (if and only if) $s[\![P]\!]$ equals *true*.

An *action* is a boolean-valued expression containing primed and unprimed variables, such as $x' = x + y$. An action denotes a relation between states, representing a transition from 'old' states to 'new'. By $s[\![\mathcal{A}]\!]t$ we denote the value of $[\![\mathcal{A}]\!]$ applied to the pair of states $\langle s, t\rangle$. This is defined to be the value of $\mathcal{A}$ where we substitute $s[\![x]\!]$ for all unprimed appearances of the free variable x, and $t[\![x]\!]$ for all primed. Intuitively, x represents the

value of x in the old state s and x' the value of x in the new state t. We say that $\langle s, t\rangle$ *satisfies* $\mathcal{A}$, iff $s[\![\mathcal{A}]\!]t$ equals *true*. The transition from s to t is then called an $\mathcal{A}$ step. An action $\mathcal{A}$ is *valid*, denoted $\models \mathcal{A}$ iff all steps are $\mathcal{A}$ steps, i.e. iff all pairs $\langle s, t\rangle$ of states satisfies $\mathcal{A}$.

As a predicate can be viewed as an action containing no primed variables, we may extend the interpretation of a predicate P to be a mapping from pairs of states, $s[\![P]\!]t$ being equal to $s[\![P]\!]$.

We allow that state functions be primed, and define the primed state function f' to equal f in which any free variable x is replaced by its primed counterpart x'.

Unchanged f is defined as the action $f = f'$, the *stuttering* step in which f does not change. By $[\mathcal{A}]_f$ we denote the action $\mathcal{A} \vee$ *Unchanged* f, the action that is either an $\mathcal{A}$ step or a stuttering step (with respect to f). The dual form $\langle\mathcal{A}\rangle_f$ denotes $\mathcal{A} \wedge \neg($*Unchanged* $f)$, an $\mathcal{A}$ step that necessarily changes f.

Enabled $\mathcal{A}$ is defined for any action $\mathcal{A}$ to be the predicate that is true for all states s such that there exists a state t for which $\langle s, t\rangle$ satisfies $\mathcal{A}$.

3.2 The temporal logic

A *temporal formula* is an expression built from elementary formulas, the boolean operators $\wedge$ and $\neg$ and the unary operator $\Box$ (read *always*). In what we call the *Raw* logic, elementary formulas are just actions, while in the refined version of the logic we are more restrictive, demanding that elementary formulas are either predicates or of the form $\Box[\mathcal{A}]_f$. Writing specifications in this refined logic makes them *invariant under stuttering* [20, page 39] which is helpful when showing the correctness of refinements. TLP allows reasoning in Raw TLA.

The semantics of the temporal logic is based on *behaviours*, which are infinite sequences of states. A temporal formula is interpreted as a predicate over behaviours, i.e. $[\![F]\!]$ is a mapping from behaviours to *true* and *false*. We give a structural definition of the meaning of F for the behaviour σ, denoted $\sigma[\![F]\!]$. $\sigma[\![F \wedge G]\!]$ equals *true* iff $\sigma[\![F]\!]$ and $\sigma[\![G]\!]$ both equal *true*. $\sigma[\![\neg F]\!]$ equals *true* iff $\sigma[\![F]\!]$ does not. $\sigma[\![\Box F]\!]$ equals *true* iff for all *tails* σ_i of σ, $\sigma_i[\![F]\!]$ equals *true*. A tail of the behaviour σ being the infinite sequence $\langle\!\langle s_0, s_1, s_2, \ldots\rangle\!\rangle$, is a behaviour $\langle\!\langle s_i, s_{i+1}, s_{i+2}, \ldots\rangle\!\rangle$ for any i. Finally, the meaning $\sigma[\![\mathcal{A}]\!]$ of an action $\mathcal{A}$ is the value $s_0[\![\mathcal{A}]\!]s_1$ where s_0 and s_1 are the initial two states of σ.

If $\sigma[\![F]\!]$ equals *true*, we say that σ *satisfies* F and write $\sigma \models F$. The temporal formula F is said to be *valid*, written $\models F$ iff all behaviours satisfy F.

For convenience, we extend the temporal logic with the boolean operators $\vee$, $\Rightarrow$, and $\Leftrightarrow$, defined as usual. The unary operator $\Diamond$ (read *eventually*) is defined so that $\Diamond F$ equals $\neg\Box\neg F$. $F \rightsquigarrow G$ (F *leads to* G) is furthermore defined to equal $\Box(F \Rightarrow \Diamond G)$.

For reasoning about liveness, we add notations for fairness. *Weak fairness* of the action $\langle\mathcal{A}\rangle_f$ means that $\langle\mathcal{A}\rangle_f$ steps are eventually taken as long as they remain possible, i.e. as long as $\langle\mathcal{A}\rangle_f$ remains enabled. Formally, $\mathrm{WF}_f(\mathcal{A})$ is defined as $\Box\Diamond\langle\mathcal{A}\rangle_f \vee \Box\Diamond\neg$*Enabled* $\langle\mathcal{A}\rangle_f$. *Strong fairness* of the action $\langle\mathcal{A}\rangle_f$ means that $\langle\mathcal{A}\rangle_f$ steps are performed eventually, if the action is enabled *infinitely often*; formally $\mathrm{SF}_f(\mathcal{A})$ is $\Box\Diamond\langle\mathcal{A}\rangle_f \vee \Diamond\Box\neg$*Enabled* $\langle\mathcal{A}\rangle_f$.

The full logic described by Lamport extends the logic described above with quantification over both rigid and flexible variables. The logic without quantification is called

Syntax

$\langle formula \rangle \triangleq \langle predicate \rangle \mid \Box[\langle action \rangle]_{\langle state\ function \rangle} \mid \neg\langle formula \rangle$
$\mid \langle formula \rangle \wedge \langle formula \rangle \mid \exists\, \langle rigid\ variable \rangle : \langle formula \rangle$
$\mid \Box\, \langle formula \rangle$

$\langle action \rangle \triangleq$ boolean-valued expression containing constants, variables, and primed variables

$\langle predicate \rangle \triangleq$ boolean-valued $\langle state\ function \rangle \mid Enabled\ \langle action \rangle$

$\langle state\ function \rangle \triangleq$ expression containing constants and variables

Semantics

$$s[\![f]\!] \triangleq f(\forall\, `v' : s[\![v]\!]/v) \qquad \sigma[\![F \wedge G]\!] \triangleq \sigma[\![F]\!] \wedge \sigma[\![G]\!]$$
$$s[\![\mathcal{A}]\!]t \triangleq \mathcal{A}(\forall\, `v' : s[\![v]\!]/v, t[\![v]\!]/v') \qquad \sigma[\![\neg F]\!] \triangleq \neg\sigma[\![F]\!]$$
$$\sigma[\![\exists\, \mathsf{c} : F]\!] \triangleq \exists \mathsf{c} \in \mathsf{Val} : \sigma[\![F]\!]$$
$$\models \mathcal{A} \triangleq \forall s, t \in \mathbf{St} :\models s[\![\mathcal{A}]\!]t \qquad \models F \triangleq \forall \sigma \in \mathbf{St}^\infty :\models \sigma[\![F]\!]$$
$$s[\![Enabled\ \mathcal{A}]\!] \triangleq \exists t \in \mathbf{St} : s[\![\mathcal{A}]\!]t$$
$$\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle [\![\Box F]\!] \triangleq \forall n \in \mathsf{Nat} : \langle\!\langle s_n, s_{n+1}, \ldots \rangle\!\rangle [\![F]\!]$$
$$\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle [\![\mathcal{A}]\!] \triangleq s_0[\![\mathcal{A}]\!]s_1$$

Additional notation

$$f' \triangleq f(\forall\, `v' : v'/v) \qquad \Diamond F \triangleq \neg\Box\neg F$$
$$[\mathcal{A}]_f \triangleq \mathcal{A} \vee (f' = f) \qquad F \rightsquigarrow G \triangleq \Box(F \Rightarrow \Diamond G)$$
$$\langle \mathcal{A} \rangle_f \triangleq \mathcal{A} \wedge (f' \neq f) \qquad \mathrm{WF}_f(\mathcal{A}) \triangleq \Box\Diamond\langle \mathcal{A} \rangle_f \vee \Box\Diamond\neg Enabled\ \langle \mathcal{A} \rangle_f$$
$$Unchanged\ f \triangleq f' = f \qquad \mathrm{SF}_f(\mathcal{A}) \triangleq \Box\Diamond\langle \mathcal{A} \rangle_f \vee \Diamond\Box\neg Enabled\ \langle \mathcal{A} \rangle_f$$

where f is a $\langle state\ function \rangle$; s, s_0, s_1, ... are states
c is a $\langle rigid\ variable \rangle$; σ is a behaviour
$\mathcal{A}$ is an $\langle action \rangle$; $(\forall\, `v' : \ldots/v, \ldots/v')$ denotes substitution
F, G are $\langle formula \rangle$s; for all variables v

Figure 3.1: Simple TLA extended with quantification over rigid variables.

Simple TLA. The logic we use when reasoning in TLP does not yet contain quantification over flexible variables, positioning it between the two.

The syntax and semantics of Simple TLA extended with quantification over rigid variables and with some of the extended notation described above is summarized in figure 3.1, as specified by Lamport [20].

3.3 Proof rules

The TLA proof rules are listed in figure 3.2. Rules STL1–STL6, the Lattice Rule and the basic rules TLA1 and TLA2 constitutean independent axiom system for reasoning about the Simple Logic, which is easily shown to be sound with respect to the semantics. Lamport furthermore claims [20, page 20] that this system is complete relative to the ability to prove all valid action formulas (the logic of which can be translated into pure

The Rules of Simple Temporal Logic

STL1. $\dfrac{F \text{ provable by propositional logic}}{F}$

STL2. $\vdash \Box F \Rightarrow F$

STL3. $\vdash \Box\Box F \equiv \Box F$

STL4. $\dfrac{F \Rightarrow G}{\Box F \Rightarrow \Box G}$

STL5. $\vdash \Box(F \wedge G) \equiv (\Box F) \wedge (\Box G)$

STL6. $\vdash (\Diamond\Box F) \wedge (\Diamond\Box G) \equiv \Diamond\Box(F \wedge G)$

LATTICE. $\succ$ well-founded partial order on nonempty set S

$$\frac{F \wedge (\mathsf{c} \in \mathsf{S}) \Rightarrow (H_{\mathsf{c}} \rightsquigarrow (G \vee \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}}))}{F \Rightarrow ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \rightsquigarrow G)}$$

Rules of Quantification

$F1$. $\vdash F(\mathsf{e}/\mathsf{c}) \Rightarrow \exists\, \mathsf{c} : F$

$F2$. $\dfrac{F \Rightarrow G \qquad \mathsf{c} \text{ does not occur free in } G}{(\exists\, \mathsf{c} : F) \Rightarrow G}$

The Basic Rules of TLA

TLA1. $\vdash \Box P \equiv P \wedge \Box[P \Rightarrow P']_P$

TLA2. $\dfrac{P \wedge [\mathcal{A}]_f \Rightarrow Q \wedge [\mathcal{B}]_g}{\Box P \wedge \Box[\mathcal{A}]_f \Rightarrow \Box Q \wedge \Box[\mathcal{B}]_g}$

Additional Rules

INV1. $\dfrac{I \wedge [\mathcal{N}]_f \Rightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$

INV2. $\vdash \Box I \Rightarrow (\Box[\mathcal{N}]_f \equiv \Box[\mathcal{N} \wedge I \wedge I']_f)$

WF1.
$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \Rightarrow (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \Rightarrow Q' \\ P \Rightarrow \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \Rightarrow (P \rightsquigarrow Q)}$$

WF2.
$$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \Rightarrow \langle \mathcal{M} \rangle_g \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \Rightarrow \mathcal{B} \\ P \wedge \mathit{Enabled}\ \langle \mathcal{M} \rangle_g \Rightarrow \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \Rightarrow \Diamond\Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \Rightarrow \mathrm{WF}_g(\mathcal{M})}$$

SF1.
$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \Rightarrow (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \Rightarrow Q' \\ \Box P \wedge \Box[\mathcal{N}]_f \wedge \Box F \Rightarrow \Diamond \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{SF}_f(\mathcal{A}) \wedge \Box F \Rightarrow (P \rightsquigarrow Q)}$$

SF2.
$$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \Rightarrow \langle M \rangle_g \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \Rightarrow \mathcal{B} \\ P \wedge \mathit{Enabled}\ \langle \mathcal{M} \rangle_g \Rightarrow \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathrm{SF}_f(\mathcal{A}) \wedge \Box F \Rightarrow \Diamond\Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{SF}_f(\mathcal{A}) \wedge \Box F \Rightarrow \mathrm{SF}_g(\mathcal{M})}$$

where F, G, H_{c} are TLA formulas; P, Q, I are predicates; $\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{M}$ are actions; f, g are state functions

Figure 3.2: The axioms and proof rules of Simple TLA without refinement mappings and extended with quantification over rigid variables.

$$
\begin{array}{l}
\textbf{var natural}\ \ x, y = 0\ ; \\
\textbf{do}\ \ \langle\ \mathsf{true} \rightarrow x := x + 1\ \rangle \\
\quad \square \\
\quad\ \langle\ \mathsf{true} \rightarrow y := y + 1\ \rangle \qquad \textbf{od}
\end{array}
$$

Figure 3.3: Program 1 of the Increment Example

$$
\begin{array}{l}
\mathit{Init}_\Phi \ \triangleq\ (x = 0)\ \wedge\ (y = 0) \\
\mathcal{M}_1 \ \triangleq\ (x' = x + 1)\ \wedge\ (y' = y) \qquad \mathcal{M}_2 \ \triangleq\ (y' = y + 1)\ \wedge\ (x' = x) \\
\mathcal{M} \ \triangleq\ \mathcal{M}_1 \vee \mathcal{M}_2 \\
\Phi \ \triangleq\ \mathit{Init}_\Phi \wedge \Box[\mathcal{M}]_{(x,y)} \wedge \mathrm{WF}_{(x,y)}(\mathcal{M}_1) \wedge \mathrm{WF}_{(x,y)}(\mathcal{M}_2)
\end{array}
$$

Figure 3.4: A TLA representation of Program 1

first order predicate logic).

3.4 An example: Increment

The main example that we will be using for describing the TLP system is the Increment Example that Lamport uses in the TLA report, and which is also quite suitable for illustrating the use of TLA in general. We give a brief introduction here; for a full description see the TLA Report.

Program 1 is a simple program that when executed keeps incrementing the two only variables x and y, initially set to 0, nondeterministicly choosing which one to increment. The program is shown in figure 3.3 as specified in some conventional programming language.

The formula Φ shown in figure 3.4 is a TLA representation of Program 1. Init_Φ is the predicate asserting that x and y are both set to 0, representing the initial state of the program execution. The actions $\mathcal{M}_1$ and $\mathcal{M}_2$ represents the two possible steps of incrementing either x or y, leaving the other variable unchanged. $\mathcal{M}$, being the disjunction of the two, is thus the action that nondeterministically increments one of the variables. Φ is a logical formula that represents correct executions of Program 1. The formula is satisfied by behaviours such that Init_Φ is satisfied by the first state and any subsequent pair of states either satisfies $\mathcal{M}$ or leaves the variables unchanged (representing no progress in the program execution). The weak fairness part asserts that only behaviours are accepted in which infinitely many incrementations of x and y are performed, representing the fact that the execution never stops.[1]

A property of Program 1 that may be shown with this TLA representation is for instance the invariance of the predicate T stating that x and y are both natural numbers;

[1] A precise representation of Program 1 should only demand that infinitely many incrementations of *either* x or y are performed; we use the more strict specification as it is the one discussed in the TLA report.

in TLA $\Box((x \in \mathsf{Nat}) \wedge (y \in \mathsf{Nat}))$. The theorem that Φ satisfies this property is just $\Phi \Rightarrow T$, and is easily proved from the lemmas $Init_\Phi \Rightarrow T$ and $T \wedge [\mathcal{M}]_{(x,y)} \Rightarrow T'$ by application of the INV1 rule, as explained in the TLA report.

Program 2 is a refinement of Program 1 introducing two processes that take care of the incrementation of the variables. A semaphore ensures that only one of the processes can at any point increment its variable. This is shown in a conventional language notation in figure 3.5.

Representing Program 2 in TLA requires the addition of a semaphore variable *sem* as well as two 'program counters' pc_1 and pc_2 indicating at any time where the two processes are in their execution of the loops. As values that the program counters may take we use "a", "b", and "g", indicating that the control is at α, β, or γ, respectively. We use strong fairness to specify an assumption that the 'cobegin' construct is fair in the sense that each process will eventually execute its α action (and continue with β and γ) if it is repeatedly (not necessarily continuously) enabled. The full TLA specification is shown in figure 3.6.[2]

We are interested in showing that Program 2 is really a refinement of Program 1, i.e. that any behaviour satisfying Program 2 would also satisfy Program 1. The theorem we thus want to show is in TLA simply $\Psi \Rightarrow \Phi$. The proof is based on the three lemmas

$$\begin{array}{ll} Init_\Psi & \Rightarrow Init_\Phi \\ \Box[\mathcal{N}]_w & \Rightarrow \Box[\mathcal{M}]_{(x,y)} \\ \Psi & \Rightarrow \mathbf{WF}_{(x,y)}\mathcal{M}_1 \wedge \mathbf{WF}_{(x,y)}\mathcal{M}_2 \end{array}$$

The first lemma states that the initial condition of Ψ implies the initial condition of Φ, and the second that each step $\mathcal{N}$ that Ψ may take simulates an $\mathcal{M}$ step or a stuttering step of Φ. These two lemmas are rather trivial, while the last one, stating that Ψ is satisfied only by behaviours which simulate a (weakly) fair execution of $\mathcal{M}_1$ and $\mathcal{M}_2$ is the hard part involving both of the strong fairness rules. To show fairness we first have to prove an invariant implying mutual exclusion between the two processes. The invariant, I_Ψ, is defined as

$$\begin{array}{lllll} \vee & (sem = 1) & \wedge & (pc_1 = pc_2 = \text{"a"}) \\ \vee & (sem = 0) & \wedge & \vee & (pc_1 = \text{"a"}) \wedge (pc_2 \in \{\text{"b"}, \text{"g"}\}) \\ & & & \vee & (pc_2 = \text{"a"}) \wedge (pc_1 \in \{\text{"b"}, \text{"g"}\}) \end{array}$$

The proofs are described informally in the TLA Report. Later we will see how they can be done within TLP, the mechanical verification system described in this thesis.

[2] We adopt the convention that a list bulleted with $\wedge$'s denotes the conjunction of the items, and we use indentation to eliminate parentheses – a style that Lamport advocates [21], and that has also been adopted in the TLP language.

```
var integer  x, y      = 0 ;
    semaphore  sem     = 1 ;
cobegin  loop  α1: ⟨ P(sem) ⟩ ;
               β1: ⟨ x := x + 1 ⟩ ;
               γ1: ⟨ V(sem) ⟩      endloop
  □
         loop  α2: ⟨ P(sem) ⟩ ;
               β2: ⟨ y := y + 1 ⟩ ;
               γ2: ⟨ V(sem) ⟩      endloop
coend
```

Figure 3.5: Program 2 of the Increment Example

$$
\begin{array}{lcl}
Init_\Psi & \triangleq & \land\ (pc_1 = \text{“a”}) \land (pc_2 = \text{“a”}) \\
 & & \land\ (x = 0) \land (y = 0) \\
 & & \land\ sem = 1 \\
\alpha_1 & \triangleq & \land\ (pc_1 = \text{“a”}) \land (0 < sem) \\
 & & \land\ pc_1' = \text{“b”} \\
 & & \land\ sem' = sem - 1 \\
 & & \land\ Unchanged(x,\, y,\, pc_2) \\
\beta_1 & \triangleq & \land\ pc_1 = \text{“b”} \\
 & & \land\ pc_1' = \text{“g”} \\
 & & \land\ x' = x + 1 \\
 & & \land\ Unchanged(x,\, y,\, pc_2) \\
\gamma_1 & \triangleq & \land\ pc_1 = \text{“g”} \\
 & & \land\ pc_1' = \text{“a”} \\
 & & \land\ sem' = sem + 1 \\
 & & \land\ Unchanged(x,\, y,\, pc_2) \\
\mathcal{N}_1 & \triangleq & \alpha_1 \lor \beta_1 \lor \gamma_1
\end{array}
\qquad
\begin{array}{lcl}
\alpha_2 & \triangleq & \land\ (pc_2 = \text{“a”}) \land (0 < sem) \\
 & & \land\ pc_2' = \text{“b”} \\
 & & \land\ sem' = sem - 1 \\
 & & \land\ Unchanged(x,\, y,\, pc_1) \\
\beta_2 & \triangleq & \land\ pc_2 = \text{“b”} \\
 & & \land\ pc_2' = \text{“g”} \\
 & & \land\ y' = y + 1 \\
 & & \land\ Unchanged(x,\, y,\, pc_1) \\
\gamma_2 & \triangleq & \land\ pc_2' = \text{“a”} \\
 & & \land\ pc_2 = \text{“g”} \\
 & & \land\ sem' = sem + 1 \\
 & & \land\ Unchanged(x,\, y,\, pc_1) \\
\mathcal{N}_2 & \triangleq & \alpha_2 \lor \beta_2 \lor \gamma_2
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & \mathcal{N}_1 \lor \mathcal{N}_2 \\
w & \triangleq & (x, y, sem, pc_1, pc_2) \\
\Psi & \triangleq & Init_\Psi \land \Box[\mathcal{N}]_w \land \mathbf{SF}_w\mathcal{N}_1 \land \mathbf{SF}_w\mathcal{N}_2
\end{array}
$$

Figure 3.6: A TLA representation of Program 2

4

Logic engineering

When we began the project of mechanical verification of TLA specifications, we had to make a choice on which tool to use as our major proof engine. We decided to initially try LP, the Larch Prover, developed by John Guttag and Steve Garland at MIT [13, 14]. Speaking for this were a number of reasons:

- LP contains an efficient and well supported rewriting system, with which it seemed to be possible to get much of the reasoning done automatically. Besides the rewrite rules, it is also possible to specify deduction rules, which makes it possible to add the axioms and rules of TLA directly to the prover.

- LP has built-in support for the natural deduction style proofs that we have been doing as hand proofs when working with TLA.

- Unlike in HOL [15, 16] and Isabelle [28, 29], it is in LP possible to add proof rules together with theories freely (the rules do not have to be proved from a semantics), making the outcome less reliable, but the process towards a useful system less painful – we get efficiency and flexibility at the cost of security.

- LP is referred to, by its creators, as a 'proof debugger' rather than just a 'proof checker'. It is well suited for interactive work on proofs, helping in the design. It also supports regressive testing of proofs whenever specifications are changed, facilitating the maintaining of proofs.

- At the time when we started the project, we had direct contact to the developers of LP at Digital Systems Research Center through Jim Saxe and Jim Horning, both working on the Larch project, with the consequence that we could get good help, and have impact on how the prover itself developed.

The last reason might have been the one that influenced us the most, while the others, in retrospect, certainly were very important, in that we were able to do real reasoning about examples such as the Spanning-Tree in just a few weeks, without first having to spend time on things like specifying the semantics and proving the soundness of the proof rules of TLA. We just had to find a reasonable encoding of the logic and specify the proof rules.

On the other hand, LP misses several features that could have been helpful:

- It does not contain a meta-language in which to specify the meaning of certain computable functions (e.g. *Enabled* could be computed internally in HOL).
- It misses features corresponding to the *tacticals* of HOL and Isabelle, making it possible to try different proof strategies whenever problems appear, or the *heuristics* for formulating additional proof goals of the Boyer-Moore prover [5, 6]. Features such as these can make proof scripts more general, automating some of the reasoning.
- Its support of quantification and set theory is minimal.
- It does not contain libraries with support for reasoning about data types, which is the case for HOL and Isabelle.
- It does not guarantee soundness, as mentioned above. The cost of being able to add rules freely has to be paid by rigorous, manual reasoning about the encodings and specifications of proof rules, to ensure soundness and consistency.

As the aim of the project from the beginning was the pragmatic one to investigate the *feasibility* of doing mechanical reasoning about TLA specifications, rather than to build a provably consistent system, we attached more importance to the efficiency of LP and its support of interactive reasoning and regression testing, than the guaranteed consistency of HOL. As we started thinking about the implementation of a translator and a front-end, creating the TLP system, the meta-language and tactical issues seemed to be less important. And again in retrospect we see that it will be possible to add back-ends to the TLP system that supply better handling of the reasoning about data types. In any case, although we started out using the Larch Prover, we at no point felt that this prevented switching to some other proof engine later in the project.

4.1 The Larch Prover

The following is a brief introduction to the Larch Prover, LP, based in part on the LP guide [14].

LP is a theorem prover for a subset of multisorted first-order logic. A *term* of the logic is either a *variable* or an *operator* applied to a sequence of terms known as *arguments*. Variables are assigned a *sort* and operators a *signature* which consist of a *domain* (a list of sorts) and a *range* (a single sort), syntactically connected by the symbol $\rightarrow$.

There is one built-in sort *Bool* representing the set of booleans. Built-in operators include the boolean constants *true* and *false*, the boolean operators not, &, |, $\Rightarrow$, $\Leftrightarrow$ (for negation, conjunction, disjunction, implication and equivalence), the overloaded equality operator $=$, and the overloaded conditional *if*. If i and j are variables of sort *Val*, 1 is an operator with signature $\rightarrow$ *Val*, and $\times$ is an operator with signature *Val*, *Val* $\rightarrow$ *Val*, then e.g. $i \times 1$ and $\mathit{true} \Rightarrow i \times j = j \times i$ are terms of sort *Val* and *Bool* respectively.

The basis for proofs is a logical system of *declared operators*, *equations*, *rewrite rules*, *induction rules*, and *deduction rules*. *Equations* are pairs of terms connected by either of the logical symbols $==$ or $\rightarrow$, the latter indicating a suggested orientation when the equation is ordered into a rewrite rule. *Rewrite rules* are *ordered* equations, sets of which constitute *rewriting systems*. *Induction rules* are rules connected to operator theories, which are not described here; see the LP guide for an explanation. *Deduction rules*,

finally, consist of two lists of equations, the *hypotheses* and the *conclusions*, syntactically expressed as **when** $h_1, \ldots, h_m$ **yield** $c_1, \ldots, c_n$.

Proofs rely on built-in tactics introducing different mechanisms for forward and backward inference. The forward inference methods are:

Normalization: LP uses rewrite rules to manually or automatically *rewrite* equations, rewrite rules and deduction rules of its logical system to normal forms. Informally, a rewriting using the rule $l \rightarrow r$ can take place whenever l matches (by unification) any subterm of an equation or rule, with any variables substituted by new terms, rewriting the subterm to r with the same substitutions.

Application of deduction rules: Likewise, LP uses deduction rules to manually or automatically *deduce* new equations from the equations and rewrite rules of its logical system. An application of the deduction rule **when** $h_1, \ldots, h_m$ **yield** $c_1, \ldots, c_n$ can be performed whenever h_1 to h_m with the same variable substitutions matches equations or rewrite rules of the current logical system, creating the new equations c_1 to c_n with the used substitutions.

Instantiation: Explicit *instantiation* of variables in equations, rewrite rules, and deduction rules creates new instances of equations and rules that may be normalized or applied to other equations and rules.

Critical-pair equations and **completion:** LP can compute *critical-pair equations* of groups of rewrite rules. See the LP guide [14] for an explanation of this featue.

Among the backward inference methods are:

Proofs by normalization: LP also uses rewrite rules to rewrite conjectures. Whenever a conjecture is normalized to *true* (or more precise, to an *identity*), the proof has succeeded and the original conjecture is said to be a theorem.

Proofs by cases: When instructed to do a proof by a list of different *cases*, LP divides the proof into a number of subgoals corresponding to the cases and the fact that these are extensive. As we note when implementing the similar inference method in TLP, this is really the backward inference resemblance of the disjunction elimination rule of natural deduction (see e.g. Prawitz [30]).

Proofs by contradiction: When instructed to do a proof by *contradiction*, LP attempts to prove an inconsistency (i.e. to prove *false*) from assuming the negation of the conjecture. This again resembles the forward inference that could be done with a rule of natural deduction, the *false* rule.

Proofs by implication: In a proof by *implication* LP attempts to prove a conjecture of the form $A \Rightarrow B$ by assuming A and proving B. This resembles the implication introduction rule of natural deduction.

Other backward inferences work on conditionals and conjunctions, and deduction and induction rules.

The mechanism of performing forward inferences automatically is based on two properties that are connected individually to each rule and equation of the logical system,

called *activity* and *immunity* respectively. When a rule is *active*, it will automatically rewrite or be applied to any equation or rule that it matches, and which is not *immune*. LP thus maintains an invariant, that all currently active rules have been applied as far as possible to all currently nonimmune equations or rules. Manual use of forward inferences is indifferent to these properties, and is done whenever requested by using the standard tactics *normalize*, *apply*, etc.

Automatic application of backward inference methods is possible through a user-definable parameter that always specifies a list of methods to use whenever applicable. It is common to let normalization be applied automatically while the other backward inference methods are applied only manually.

A major feature of LP is its ability to automatically *order* equations into rewrite rules, based on different algorithms and *registering hints* on the declared operators. The commonly used methods for having equations ordered ensure that no infinite rewriting sequences can occur (but, by consequence, are not always able to order all equations of the logical system).

Reasoning in LP can be done either in an interactive session or by executing a *script*. Sorts, variables and operators are declared by the **declare** command. Axioms and proof rules are specified by equations and rules, using **assert**. A proof is initiated by the **prove** command followed by an equation (or rule) to be proved, the *conjecture* of the proof. LP always maintains the invariant mentioned above, namely that all active facts have been applied to all nonimmune, and if so instructed, tries to order all existing equations into rewrite rules. When starting a proof, the currently specified backward inference methods are tried on the conjecture. When this has been done, if the conjecture has not been proved, LP waits for instructions on which backward and forward methods to use for resuming the proof. At any point, it is possible to get the elements of the current logical system and the status of the proof displayed in different ways, using the **display** command.

When using proof scripts, it is possible to insert into the script special symbols that indicate what is supposed to happen after certain instructions. Whenever LP meets a line beginning with the symbol [] it expects the current top-most conjecture (there may be several levels) to be proved. If this is not so, LP will stop running the script and complain. Likewise, the symbol $\langle\rangle$ on the beginning of a line instructs LP to expect one or more subgoals to have been created by some backward inference method. Extensive use of these symbols facilitates maintaining of proofs whenever specifications of the logical system change, as it is possible to detect where the problems arise. This is a feature that is central for the TLP system, as we will later show.

A feature that as well will be used extensively in the TLP system is the ability to save images of any logical system (and any unfinished proofs), called *freezing*. When reasoning depends on an earlier frozen system, this system can be *thawed* and reasoning resumed as from the point where the earlier system was frozen.

4.2 Formalizing TLA for a theorem prover

It is not a trivial to task to find a proper encoding of the TLA logic for use in a theorem prover, and there probably isn't any such thing as the *right* formalism. In this section we will first present some different formalizations as done by others, before turning to a

description and discussion of the one we have chosen to use.

4.2.1 The technique of von Wright and Långbacka

Joakim von Wright and Thomas Långbacka (W&L) in 1991 and 1992 described a formalization of TLA for reasoning with the HOL theorem prover [34, 35]. Their implementation differ from the general idea in TLA in that variables have types, this being dictated to some extent by the HOL system. Otherwise it is a very straightforward implementation of the semantics as specified by Lamport. States are represented as tuples of values, each representing a variable of the specification that we want to reason about, with a last element representing "the rest of the universe". Predicates and actions are formalized as lambda expressions taking one and two states respectively and returning a boolean. To be able to specify proof rules containing predicates and actions, W&L define boolean connectives for each sort, *pand*, *pnot*, *aand*, *anot*, and so forth, which are lifted to the ordinary boolean operators of HOL (e.g. p *pand* q is lifted to $\lambda s.\ p\ s \wedge q\ s$ where s represents a state). Likewise, they define validity functions *pvalid* and *avalid*, e.g. *pvalid* p being defined as $\forall s.\ p\ s$.

To reason about temporal logic, a behaviour is represented as a function from the natural numbers to states. Temporal formulas can then be represented as lambda expressions taking a behaviour and returning a boolean. The box operator is defined so that *box* f t equals $\forall i.\ f(\lambda n.\ t(i+n))$. As for predicates and actions, W&L define boolean connectives *tand* and *tnot* and a validity function *tvalid*. It is essential to be able to lift, or coerce, predicates and actions into temporal formulas, for which four functions, *liftpa*, *liftpa′*, *liftpt*, and *liftat* are defined. E.g. *liftpt* p t is defined as $p\ (t\ 0)$. *liftpa′* actually implements the prime operator for predicates, as it lifts a predicate into its primed version as an action.

With this formalism, Program 1 of the Increment Example may be specified in HOL as

$$
\begin{array}{ll}
\vdash_{def} \mathit{InitPhi} & = \lambda\,(x,y,z).\ (x=0) \wedge (y=0) \\
\vdash_{def} \mathcal{M}1 & = \lambda\,(x,y,z)\,(x',y',z').\ (x'=x+1) \wedge (y'=y) \\
\vdash_{def} \mathcal{M}2 & = \lambda\,(x,y,z)\,(x',y',z').\ (y'=y+1) \wedge (x'=x) \\
\vdash_{def} \mathcal{M} & = (\mathcal{M}1\ \mathit{aor}\ \mathcal{M}2) \\
\vdash_{def} v & = \lambda\,(x,y,z).\ (x,y) \\
\vdash_{def} \mathit{Phi} & = (\mathit{liftpt}\ \mathit{InitPhi})\ \mathit{tand} \\
 & \quad\ (\mathit{box}\ (\mathit{liftat}\ (\mathit{square}\ M\ v)))\ \mathit{tand} \\
 & \quad\ (\mathit{WF}\ \mathcal{M}1\ v)\ \mathit{tand}\ (\mathit{WF}\ \mathcal{M}2\ v)
\end{array}
$$

and the INV1 rule as

$$
\begin{array}{l}
\vdash \mathit{avalid}(((\mathit{liftpa}\ I)\ \mathit{aand}\ (\mathit{square}\ N\ f))\ \mathit{aimp}\ (\mathit{liftpa'}\ I)) \Rightarrow \\
\quad \mathit{tvalid}\ (((\mathit{liftpt}\ I)\ \mathit{tand}\ (\mathit{box}\ (\mathit{square}\ N\ f)))\ \mathit{timp}\ (\mathit{box}\ (\mathit{liftpt}\ I)))
\end{array}
$$

The TLA rules all have to be proved from the semantics (or from already proved rules), to be used in HOL. The security achieved by this is one of the features that are emphasized by the HOL creators. Once the rules are proved, one can use them to reason about algorithms, e.g. showing that Program 2 of the Increment Example implements Program 1. W&L had done part of this proof when presenting their paper.

W&L admit that their way of representing rules may look ugly with its use of different connectives and lifting functions (remember that the WF2 rule, for instance, is consider-

ably more complicated than the INV1 rule shown above). They also state, however, that this may be seen as an advantage, in that no confusion arises when the same term is used both as a predicate and as a temporal formula. We will see later a quite different way to deal with these problems in the TLP system.

It is worth noting that W&L in one of their conclusions state that "[the] formalisation does not permit meta-level reasoning about the TLA logic. It is possible to formalise TLA in HOL in a way which would permit such reasoning, but such a formalisation would be much more difficult to use in practice." [35].

4.2.2 The technique of Chou

Ching-Tsun Chou has in two papers in 1993 [7, 8] presented a similar formalization of TLA in HOL. Chou represents states and behaviours in the same way as W&L, uses lifting to go from connectives of predicates, actions, and temporal properties to the logic of HOL, and uses *projections* from state pairs to states, and from behaviours to state pairs or states, combined with an *inverse image operator*, to implement coercions, so that predicates, for instance, may be interpreted as temporal formulas. A difference is that W&L use *curried* lambda expressions for representing actions, which Chou does not. This may not seem very important, but making actions uncurried gives the formalism a homogeneity, in that predicates, actions, and temporal properties all can be represented by the same *general predicate* type $* \rightarrow bool$ (where $*$ is a *type variable*). This, in turn, gives a uniform treatment of the logical connectives at the different levels, and Chou consequently only has to declare one set of connectives with corresponding lifting functions.

Chou also himself accentuates [8] that W&L use a Hilbert style calculus (similar to the one used by Lamport in the TLA Report) to express the validity of single predicates, i.e. expressing the validity of a predicate as

$$\models P = \forall s.\ P(s)$$

while Chou himself uses a sequent formulation, expressing validity with respect to assumptions, i.e.

$$[P_1, P_2, \ldots, P_n] \models Q = \forall s.\ P_1(s) \wedge P_2(s) \wedge \ldots \wedge P_n(s) \Rightarrow Q(s)$$

According to Chou, the latter formalization permits a more straightforward lifting to the HOL tactics, thus saving unnecessary work.

4.2.3 The initial technique of Lamport

The first attempt that Leslie Lamport made in 1990 on mechanizing the reasoning in TLA using the Larch Prover is somewhat different from the formalisms of W&L and of Chou. Lamport, in accordance with his views in TLA, makes variables type-free, declaring them all to be of the general sort *Any*. Rigid variables are represented by LP variables, while flexible variables are represented by LP operators. Believing the sort system of LP to be insufficient for his purposes, Lamport in fact uses the single sort *Any* for everything not being a boolean or a function producing a boolean, and instead implements his own functions for checking whether an expression is a constant, predicate, an action, and so forth. This is only possible in LP as it is possible to declare new constants and variables

and *assert* that they are constants and variables. Something similar may well have been achieved by using separate sorts for the different expressions, but the use of only one general sort has advantages, e.g. by permitting the use of only a single set of connectives ($\wedge$, $==$, $\neg$, etc.) for constants, predicates, actions, and temporal properties. In Lamport's formalization, these are lifted into LP's boolean operators only when used on constants, i.e. after the expressions have been applied to a state (state pair or behaviour, respectively). Lamport also does not make any concrete representation of states, but simply lets these be unspecified variables of sort *Any*.

Application of a predicate to a state is done by the application operator '.' of sort $Any, Any \rightarrow Any$, so that the semantic value $s[\![P]\!]$ of the predicate P in state s is written as

$$P \,.\, s$$

Application is defined to distribute over the the connectives, by rewrite rules:

$$\begin{aligned} (u \wedge v) \,.\, s &\rightarrow (u \,.\, s) \wedge (v \,.\, s) \\ (u == v) \,.\, s &\rightarrow (u \,.\, s) == (v \,.\, s) \\ \neg(u) \,.\, s &\rightarrow \neg(u \,.\, s) \end{aligned}$$

Lamport, like W&L, uses currying for representing the semantic value of an action, so that $s[\![\mathcal{A}]\!]t$ is represented as

$$\mathcal{A} \,.\, s \,.\, t$$

('.' associates to the left). He furthermore extends this scheme to include temporal properties, in that a behaviour is represented by its first two states and 'the rest', so that $\sigma[\![F]\!]$ can be represented as

$$F \,.\, s \,.\, t \,.\, r$$

where s is the first state of σ, t the second, and r the behaviour containing the rest.

We now define two functions of sort $Any \rightarrow Bool$, *IsConstant* and *IsBool*, that tell whether a given expression is a general or boolean constant respectively. *IsConstant* and *IsBool* are defined structurally in a straightforward way, by rules such as

TypeCheck.6:

$$IsBool(u == v) \;\rightarrow\; IsConstant(u) \;\&\; IsConstant(v)$$

'&' being LP's logical *and*. The *bottoms* of the definitions are constants and variables declared and asserted to be so by the user writing a specification (i.e. by asserting the fact $IsConstant(a)$ for a constant a and $IsConstant(x \,.\, s)$ for a variable x), and the non-lifted boolean constant *vtrue*.

From *IsBool* we can compute functions *IsPredicate*, *IsAction*, and *IsTemporal* of sort $Any \rightarrow Bool$, that tell us if an expression is of the respective sort. As LP does not support general quantification[1], we provide deduction rules for reasoning about these functions, rather than give definitions. For *IsPredicate* we thus have the rule

[1]The forthcoming version of LP will probably contain quantification, but the versions we have been using do not.

ProveTLADecls.1:

when (**forall** s) $IsBool(u\,.\,s)$
yield $IsPredicate(u)$

and the reverse

UseTLADecls.1:

when $IsPredicate(u)$
yield $IsBool(u\,.\,s)$

(in the latter case, s is a free variable, while in the former it is explicitly bound by the universal quantifier that is allowed in LP for use in deduction rules only).

Lifting is implemented by the function *MakeBool* that of course again is of sort $Any \rightarrow Bool$. *MakeBool* is defined by rules such as

MakeBool.6:

$$MakeBool(u \wedge v) \;\rightarrow\; MakeBool(u)\;\&\;MakeBool(v)$$

where it is worth noting that there appear no states or behaviours. As stated above, the lifting function works on all levels, as u and v can be instantiated with e.g. $P\,.\,s$ as well as $F\,.\,s\,.\,t\,.\,r$.

This finally lets us express the functions representing *validity* for the different sorts. These are called *PValid*, *AValid*, and *TValid*, and are all expressed by deduction rules, so for example the *TValid* function by the two rules

ProveValid.3:

when (**forall** s, t, r) $IsTemporal(u)\;\&\;MakeBool(u\,.\,s\,.\,t\,.\,r)$
yield $TValid(u)$

and

UseValid.3:

when $TValid(u)$
yield $IsTemporal(u)\;\&\;MakeBool(u\,.\,s\,.\,t\,.\,r)$

A strong feature of Lamport's formalization is that no coercions are needed. Any constant boolean expression may be used as a predicate, any predicate as an action, and so forth, as we may just disregard the application to a state whenever we have reached a constant expression, using the rule

ApplyDecls:

when $IsConstant(u)$
yield $(u\,.\,s) \rightarrow u$

What may not seem so nice is that the representation of behaviours does not permit the semantics of the box operator to be specified. Instead, Lamport has to *assert* the rules of simple temporal logic and of TLA, so that we can reason about temporal properties. As these rules, according to Lamport, constitute a relatively complete proof system, the strength and flexibility of the formalization should be the same. The security of the two HOL implementations, where all rules have to be proved from the semantics, is not achieved, but if we don't allow asserting any new rules other than the basic ones of our formalization, then this security hole can be considered a static assumption, that we only have to deal with once.

Program 1 of the Increment Example may be specified in Lamport's formalism as

$$
\begin{array}{lll}
\textbf{assert} & \mathit{IsConstant}(x\,.\,s) & \\
\textbf{assert} & \mathit{IsConstant}(y\,.\,s) & \\
\textbf{assert} & \mathit{InitPhi} & == (x = 0) \wedge (y = 0) \\
\textbf{assert} & \mathcal{M}1 & == (\mathit{Prime}(x) = x + 1) \wedge (\mathit{Prime}(y) = y) \\
\textbf{assert} & \mathcal{M}2 & == (\mathit{Prime}(y) = y + 1) \wedge (\mathit{Prime}(x) = x) \\
\textbf{assert} & \mathcal{M} & == \mathcal{M}1 \vee \mathcal{M}2 \\
\textbf{assert} & v & == (x * y) \\
\textbf{assert} & \mathit{Phi} & == \mathit{InitPhi} \wedge \mathit{BoxAct}(\mathcal{M}, v) \wedge \mathit{WF}(v, \mathcal{M}1) \wedge \mathit{WF}(v, \mathcal{M}2)
\end{array}
$$

and the INV1 rule as

$$
\begin{array}{ll}
\textbf{assert} & \\
\quad \textbf{when} & \mathit{AValid}((I \wedge (\mathcal{N} \vee \mathit{Unchanged}(f))) \Longrightarrow \mathit{Prime}(I)) \\
\quad \textbf{yield} & \mathit{TValid}((I \wedge \mathit{BoxAct}(\mathcal{N}, f)) \Longrightarrow \mathit{Box}(I))
\end{array}
$$

4.3 Our formalization

This section describes the basic techniques for encoding TLA in LP, that were developed in cooperation with Peter Grønning and Leslie Lamport during 1991. In our joint work, Grønning and the author experimented with different methods of encoding TLA, so that the managing of large proofs would become feasible. The results of our first attempts were discomforting, in that it took us several weeks to finish some logically simple proofs of refinement of a very small example. The work in getting rules applied in the right way and lifting statements about actions into temporal formulas seemed to take much more time than the real work of writing the proof. During discussions with Lamport, the idea came up to use different encodings for different kinds of reasoning, and creating some sort of front-end for doing the tedious work that didn't really seem to be part of the theorem proving. Constituting one of the main differences from the initial technique of Lamport and the HOL encodings described earlier, this was an idea that Grønning and the author found appealing. In the following experiments we found that the special-purpose encodings made our proofs much simpler. The first prototype translator that the author wrote for combining the reasoning of the different environments, and so that we wouldn't have to manually write different encodings of the same specification, made the reasoning process itself more manageable, while adding the flexibility of a high-level language.

4.3.1 Basic techniques

The encoding of TLA for the Larch Prover that we used initially was based on the described technique of Lamport. This has advantages of reflecting TLA correctly, and being homogeneous and easy to use; you don't have to deal with different levels of connectives and lifting functions when reasoning. Still, experiments showed that much time in proofs were spent on things that were not really related to the theorems themselves.

As an example, a very simple proof in LP based on our initial encoding is shown in figure 4.1. We prove validity of the theorem $Init0 \Longrightarrow T0$. The predicates *Init0* and *T0*

```
set name Theorem_tc_1
prove PValid(Init0 ⟹ T0)
  prove IsPredicate(Init0 ⟹ T0)
    instantiate
      u by Init0 ⟹ T0
      in ProveTLADecls.1
      ..
    prove IsBool((Init0 ⟹ T0) . s)
      make inactive ProveTLADecls
      make active TypeCheck UseTLADecls
      [] conjecture
    [] conjecture
  prove MakeBool((Init0 ⟹ T0) . s)
    make active MakeBool Apply ApplyDecls
    make active Init0 T0
    resume by ⇒
    ⟨⟩ 1 subgoal for proof of ⇒
      [] ⇒ subgoal
    [] conjecture
  instantiate
    u by Init0 ⟹ T0
    in ProveValid.1
    ..
  [] conjecture
```

Figure 4.1: A simple proof in our initial encoding.

are defined elsewhere; *Init0* is just $pc = \mathsf{a}$ and *T0* the disjunction $pc = \mathsf{a} \vee pc = \mathsf{b}$. The real reasoning appears in the lines

```
make active Init0 T0
resume by ⇒
```

where we, basically, ask that the goal be rewritten using the definitions of *Init0* and *T0*, and proved by implication introduction (see section 4.3.2.) The rest of the proof is used to show that the goal is a predicate by applying it to a state and showing the result to be a boolean, and making sure that the right lifting and validity functions get applied.

Not only is the proof longer than what one would think necessary, considering the logical implications of showing $Init0 \Rightarrow T0$, it also involves a considerable amount of work to be done inside LP, that takes up a lot of time during verification.

Furthermore, the example illustrates that the parts of the proofs dealing with other things than the real reasoning are severely distracting; it is very difficult to get an idea of what is being proved, and how the proof works. This problem gets worse as the complexity of proofs gets bigger.

Part of the solution to this problem is based on the observation that not all reasoning has to be done inside LP. It is a trivial task to analyze a TLA expression syntactically, determining whether it is e.g. a well-formed predicate. If we can thus do the sort-checking part elsewhere, determining $IsPredicate(Init0 \Longrightarrow T0)$, we might just assert its validity in LP, whereby we would only have to show the more essential part, $MakeBool((Init0 \Longrightarrow T0)\,.\,s)$, that has to do with the validity of the theorem.[2]

The proof may be made simpler yet, however. When reasoning about the validity of a predicate or action, we always go through the same steps: assuming a universally quantified state or state pair to which we apply the formula, lifting the boolean operators, and finally instantiating the ProveValid rule. These steps may all be removed from the LP session.

Our improvement is based on the fact that there are two different kinds of reasoning involved in a TLA proof. *Action reasoning* is reasoning not involving any temporal operators, such as the reasoning done in any invariance proof, showing that the initial conditions satisfy the invariant and that the possible actions maintain it. *Temporal reasoning* may involve temporal operators, exemplified by the application of the invariance rule and the following reasoning to conclude that the specification implies that the invariant is always true. When doing a fairness proof, using the WF2 rule, the first three steps obviously contain solely action reasoning, while the fourth, its goal being a temporal formula, and the conclusion are the only steps involving temporal reasoning. Generally, it is clearly the case that action reasoning constitutes the longest and most difficult parts of a proof. It therefore makes sense to give this some special attention, possibly using a separate encoding. In designing the TLP system, we found the resulting simplification of action reasoning to be worth the inconvenience of having two different encodings.

The idea is to represent actions and predicates in the action reasoning encoding in a context where they are already applied to a (free) state pair, say (s, t). But instead of writing $x\,.\,s$ and $x\,.\,t$ for the variable x and the primed x', we might as well just use x and x' as constant names referring to the values $x\,.\,s$ and $x\,.\,t$. Thus we get rid of the state notation altogether. We don't need to lift the boolean operators, as we are already dealing with boolean values, and can thus use LP's boolean operators directly. The prime operator may no longer be represented as an operator in LP, as its meaning in the new encoding is purely syntactical. Instead, we have to compute the value of each primed predicate or state function in our encoding, so that each predicate or state function will be represented with two definitions, one for the unprimed version and one for the primed.

With an encoding like this, the proof of $Init0 \Rightarrow T0$ is getting closer to what we are used to in manual reasoning, as shown in figure 4.2. Note that the $\Rightarrow$ symbol in this

[2]This is possible in LP as we can freely *assert* any facts we need, making it easy to combine LP with other tools. In HOL or Isabelle something similar should also be possible, writing a sort-checking function in the metalanguage as part of the semantic definition.

proof represents ordinary LP implication (in ASCII written =>), while $\Longrightarrow$ in figure 4.1 represents the unlifted implication operator of TLA (in ASCII ==>).

set name Theorem_tc_1

prove *Init0* $\Rightarrow$ *T0*
  make active *Init0 T0*
  resume by $\Rightarrow$
  $\langle\,\rangle$ 1 subgoal for proof of $\Rightarrow$
    [ ] $\Rightarrow$ subgoal
  [ ] conjecture

Figure 4.2: The same proof in an action reasoning encoding.

Clearly, the described encoding doesn't allow temporal reasoning. For temporal reasoning we still need an encoding in which behaviours have a representation, and the prime is a semantic operator. As we don't have to do any pure action reasoning inside the temporal environment, however, we no longer need to represent single states, and only have to deal with a single validity function, namely that of temporal formulas.

The problem of having two different encodings is solved by using a preprocessing tool, a *translator* that from a specification written in a general language generates encodings for both environments. The translator also may perform the tasks described above: sort-checking the specifications and proofs, and for the action reasoning environment computing the primed versions of predicates and state functions. Finally, being able to sort-check formulas, it is also able to split up proofs in parts to be verified in the action environment and parts to be verified in the temporal environment. That means that we may provide proofs that contain steps that should be proved by action reasoning, and steps to be proved by temporal reasoning; the translator will then produce proof scripts for the action parts in the action reasoning encoding, while *asserting* the validity of the goals of these steps together with proof scripts for the temporal steps, all in the temporal reasoning encoding.

With the translator we are also able to write the specifications and proofs in a language using a higher level of abstraction, hiding the technicalities of LP and concentrating on the logic. The language that we use for this is described in chapter 5. The following sections contain a more extensive description of the encodings and the basis for our reasoning.

4.3.2 Action reasoning

The encoding for action reasoning is straightforward. We represent rigid variables by LP variables and flexible variables by constants (LP operators), using two distinct constants for representing the unprimed and primed version of each variable. Variables can be of the general type *Val* or of the built-in boolean type *Bool*. States are not represented, but flexible variables represent their value in a universally quantified state. Unprimed and primed versions of state functions and predicates are computed and encoded separately. The boolean connectives are represented directly by the ones of LP. Tuples are represented as ordered pairs, using the pairing operator $*$.

The non-temporal definitions of Program 1 of the Increment Example may be specified and encoded for action reasoning in LP as

$$\begin{array}{lll}
\textbf{assert} & \mathit{InitPhi} & == (x = 0)\ \&\ (y = 0) \\
\textbf{assert} & \mathit{InitPhi}' & == (x' = 0)\ \&\ (y' = 0) \\
\textbf{assert} & \mathcal{M}1 & == (x' = x + 1)\ \&\ (y' = y) \\
\textbf{assert} & \mathcal{M}2 & == (y' = y + 1)\ \&\ (x' = x) \\
\textbf{assert} & \mathcal{M} & == \mathcal{M}1 \mid \mathcal{M}2 \\
\textbf{assert} & v & == (x * y) \\
\textbf{assert} & v' & == (x' * y')
\end{array}$$

where $=$, $\&$, and $\mid$ are the ordinary LP operators for equality, conjunction and disjunction.

The expression stating that a formula is valid is represented by the formula itself. Proving validity of a formula is thus just proving the formula as a conjecture in LP, using the built-in natural deduction techniques described in section 4.1. This corresponds directly to performing a natural deduction proof using the traditional Gentzen-style inference rules shown in figure 4.3. LP, however, performs all the inferences based on the rules $\wedge I$, $\wedge E$, $\vee I$, and $\Rightarrow E$ by (automatic) normalization using built-in rewrite rules, while $\vee E$, $\Rightarrow I$, and *False* are implemented by the backward inference methods 'proof by cases', 'proof by implication', and 'proof by contradiction' respectively. The normalization technique serves to simplify proofs a lot. In the simple proof shown in figure 4.2 on page 42, we thus just have to explicitly apply the implication introduction rule ($\Rightarrow I$), by saying '**resume by** $\Rightarrow$', while the application of the disjunction introduction rule ($\vee I$) to show $pc = \mathsf{a} \mid pc = \mathsf{b}$ from $pc = \mathsf{a}$ is implicit.

Much of the work we do in the action reasoning environment tends to be reasoning about datatypes. Part of this is trivial exercises such as proving that x' is not equal to x when x is a natural number and x' equals $x + 1$. Within the action reasoning encoding this is just applying rules such as

$$\frac{x \in \mathrm{Nat}}{x \neq x + 1}$$

to the assumptions (here $x \in \mathrm{Nat}$ and $x' = x + 1$). Indeed, it would be possible to use general libraries (not specific to TLA nor our encodings) describing different datatypes as the basis of this kind of reasoning in TLP.

4.3.3 Temporal reasoning

Also in the temporal encoding we represent rigid variables by LP variables and flexible variables by LP constants. Variables always have the type *Any*. The boolean connectives and the equality operator of TLA are represented by operators ($\wedge$, $==$, $\neg$, etc.) of sort $\mathit{Any}, \mathit{Any} \rightarrow \mathit{Any}$ or $\mathit{Any} \rightarrow \mathit{Any}$. The prime and box operators are just operators of sort $\mathit{Any} \rightarrow \mathit{Any}$. We don't represent single states, as we are only interested in temporal aspects; we may thus regard all formulas as temporal, using an implicit coercion on predicates and actions. We use the sort *Behaviour* to represent behaviours and by $\sigma \models F$ denote the fact that the formula F is satisfied by the behaviour σ. Program 1 of the

$$\wedge I: \quad \frac{A \quad B}{A \wedge B}$$

$$\wedge E: \quad \frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

$$\vee I: \quad \frac{A}{A \vee B} \qquad \frac{A}{A \vee B}$$

$$\vee E: \quad \frac{A \vee B \quad \begin{matrix}(A)\\ C\end{matrix} \quad \begin{matrix}(B)\\ C\end{matrix}}{C}$$

$$\Rightarrow I: \quad \frac{\begin{matrix}(A)\\ B\end{matrix}}{A \Rightarrow B}$$

$$\Rightarrow E: \quad \frac{A \quad A \Rightarrow B}{B}$$

$$\mathit{False}: \quad \frac{\begin{matrix}(\neg A)\\ \text{false}\end{matrix}}{A}$$

Figure 4.3: Inference rules of Gentzen-style natural deduction (without quantification).

Increment Example is encoded for temporal reasoning as

$$\begin{array}{lll}
\textbf{assert} & \mathit{InitPhi} & == (x = 0) \wedge (y = 0) \\
\textbf{assert} & \mathcal{M}1 & == (\mathit{Prime}(x) = x + 1) \wedge (\mathit{Prime}(y) = y) \\
\textbf{assert} & \mathcal{M}2 & == (\mathit{Prime}(y) = y + 1) \wedge (\mathit{Prime}(x) = x) \\
\textbf{assert} & \mathcal{M} & == \mathcal{M}1 \vee \mathcal{M}2 \\
\textbf{assert} & v & == (x * y) \\
\textbf{assert} & \mathit{Phi} & == \mathit{InitPhi} \wedge \mathit{Box}(\mathit{BAct}(\mathcal{M}, v)) \wedge \mathit{WF}(v, \mathcal{M}1) \wedge \mathit{WF}(v, \mathcal{M}2)
\end{array}$$

and the INV1 rule as

$$\begin{array}{ll}
\textbf{assert} & \\
\quad \textbf{when (forall } \sigma) & \sigma \models ((I \wedge \mathit{BAct}(\mathcal{N}, f)) \Longrightarrow \mathit{Prime}(I)) \\
\quad \textbf{yield} & \sigma \models ((I \wedge \mathit{Box}(\mathit{BAct}(\mathcal{N}, f))) \Longrightarrow \mathit{Box}(I))
\end{array}$$

where we may use rewrite rules for rewriting $\mathit{BAct}(\mathcal{N}, f)$ to $\mathcal{N} \vee \mathit{Unchanged}(f)$ and $\mathit{Unchanged}(f)$ to $f = \mathit{Prime}(f)$.

The validity of formula F may now be represented by $\forall\sigma.\ \sigma \models F$, or in LP as just $\sigma \models F$, universal quantification over the free variable σ being implicit. We don't specify any

semantics of operators such as *Prime* and *Box*, but assert proof rules for reasoning about them. When reasoning about temporal properties, we seldom have to use much ordinary predicate logic or to apply the definitions of *BAct* and *Unchanged*, as all reasoning about actions can be done in the action encoding. Only when we want to combine different temporal lemmas do we need to lift the operators such as $\wedge$ and $=$, for which we use rules such as

$$\sigma \models F \wedge G \quad \rightarrow \quad (\sigma \models F) \;\&\; (\sigma \models G)$$

These normalization rules are in fact always 'active', so that expressions appear in their lifted form whenever we need to reason about them.

Applying a TLA rule is usually done by instantiating the variables of the premises so that they match the lemmas that the rule should be applied to. The premises of the instantiated rule are then automatically reduced to identities, producing the instantiated consequence as a new equation.

4.3.4 Quantification, functions, and sets

To make our encoding powerful enough for reasoning about more interesting examples, we need a representation of quantification over rigid variables. Quantification over flexible variables has not been considered in our work so far.

The versions of LP that we have been working with (releases 2.2–2.4) do not support quantification. That means that we cannot translate quantification over rigid variables directly, as would otherwise be possible in the action reasoning environment. Fortunately it is trivial, although less practical, to encode quantification separately and allow reasoning by inference rules.

We have chosen to represent quantified formulas in a variable-free fashion by introducing the two operators $\forall$ and $\exists$ (in LP written as *for_all* and *exists*), that take a boolean-valued function as an argument. We thus encode the existentially quantified TLA formula

$$\exists\, \mathsf{c} : x = \mathsf{c} \wedge x' = \mathsf{c} + 1$$

as just

$$\exists(f)$$

with the added definition

$$\textbf{assert} \quad f(\mathsf{c}) == x = \mathsf{c} \;\&\; x' = \mathsf{c} + 1$$

For flexibility, we add an encoding for *bounded* quantification, so that for instance the formula

$$\forall\, \mathsf{c} \in S : \mathsf{c} \leq x$$

may be encoded as

$$\forall(S, g)$$

with the definition

$$\textbf{assert} \quad g(\mathsf{c}) == \mathsf{c} \leq x$$

Reasoning about quantified formulas is based on the traditional Gentzen-style introduction and elimination rules, with one set of rules for each of the four quantification

$$\forall I: \quad \frac{f(\mathsf{x})}{\forall(f)} \qquad \frac{\mathsf{x} \in S \Rightarrow f(\mathsf{x})}{\forall(S,f)}$$

$$\forall E: \quad \frac{\forall(f)}{f(x)} \qquad \frac{\forall(S,f) \quad x \in S}{f(x)}$$

$$\exists I: \quad \frac{f(x)}{\exists(f)} \qquad \frac{f(x) \quad x \in S}{\exists(S,f)}$$

$$\exists E: \quad \frac{\exists(f) \quad f(\mathsf{x}) \Rightarrow B}{B} \qquad \frac{\exists(S,f) \quad (\mathsf{x} \in S \wedge f(\mathsf{x})) \Rightarrow B}{B}$$

In the rules $\forall I$ and $\exists E$, x should be a free variable that can not appear in B, f, or in any assumption on which $f(\mathsf{x})$ depends (in LP indicated by writing (**forall** x)), while in $\forall E$ and $\exists I$, x is a variable that may be instantiated by any value.

Figure 4.4: Inference rules for reasoning about quantification.

types, unbounded and bounded existential, and unbounded and bounded universal quantification. The rules that we need for reasoning in the action reasoning encoding are shown in figure 4.4. As none of these rules are built-in as inference methods in LP, we have to add them as ordinary deduction rules, which are applied (or instantiated) manually at each step. As an example of a proof using these rules (as well as some of the other natural deduction rules), we take the one given by Prawitz [30], showing

$$((\forall \mathsf{x} : \exists \mathsf{y} : P(\mathsf{x},\mathsf{y})) \wedge (\forall \mathsf{x} : \forall \mathsf{y} : P(\mathsf{x},\mathsf{y}) \Rightarrow P(\mathsf{y},\mathsf{x}))) \Rightarrow (\forall \mathsf{x} : \exists \mathsf{y} : P(\mathsf{x},\mathsf{y}) \wedge P(\mathsf{y},\mathsf{x}))$$

Figure 4.5 illustrates the deduction that is done in this proof. The LP encoding of the proof follows the structure almost exactly by applying a rule to one or two sub-conjectures for each horizontal line in the diagram, the exceptions being the implication and conjunction eliminations which are handled by automatic normalization, and the two implication introductions which are done using the 'proof by implication' method. In figure 6.7 on page 77 we show how the proof may be written in the higher-level TLP language.

The rules that we need for the temporal reasoning encoding are similar, although referring to the validity with respect to a behaviour, so that e.g. the introduction rule for unbounded universal quantification becomes

$$\frac{\sigma \models f(\mathsf{x})}{\sigma \models \forall(f)}$$

Fortunately, experience tells us that it is seldom (if at all) necessary to perform reasoning about quantified expressions in the temporal encoding.

We represent functions (i.e. array values) by lambda expressions, and chose the same variable-free encoding as used for quantified expressions. This means that the function

$$f(\mathsf{c}) = \mathsf{c} + 1$$

$$\dfrac{\dfrac{\dfrac{(\forall(p11))}{\exists(p12(\mathsf{a}))}\forall E \quad \dfrac{\dfrac{\dfrac{\dfrac{(\forall(p21))}{\forall(p22(\mathsf{a}))}\forall E}{P(\mathsf{a},\mathsf{b}) \Rightarrow P(\mathsf{b},\mathsf{a})}\forall E \quad (P(\mathsf{a},\mathsf{b}))}{P(\mathsf{b},\mathsf{a})}\Rightarrow E \quad (P(\mathsf{a},\mathsf{b}))}{P(\mathsf{a},\mathsf{b}) \;\&\; P(\mathsf{a},\mathsf{b})}\wedge E}{\exists(p32(\mathsf{a}))}\exists I}{P(\mathsf{a},\mathsf{b}) \Rightarrow \exists(p32(\mathsf{a}))}\Rightarrow I}{\exists(p32(\mathsf{a}))}\exists E}{\forall(p31)}\forall I}{(\forall(p11) \wedge \forall(p21)) \Rightarrow \forall(p31)}\Rightarrow I$$

Definitions:

$$\begin{aligned}
p11(\mathsf{x}) &== \exists(p12(\mathsf{x})) \\
p12(\mathsf{x})(\mathsf{y}) &== P(\mathsf{x},\mathsf{y}) \\
p21(\mathsf{x}) &== \forall(p22(\mathsf{x})) \\
p22(\mathsf{x})(\mathsf{y}) &== P(\mathsf{x},\mathsf{y}) \Rightarrow P(\mathsf{y},\mathsf{x}) \\
p31(\mathsf{x}) &== \exists(p32(\mathsf{x})) \\
p32(\mathsf{x},\mathsf{y}) &== P(\mathsf{x},\mathsf{y}) \;\&\; P(\mathsf{y},\mathsf{x})
\end{aligned}$$

Figure 4.5: Proving $\forall \mathsf{x} : \exists \mathsf{y} : P(\mathsf{x},\mathsf{y}) \wedge P(\mathsf{y},\mathsf{x})$ from the assumptions $\forall \mathsf{x} : \exists \mathsf{y} : P(\mathsf{x},\mathsf{y})$ and $\forall \mathsf{x} : \forall \mathsf{y} : P(\mathsf{x},\mathsf{y}) \Rightarrow P(\mathsf{y},\mathsf{x})$ by natural deduction in the action encoding (example from Prawitz: *Natural deduction* [30])

with domain *Nat* is represented by

$$\lambda\, \mathsf{c} \in Nat : \mathsf{c} + 1$$

and encoded in the action reasoning environment in LP as

$$\lambda(Nat, f)$$

with the definition

$$\textbf{assert} \quad f(\mathsf{c}) == \mathsf{c} + 1$$

For reasoning about lambda expressions we have constructed a set of rules *ad hoc* which can be seen in appendix B.1.2.

We also provide support for some set comprehensive constructs that may be practical in TLA specifications, the '*setof*' construct

$$\{\mathsf{c} \in S \mid P(\mathsf{c})\}$$

describing the subset of S for which the logical predicate P holds, and the '*setexp*' construct

$$\{F(\mathsf{c}) \mid \mathsf{c} \in S\}$$

describing the range of the function F on the set S. The first is encoded as

$$setof(S, f)$$

with the definition

$$\textbf{assert} \quad f(\mathsf{c}) == P(\mathsf{c})$$

and the second as

$$setexp(S, g)$$

with the definition

$$\textbf{assert} \quad g(\mathsf{c}) == F(\mathsf{c})$$

Appendix B.1.3 contains introduction and elimination rules for these two set combinators. Generally it should be mentioned that the rules for reasoning about sets and functions are not complete. A theory for sets can freely be added to the LP system, and is unrelated to the work of encoding TLA.

4.3.5 Enabled

TLA for any action $\mathcal{A}$ defines *Enabled* $\mathcal{A}$ to be the predicate that is true for a state s if and only if it is possible to take an $\mathcal{A}$-step starting in that state. This means that there should exist a state t, so that $s[\![\mathcal{A}]\!]t$ is true. Neither of the encodings we have chosen for efficient reasoning about actions and temporal properties in LP allow a simple definition of *Enabled* in the way that it can be done in e.g. the HOL encoding of Wright and Långbacka [35]. This is clearly a drawback of our encodings. On the other hand, there are several ways in which *Enabled* predicates could be handled. With the introduction of a translator we may choose to compute *Enabled* when translating a specification. E.g. for reasoning about *Enabled* $\mathcal{M}1$ in the encoding of Program 1 of the Increment Example, we would produce the additional definitions

$$\begin{array}{lll}
\textbf{assert} & E_\mathcal{M}1(\mathsf{c},\mathsf{d}) & == (\mathsf{c} = x + 1)\ \&\ (\mathsf{d} = y) \\
\textbf{assert} & Enabled_\mathcal{M}1 & == \exists(e_\mathcal{M}1) \\
\textbf{assert} & e_\mathcal{M}1(\mathsf{c}) & == \exists(e1_\mathcal{M}1(\mathsf{c})) \\
\textbf{assert} & e1_\mathcal{M}1(\mathsf{c})(\mathsf{d}) & == E_\mathcal{M}1(\mathsf{c},\mathsf{d})
\end{array}$$

and then translate *Enabled* $\mathcal{M}1$ in the specifications and proofs to *Enabled*_$\mathcal{M}1$. The results from such computations unfortunately tend to be rather complicated expressions compared with the predicates that we may manually come up with; e.g. in the case above, *Enabled* $\mathcal{M}1$ may be simplified to *true*.

Another solution would be to produce the *Enabled* predicates manually, achieving the wanted simplicity, and proving the correctness of the predicates within an encoding that allows a direct representation.

In the current TLP system we have chosen the second solution, although we have not yet been concentrating on finding a practical way to reason about the *Enabled* predicates.

4.3.6 Refinement mappings

Refinement mappings were introduced by Abadi and Lamport as a powerful tool when reasoning about refinements [1]. A refinement mapping is a mapping from the variables

of a specification to state functions referring to variables of a refined specification. Refinement mappings may be represented directly as functions in our temporal reasoning encoding, but the same is not possible in the action reasoning encoding. On the other hand, it is trivial to represent them there by additional definitions generated by the translator. E.g. when encoding the refinement mapping of the cached memory example of the TLA report [20, page 47], we would write the definition of the mapping as

$$\textbf{assert} \quad \mathit{Bar\_memory} == \mathrm{if}(\mathit{cache}(m) = \mathit{bottom}, \mathit{main}(m), \mathit{cache}(m))$$

and for each definition in the containing specifications produce an additional 'barred' version, where all appearances of *memory* on the right-hand side of the '==' are replaced by the state function *Bar_memory*, and all appearances of other state functions, predicates, etc., are replaced by their 'barred' version.

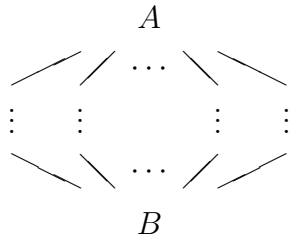
4.3.7 Lattice proofs

When reasoning about liveness we often perform an inductive argument on a well-founded set. In TLA this can be done by application of the Lattice rule (see figure 3.2 on page 26). The encoding of the Lattice rule in the temporal reasoning environment is straightforward. It would indeed be possible to include a theory for reasoning about well-foundedness in LP, but we have found that this is outside the scope of our work so far. The Lattice rule that is used in TLP (shown in appendix B.2.3) therefore just includes the premise

$$\mathit{Wellfounded}(S, s)$$

stating that s is a well-founded ordering on the set S. With the translator it would be possible to perform the verification of this premise with the verification tool and encoding that seem most fit. In the examples where we have been using the Lattice rule, however, well-foundedness of the used set and ordering is left as an assumption. An example of this can be seen in the liveness proof of the Spanning-Tree Example, presented in section 10.3.

For reasoning about *finite* lattices, we use a set of rules referred to as the Grønning lattice rules (appendix B.2.4), based on an idea by Peter Grønning. These rules are easily shown to form a complete system for reasoning about finite lattices, meaning that from any finite lattice



where a set of edges from a node N to a set of nodes $M_1, \ldots, M_n$ below it represents the formula $N \leadsto (M_1 \vee \ldots \vee M_n)$, and where A is an upper bound and B a lower bound, we are able to deduce $A \leadsto B$. The Grønning lattice rules are used e.g. in the fairness proof of the Increment Example, presented in section 9.5.

4.3.8 Automatic verification of finite-state systems

The encodings described so far together form a basis for sound and complete reasoning about finite-state as well as infinite-state systems in TLA, with the exceptions mentioned in sections 4.3.5 and 4.3.7. Restricting our view to finite-state systems, there has been much recent research in the development of powerful model-checking algorithms for *automatic* verification, among others by Clarke et al. [10, 9]. It seems reasonable that we should not insist on creating large proofs for systems that can be verified automatically, and that a system for mechanical reasoning would benefit from integration with tools for automatical verification.

One such tool is an implementation by David Long of *decision procedures* for linear-time temporal logic (LTL) based on so-called binary decision diagrams. These allow direct verification of formulas in a subset of the Computation Tree Logic, CTL*, described by Clarke et al. [10].

It is straightforward to encode a subset of TLA in this logic, in which formulas are constructed from *true*, *false*, boolean variables, $\wedge$, $\neg$, and $\Box$ (as well as derived operators such as $\vee$ and $\Diamond$). The validity of any such formula may be *checked* by an extensive search for counter examples – behaviours not satisfying the formula.

With the translation scheme described earlier, it is possible to let substeps of proofs suitable for such verification be checked automatically by Long's LTL-checker. The conjectures of these steps may then be *asserted* as valid in both the LP action encoding (if the formulas considered do not contain temporal operators) and the temporal encoding of the proof.

A simple example of the use of the LTL checker appears in the Multiplier refinement proof, where three previous results are combined with the help of a lemma that can easily be checked (A and B are rigid variables):

$$\begin{array}{l} \wedge\ \Box(\neg(\neg A \wedge B)) \\ \wedge\ (\neg A \wedge \neg B) \leadsto (A \wedge \neg B) \\ \wedge\ \Box((A \wedge \neg B) \Rightarrow \Box(A \wedge \neg B)) \\ \Rightarrow \Diamond\Box A \end{array}$$

Even in such simple an example, a proof based on deduction would be complicated and time-consuming.

The LTL-checker used in the current TLP system is only able to check rather simple formulas, but other tools can easily be built in when they become available.

5

Interface engineering

In the previous chapter we discussed how TLA may be encoded in different ways so that different kinds of reasoning may be performed efficiently. Using such encodings as the basic specification language would not be practical for the systems designer, as distracting the attention from the logic of the specification towards subtleties of the language and the underlying verification tool. When using more than one encoding, this problem becomes even more evident, and we are furthermore presented with the possibility of human errors causing inconsistency between the different encodings.

The solution is to provide a general language on a level above the encodings. Preferably, this language should be *transparent* with respect to features of the tools and encodings, thus letting you write specifications in pure TLA (or some superset thereof, such as TLA$^+$ [23]) without knowing or having to consider any such details. Such a solution is as well mentioned by Wright & Långbacka in the conclusion of one of their articles on the encoding of TLA in HOL [35].

As we want to reason about the specifications, the language should also permit expression of conjectures and of reasoning strategies. As we know, conjectures in TLA are just TLA formulas, which can already be expressed. Reasoning strategies may be simple instructions on how to verify a conjecture, or they may be more complicated structures showing how a detailed proof can be carried out. The level of detail depends on the verification tool that is used; when using automatic reasoning tools such as decision procedures for temporal logic or arithmetic, nothing more than the conjectures need be specified, whereas when proving a system by natural deduction we may need to represent each individual step.

The possibility of expressing proofs to be verified opens up yet another problem. While it is possible to write simple, correct proofs, that may be verified in the first try, producing more complicated proofs will always be a matter of incremental development and interaction with the verification tool. Using the interactive capabilities of a verification tool such as LP or HOL is possible, but this again dims the logic on behalf of technical aspects of the tool, hampering the development of the proof. A better solution is to provide a front-end that takes care of the interaction between the proof designer and the tools that are used. The front-end should provide a uniform surface, again focusing on *transparency*, meaning that the designer should be able to work on the logical aspects of the proof he is designing, ideally knowing nothing about the verification back-end in use.

In the next sections we discuss the design of an interface for mechanical reasoning in TLA: the high-level language and the interactive front-end. Detailed descriptions of the

language and front-end used in TLP are given in part II.

5.1 The language

The high-level language should make it possible to express TLA specifications and reasoning strategies. We would like the language to be transparent in the sense that the specific encodings and technical aspects of the tools used for reasoning should not disturb the presentation. Specifications should be written in TLA, or some superset thereof, with a potential of expressing e.g. sets and functions by higher-level constructs and with some overlaying structures such as the modules of TLA$^+$ [23].

The language we have constructed for writing specifications in the TLP system is close to being transparent in the above sense. We allow expressions to be written in pure TLA, and extend this with constructs for expressing *declarations* and *definitions*, both being parameterizable. We are bound to using the ASCII character set, so a TLA operator such as $\Box$ has to be written as `[]`, but this is not a major problem. Other than that, the most significant exception from true transparency is the representation of quantified formulas and set expressions, where we need to manually specify the name of the so called *quantifier function* that is used in the encoding for the Larch Prover (see section 4.3.4). When reasoning about quantified formulas, we need to be able to refer to this function, so automatic generation of the names is not desirable.

In figure 6.3 on page 70 we show the specification of Program 1 of the Increment Example, which closely corresponds to the original TLA specification in figure 3.3 on page 27. The only significant difference is the use of *headers* indicating whether a defined expression is a state function, an action, etc., and the *directives* giving the specification a name and expressing its dependency on declarations given elsewhere. For structuring purposes, we also use an additional definition, v, for the state function (x, y).

5.1.1 Proofs

The design of a language for expressing proofs is more subtle. It is clear that for a proof to be mechanically checkable, it needs to be presented in a structural and precise fashion, the level of detail varying with the power of the verification back-end. Fortunately, presenting proofs in a structural manner may be preferable not only for processing by mechanical tools; it can also help the designer avoiding mistakes and errors, while making the proofs easier to read and understand. Work that we put into the design of a structured language for TLP was in fact used by Lamport in his note on how to write *manual* proofs [22].

As a basis for our structural proofs, we found the *natural deduction* style to be well suited. Natural deduction provides a flexible way of treating inferences from assumptions in a way that we are used to. We prove $A \Rightarrow B$ by assuming A and deducing B – *discharging* the assumption when the conjecture has been proved. The structure is simple; a natural deduction proof is essentially just an application of one of the deduction rules (figure 4.3) applied to a number of subproofs, that each may depend on the assumptions indicated by the rule.

Although providing a good basis, natural deduction in its simplicity is however not good enough for making reasoning manageable in practice. We should not need to explicitly state that A is proved from $A \wedge B$ by application of the $\wedge$ elimination rule, when

a step like this could be handled automatically by a verification back-end such as LP. We have thus made our style a refinement of ordinary natural deduction. Having chosen LP as our primary back-end, it is not a coincidence that the explicit deductions that we are left with closely corresponds to LP's backward inference methods. These represent the deduction rules for $\vee$ elimination, $\Rightarrow$ introduction, and the rule *False*. In a logical perspective, the other rules can be assumed to be applied automatically whenever needed.

We simplify things a bit more by generalizing the first two of the remaining rules. A *case* proof is a generalization of the $\vee$ elimination rule, where we deduce B from $A_1 \vee \ldots \vee A_n$ and the subproofs deducing B from each of the A_i. Likewise, an *implication* proof is a generalization of the $\Rightarrow$ introduction rule, deducing $A_1 \Rightarrow (A_2 \Rightarrow \ldots (A_n \Rightarrow B)\ldots)$ by assuming each of the A_i and proving B.

A deduction is denoted by a proof step, which begins with a conjecture, and ends with the keyword **Qed**. In between the conjecture and the **Qed** are the subproofs represented by substeps with the same structure. In the cases where simple boolean logic is all that is needed to finish a proof step or where we need to apply special rules for reasoning about data types etc., we may indicate how to do this by applying typed, parameterized macros, which we refer to as *methods* (see section 6.9).

The *kind* of the conjecture sometimes indicates what kind of a deduction is to be done. The conjecture **Assume** A_1**,** $\ldots$**,** A_n **Prove** B is logically equivalent to $A_1 \Rightarrow (A_2 \Rightarrow \ldots (A_n \Rightarrow B)\ldots)$, but is to be proved by $\Rightarrow$ introduction. The conjecture **Case** A_1**,** $\ldots$**,** A_n is likewise to be proved as an implication $A_1 \Rightarrow (A_2 \Rightarrow \ldots (A_n \Rightarrow C)\ldots)$, where C is the *current* goal, i.e. the conjecture of the proof of which this is a subproof. Case proofs are indicated by the *method application* **By-Cases**, which implicitly takes the contained **Case**-steps to be the premises of the $\vee$ elimination rule.

The main advantage of the general proof language is not the natural deduction style; this is already present in LP and other verification tools. But with the specially adapted TLP language we get transparency with respect to the conjectures and other formulas that we need specify – these may all be written in pure TLA – and uniformity with respect to the different back-ends that we choose to use. The language is structured in a sense that is well known from programming languages; it visualizes the proof steps rather than merely a sequence of instructions. The static structure furthermore makes it easy to refer to different parts of proofs without the explicit naming directives used in pure LP; proof steps and hypotheses are referenced by a simple naming scheme, relative to the position in the proof.

A detailed description of the TLP language of proofs with examples is given in section 6.8.

5.2 The translator

With a high-level language for expressing specifications and proofs, we need a translator for generating the encodings that can be interpreted by the verification back-ends. Such a tool is easy to construct, in that it is basically just a parser for the high-level language combined with an array of code generators, one for each back-end and encoding.

As mentioned in chapter 4, we may however also let the translator take care of some of the simpler tasks connected to verification. Such tasks are the checking of wellformedness and of sort correctness of the used expressions. Checking wellformedness is a task that is

already handled by the parser, while sort correctness may be handled by a sort checker built into the translator. This checks whether a defined action can really be interpreted as a mapping from two states to a boolean and so on. It may also check whether proofs of temporal formulas are erroneously used as substeps of action proofs and complain if so is the case. The most important aspect of sort checking is probably that the translator with sort information is able to split proofs into different parts that it knows to be checkable by either action or temporal reasoning. Automating the splitting process ensures that the combined result of action and temporal reasoning is sound, when both kinds have been verified separately.

5.3 The interactive front-end

The high-level language makes us able to write well-structured specifications and proofs. With a translator, we may check the wellformedness of both, and generate encodings for use with various verification tools. With an appropriate setup of the verification tools containing a representation of the TLA logic, we are finally able to verify the proofs, or, while in the phase of development, detect the erroneous and non-verifiable parts. We thus have a working system for specifying and reasoning about systems in TLA.

While the use of a general high-level language, with the possibility of different encodings and a variable number of back-ends, is an advancement from the stage where we were reasoning directly in the formalism of a single proof-checker, it clearly increases the complexity of the reasoning process. It is harder to work interactively on a proof, getting useful information back from the proof-checker while trying different strategies. In the setup we have described, whenever the proof checker stumbles on a verification problem, we have to re-edit and re-translate the structured proof, generating a modified encoding for the proof checker. We may have to manually instruct the proof checker to undo some unsuccessful steps, before executing the correct part of the new code, or in the worst case to redo the verification of the complete proof from the start. Getting information from the proof checker is only possible by sending it internal commands in its own input language, and manually interpreting the information it is able to present using its specific encoding of the logic.

All the work we have to do when encountering verification problems again makes the proof development diverge from the essence of the logic – it gets harder to understand the proof itself when having to deal with technical aspects of the verification system. Ideally, we should be able to work on proofs in a structural manner, dealing with the problems of each single step without bothering about the rest of the proof. When encountering a problem, it should be possible to go on by giving a few additional instructions or changing the step that could not be verified, with no knowledge about how the verification session is to be resumed.

Luckily, the problem of the added complexity can be solved, as we may provide the simple interface that we would like by building a front-end. This front-end should be the single, uniform interface between the user, the translator, and the verification tools with their different setups. To start a verification session, the user should only have to execute a single command, while the front-end should take care of the translation of the files that are needed, start the relevant verification back-ends and feed them their input. When a verification problem occurs, this should be presented to the user by an indication

of the step that caused the problem, allowing the user to change it, or merely to add instructions. The user should also be able to get information from the back-ends that helps in finding the right verification strategy, and to go backwards in the proof when a strategy has shown to be unsuccessful. After the required changes have been performed, the user again should have to execute a single command only, to resume verification from the point where it was stopped. The front-end should then take care of any required re-translation, feeding the right part of the generated code to the back-end. In some cases it may redo the complete proof session if this is needed; the important thing is that the user always sees it as if the session was continued from the point where it stopped.

The front-end that is used in the TLP system has been built on top of the real-time display editor Emacs [32], which besides allowing ordinary and syntax oriented editing of text, provides an extension language in which advanced and specialized features may be programmed. Using a customizable editor like Emacs as our front-end is advantageous, in that it offers the same specialized environment for editing TLP specifications and proofs when in the middle of a verification session, as is available at other times. The TLP front-end built on top of Emacs uses different windows for displaying the source files, messages from the front-end about the status of verification sessions including syntax errors and verification problems, and output from the verification back-ends. It does not yet provide interpretation of the output from back-ends, which can only be given in the format of the used encoding.

Figure 5.1 gives a schematic overview of the complete TLP system, showing how the front-end glues the different parts together. The front-end interacts with the user at one end, and at the other with the file-system containing the source files that the user wants to work on (consisting of TLP and LP code), the translator that translates the source files to input for the different back-ends, and with each of the back-ends. When instructed to do so by the user, it may perform translation and verification sessions, by running the translator and verification back-ends, feeding them input from either the source files or the generated code and setup files, provided as part of the system. The current system uses two back-ends based on the Larch Prover for action and temporal reasoning, and one based on BDDT for automatic checking of linear-time temporal logic. The front-end is described in detail in chapter 8.

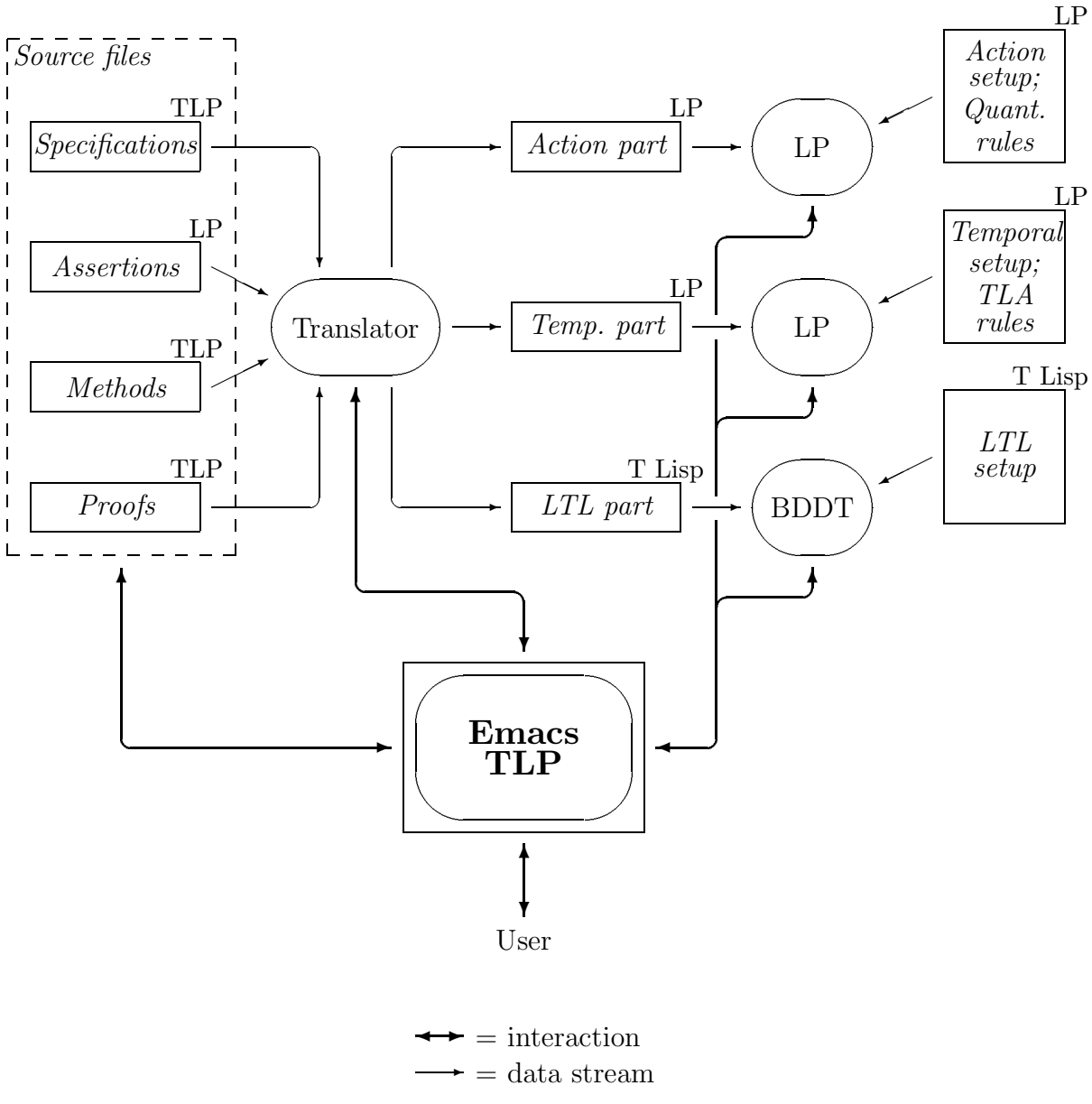


Figure 5.1: Overview of the TLP system

Part II

The TLP system

6

The language

The TLP language is a dialect of TLA extended with certain declarations and directives for parsing and verification purposes, and a language for writing structured proofs. By a TLP *script* we refer to any file of text written in the TLP language. A TLP script may contain an arbitrary number, in any order, of

directives, specifying e.g. which other files the script depends on;

specifications, consisting of *declarations* of constants and variables, and *definitions* of state functions, predicates, actions, temporal formulas, etc.; and

proofs, being structured constructs that specify how lemmas and theorems may be proved, to be verified in the TLP system.

The contents may furthermore be grouped under common *headers*, which are used for naming, and thus enabling reference to certain facts contained in the related specifications and proofs.

In this chapter we go through the different parts of the TLP language. Section 6.1 describes the idea of *types* and *sorts* in TLP. Section 6.2 goes on to describe well-formed TLP *expressions*, specifying their interpretation and syntax. Then follows descriptions of the more structurally important constructs of the language, the *headers* and *directives*, *declarations*, and different kinds of *definitions* in sections 6.3–6.7. In section 6.8 we describe the fundamental idea and interpretation of *proofs* in TLP, and finally in section 6.9 we introduce the concept and syntax of verification *methods*, how they are defined and used.

6.1 Types and sorts

6.1.1 The type system

TLA is an untyped logic. Variables have no type, and *states* are just mappings from variable names to the set Val of all values. In his initial papers on TLA, Lamport regarded the set containing the booleans true and false, Bool, as a subset of Val. Lately [24], this has changed to a view in which Val and Bool are distinct sets, which is also the current view in TLP.

As a feature of TLP, variables can also take boolean values. For this purpose we introduce two *types*, **Val** and **Bool** which correspond to the sets Val and Bool. Variables and constants are declared to be of type **Val** or **Bool**, and take their values exclusively from the corresponding set. User-declared operators are declared as taking one or more operands of type either **Val** or **Bool**, and producing a **Val** or **Bool**, so that e.g. arithmetic addition would be declared as **Val, Val -> Val**.

6.1.2 The sort system

As in TLA we let a *state* be a mapping from *variable names* to *values*. As we in TLP also allow *boolean* variables, *values* here means the union of the sets Val and Bool and thus define the set of states, **St**, as the set of mapping from variables names to Val ∪ Bool.

Elements of the set **St**×**St** are classified as *transitions*, and infinite sequences of states, elements of the set $\mathbf{St}^\infty$, as *behaviours*.

When reasoning in TLP we assign to all well-formed expressions a *sort* indicating our interpretation of the expression as an element of Val or Bool, or as a mapping from states, transitions, or behaviours to one of these two sets. The interpretation as an element of Val is designated the sort **Constant**, while expressions interpreted as elements of Bool are of sort **Boolean**. The rest of the sorts are named consistently with TLA as **Statefunction**, **Predicate**, **Action**, and **Temporal** (formula), and furthermore the implied sorts, **Transitionfunction**, and **Sequencefunction**. The corresponding interpretations are shown in table 6.1.

Sort	Interpretation
Constant	Val
Boolean	Bool
Statefunction	$\mathbf{St} \to$ Val
Predicate	$\mathbf{St} \to$ Bool
Transitionfunction	$\mathbf{St} \times \mathbf{St} \to$ Val
Action	$\mathbf{St} \times \mathbf{St} \to$ Bool
Sequencefunction	$\mathbf{St}^\infty \to$ Val
Temporal	$\mathbf{St}^\infty \to$ Bool

Table 6.1: The sorts of TLP.

Any non boolean-valued constant expression (i.e. of sort **Constant**) can also be interpreted as a state function, interpreting it as the constant function from **St** to Val. Similarly, any state function can be interpreted as a transition function, and any transition function as a sequence function. We say that the sorts are *ordered* transitively, so that **Constant** is the least element in the ordering and **Sequencefunction** the greatest. Likewise for boolean expressions, predicates, actions, and temporal formulas. The ordering is thus as shown in figure 6.1.

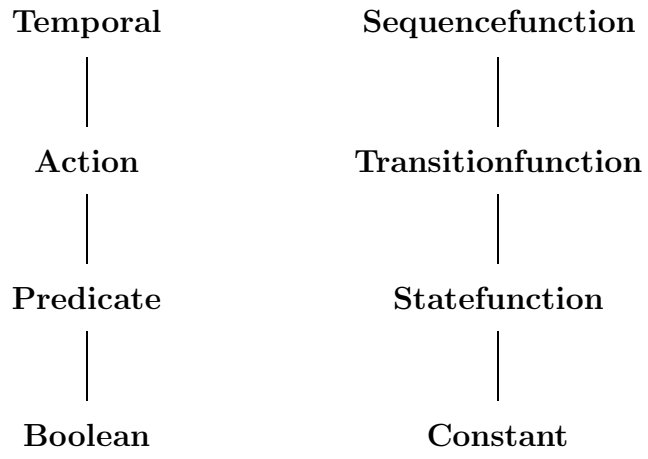


Figure 6.1: Ordering of the TLP sorts.

6.2 Expressions

The *basic expressions* in TLP are constructed from user-declared variables, constants, and operators, the built-in constants **True** and **False**, and the operators = (equality), ~= (inequality), **in** (set inclusion), /\ (conjunction), \/ (disjunction), ~ (negation), => (implication), and <=> (boolean equivalence). Examples are

$$x \texttt{ + } 1 \texttt{ - } y$$
$$y \texttt{ >> } z \texttt{ /\ } a \texttt{ => } \sim b$$
$$z \textbf{ in } S \texttt{ \/ } \textbf{True}$$

The built-in operators are ordered with respect to precedence as

=> <=>
\/
/\
= ~=
in

with **in** binding tightest. All user-declared operators bind tighter than the built-in operators. The expressions above is thus interpreted as their explicitly parenthesized analogues

$$(x \texttt{ + } 1) \texttt{ - } y$$
$$((y \texttt{ >> } z) \texttt{ /\ } a) \texttt{ => } \sim b$$
$$z \textbf{ in } (S \texttt{ \/ } \textbf{True})$$

For a basic expression to be *well-formed*, all the contained operators should be applied to the correct number of operands ranging over the right set of values, Val or Bool. A well-formed expression is *boolean-valued*, if its range is Bool. The first two expressions above are well-formed, if + and - are declared as **Val, Val -> Val** and >> as **Val, Val -> Bool**, the second is furthermore boolean-valued. The third cannot be well-formed, as **in** has type **Val, Val -> Bool**.

On top of the basic expressions, TLP contains several complex constructs, namely *tuples*, *conditionals*, *sets*, *functions*, and *quantified expressions*. We will return to these

in the end of this section, but first we will describe the different kinds of expressions that we talk about relating to the different *sorts* of TLP, and introduce the TLA *prime* and *always* operators, and some syntactic sugar.

In the following we will use the meta-variables listed below – sometimes with additional subscripts as in v_0, v_1:

$\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$: any well-formed expressions

$\mathcal{B}$: any boolean-valued expression.

f: a state function or predicate

$\mathcal{A}$: an action

F, G: temporal formulas

c: a rigid variable

6.2.1 Constant expressions

Constant expressions are well-formed expressions containing no variables. Their interpretations are as values of the sets Val and Bool. We say that a boolean valued expression has sort **Boolean**, and a non boolean-valued expressions sort **Constant**.

6.2.2 State functions and predicates

State functions and *predicates* are well-formed expression built from constants, operators, and variables. In TLP, a *state function* is always *non* boolean-valued, meaning that its interpretation is as a mapping from **St** to Val. Likewise, a *predicate* is boolean-valued and is interpreted as a mapping from **St** to Bool. We say that state functions have sort **Statefunction**, and predicates sort **Predicate**.

The first example on the preceding page is a state function interpreted as the mapping that assigns to a state s the value $(s[\![x]\!] + 1) - s[\![y]\!]$, where $s[\![x]\!]$ denotes the value of x in s, and $+$ and $-$ are the semantical interpretations of + and -. The second example is a predicate, while the third, being non well-formed, is neither.

Any constant expression is also a state function or predicate, as it can be interpreted as a (constant) function from **St** to either Val or Bool.

6.2.3 Transition functions and actions

Transition functions and *actions* are well-formed expressions built from constants, variables, and *primed* variables. They map pairs of states to Val and Bool respectively, by mapping unprimed variables to their values in the first state and primed variables to their values in the second. An example of a transition function is x', and an example of an action is $(x' = x + 1 \ /\backslash \ y' = y)$. We say that transition functions and actions have sorts **Transitionfunction** and **Action**.

As in TLA, we allow any state function and predicate to be primed. A primed expression is equivalent to the expression with all variables replaced by their primed counterparts.

Unchanged(f) is semantically equivalent to the action f' ~= f. [$\mathcal{A}$]_f is the action equivalent to $\mathcal{A}$ \/ **Unchanged**(f), and <$\mathcal{A}$>_f the action equivalent to $\mathcal{A}$ /\ (~**Unchanged**(f)).[1]

Enabled<$\mathcal{A}$>_f is the *enabled* predicate of the action <$\mathcal{A}$>_f which is the predicate that maps a state s to **True**, if and only if there exists a state t so that <$\mathcal{A}$>_f evaluates to **True** on $\langle s, t\rangle$. There is currently no way to compute enabled predicates in TLP. Until this has been implemented, we allow enabled predicates to be *defined*, as described in section 6.7, page 72.

State functions and predicates can also be considered as transition functions and actions respectively, by interpreting them as functions from $\mathbf{St}\times\mathbf{St}$, depending only on their first parameter.

6.2.4 Temporal expressions

Temporal expressions are built from transition functions and actions (and thus also from constant expressions, state functions, and predicates) and the temporal operator [] (*always*). A boolean-valued temporal expression such as []F is known as a *temporal formula* (with sort **Temporal**), and is interpreted as a mapping from behaviours, $\mathbf{St}^\infty$, to Bool. The mapping described by the formula []F is the function mapping an infinite sequence of states $s_0, s_1, \ldots$ to **True** if and only if the formula F holds for each tail of the sequence, i.e. for any i the behaviour $s_i, s_{i+1}, \ldots$. An action holds for a behaviour if and only if it holds for the pair described by the first two states, and a predicate thus holds for the behaviour if it holds for its first state.

<>F is equivalent to ~[]~F, and F ~> G to [](F => <>G). The fairness expression **WF**(f, $\mathcal{A}$) is equivalent to []<><$\mathcal{A}$>_f \/ []<>~**Enabled**<$\mathcal{A}$>_f and **SF**(f, $\mathcal{A}$) to []<><$\mathcal{A}$>_f \/ <>[]~**Enabled**<$\mathcal{A}$>_f.

6.2.5 Tuples

A tuple in TLP is an ordered list of boolean and non-boolean expressions. It is written as in standard TLA, enclosed in parentheses and with its elements separated by commas. (If the tuple itself is directly enclosed within the parentheses of some other construct, such as **Unchanged**(...), the parentheses can be left out.) Equality of tuples is defined as the element-wise equality, as usual. A tuple of elements of sort **Constant** and **Boolean** has sort constant, a tuple of elements of sort **Statefunction** and **Predicate**, sort **Statefunction**, and so forth. Tuples are used throughout all the examples in part III.

6.2.6 Conditionals

A conditional is an expression of the form **If** $\mathcal{B}$ **then** $\mathcal{U}$ **else** $\mathcal{V}$. It is interpreted as the expression equal to the value of $\mathcal{U}$ whenever $\mathcal{B}$ equals true, and the value of $\mathcal{V}$ whenever

[1] In TLA, these constructs are regarded as *syntactic sugar*, defined to equal the respective actions. In TLP, they may be handled differently by different back-ends (although always preserving the semantical equivalence), so they cannot be regarded purely as definitions. E.g. when doing temporal reasoning, using the notation [$\mathcal{A}$]_f is not equivalent to writing $\mathcal{A}$ \/ **Unchanged**(f), as a proof depending on the actions would differ on the application of a rewrite rule (the rule *BAct*, to be precise).

$\mathcal{B}$ equals false. If $\mathcal{B}$ has sort **Boolean** and $\mathcal{U}$ and $\mathcal{V}$ sort **Constant** (or **Boolean**, respectively) then **If** $\mathcal{B}$ **then** $\mathcal{U}$ **else** $\mathcal{V}$ has sort **Constant** (**Boolean**, respectively). Likewise for **Predicate** and **Statefunction**, etc.

If x = 0 **then** *pc1* **else** *pc2* is thus the *state function* that produces the value of *pc1* for all states in which the variable x has the value 0 and the value of *pc2* in all other, while **If** *sem* = 1 **then** *pc1*′ = b **else Unchanged(***pc1***)** is the *action* that sets *pc1* to the value b whenever *sem* is 1 and keeps it unchanged otherwise. A real example where a conditional expression is used as a state function can be seen in the specification of the spanning-tree algorithm on page 133.

6.2.7 Quantification

TLP supports universal and existential quantification over *rigid* variables. Quantification over *flexible* variables, used for e.g. *hiding* variables in a specification, is not supported.

A universally quantified formula is written either as

Forall c **in** $\mathcal{U}$: $\mathcal{B}$ [* m *]

or just

Forall c : $\mathcal{B}$ [* m *]

and similarly for existential quantification, using the keyword **Exists**. We call the first form *bounded* quantification, and the other *unbounded*. The rigid variable c need not be declared. The meta-variable m inside the special brackets [* ... *] stands for the name of the *quantifier function* which should be an identifier that is not used elsewhere as a quantifier function. The quantifier function is a function that is automatically defined from the quantified formula when we translate it into the language of the Larch Prover in the current TLP system. If the boolean-valued expression $\mathcal{B}$ contains *free* rigid variables (i.e. variables different from c and not bound by quantification inside $\mathcal{B}$), then these should be appended as parameters to the quantifier function (this could be automated in later releases). See section 4.3.4 on page 45 describing how quantification is encoded for an explanation to why the function is needed. An example of bounded, existential quantification is shown in the specification of the spanning-tree algorithm, page 133.

When a quantified expression has been used once, there is a useful hack for referring to the same expression elsewhere, by writing just

Forall($\mathcal{U}$, m**)**

or

Forall(m**)**

where m is the (possibly parameterized) quantifier function name of the original expression. This is useful, as one may not otherwise use the same quantifier name twice, and showing with the Larch Prover that the same two quantified expressions are equivalent would otherwise be unnecessarily cumbersome. This is used in e.g. some of the Spanning-Tree proofs; see for example the lemma 'DoneOneDoneAll' on page 150 (step <2>1).

6.2.8 Sets and functions

TLP also has a number of built-in constructs for expressing sets and functions. The construct

$\{\mathcal{V}$ | c **in** $\mathcal{U}$ [* m *]$\}$

denotes the set of elements $\mathcal{V}$ so that there exists a c in the set $\mathcal{U}$. Likewise, the construct

$\{$c **in** $\mathcal{U}$: $\mathcal{B}$ [* m *]$\}$

denotes the set of elements c in the set $\mathcal{U}$ so that $\mathcal{B}$ is true. In both cases, m should be a fresh function identifier which is to be used just like the quantifier function name described in section 6.2.7 above, and should thus be parameterized when $\mathcal{V}$ ($\mathcal{B}$, respectively) contains free rigid variables. TLP does not enforce the requirement that $\mathcal{U}$ be a non-boolean valued expression, although it will make little sense if it is not.

An example of the use of the first set construct, together with bounded existential quantification and function application (see below), is the definition of 'NotDoneSet' from the correctness proof of the Spanning-Tree Example (page 149). Note that the rigid variable x is free inside the quantified expression, and thus supplied as a parameter to the quantifier function *pnotdone*.

```
NotDoneSet == {x in Nat :
                  Exists p in Node : x = dDone[p] /\ dDone[p] << d[p]
                     [* pnotdone(x) *]
                  [* notdoneset *]}
```

in may also be used as an infix operator for specifying set inclusion, as in

$\mathcal{U}$ **in** $\mathcal{V}$

For expressing functions (or arrays) we use lambda notation, and let

Lambda c **in** $\mathcal{U}$: $\mathcal{V}$ [* m *]

denote the function with domain $\mathcal{U}$ mapping an element c to the expression $\mathcal{V}$. Again, m should be a fresh function identifier as explained above.

Lambda i **in** interval(0, n): square(n) [* sqarray *]

would thus be the function mapping the subset of the natural numbers from 0 to n to their square, with the right definitions of the operators. Another example may be seen, again, in the specification of the Spanning-Tree algorithm. You may use **Array** as a synonym for **Lambda**.

The expression

$\mathcal{U}$[$\mathcal{V}$]

denotes the application of the function $\mathcal{U}$ to the expression $\mathcal{V}$, which may alternatively be written as $\mathcal{U}$ @ $\mathcal{V}$. The expression

[$\mathcal{U}$ -> $\mathcal{V}$]

expresses the set of functions from $\mathcal{U}$ to $\mathcal{V}$. It is common to declare the operator **Dom** for returning the domain of a function. In the LP files containing rules for reasoning about functions and sets, this operator is given a meaning by deduction and rewrite rules, although it is not built-in as a predefined operator in the current TLP system.[2]

When specifying actions, we often want to update a variable whose value is a function (array). For this purpose we have two constructs. The first,

$$\mathcal{U}[\mathcal{V} := \mathcal{W}]$$

denotes the function with the same domain as the function denoted by $\mathcal{U}$, which maps the value of $\mathcal{V}$ to the value of $\mathcal{W}$ (if the value of $\mathcal{V}$ is in the domain of the function $\mathcal{U}$), and is equal to $\mathcal{U}$ on all other values. The second construct,

$$\mathcal{U}[\mathcal{V} :\mathbf{in}\ \mathcal{W}]$$

is the adaption of the first to sets; it denotes the *set* of functions with domain $\mathcal{U}$, which map the value of $\mathcal{V}$ to any value in the *set* $\mathcal{W}$ (if $\mathcal{V}$ is in the domain of $\mathcal{U}$), and all other values to the one of $\mathcal{U}$. Both constructs appear in the specification of the Spanning-Tree algorithm (page 133).

6.2.9 Indentation sensitive expressions

As used in TLA, we introduce a concept of *indentation sensitive* expressions in TLP. This makes long formulas easier to read and write, as the structure becomes evident without the need of parentheses. A list of expressions separated by newlines and 'bulleted' by /\ or \/, all indented to the same level, is interpreted as the conjunction respectively disjunction of the expressions. Thus, the expression

$$\begin{array}{l} /\backslash\ \mathcal{U}_1 \\ /\backslash\ \backslash/\ \mathcal{U}_2 \\ \quad \backslash/\ /\backslash\ \mathcal{U}_3 \\ \quad\quad /\backslash\ \mathcal{U}_4 \\ /\backslash\ \mathcal{U}_5 \end{array}$$

is equivalent to

$$\mathcal{U}_1\ /\backslash\ (\mathcal{U}_2\ \backslash/\ (\mathcal{U}_3\ /\backslash\ \mathcal{U}_4))\ /\backslash\ \mathcal{U}_5$$

The implication $(\mathcal{U}_1\ \backslash/\ \mathcal{U}_2\ \backslash/\mathcal{U}_3)$ => $(\mathcal{U}_4\ /\backslash\ \mathcal{U}_5)$ may be written as

$$\begin{array}{l} \backslash/\ \mathcal{U}_1 \\ \backslash/\ \mathcal{U}_2 \\ \backslash/\ \mathcal{U}_3 \\ => /\backslash\ \mathcal{U}_4 \\ \quad /\backslash\ \mathcal{U}_5 \end{array}$$

and a boolean equivalence, using <=>, likewise. For further explanation on this notation, see Lamport's note on how to write long formulas [21]. The parser that is used in the current release of TLP only allows indentation sensitive expressions on the outermost level of expressions, which means that they among other places can't be used inside quantification. This is unfortunate, but will not be solved until a new parser is written.

[2] Generally, we prefer to keep the amount of built-in constructs as low as possible, to gain flexibility. The intention is that operators such as **Dom** should be included in special 'packages' that can be used in different examples.

6.3 Headers and directives

We now go back to some of the more distinct constructs of TLP. *Headers* and *directives* are constructs that are used by the translator and the front- and back-ends when translating and verifying.

6.3.1 Name

The **Name** construct is semantically insignificant. It is used as a header to specify a group of definitions and proofs that belong together. It is furthermore transfered to the relevant code generators, so that it can be used for naming groups of definitions. This is used e.g. for naming the 'quantifier functions' used to specify quantified formulas in LP (described in section 6.2.7). This will become obsolete when a module system is introduced in a future version.

6.3.2 Use and Bottom

The **Use** directive is used to specify a transitively closed dependency relation between the containing file and a set of other files. Specifying

`%-` **Use** $file_1$ $file_2$ ... $file_n$

anywhere in a file means that formulas used in definitions and proofs in the file may depend on definitions, theorems, and lemmas in one or more of the files $file_1$ to $file_n$, and all the files that these refer to, etc. This means that information from any of these files will be used when analyzing and verifying the well-formedness of predicates, actions, temporal formulas etc. The code generators may or may not take advantage of this information; the two currently supplied LP code generators do while the LTL generator does not (see chapter 7 for information on how the code generators use the dependency specification). The dependency relation is also used when redoing the verification of proofs from inside the front-end; whenever any files that the current file depend on have been modified, the front-end assures that these are re-translated and verified as well as the current file. This is described thoroughly in section 8.4.1.

The **Bottom** directive, which is written simply as

`%-` **Bottom**

is a technicality that may direct the verification back-end to change its ordinary strategy to save space to the cost of execution-time. For an explanation, see section 8.4.1.

A TLP file should only contain *one* **Use** directive. Supplying more than one **Bottom** directive has no effect.

6.3.3 Include

The directive

`%-` **Include** $file_1$ $file_2$ ... $file_n$

simply includes the contents of $file_1$ to $file_n$, in that order and at the position of the directive, whenever the file is parsed. This is especially convenient and mostly used for including files declaring methods (see section 6.9, page 78), as it makes it possible to add and change methods while in the middle of a verification session.

6.4 Declarations

TLP contains only a few pre-declared constants and operators, as presented in section 6.2 above: the constants **True** and **False**, and the operators `=`, `~=`, **in**, `/\`, `\/`, `~`, `=>`, and `<=>`, as well as the special action and temporal operators.

All other constants, operators, and variables need to be declared, to be used within TLP expressions.[3] A set of declarations begin with the header **Constants**, **Values**, **Operators**, **Rigid Variables**, or **Variables**, specifying the kind of identifiers to be declared. Following the header should be a number of lines, each containing a list of identifiers and possibly ended by a type signature. Below we describe the different kinds of declarations.

6.4.1 Constants

Constants are not defined to have any particular value or property; if we want to make assertions about certain constants, these should be added either by a theory that we load into the back-end (as e.g. the rules we add for reasoning about arithmetic in the Increment Example – see chapter 9) or by explicit assumptions in the theorems we prove.

Constants, just like variables, can have type **Val** or **Bool**. The declaration

Constants
P, *Q* : **Val**
Flag : **Bool**

has the obvious interpretation. Boolean constants are rarely used, but are allowed mostly for consistency.

Constants can be parameterized, so that you may create different instances of them, and of whole system specifications. This is explained in section 6.6, page 70.

6.4.2 Values

Values in TLP are just constants, although their declaration differs semantically from that of ordinary constants. As value constants are meant to be used for representing different values of declared data types, a declaration of a set of values (specified as a sequence of identifiers separated by commas), implicitly states that the values are distinct. Thus, the declaration

Values
0, 1, 2
Nat

declares 0, 1, 2, and *Nat* as constants (which from then on can be used freely in formulas) and let the code generators produce assertions that 0, 1, and 2 are mutually different. The value constants are, of course, always of type **Val**. Values may be parameterized in the same way as ordinary constants, although the use of this seem to be more uncommon.

[3] An exception is the *bound* rigid variables used in quantified expressions and set and lambda expressions, which do not have to be declared.

6.4.3 Operators

Constant *operators* can be defined much like ordinary constants. Operators take one or more parameters of type **Val** or **Bool** and produce a **Val** or **Bool**. Like in LP, operators that consist of only special characters, or that are prefixed by a 'backslash' ('\'), are conveniently defined to be infix, so that

Operators
`+, -` : **Val, Val -> Val**
Positive : **Val -> Bool**

will let you write e.g. "x+y" and "*Positive*(x)". As with constants, the declaration in the TLP script is only made so that the TLP translator will be able to syntactically check the well-formedness of expressions containing the operators; the semantic definition of the operators should be added as theories known to the back-end.

6.4.4 Rigid variables

What we define as *rigid variables* are again, logically, just constants. However, while we allow users to assert statements about *constants* as declared above, to e.g. say that 1 is greater than 0, the *rigid variables* are real mathematical 'unknowns'. The declaration

Rigid Variables
n : **Val**

would thus in LP be translated into a declaration of n as an LP *variable*, while the constant P and the value *Nat* above would become LP *operators*. Rigid variables can be of type **Val** or **Bool**.

6.4.5 Variables

Flexible variables, or *program variables*, are declared as just *variables*. Like rigid variables, they can have type **Val** or type **Bool**. Like constants, they can be parameterized, see section 6.6 on the next page.

An example of a complete declarations part, the one of the Increment Example, is shown in figure 6.2.

6.5 Definitions

Definitions in TLP are written in a straightforward way. A set of definitions begins with the header **Statefunctions**, **Predicates**, **Transitionfunctions**, **Actions**, or **Temporal**, which declare the identifiers to be defined to be of the given sort. The definitions should all be sort-correct, which means that e.g. definitions as **Actions** should all be functions from **St**×**St** to **Bool**. This means that it is quite all right, although of dubious use, to define e.g. a predicate as being an action – as any predicate is also a function from **St**×**St** to **Bool**.

The order of the definitions under a common header is insignificant. This is so that definitions can be written in a straightforward way just as you would like them. It also

Name Declarations
%-Use frame
Values
  0, 1, 2
  Nat
Operators
  +, - : **Val, Val -> Val**
  << : **Val, Val -> Bool**
Variables
  x, y : **Val**

Figure 6.2: Initial declarations for the Increment Example.

makes it possible to write circular definitions, which are not detected by the current system; this should of course be avoided.

The specifications of Program 1 and Program 2 of the Increment Example (figure 6.3 and figure 6.4) illustrate the common use of definitions.

Name Def0
%-Use declarations
Predicate
  InitPhi == (x = 0) /\ (y = 0)
Actions
  M1 == (x' = x + 1) /\ (y' = y)
  M2 == (y' = y + 1) /\ (x' = x)
  M == *M1* \/ *M2*
Statefunction
  v == (x, y)
Temporal
  Phi == *InitPhi* /\ [][*M*]_v /\ **WF**(v, *M1*) /\ **WF**(v, *M2*)

Figure 6.3: The specification Φ of Program 1 of the Increment Example.

6.6 Parameterized definitions and declarations

Constants, values, variables, and definitions can all be *parameterized*. This is essential for convenient specification of non finite-state systems such as the one described in the Spanning-Tree Example. Let us e.g. assume a set of nodes in a graph, for each pair of neighbours (n, m) of which we want to be able to perform the parameterized action $N(n, m)$. The disjunction of the possible actions may then be defined as the action *NAll*, as follows:

Name Def1
%-Use declarations
Variables
pc1, *pc2*, *sem* : **Val**
Values
a, *b*, *g*
Predicate
InitPsi == /\ (*pc1* = *a*) /\ (*pc2* = *a*)
/\ (*x* = 0) /\ (*y* = 0)
/\ *sem* = 1
Actions
alpha1 == /\ (*pc1* = *a*) /\ (0 << *sem*)
/\ *pc1'* = *b*
/\ *sem'* = *sem* - 1
/\ **Unchanged**(*x*, *y*, *pc2*)
beta1 == /\ *pc1* = *b*
/\ *pc1'* = *g*
/\ *x'* = *x* + 1
/\ **Unchanged**(*y*, *sem*, *pc2*)
gamma1 == /\ *pc1* = *g*
/\ *pc1'* = *a*
/\ *sem'* = *sem* + 1
/\ **Unchanged**(*x*, *y*, *pc2*)
alpha2 == /\ (*pc2* = *a*) /\ (0 << *sem*)
/\ *pc2'* = *b*
/\ *sem'* = *sem* - 1
/\ **Unchanged**(*x*, *y*, *pc1*)
beta2 == /\ *pc2* = *b*
/\ *pc2'* = *g*
/\ *y'* = *y* + 1
/\ **Unchanged**(*x*, *sem*, *pc1*)
gamma2 == /\ *pc2* = *g*
/\ *pc2'* = *a*
/\ *sem'* = *sem* + 1
/\ **Unchanged**(*x*, *y*, *pc1*)
N1 == *alpha1* \/ *beta1* \/ *gamma1*
N2 == *alpha2* \/ *beta2* \/ *gamma2*
N == *N1* \/ *N2*
Statefunction
w == (*x*, *y*, *pc1*, *pc2*, *sem*)
Temporal
Psi == *InitPsi* /\ [][*N*]_*w* /\ **SF**(*w*, *N1*) /\ **SF**(*w*, *N2*)

Figure 6.4: The specification Ψ of Program 2 of the Increment Example.

Actions
NAll == **Exists** n in *Node*
Exists m in Nbrs(n)
$N(n, m)$ [* *n2* *] [* *n1*(n) *]

Parametrization can of course also be used for easy definition of multiple instances of a specification. As an example, consider the specification

Variables
$x(i)$: **Val**
Predicates
Init(i) == $x(i) = 0$
Actions
$M(i)$ == $x(i)' = x(i) + 1$
Temporal
Counter(i) == *Init*(i) /\ [][$M(i)$]_$x(i)$

By introducing a set of *values*, $A, B, C, \ldots$, this will now refer to a set of different counters, $Counter(A)$, $Counter(B)$, $Counter(C), \ldots$, each incrementing a distinct variable. An array (of variable length) of counters is specified by

Temporal
Counters(n) == **Forall** i **in** *NatsLowerThan*(n) :
Counter(i) [* *counters*(n)*]

6.7 Enabled and Bar

In addition to the ordinary definitions, it is possible to define *enabled* predicates and *refinement mappings*.

The *enabled* predicate **Enabled**<A>_f for a given action A and state function f is a computable formula, defined as the predicate that is true for any state s, such that there exists a state t so that <A>_f is true on $\langle s, t \rangle$. There isn't yet any way to either compute or prove the correctness of enabled predicates in TLP, so we have been forced to add a possibility to directly include assumptions about enabled predicates as definitions. We would like to fill this hole in the safety of TLP as soon as possible; but until that is done one should regard the definitions of enabled predicates as open assumptions, on which the soundness of the proofs rely. In the Increment Example, we state the value of the enabled predicates of the actions of the first program as:

Predicates
Enabled<*M1*>_v == x ~= $x + 1$
Enabled<*M2*>_v == y ~= $y + 1$
Enabled<M>_v == (x ~= $x + 1$) \/ (y ~= $y + 1$)

Predicates

Enabled<*alpha1*>_*w* == (*pc1* = *a*) /\ (0 << *sem*)
Enabled<*beta1*>_*w* == (*pc1* = *b*)
Enabled<*gamma1*>_*w* == (*pc1* = *g*)
Enabled<*alpha2*>_*w* == (*pc2* = *a*) /\ (0 << *sem*)
Enabled<*beta2*>_*w* == (*pc2* = *b*)
Enabled<*gamma2*>_*w* == (*pc2* = *g*)
Enabled<*N1*>_*w* == \/ **Enabled**<*alpha1*>_*w*
\/ **Enabled**<*beta1*>_*w*
\/ **Enabled**<*gamma1*>_*w*
Enabled<*N2*>_*w* == \/ **Enabled**<*alpha2*>_*w*
\/ **Enabled**<*beta2*>_*w*
\/ **Enabled**<*gamma2*>_*w*
Enabled<*N*>_*w* == **Enabled**<*N1*>_*w* \/ **Enabled**<*N2*>_*w*

One can refer to these definitions in proofs by the the name of the action prefixed by 'Nab_'; e.g. the definition of **Enabled**<*M*>_*v* is referred to as 'Nab_M'.

Refinement mappings of flexible variables are defined as state functions, using the **Bar** operator. E.g. the refinement mapping of the cached memory example of the TLA report [20, page 47] would be defined in TLP as

Statefunctions

Bar(*memory*(*m*)) == **If** *cache*(*m*) = ⊥ **then** *main*(*m*)
else *cache*(*m*)

When you reason with refinement mappings, TLP generates barred versions of all the used expressions so that you can refer to e.g. the formula **Bar**(*Psi*) if you have specified *Psi* as in the cached memory example.[4] The definitions of the barred identifiers can be referred to by the identifier prefixed by 'Bar_', in the case above thus as 'Bar_Psi'.

6.8 Proofs

Proofs in TLP are written in a structural, natural deduction style, essentially as introduced by Gentzen (see e.g. Prawitz [30] and Scott [31]).[5] A proof in TLP consists of a *head goal* denoted by the keyword **Theorem** or **Lemma** with a unique name for later reference, a goal (see below) that is to be proved valid (with respect to the built-in axioms and rules of TLP and to already proven facts as well as *assumed* facts asserted by the user). Finally it contains a list of substeps and method applications (see section 6.9), starting with the keyword **Proof** and ending with **Qed**. Each substep is denoted by a label <*level*>*item* indicating its *depth* within the proof and its number within the containing step.[6] Like the head proof, it contains a list of substeps and method applications ended by **Qed**. The general outline of a proof is thus as shown in figure 6.5.

[4]To use this in the current version of TLP, you have to explicitly tell the translator to generate barred versions of expressions, as described in section 8.3.1.

[5]Readers familiar with the Larch Prover will also see a strong resemblance to the LP proof style, upon which TLP is heavily based; TLP however uses a much more strict, structural language for the proofs.

[6]This label will be unique with respect to any reference inside the proof, and is usually much more convenient than a traditional label, denoting the step by its *path*, as e.g. 1.3.2.2.5.

```
Theorem theorem-name
  head goal
Proof
...
method application
...
        <1>1 subgoal 1
        ...
        method application
        ...
                <2>1 subgoal 1.1
                ...
                        <3>1 subgoal 1.1.1
                        ...
                        Qed
                ...
                        <3>k subgoal 1.1.k
                        ...
                        Qed
                ...
                Qed
        ...
        method application
        ...
                <2>l subgoal 1.l
                ...
                Qed
        ...
        method application
        ...
        Qed
...
method application
...
        <1>2 subgoal 2
        ...
        Qed
...
        <1>m subgoal m
        ...
        Qed
...
method application
...
Qed
```

Figure 6.5: The general outline of a TLP proof.

The proof goals can be simple TLP formulas, which constitute a goal to be proved with the currently known hypotheses, or they can be either *implication goals* of the form **Assume** *formula-list* **Prove** *formula* or just **Assume** *formula-list*, or *case goals* of the form **Case** *formula*. The latter special goal types are intended for making certain common parts of the natural deduction proof more easily readable as well as significantly more efficient.

An *implication goal* **Assume** $A_1, A_2, \ldots, A_n$ **Prove** B is used for proving the implication $A_1 \Rightarrow (A_2 \Rightarrow \ldots (A_n \Rightarrow B)\ldots)$ by proving $\hat{B}$ in a model where all of the $\hat{A}_i$ are assumed to be true (where $\hat{A}_i$ and $\hat{B}$ are the formulas A_i and B with all free occurrences of rigid variables replaced with *fresh* constants, like in LP [14, page 43]). This resembles multiple use of the *implication introduction* rule of a traditional Gentzen style inference system, sometimes written as

$$\Rightarrow I : \frac{\begin{array}{c}(A)\\ B\end{array}}{A \Rightarrow B}$$

An implication goal being just **Assume** $A_1, A_2, \ldots, A_n$, i.e. without the **Prove** clause, refers to the implication $A_1 \Rightarrow (A_2 \Rightarrow \ldots (A_n \Rightarrow G)\ldots)$ where G is the *current* goal, the (possibly reduced) goal of the surrounding step.

The *case goal* **Case** C uses the same method for proving the implication $C \Rightarrow G$, where G is again the current goal. When a number of case steps with goals **Case** C_1 ... **Case** C_n is succeeded by the built-in method application **By-Cases** (see section 6.9.2), the current goal is proved by proving $C_1 \vee \ldots \vee C_n$, if n is greater than 1, and G under the assumption that $\sim C_1$, if $n = 1$. This resembles multiple use of of the *disjunction elimination* rule of a Gentzen style inference system:

$$\vee E : \frac{\begin{array}{ccc} & (A) & (B)\\ A \vee B & C & C\end{array}}{C}$$

As the *head goal* is the outermost goal, it doesn't make sense to make it neither a **Prove**-less implication goal nor a case goal; thus this is not allowed.

A typical, short proof of an invariant, the proof of the so called *type correctness* theorem for Program 1 of the Increment Example, is shown in figure 6.6. This illustrates the structure of a TLP proof nicely, and contains the three basic goal types. The reasoning is explained in detail in chapter 9, which contains the complete Increment Example. An example of a more deeply nested proof is the the one we used for explaining reasoning with quantifiers in figure 4.5 on page 47. This is redone in the TLP language in figure 6.7. You may notice how the outline made by the indentation of the proof resembles the shape of the deduction tree, rotated 90 degrees clockwise.

6.8.1 Asserted goals

Writing ** in front of a goal, as in the substeps of <1>2 below

Name TPhi

%-**Use** def0
%-**Include** methods

Predicates
TPhi == x in *Nat* /\ y in *Nat*

Theorem TPhi
Phi => [] *TPhi*
Proof

<1>1 **Assume** *InitPhi* **Prove** *TPhi*
Expand *InitPhi* in **Hyp**
Expand *TPhi* in **Goal**
Qed

<1>2 **Assume** *TPhi*, [*M*]_*v* **Prove** *TPhi'*

Activate *TPhi*

<2>1 **Case** *M1*
Expand *M1* in **Hyp**
Instantiate NatRule2 with i_u <- x
Qed

<2>2 **Case** *M2*
Expand *M2* in **Hyp**
Instantiate NatRule2 with i_u <- y
Qed

<2>3 **Case Unchanged**(v)
Expand v in **Hyp**
Qed

By-Cases
Expand *M* in **Hyp**
Qed

<1>3 *TPhi* /\ [][*M*]_*v* => [] *TPhi*
INV1 with pred_I <- *TPhi*, act_N <- *M*, sf_f <- v
Qed

By-Implication
Expand *Phi* in **Hyp**
UseTempFact **Step**1, **Step**3
Qed

Figure 6.6: The proof of $\Phi \Rightarrow \Box T_\Phi$, of the Increment Example.

```
Name Prawitz
%-Use frame
%-Include methods, quantmethods
Rigid Variables
  a, b : Val
Operators
  P : Val, Val -> Bool
Predicates
  Prawitz == /\ Forall x : Exists y : P(x, y)
                           [* praw12(x) *] [* praw11 *]
             /\ Forall x : Forall y : P(x, y) => P(y, x)
                           [* praw22(x) *] [* praw21 *]
             => Forall x : Exists y : P(x, y) /\ P(y, x)
                           [* praw32(x) *] [* praw31 *]
Theorem Prawitz
  Prawitz
Proof
  <1>1 Assume Forall(praw11) /\ Forall(praw21) \
       Prove  Forall(praw31)
    <2>1 Exists(praw32(a))
      <3>1 Exists(praw12(a))
      UseForall on praw11 with a
      Qed
      <3>2 Assume P(a, b) Prove Exists(praw32(a))
        <4>1 P(b, a)
          <5>1 Forall(praw22(a))
          UseForall on praw21 with a
          Qed
        UseForall on praw22[a] with b
        Qed
      ProveExists on praw32[a] with b
      Qed
    UseExists on praw12[a]
    Qed
  ProveForall on praw31
  Qed
Expand Prawitz in Goal
Qed
```

Figure 6.7: The TLP proof representing the deduction of figure 4.5.

<1>2 **Assume** *TPhi*, [*M*]_*v* **Prove** *TPhi'*

<2>1 ** **Case** *M1*
Qed

<2>2 ** **Case** *M2*
Qed

<2>3 ** **Case Unchanged(***v***)**
Qed

By-Cases
Expand *M* in **Hyp**
Qed

tells the back-end to accept the goal without verification. This can be used in incremental proof development where one wants to check the outer goal before the subgoals,[7] or for efficiency, when one wants to avoid re-verification of a large proof that hasn't been changed. When a goal is 'starred', any methods and substeps in the proof of it are ignored by the verification back-end.

6.9 Methods

Directions to the verification back-ends on how to verify a goal is given by application of *methods*. TLP contains a few built-in methods; others can be defined by the user. The user-defined methods are really just *macros* that produce pieces of text to be given to the back-end, although with one important feature: it is possible to specify parameters of different *kind*. The kind of a parameter may be interpreted by the TLP translator, so that e.g. a TLA formula used as parameter is translated with respect to the encoding of the back-end being used and the context where the application appears.

6.9.1 Method applications

A method application consists of a *method name* followed by a possibly empty sequence of *keywords* and *parameters*. The name is the unique identifier of the method. Keywords that follow have to match the keywords that are given in the definition; their purpose is purely mnemonic. Parameters can be of the following kinds:

Name: An identifier containing letters, digits, and the special characters '_' and '*', or alternatively a **Goal**, **Step**, **Hyp**, or **ContraHyp** parameter, as described below.

Name list: A list of name parameters, separated by commas.

TLA expression: A parenthesized TLA expression (see section 6.2).

Instantiation lists: Closely related to the instantiations used in LP, a comma-separated list of pairs *n* <- *t* where *n* is an identifier and *t* a parenthesized TLA expression.

[7] The interactive front-end also supports a 'post-order' mode for doing this automatically, see section 8.4.

Quoted string: A string inside double-quotes (" ... ") containing any ASCII characters other than ".

The parameters given should match the parameter *specifications* in the method definition with respect to their kind, number, and position within the parameter list. Some of the parameter kinds may however match more than one parameter *specification*, as will become clear.

The **Goal**, **Step**, **Hyp**, and **ContraHyp** parameters refer to equations and conjectures in the proof where they appear, with the following interpretations (n and m are arbitrary integers and i the *level* of the step where the method parameter appears):

Goal: The current conjecture.

Step: All previous steps, lemmas, and theorems, in the scope of the method application.

Step<*n*>: All previous steps on level n (i.e. with a label beginning with <*n*>) in the current scope.

Step*m*: The step in the current scope with label <*i*>*m*.

Step<*n*>*m*: The step in the current scope with label <*n*>*m*.

Hyp: The implication or case hypotheses of the current step.

Hyp<*n*>: The implication or case hypotheses of level n in the current scope.

Hyp.*m*: Same as **Hyp**, but referring only to hypothesis number m.

Hyp<*n*>.*m*: Same as **Hyp<*n*>**, but referring only to hypothesis number m.

ContraHyp: The contradiction hypothesis of the current step.

ContraHyp<*n*>: The contradiction hypothesis of level n in the current scope.

6.9.2 Built-in methods

TLP currently supports three built-in *tactical* methods. The purpose of these is to direct the verification back-end into using a special inference technique to complete the proof in which they appear. As TLP use LP as its main back-end, it is not a coincidence that the tactical methods are very similar to the main backward inference methods of LP:

By-Cases: This is the method described in section 6.8 above, which ends a *case* proof (a proof using the disjunction elimination rule) by creating a new goal being the disjunction of all the special cases used above it on the same proof level.

By-Implication: This ends a proof of a current goal being an implication $A \Rightarrow B$, in the same way as if the goal **Assume** A **Prove** B was inserted explicitly (which is really proving the goal by the implication introduction rule).

By-Contradiction: This similarly creates the new goal **False** after assuming $\sim A$, where A is the current goal. This corresponds to the traditional Gentzen style rule *False*:

$$\frac{\begin{array}{c}(\neg A)\\ \text{false}\end{array}}{A}$$

The other built-in methods are:

By-LTL: In spite of the resemblance of the tactical methods above, this is not really a method, but rather a directive telling TLP to handle the current goal by automatic LTL (linear-time temporal logic) checking. As the LTL checker does not need directions, **By-LTL** should always be the *only* method application in a proof step (other applications will be ignored).

Do: This is the only primitive method for producing ordinary text input to the back-end. **Do** takes an arbitrary number of parameters of arbitrary kind, and processes these according to their kind. Parenthesized TLA expressions given as parameters are thus translated according to the encoding of the used back-end, while name parameters and quoted strings are copied directly to the back-end. Name lists and instantiation lists may be treated in different ways with respect to the back-end; the current LP translations generate space-separated name lists and LP instantiations from instantiation lists.

6.9.3 User-defined methods

User-defined methods can be defined anywhere in a TLP file, and used anywhere below the definition. It is very common, however, to put the definitions in a file of their own (often called `methods.tlp`), to be included in the beginning of all files containing proofs (i.e. by the directive **%-Include** methods), allowing you to use the same methods in all proofs, and as an important extra benefit, to change and add definitions of methods while working interactively on a proof.

A method definitions section begins with the header **Methods** and may contain any number of method definitions. A definition begins with a method *header* and ends with the keyword **End**. The header consists of a *unique* method identifier followed by a sequence of *keywords* and *parameter specifiers* ended by the word **is**.

The method *body*, between the header and **End**, is simply a list of method applications, differing from the ones used in proofs by that they may contain references to the method parameters.

Below we show the definitions of two of the most common methods for reasoning with LP, Expand and Instantiate, and a method, UseTempFact, that is often used in the concluding steps of a temporal proof:

Methods

Expand `#nl` in `#nl` **is**
Do normalize `#2` with `#1`
End

Instantiate `#nl` with `#il` **is**
Do instantiate `#2` in `#1`
End

UseTempFact `#nl` **is**
Passify `#1`
Instantiate `#1` with σ `<-` σc
End

The 'in' of the first definition, and the 'with' of the second are keywords, which serve merely mnemonic purposes, making the applications more easily understandable. The parameter specifiers indicate the kind of parameter expected at each position, and thus how parameters should be handled in the translation. Options are:

`#n:` Name, i.e. the name of a definition, rule, equation, conjecture, or hypotheses. When generating LP code, no translation is done on parameters that are ordinary method identifiers (which may contain letters, digits, and the special characters '_' and '*'), while special **Step**, **Hyp**, **ContraHyp**, and **Goal**-parameters are translated according to the context as described in section 6.9.1. For variable and constant names it is more correct to use the `#t` specifier (see below), to ensure correct translation.

`#nl:` Name list, i.e. a list of names as above, separated by commas. Each name is handled as a name parameter, while the list itself may be processed in different ways. In the current LP translations, a space-separated list of the given parameters is generated.

`#t:` A TLA expression. The parameter should be a single alphanumeric identifier, or a parenthesized compound expression. This is always translated according to the encoding in the current back-end. (This means that rigid variables may be instantiated with constants when the application appears inside e.g. a case proof.)

`#f:` A so-called quantifier function name used in connection with quantified, lambda, and set expressions (see section 6.2.7). This differs from an ordinary name in the way it is translated; in LP function names are thus given a special prefix '`f_`'.

`#fl:` A quantifier function name list, separated by commas.

`#il:` An instantiation list. This is closely related to the instantiations used in LP; an instantiation list is a comma-separated list of pairs n `<-` t where n is a name and t a TLA expression, handled as described above (see `#n` and `#t` respectively). The translations in LP contain the appropriate code to instantiate a rule or equation accordingly.

E.g. Expand is defined to take two lists of names (of rules) and, in LP, normalize the rules indicated by the second list with the ones of the first. (If the first list contains names of definitions, this means 'expanding' identifiers in the rules of the second list by these

definitions, thus the name.) In the definitions of Expand and Instantiate, the method body is just a single **Do**-application.

Other examples can be found in the standard methods file supplied with the TLP distribution in the "base" directory, also shown in appendix C.

6.9.4 Using methods outside proofs

Sometimes it's useful to give directions directly to the back-end from within a TLP script. In LP, we might e.g. be interested in making certain facts active, using a non-default ordering, or even asserting facts that we haven't been able to specify otherwise. This is also done by *methods*. When you want to apply a method outside a proof, you need to specify which back-end is to receive the generated output. Currently, you can direct output to the Larch Prover only, by using the method-headers **Act** and **Temp** for action and temporal reasoning respectively. Thus, to specify an alternative ordering of rules in the temporal environment before doing a proof, you could write

```
Temp
  Do "set ordering left-to-right"
End
```

7

The translator

The TLP translator is the the tool that parses the TLP specifications and proofs, checks syntax and sort correctness of constructs and expressions, and generates output for the different verification back-ends. It can be run either from a shell or from the interactive front-end (see chapter 8). When run from a shell, it is called as just 'tlp', taking as parameters a number of options and a list of files to translate. The TLP files should all have the extension '`.tlp`', and the generated output will be written to files with the same name as the input files, but with the '`.tlp`' extension substituted by either '`_act.lp`', '`_temp.lp`', or '`_ltl.t`' depending on which verification back-end the output is aimed for.

The translation of a TLP file is done in a number of passes:

- The file is parsed, creating an abstract syntax tree representing the contents. Files included by an **Include** directive (and files included transitively by those files, etc.) are included as if they were part of the file, at the point of the directive. The parser will try to recover from any syntax errors, as it produces information on the standard output.

- The file contents, represented by the abstract syntax tree, are sort checked (see section 7.2 below). To do the sort checking, we need to know the sorts of identifiers declared in files that the current one depend on, i.e. any files mentioned in a **Use** directive in the parsed file, or, transitively, in a **Use** directive in any of those, etc. We call these files the *ancestors* of the current file. To obtain the sorts of identifiers declared in the ancestor files, we parse each of them, and extract from the generated abstract syntax trees, the *declared* sort information for all declared identifiers. We do not do any sort checking on the ancestors. From the sort checking we generate a table of sort information for all identifiers declared in the parsed file and any of its ancestors. Information about errors is written to the standard output.

- If the file could be parsed and the sort checker didn't detect any errors, code is generated from the abstract syntax tree for each of the back-ends (or those specified explicitly by options given on the command line). The sort information is used to determine which declarations and definitions to generate code for. With the help of the sort checker, using the sort information table, it is determined which parts of a proof belong to action and temporal reasoning respectively, so that the right steps are encoded for being verified by the respective back-ends.

The translator is written in the New Jersey dialect of Standard ML [27, 3].

7.1 Parsing

The parsing is itself done in two passes. The lexer generator ML-Lex [2] and the parser generator ML-Yacc [33] have been used for implementing the first pass. As part of the TLP language, namely the indentation sensitive formulas described in section 6.2.9 on page 66, is not context-free, the Yacc generated parser cannot do the complete analysis itself. Thus we let the first pass generate an intermediate syntax tree, in which indentation sensitive formulas are just indicated with their level of indentation. A second pass then parses this information and generates the final syntax tree.

The Yacc generated parser is able to do some error correction on-the-fly, and produces error information, indicating the position in the input file where the error appeared. The second pass, however, does not produce information about where so-called 'offside' errors appear, but just indicates the kind of error that was detected.

7.2 Sort checking

The translator checks that all parts of a TLP specification are sort correct, so that any identifier declared to be of a given sort is defined to equal an expression of this sort or any of the sorts below it in the sort ordering (see figure 6.1 on page 61). It also checks all uses of parameterized variables and constants, as well as uses of user declared functions, making sure the right number and kind of parameters are being applied.

7.3 Translator options

`(+/-)a`<*directory*>**:** A `+` indicates that the translator should generate output for use by the LP back-end, for doing action reasoning. The translator will not generate any code for definitions containing temporal expressions, and inside proofs it singles out the steps that can be determined as dealing solely with action reasoning and generates code for these alone. If a step contains the method **By-LTL** on outermost level, the code generated will be an *assertion* that the goal of the step is valid, otherwise it will be an attempted LP proof. The output will be written to the file with the same name as the translated file, but with '`_act.lp`' substituted for the original '`.tlp`' extension. If a *directory* is supplied, the file containing the action code is placed there – the directory can be specified relatively to the current one. `+a` is the default, so that without any options, action code will be generated in a file in the current directory.

`(+/-)t`<*directory*>**:** Similarly, a `+` here indicates that code should be generated for the LP back-end, for doing temporal reasoning, and written to a file with the extension '`_temp.lp`', in *directory* if present. The translator generates code for all definitions, declarations etc. All steps that have to do with action reasoning only, or which are proved by the method **By-LTL**, are *asserted* in the output, while ordinary temporal

steps are translated to proofs. `-t` is the default, so that no temporal code will be generated unless you explicitly ask for it.

(+/-)`l`<*directory*>**:** Just as the two previous options, indicating whether to generate code for the linear temporal logic checker. The only code that is generated relates to the goals of steps (either temporal or action) that contain the **By-LTL** method on their outermost level. Output goes to a file with extension '`_ltl.t`', and the default is `-l`.

(+/-)`i`**:** The `i` option indicates whether to automatically generate so-called quantifier rule *instantiations*. This means to instantiate rules such as ProveForall and UseForall, whenever a quantified expression (or set expression) is used. This can be helpful, as it will eliminate the need of explicitly making instantiations by methods inside proofs. It also can cause some problems relating mostly to efficiency in LP, and so it is an approach that has mostly been abandoned. For small and simple proofs it can still be used, if desired. Automatic instantiation applies only to the action reasoning code, and the default is not to generate any instantiations.

(+/-)`b`**:** The `b` option indicates whether to generate *barred* versions of all definitions and flexible variable declarations. This is needed when you have specified and want to reason about a refinement mapping. The barred versions will be exactly the same as the non-barred, but with the prefix 'Bar' added to all non-constant identifiers. Barring applies only to the LP code environments, and the default is not to generate barred versions.

(+/-)`c`**:** In the newer versions of TLP, case steps are proved as deduction rules in LP, solving some problems with the old style, using ordinary implications (with proof by implication). To make sure that old proofs will always work, it is still possible to prove case steps as implications. This is done when you specify `-c`.

(+/-)`r`**:** LP sometimes has problems with finding acceptable orderings of new equations. This can be improved by specifying so-called *registering hints*. These assist LP in giving the identifiers different 'heights', by specifying pairwise relationships. If an identifier f is higher in the ordering than g, then LP will try to order equations containing f and g on either side into rewrite rules that replace the expression containing f to the expression containing g. The translator usually does this automatically by assigning different levels to different kinds of identifiers. Sometimes, however, adding registering hints can be a burden, making LP spend enormous amounts of time on otherwise simple orderings. `-r` turns the generating of registering hints off so that this can be avoided. The default is to generate the hints.

(+/-)`p`**:** The `p` option specifies whether, in LP, newly proved theorems and lemmas should be made 'passive' and 'immune'. `+p` is the default, as keeping as many facts as possible from interfering with the reasoning you are doing is the recommended way to work in TLP. When in a proof you need to use a lemma that you proved before, you will know that you need it, and indicate this by instantiating it or making it active at that point in the proof. Making all theorems and lemmas active (and non-immune) can make some things go through with less instructions, but also often

makes the verification slower and harder to understand, and sometimes even breaks your line of reasoning by rewriting things you didn't want rewritten.

(+/-)o: This last option is only useful inside the TLP interactive front-end. +o specifies that proofs should be verified depth-first, pre-order, meaning that you verify each step from its substeps, before you verify the substeps, while the default, -o, specifies that proofs should be verified depth-first, post-order, meaning that the substeps are always verified first. Post-order is the way LP does things by 'birth', while pre-order is accomplished by a more complicated translation, where the substeps of a proof are first asserted, the outer step verified, after which the proved facts are deleted and the whole thing redone, now, recursively, first verifying the substeps in pre-order.

8

The interactive front-end

The TLP interactive front-end is the place where you will usually write and edit your specifications, and incrementally develop and maintain your proofs. It is truly interactive, meaning that you can interact with the TLP translator, correcting syntactical and semantical errors, and with the verification back-ends, discovering in a step-by-step manner what information needs to be conveyed to have your theorems successfully verified.

It is possible to use the TLP tools without the interactive front-end, by simply calling the translator from a UNIX shell, and by feeding the generated output to e.g. LP, running as a batch job – this was the only way to use TLP in its first release. The front-end however makes things a lot easier, and when constructing large proofs is indispensable.

The TLP front-end is built on top of GNU Emacs, the "GNU incarnation of the advanced, self-documenting, customizable, extensible real-time display editor Emacs" [32]. Emacs is first of all an editor that makes it possible to write and edit TLP specifications and proofs, but furthermore is useful as an application platform on which it has been possible to build an advanced interface to the TLP translator and the verification back-ends. The TLP front-end is even more than this, as it also handles the linking of groups of TLP files depending on each other, assisting you by re-translating and re-verifying the necessary files in the right order, in all different verification modes, whenever any changes have been done. It should be noted that you need to be acquainted with Emacs to be able to use the features of the TLP interface fully. Some knowledge of the basic Emacs concepts is therefore also required for understanding this chapter.

8.1 Getting started

The TLP front end is started simply by loading a TLP file into Emacs, or by opening a new file with the extension '`.tlp`'.[1] Emacs will switch to the major mode "tlp-mode", indicating this by saying something like "`(TLP:act)`" in the mode-line. You can now edit your specifications and proofs as in any ordinary Emacs buffer, having at your disposal a few extra features and commands for moving around, inserting common constructs, indenting and filling sections of proofs, etc. (see the section on editing commands below).

[1] TLP should be properly installed, and Emacs set up so that it automatically switches to tlp-mode whenever a TLP file is loaded – see appendix E for instructions on how to do this.

When you are ready to start reasoning with TLP, you just type 'C-c C-c'[2], executing the function 'tlp-command' on the file visited in the active buffer, which we will refer to as the *master-file*. Emacs will then ask you for a command to perform, indicated by a unique letter. If you type a question mark or press the space-bar, Emacs will open a 'completions' buffer, telling you which commands are available, as shown in table 8.1.

Command	Effect
a - verify all modes	*Starts verification of the master file successively going through all available verification modes.*
m - make test	*Just shows what would be done if you executed a verification command.*
p - toggle verification mode	*Switches to the next mode of verification (the modes are action reasoning, temporal reasoning, and LTL checking).*
q - quit	*Quits any ongoing verification.*
r - relabel proofs	*Relabels the proofs in the current buffer.*
t - translate current mode	*Translates the file, displaying any syntax and sort errors.*
v - verify current mode	*Starts verification in the current verification mode.*

Table 8.1: The main TLP commands

If you type a 'v' followed by 'return', a verification session translating and verifying the specifications and proofs of the file you are visiting will be started. You will be asked to correct syntax and sort errors, and eventually, to supply information to the back-end, assisting it in successfully verifying your proofs. Whenever the back-end gets into trouble, trying to verify a step, it will stop, supply you with information on the current proof status, and let you do any changes you think necessary to make the step go through.

8.2 Editing TLP files

The primary objective of the Emacs TLP mode is to provide an environment for interactive reasoning. The mode, however, also contains a number of commands and functions that make editing TLP files an easier task. The major commands that are available are listed in table 8.2; other commands can be found by typing 'C-h b' in a TLP buffer, listing the mode-specific key bindings. First of all there are commands for indenting and 'filling' TLP specifications and proofs according to the TLP standards. Changing these standards is possible through an array of variables declared in the file `tlp-local.el` under the *Indentation* header. A few commands are available for moving around in proofs, which is helpful when working on large proof steps. Most useful of them all is probably the

[2] 'C-c' is standard Emacs nomenclature, meaning 'control-c', i.e. the action where you press 'c' holding down the 'control'-key.

command invoked by typing 'C-c C-d' for inserting complete syntactical units, using the *dmacro* package by Wayne Mesard, supplied as part of the TLP release.

Key event	Effect
tab	*Indent the current line according to the standard TLP indentation style.*
M-C-q	*Indent the current step.*
M-q	*Fill the current paragraph. Works properly for indented comments, which is very helpful when writing 'literate' proofs, as e.g. the TPhi proof of the Spanning-Tree Example supplied with the TLP release.*
M-a	*Move to the beginning of the current step. Will move to the previous or encapsulating step, if at the beginning of this one. The beginning is defined as the beginning of the line containing the "Theorem" or label.*
M-e	*Move to the end of the current step, or more precisely, to the beginning of the first line after the current step. Will move to the next substep of the current step if such exists. The end of the current step is defined as the end of the line containing the "Qed" of the current step.*
M-C-h	*Mark the current step as the Emacs region, moving to the beginning.*
C-c C-d	*Insert a syntactical unit, using one of the TLP dmacros.*

Table 8.2: Special TLP editing commands

8.2.1 Using the dmacro package

The dmacro package must be properly installed on your system for TLP to use it. See appendix E for instructions on how to do this. When this has been done and you have declared the tlp-dmacro variable in `tlp-local.el` with the initial value 't', you will be provided with a set of commands that inserts pieces of syntax with or without interaction.

Already when you enter a new, empty file, with the necessary '`.tlp`' extension, you will see an effect of using the dmacros, in that a *header* is inserted in your buffer, looking something like

```
% Filename:    phi.tlp
% Created:     5 September 1995

Name

%-Use
%-Include
```

with the cursor positioned right after the **Name** specifier. If you want to change the default header, you can do so by making a private copy of `tlp-dmacro.el` and edit the string defining the 'tlphead' macro.

When you are editing a TLP file, you insert syntactical units by typing 'C-c C-d' executing the TLP insert dmacro command. This will always ask you for a macro to insert, by displaying "Dmacro: " in the minibuffer. If you type a question mark or hit the space bar, a completions buffer will be opened listing the available macros, currently the ones shown in table 8.3. Typing one more question mark also shows a documentation

Macro	Documentation
INV1.1	*Insert an instantiation of the INV1 rule (without assumptions).*
INV1.2	*Insert an instantiation of the INV1 rule (with an action assumption).*
INV1.3	*Insert an instantiation of the INV1 rule (with a predicate assumption).*
INV2	*Insert an instantiation of the INV2 rule.*
SF1	*Insert an instantiation of the SF1 rule.*
SF2	*Insert an instantiation of the SF2 rule.*
WF1	*Insert an instantiation of the WF1 rule.*
WF2	*Insert an instantiation of the WF2 rule.*
actcode	*Insert some LP-code for the action reasoning back-end.*
action	*Declare an action.*
aproof	*Start a proof.*
assumpstep	*Insert a new* **Assume Prove** *step on the current level.*
bcomment	*Insert a bracketed comment.*
casestep	*Insert a new* **Case** *step on the current level.*
elcomment	*Insert a comment at the end of the current line.*
include	*Insert an* **Include** *directive.*
lemma	*Propose a lemma.*
method	*Declare a method.*
predicate	*Declare a predicate.*
starproof	*"Star" the current proof.*
statefunction	*Declare a statefunction.*
step	*Insert a new proof step on the current level.*
tempcode	*Insert some LP-code for the temporal reasoning back-end.*
temporal	*Declare a temporal expression.*
theorem	*Propose a theorem.*
tlphead	*Insert a TLP header.*
transitionfunction	*Declare a transition function.*
use	*Insert a* **Use** *directive.*

Table 8.3: TLP dmacros

string, stating what each macro does. You should then type the first significant letters of the macro you would like inserted, followed by a 'return'.

Macros for inserting TLA rules are probably the most helpful, as you don't have to remember what names are used for the different variables, and as they insert a description of the rule in a comment, displaying the derivation you are performing. The macros for inserting substeps are also quite handy, as they always give the newly inserted step the

correct label, when inserted in the right context. You can add new macros or change the existing ones by editing your private copy of `tlp-dmacro.el`.

8.2.2 Relabeling files

The relabeling facility is meant for resetting the labels of all proof steps, whenever changes have been done that make these incorrect. If you, for instance, move a step from one place to another, or copy a substep of one proof to another, you just need to use the relabeling command to get all the labels updated correctly. The most significant advantage of using this function, besides saving you time by doing the trivial updating of the proof labels themselves, is that it also remembers to change references to steps and hypotheses inside method calls, so that proofs should continue to work. The way this works is by replacing references such as **Step**<*i*>*k* and **Hyp**<*i*> by **Step**<*j*>*l* and **Hyp**<*j*>, if the reference before the relabeling referred to a step with label <*i*>*k* which was changed to <*j*>*l*; similarly for references such as **Step***k* and **Step**<*i*>.

Relabeling is done with the command 'tlp-relabel', which is invoked by typing 'C-c C-c r return'.

8.2.3 Highlighting TLP keywords

With the hilit19 package that is a part of Emacs 19 (the newest Emacs release) it is possible to have files *highlighted* in different ways, based on the file syntax. By highlighting is meant some sort of *emphasizing* of keywords and syntactical constructs, either by use of different fonts or, on displays that support this, by different colours. The TLP front-end comes with a setup for highlighting keywords such as **Predicates**, **Actions**, **Proof**, as well as proof labels and comments. It can make proofs a lot easier to read and is therefore recommended. Its use is automatic when the installation is done as requested.

8.3 Translating TLP files

The translation command does not do any verification, but can be used to check the syntax and sort correctness of specifications and proofs. Executing the translation command starts the TLP translator, which parses and sort checks the master file and generates output for the current verification mode. Emacs first splits the frame into two separate windows, with the master file displayed in one and the so-called 'make' buffer displayed in the other. The TLP 'make' buffer is used for showing what actions are performed during translation and verification, and also shows any information produced by the translator.

If the file in question contains any **Use** directives (as most do) the files quoted by these, and files quoted recursively by directives in the used files, will be parsed as well. This is something the translator does automatically for figuring out the sorts of declared and defined identifiers, that are used in the master file. If there are **Include** directives, the quoted files are included, and thus also parsed.

If syntax errors are found in one or more of the parsed files, Emacs will switch to the file containing the first error, put the cursor at the place were the error seems to be, and, if possible, highlight the lexeme that the translator didn't like. It should also tell you

what the problem seems to be in Emacs' minibuffer. You are then able to edit the file, and when done you can go to the next error by pressing 'M-n'[3], and so forth.

The translator is not yet very good at locating syntax errors that appear to be caused by wrongly indented formulas. When such errors are found, Emacs will instead switch to the 'make' buffer, displaying the error message of the translator. This should hopefully be enough for you to figure out what the problem is and to fix it.

If sort errors are detected, Emacs also switches to the 'make' buffer, the translator giving a detailed description of what was wrong. A sort error will appear in any expression that cannot be sort-checked correctly with respect to the sort it is declared to have, or should have from the context. Undeclared variables, constants, or operators also lead to sort errors, as the sort of the used identifier is not known to the sort checker.

8.3.1 Options to the translator

To change the verification mode, so that the translator will generate output for one of the other modes (and back-ends) you should use the command 'tlp-toggle-sort', which can be done inside TLP buffers by typing 'C-c C-c p return'.

The other options to the translator are also reachable from inside the front-end: 'tlp-instantiating' is the command for toggling the '-i' option, which allows you to have quantifier functions instantiated automatically; 'tlp-barring' toggles the '-b' option, controlling the generating of barred definitions (for refinement mappings); 'tlp-registering' toggles the '-r' option, controlling the generating of registering hints in LP; and 'tlp-immunizing' toggles the value of the '-p' option, controlling whether to make new theorems immune inside LP. The command 'tlp-proof-order' switches between 'pre-order' and 'post-order' verification, which is only interesting while doing interactive verification; this is described in section 8.4.3. Commands like these are all invoked from an Emacs buffer by typing 'M-x' followed by the command name.

The translator options can also be set up to have different default values than the ones used by the translator and the front-end. To do that you should edit the file `tlp-local.el` as described in appendix E.4.

8.4 Verifying TLP files

Executing a verification command, either 'verify current mode' or 'verify all modes' starts a TLP *make session*. At the same time, it makes all TLP buffers switch to *verification mode*, which is signified on the mode-line, by the addition of '-verifying' right after the verification mode indicator.

When starting a make session, the front-end will detect if the master file or any of its ancestors (files that it depends on, through the **Use** directive) have changed since their latest verification, and act accordingly to this information. Successfully verifying the file means that it and all the files it depends on are verifiable in their current state, meaning that the files are all syntax and sort correct, and that all contained proofs are verifiable with the given assumptions. Verifying in 'current mode' means verifying only parts of the proofs, namely the parts that are to be verified in the verification mode specified

[3] 'M-n' means 'meta-n', which is typed either by holding down the 'meta'-key and pressing 'n' or by pressing 'escape' and then 'n'.

by the variable tlp-verification-mode and shown on the Emacs mode-line. The existing modes are 'act' for action reasoning with the Larch Prover, 'temp' for temporal reasoning with the Larch Prover, and 'ltl' for linear temporal logic checking. Verifying 'all modes' means verifying in all existing modes, one by one, thus ensuring that everything has been completely verified.

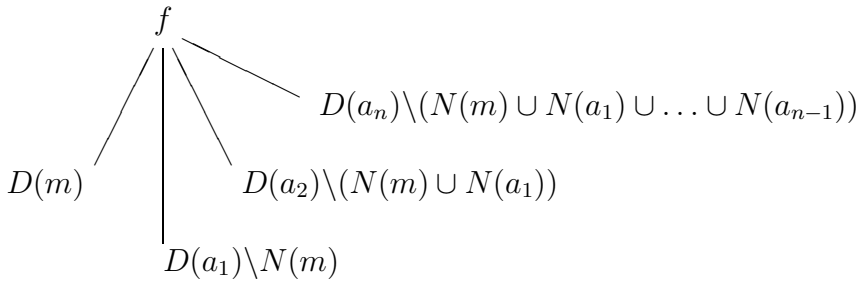
8.4.1 The make session

The way a make session is performed is not completely obvious. When we want to verify a file, we first create a tree describing the dependencies of the file. Let $D(m)$ be the dependency tree of the file m and $N(m)$ the set of nodes that appear in the dependency tree $D(m)$. Also, let $T\backslash\{a_1, \ldots, a_n\}$ be the tree T from which we have cut off all branches to subtrees, the roots of which are in the set $\{a_1, \ldots, a_n\}$ (if the root of T is itself in the set, then $T\backslash\{a_1, \ldots, a_n\}$ is the empty tree).

If the file we want verified is called f, and its **Use** directive looks like

%-Use $m\ a_1\ \ldots\ a_n$

then its dependency tree, $D(f)$ is defined recursively as



where we ignore any of the root's n rightmost branches that point to empty subtrees. We will refer to m as the *mother* node of f, and to $a_1, \ldots, a_n$ as its *aunts*. Now, with this dependency tree, we are able to describe what should be done to make the verification of f up to date. First, we should verify m recursively. When this has been done, we should look for a frozen image of f. If an image exists, and

- the image is newer than f itself;
- the image is newer than the image of the mother file, m, (which must now exist); and
- the image is newer than all files of the aunt subtrees, the subtrees directly below f with roots $a_1, \ldots, a_n$;

then we are finished, finding that the image file is up to date, and that no new verification of f thus has to be done. If, on the other hand, any of the above conditions does not hold, we should re-translate and verify the file, creating a new image. How this should be done is different in the different verification modes. Thus, when using the LTL checker, we just have to re-translate f and run the checker on the output (the LTL checkable parts of a proof do not depend on formerly proved facts etc.). When using the Larch Prover, the verification may depend on facts and assumptions from any of the files in the dependency

tree, however, so we need to verify these first. This is done by first loading the image of the mother file and then executing the files of the aunt subtrees (that f depends on). The files of the aunt subtrees are executed in the order we get by going through the subtrees successively from left to right, visiting each subtree in a depth-first, post-order fashion. Each file we encounter is re-translated and executed in the ongoing Larch Prover session. Finally, we re-translate and execute f, making a frozen image when and if the execution has been successfully completed.

If the file contains a

%-Bottom

directive, this means that none of the files in the **Use** directive should be interpreted as the mother node – they are all aunts in the sense described above. This means that even though there exists a fresh image of the the file that would otherwise be interpreted as the mother file, the translation and verification of this file will always be redone. The advantage, on the other hand, is that no images of the mother file or any files that it depends on will be created in the process. As images created by the Larch Prover have sizes in the order of megabytes, this may be desirable especially for avoiding creation of images for the most basic files, the execution of which will in any case be very fast.

8.4.2 Verification interaction – building proofs

It is possible to write correct, verifiable proofs from scratch and have them verified as 'batch' jobs from inside the TLP front-end. If the proofs contain much more than a few steps, it is a very difficult task, though. More common is to build the proofs step by step in a top-down, bottom-up, or mixed style, interacting with the back-end to better understand what is needed to make the verification go through.

The TLP front-end is built to provide an interactive environment, in which you are able to do just this. A common way of building a proof is to begin with nothing more than a proof outline, containing two or three substeps that you know will be needed. You then start verification, by executing e.g. the 'verify current mode' command. After making sure that all files that the master file depends on are verified and up to date, the master file is translated and verification is begun, sending the output to the mode-specific back-end. When the back-end reaches a step that it cannot verify successfully with the information currently accessible, verification is stopped, and a message describing the problem is printed in the Emacs minibuffer.

Emacs will then split the frame into three windows (sometimes just two, depending on your default setup). The first one will contain the TLP file in which the problem appeared. This is usually the master file itself, but could also be any of the files that it depends on, given that these were changed since their last verification, or never verified at all. We will refer to this file as the *source file*. The second window displays the so called *proof correction* buffer and is the place where any changes and additions to the proof should be done – when in verification mode, no direct changes to any of the files involved are allowed. The third window, if present, displays the output from the back-end in use, and will be used for interactive purposes when trying to get the proof completed – this is where you will get information about what hypotheses have currently been assumed, etc.

The proof correction buffer may contain part of the proof where a verification problem was encountered. Whether it does this depends on the verification mode and the kind of

the problem.

Problems in 'act' and 'temp' mode:

There are a number of different error messages that you may see.

Step could not be verified: The Larch Prover is not able to reduce the current goal to true, using the active rewrite rules and deduction rules. The line in the source buffer containing the **Qed** that LP wasn't able to confirm, will be marked by either a verification error arrow `*>`, or by a different background colour, if your system supports this. The proof correction buffer will be left empty, for you to add new method calls or substeps to the current one. To figure out how to go on, you can get information about the current proof status, the (reduced) goal that should be proved, the (reduced) hypotheses that have been assumed at the current point in the proof, etc., by issuing *display* commands from any of the TLP buffers.[4] These are invoked by typing the usual 'C-c C-c', which will now give you a new list of possibilities, as shown in table 8.4. The commands execute standard LP display commands and output listings of rewrite and deduction rules that are currently known to LP. The output is thus in the format of the LP encoding of TLP expressions, being different for action and temporal reasoning. A translation back to TLP expressions together with a system that could better select the right hypotheses etc. for display would be preferable but has not yet been built.

Now to what you can do. If you find that the step should be verified by adding one or more methods before the **Qed**, you should just write these in the proof correction buffer. You can also add new substeps, or do both, just as you would type the contents of a proof step in the major (non-verifying) mode.

If you on the other hand find that something has been done wrongly, that some methods of the step should be deleted or changed, or if the goal itself is wrong, you should ask TLP to *widen the proof scope*. This will tell LP to forget about what it was you were currently trying to verify, and will let you edit the complete step where the problem appeared – thus *widening* the area, or scope, in which you are able to work. The proof correction buffer will reflect this by now containing the complete step. The new scope will also be marked in the source buffer by moving the verification error arrow to the beginning of the step, or by colouring the complete step. The widening command 'tlp-widen-correction' is normally executed by typing either 'C-c C-c' followed by a 'w' and return, or just 'C-c C-w' for short. Consecutive widening commands will widen the scope level by level, letting you do changes to the whole proof structure without having to restart verification.

When you are done with your changes, you resume the verification session, continuing from the beginning of the current proof scope, by the 'execute-correction' command, invoked by 'C-c C-c x return' or simply 'C-c C-x'. This will ask you if you want your changes committed, to which you would usually answer with a 'y' for yes. The source buffer will then be updated with your changes, and the verification session resumed. In the standard setup, the changes in the source buffer will also

[4]Right after the problem is detected, the goal that could not be properly verified will always be shown in its reduced form in the Larch Prover buffer.

automatically be indented accordingly to the context, relabeled (see section 8.2.2), and rehighlighted.

If you want to save some changes without resuming, you can do so by 'tlp-commit-correction', typing 'C-c C-c c return'. You have to do this e.g. when you want to widen the proof scope and don't want your changes to be lost.

If you don't know how to solve the problem and just want to quit the verification session, you should do so by typing 'C-c C-c q return' or just 'C-c C-q', executing the command 'tlp-quit-correction'.

Contradiction could not be verified: You have tried to do a proof by contradiction (using the **By-Contradiction** method). The Larch Prover, however, is not able to get a contradiction from the hypotheses. Your options are as in the case of an ordinary step not being verified.

Cases could not be verified as exhaustive: You have attempted to prove a step by considering a number of cases, finally applying the method **By-Cases** which implicitly uses the disjunction elimination rule of natural deduction. The Larch Prover, however, is not able to show that the cases you have chosen are exhaustive, meaning that the disjunction of all the case hypotheses reduces to true. Again you should continue as in the case of an ordinary step not being verified.

Step has been verified – expecting a Qed: A step was verified at some point where you did not expect it to be. The Larch Prover is not able to discard information that it doesn't need to get a step verified and will complain if there is any disagreement about when a proof should be complete. You may have forgotten that some rule is already active, or you may just have overlooked some trivial fact. When this happens, the proof scope is automatically set to the part of the step that seemed to be superfluous, the proof correction buffer thus displaying the steps and methods that LP didn't need. Often, you would just delete the contents, accepting what TLP tells you, and resume the session by the execute correction command. As usual, you can also investigate the current status of the proof and if needed, widen the proof scope to do changes. If you choose not to do anything, you can just resume verification as usual, and TLP will forget about the superfluous part this time. If the rest of the proof gets verified, you still end up with a proof that is correct – but re-verifying it will again complain about the extra instructions.

Don't understand *<text>* or **perhaps a missing %-Use directive:** Some method that you used generated input to LP that it didn't understand. You may have mistyped a named fact or, as the second error message proposes, tried referring to a rule that has not been loaded (e.g. referring to a quantification rule, forgetting to 'use' `quant_act.lp`).

Problems in 'ltl' mode:

The formula is not valid: In LTL checking mode there is only one verification problem, as long as the formula that is being checked is small enough to be handled by the system, namely that a step is found that was not valid. When this happen, the

proof scope is automatically widened to contain the erroneous step, so that this can be changed and a new try made.

BDD node limit exceeded: This is an internal error message from the LTL checker. The formula can not be successfully validated, as it involves too many states to be handled by the system.

If you commit new syntax or sort errors while editing the proof correction buffer, you will be notified of this as usual. You will be asked to fix the errors inside the proof correction buffer, so that you don't need to quit the verification session.

Command	Effect
x - execute correction	*Commit the changes, or insert the additions, as shown in the proof corrections buffer, and resume verifying the buffer.*
c - commit correction	*Just commit the changes or additions, so that the TLP buffer is up to date.*
w - widen scope	*Widen the proof scope one level, letting you edit the complete proof step surrounding the point where you are currently editing.*
q - quit verification	*Discard the latest changes and quit the verification session.*
h - display hypotheses	*Display the (possibly rewritten) hypotheses of the currently attempted implication, case, or contradiction proof, in the LP buffer.*
p - display proof	*Display the proof status in the LP buffer.*
g - display goal	*Display the current (possibly rewritten) goal to be proved, in the LP buffer.*
f - display facts	*Display all facts proved until know, together with all hypotheses.*
a - display all	*Display all facts currently known to the Larch Prover.*
d - display	*Query the user for some named fact or proof rule to display.*
l - execute LP-code	*Query the user for some LP code to execute, without any interference, in the LP buffer.*

Table 8.4: The TLP verification mode commands

8.4.3 Pre-order verification

Usually, when we verify a TLP file, the commands supplied to the verification back-end are issued in such a way that the conjectures and sub-conjectures will be verified one after the other in a straightforward fashion following the order in which they appear in the file. This also means that we always try to verify sub-conjectures before the conjectures that rely on them – a process that we usually refer to as bottom-up.

When working on proofs it is often more convenient to use a top-down style of development, where we assume that sub-goals are provable as we are verifying the containing step, and postpone the verification of the sub-goals till after the main step has been finished.

Fortunately it is possible in LP to assert the validity of formulas, so that we may indeed use it to simulate a top-down proof style as described. We simply let the translator generate output in which each sub-step is first asserted and the containing step carried out, after which all facts are deleted and the substeps (and then, once again, the containing step) verified. This verification process is clearly more complicated, with most steps being verified twice. The important point, however, is that the user will see no extra work being done, and will only have to worry about proving each step once. The front-end will work exactly as usual; the only difference being that when a verification problem occurs, not all steps *above* the point where the problem appeared will necessarily have been successfully verified. On the other hand, all steps *enclosing* the problematic one will have been successfully verified *on the basis* of the current one.

Viewing the proof as a tree, the usual, bottom-up style of verifying the proof corresponds to a depth-first traversal in which the nodes (i.e. conjectures) are visited in a post-order fashion. We thus call this *post-order verification*. Top-down style verification similarly corresponds to a depth-first traversal where nodes are visited in pre-order, why we refer to it as *pre-order verification*. It is possible to switch between pre- and post-order verification with the 'tlp-proof-order' command.

It should be noted that pre-order verification is only useful when interactively developing a proof; when the proof has been finished, or if any problems appear, post-order verification should be used as being the more efficient and reliable.

Part III

Examples

9

The Increment Example

The Increment Example is the example known from Lamport's TLA Report [20]. This example was also described in chapter 3. Here we present the different TLP specifications, the invariant proofs, and eventually the proof that Program 2 implements Program 1. Going through all the details from the first invariance proofs of the initial specification to the correctness of the its refinement, this example nicely illustrates the width of the TLP system. It as well gives a reallistic picture of the relation between action and temporal reasoning. Although most of the proofs are concerned with the final *liveness* theorem of the refinement, little reasoning is done outside the action reasoning environment. This serves to illustrate some of the strength we gain from splitting the TLA encodings.

The Increment Example relies on a set of LP rules for reasoning about natural numbers. We don't include those here, as we want to focus on the way we do reasoning within TLA/TLP, while the arithmetic could be done in several ways. Whenever we use one of the rules, we will however describe what it does. The natural numbers file is **Use**'d by the frame file for action reasoning, which is also not included here.

9.1 The TLP specification of Program 1

We begin with the declarations of values, operators, and variables that we are going to use in our specifications.

`declarations.tlp`:

Name Declarations

`%-` **Use** frame

Values
0, 1, 2
Nat

Operators
`+, -` : **Val, Val -> Val**
`<<` : **Val, Val -> Bool**

Variables
x, y : **Val**

'<<' is here the 'less than' operator – we can't use just '<', as this is a reserved symbol in TLP.

Then follows the specification of Program 1, which is defined under the name Def0. The specification is straightforward, and identical to the one from the TLA Report, shown in figure 3.3 on page 27.

`def0.tlp`:

Name Def0

%-Use declarations

Predicate

InitPhi == (*x* = 0) /\ (*y* = 0)

Actions

M1 == (*x'* = *x* + 1) /\ (*y'* = *y*)

M2 == (*y'* = *y* + 1) /\ (*x'* = *x*)

M == *M1* \/ *M2*

Statefunction

v == (*x*, *y*)

Temporal

Phi == *InitPhi* /\ [][*M*]_*v* /\ **WF**(*v*, *M1*) /\ **WF**(*v*, *M2*)

9.2 The safety proof of Program 1

The first part of the safety proof is the proof of the basic invariant that we sometimes refer to as 'type correctness', *TPhi*, which says that x and y are in the set of the natural numbers. This is a standard example of an invariance proof, thus containing three substeps: <1>1 showing that *InitPhi* implies the invariant, <1>2 showing that the actions maintain it, and <1>3 concluding from <1>2 that the conjunction of the invariant and the formula stating that we always execute those actions only implies that the invariant is always satisfied. From <1>1 and <1>3 we conclude that *Phi* implies always *TPhi*.

Step <1>1 is verified by action reasoning, and is done in the usual natural deduction style, by assuming that *InitPhi* is satisfied (by some arbitrary state) and showing that *TPhi* must then be satisfied as well (by the same state). *InitPhi* is thus a hypothesis of the current step, and *TPhi* the goal. The proof is done by expanding the definition of *InitPhi* in the hypothesis, and that of *TPhi* in the goal. The goal then (in LP) becomes $in(x, Nat)$ & $in(y, Nat)$ while the hypothesis becomes $x = 0$ & $y = 0$.[1] LP automatically transforms the hypothesis into two rewrite rules $x \rightarrow 0$ and $y \rightarrow 0$, whereby it rewrites the goal to $in(0, Nat)$ & $in(0, Nat)$. The rule NatsBase2 asserts that 0 and 1 are both natural numbers, and thus finally rewrites the goal to true.[2]

Step <1>2 is verified in the same way, by assuming *TPhi* and $[M]_v$ and showing *TPhi'*. As the second hypothesis is a disjunction, $M1 \vee M2 \vee Unchanged\ v$, we split the proof into cases representing each disjunct. We expand all relevant definitions and, in the cases

[1] More precisely, the LP goal (hypothesis) is the mentioned goal (hypothesis) followed by '== true'; we will consistently use the shorter form when describing LP terms.

[2] Again, what it really rewrites to is an identity, 'true == true'.

of *M1* and *M2*, perform the proof by the help of NatRule2, which states that whenever i_u is a natural, then so is $i\_u + 1$.

`tc0.tlp`:

Name TPhi

`%-` **Use** def0
`%-` **Include** methods

Predicate
TPhi `==` x in *Nat* `/\` y in *Nat*

Theorem TPhi
Phi `=>` `[]` *TPhi*
Proof
`<1>`1 **Assume** *InitPhi* **Prove** *TPhi*
Expand *InitPhi* in **Hyp**
Expand *TPhi* in **Goal**
Qed

`<1>`2 **Assume** *TPhi*, `[`M`]_`v **Prove** *TPhi'*

Activate *TPhi*

`<2>`1 **Case** *M1*
Expand *M1* in **Hyp**
Instantiate *NatRule2* with i_u `<-` x
Qed

`<2>`2 **Case** *M2*
Expand *M2* in **Hyp**
Instantiate *NatRule2* with i_u `<-` y
Qed

`<2>`3 **Case Unchanged(**v**)**
Expand v in **Hyp**
Qed

By-Cases
Expand *M* in **Hyp**
Qed

`<1>`3 *TPhi* `/\ []`[*M*]`_`v `=> []` *TPhi*
INV1 with pred_I `<-` *TPhi*, act_N `<-` *M*, sf_f `<-` v
Qed

By-Implication
Expand *Phi* in **Hyp**
UseTempFact **Step**1, **Step**3
Qed

Step `<1>`3 has to be verified within the temporal reasoning environment, and is done so by simple application of the *INV1.1* rule, a version of the TLA INV1 rule that in LP

looks as

assert
when (**forall** σ)
$\sigma \models (pred\_I \wedge (BAct(act\_N, sf\_f) \Longrightarrow Prime(pred\_I)))$
yield $\sigma \models ((pred\_I \wedge \Box(BAct(act\_N, sf\_f))) \Longrightarrow Box(pred\_I))$

To improve readability, we will in the following present rules like this in the format used in appendix B:

$$\frac{\models I \Longrightarrow ([N]_f \Longrightarrow I')}{\models (I \wedge \Box([N]_f)) \Longrightarrow \Box(I)}$$

We apply the rule by using the method INV1, which instantiates the rule with *TPhi* for I, M for N, and v for f.

We now in steps <1>1 and <1>3 have two lemmas stating the validity of the formulas that *InitPhi* implies *TPhi*, and that *TPhi* and $\Box[M]_v$ implies $\Box\, TPhi$. Validness is expressed in the temporal reasoning encoding by stating that any behaviour, indicated by a free variable, satisfies the formulas. From these lemmas we are able to deduce the goal by expanding the definition of *Phi*. There are many ways to do this part; we choose the usual natural deduction style, by explicitly stating that the proof should be done '**By-Implication**'. As this is temporal reasoning, this here means that we assume an arbitrary behaviour by which *Phi* is satisfied, and show that this also satisfies $\Box\, TPhi$. The method UseTempFact instantiates the goals of <1>1 and <1>3 to get that these are also satisfied by that significant behaviour. The rest is simple logic which is handled automatically by LP. This concludes the first invariant proof.

9.3 The specification and safety proof of Program 2

We continue with the specification of Program 2, which again is a straightforward transcription of the specification in the TLA Report which is also shown in figure 3.5 on page 29. First we declare the two new program counter variables, *pc1* and *pc2*, and the values they may take, a, b, and g (for steps *alpha*, *beta*, and *gamma*, respectively).

`def1.tlp`:

Name Def1

%-Use tc0

Variables
pc1, *pc2*, *sem* : **Val**

Values
a, b, g

Predicate
InitPsi == /\ (*pc1* = a) /\ (*pc2* = a)
/\ (x = 0) /\ (y = 0)
/\ *sem* = 1

Actions

alpha1 == /\ (*pc1* = *a*) /\ (0 << *sem*)
/\ *pc1'* = *b*
/\ *sem'* = *sem* - 1
/\ **Unchanged**(*x*, *y*, *pc2*)

beta1 == /\ *pc1* = *b*
/\ *pc1'* = *g*
/\ *x'* = *x* + 1
/\ **Unchanged**(*y*, *sem*, *pc2*)

gamma1 == /\ *pc1* = *g*
/\ *pc1'* = *a*
/\ *sem'* = *sem* + 1
/\ **Unchanged**(*x*, *y*, *pc2*)

alpha2 == /\ (*pc2* = *a*) /\ (0 << *sem*)
/\ *pc2'* = *b*
/\ *sem'* = *sem* - 1
/\ **Unchanged**(*x*, *y*, *pc1*)

beta2 == /\ *pc2* = *b*
/\ *pc2'* = *g*
/\ *y'* = *y* + 1
/\ **Unchanged**(*x*, *sem*, *pc1*)

gamma2 == /\ *pc2* = *g*
/\ *pc2'* = *a*
/\ *sem'* = *sem* + 1
/\ **Unchanged**(*x*, *y*, *pc1*)

N1 == *alpha1* \/ *beta1* \/ *gamma1*

N2 == *alpha2* \/ *beta2* \/ *gamma2*

N == *N1* \/ *N2*

Statefunction

w == (*x*, *y*, *pc1*, *pc2*, *sem*)

Temporal

Psi == *InitPsi* /\ [][*N*]_*w* /\ **SF**(*w*, *N1*) /\ **SF**(*w*, *N2*)

As in the case of Program 1 we prove a basic 'type correctness' invariant, which now also states that *sem* is a natural, and that the two program counters have one of the values a, b, or g. The structure of the proof is identical to that of *TPhi*, although with four more cases in step <1>2 in accordance with the added actions of Program 2. We use *NatRule1* to show that $sem - 1$ is a *Nat* whenever *sem* is, and *NatRule2* similarly to show that $sem + 1$ is a *Nat* whenever *sem* is.

`tc1.tlp`:

Name TPsi

%- **Use** def1
%- **Include** methods

Predicates

TPsi == /\ x in *Nat*
/\ y in *Nat*
/\ *sem* in *Nat*
/\ *pc1* = a \/ *pc1* = b \/ *pc1* = g
/\ *pc2* = a \/ *pc2* = b \/ *pc2* = g

Theorem TPsi
Psi => [] *TPsi*

Proof

<1>1 **Assume** *InitPsi* **Prove** *TPsi*
Expand *InitPsi* in **Hyp**
Expand *TPsi* in **Goal**
Qed

<1>2 **Assume** *TPsi*, [N]_w **Prove** *TPsi'*

Activate *TPsi*

<2>1 **Case** *alpha1*
Expand *alpha1* in **Hyp**
Instantiate *NatRule1* with i_u <- *sem*
Qed

<2>2 **Case** *beta1*
Expand *beta1* in **Hyp**
Instantiate *NatRule2* with i_u <- x
Qed

<2>3 **Case** *gamma1*
Expand *gamma1* in **Hyp**
Instantiate *NatRule2* with i_u <- *sem*
Qed

<2>4 **Case** *alpha2*
Expand *alpha2* in **Hyp**
Instantiate *NatRule1* with i_u <- *sem*
Qed

<2>5 **Case** *beta2*
Expand *beta2* in **Hyp**
Instantiate *NatRule2* with i_u <- y
Qed

<2>6 **Case** *gamma2*
Expand *gamma2* in **Hyp**
Instantiate *NatRule2* with i_u <- *sem*
Qed

<2>7 **Case Unchanged**(w)
Expand w in **Hyp**

Qed

By-Cases
Expand *N*, *N1*, *N2* in **Hyp**
Qed

<1>3 *TPsi* /\ [][*N*]_*w* => []*TPsi*
INV1 with pred_I <- *TPsi*, act_N <- *N*, sf_f <- *w*
Qed

By-Implication
Expand *Psi* in **Hyp**
UseTempFact **Step**1, **Step**3
Qed

The next proof shows that the invariant *IPsi* holds. This is the same invariant that we refered to as I_Ψ on page 28. The proof once again is a traditional invariant proof, only slightly more complicated than the two previous ones. In this proof we want to use the invariance of *TPsi* as a fact helping in our reasoning. We know from Theorem TPsi that *Psi* implies $\Box TPsi$, so we should indeed be able to use *IPsi* as well as $TPsi'$ as assumptions in each of our action reasoning steps. In other words, from the lemmas

$$\begin{array}{l} Psi \Rightarrow \Box TPsi \\ InitPsi \Rightarrow IPsi \\ TPsi \Rightarrow (TPsi' \Rightarrow (IPsi \Rightarrow ([N]_w \Rightarrow IPsi'))) \end{array}$$

we should be able to derive the goal, that *Psi* implies $\Box IPsi$. Among the different versions of the TLA rules that are supplied with TLP, there is a special variant of the INV1 rule that is helpful, *INV1.3:*

$$\frac{\models Assump \Longrightarrow (Assump' \Longrightarrow (I \Longrightarrow ([N]_f \Longrightarrow I')))}{\models (\Box(Assump) \wedge I \wedge \Box([N]_f)) \Longrightarrow \Box(I)}$$

We let *Assump* be just *TPsi* and are then able to perform a traditional invariant proof, where we add *TPsi* and $TPsi'$ to the assumptions of step <1>2. In the final lines of temporal reasoning we just have to add that we want to use the temporal fact, theorem TPsi.

`inv1.tlp`:

Name IPsi

%-**Use** tc1
%-**Include** methods

Predicates
IPsi == \/ *sem* = 1 /\ *pc1* = *a* /\ *pc2* = *a*
\/ /\ *sem* = 0
/\ \/ *pc1* = *a* /\ (*pc2* = *b* \/ *pc2* = *g*)
\/ *pc2* = *a* /\ (*pc1* = *b* \/ *pc1* = *g*)

Theorem IPsi

Psi => []*IPsi*

Proof

<1>1 **Assume** *InitPsi* **Prove** *IPsi*
Expand *InitPsi* in **Hyp**
Expand *IPsi* in **Goal**
Qed

<1>2 **Assume** *TPsi*, *TPsi'*, *IPsi*, [*N*]_w **Prove** *IPsi'*
Expand *TPsi* in **Hyp**<1>
Activate *IPsi*

<2>1 **Case** *alpha1*
Expand *alpha1* in **Hyp**
Instantiate *NatRule5* with i_u <- *sem*
Qed

<2>2 **Case** *beta1*
Expand *beta1* in **Hyp**
Qed

<2>3 **Case** *gamma1*
Expand *gamma1* in **Hyp**
Qed

<2>4 **Case** *alpha2*
Expand *alpha2* in **Hyp**
Instantiate *NatRule5* with i_u <- *sem*
Qed

<2>5 **Case** *beta2*
Expand *beta2* in **Hyp**
Qed

<2>6 **Case** *gamma2*
Expand *gamma2* in **Hyp**
Qed

<2>7 **Case** **Unchanged**(*w*)
Expand *w* in **Hyp**
Qed

By-Cases
Expand *N*, *N1*, *N2* in **Hyp**
Qed

<1>3 []*TPsi* /\ *IPsi* /\ [][*N*]_w => []*IPsi*
INV1 with pred_I <- *IPsi*, act_N <- *N*, sf_f <- *w*, act_Assump <- *TPsi*
Qed

By-Implication
UseTempFact **Theorem**_TPsi
Expand *Psi* in **Hyp**
UseTempFact **Step**1, **Step**3
Qed

9.4 The simulation proof

With the safety proof of Program 2 finished, we are ready to begin the refinement proof, showing that *Psi* implies *Phi*. We are going to do this in two parts that we refer to as the *simulation* part and the *fairness* part. These two parts correspond to two main lemmas:

$$\begin{array}{ll} \text{(Simulation)} & \mathit{Psi} \Rightarrow \mathit{InitPhi} \wedge \Box[M]_v \\ \text{(Fairness)} & \mathit{Psi} \Rightarrow \mathrm{WF}_v(\mathit{M1}) \wedge \mathrm{WF}_v(\mathit{M2}) \end{array}$$

First, however, we know that to complete a fairness proof containing a non-trivial fairness part, we need to be able to talk about when the different actions are enabled. Currently, TLP cannot do this by itself (cf. section 4.3.5), so we have to specify the enabled predicates manually. Wrongly specified enabled predicates would of course be a possible source of error in the proofs, so we have to compute them carefully. Fortunately, as is most often the case, the computation is straightforward, although the first three predicates might be surprising. As *Enabled* $\langle N \rangle_f$ is defined to be the predicate that is true iff we can perform an N action in which the state function f is *changed*, i.e. $f \neq f'$, *Enabled* $\langle \mathit{M1} \rangle_v$ reduces to the equation $(x, y) \neq (x', y')$, and as *M1* implies that y is unchanged, this means that x has to be different from x'. TLA is untyped, and we don't know from the action itself that x is a natural number, and thus can't deduce that x must be different from $x + 1$, hence the result. Likewise for *M2* and M.

`simulation.tlp`:

Name Simulation

`%-` **Use** inv1
`%-` **Include** methods

Predicates

Enabled<*M1*>_*v*	== x ~= x + 1
Enabled<*M2*>_*v*	== y ~= y + 1
Enabled<*M*>_*v*	== (x ~= x + 1) \/ (y ~= y + 1)

Predicates

Enabled<*alpha1*>_*w*	== (*pc1* = *a*) /\ (0 << *sem*)
Enabled<*beta1*>_*w*	== (*pc1* = *b*)
Enabled<*gamma1*>_*w*	== (*pc1* = *g*)
Enabled<*alpha2*>_*w*	== (*pc2* = *a*) /\ (0 << *sem*)
Enabled<*beta2*>_*w*	== (*pc2* = *b*)
Enabled<*gamma2*>_*w*	== (*pc2* = *g*)
Enabled<*N1*>_*w*	== \/ **Enabled**<*alpha1*>_*w*
	\/ **Enabled**<*beta1*>_*w*
	\/ **Enabled**<*gamma1*>_*w*
Enabled<*N2*>_*w*	== \/ **Enabled**<*alpha2*>_*w*
	\/ **Enabled**<*beta2*>_*w*
	\/ **Enabled**<*gamma2*>_*w*
Enabled<*N*>_*w*	== **Enabled**<*N1*>_*w* \/ **Enabled**<*N2*>_*w*

Now we continue with the simulation part. This itself consists of two lemmas,

$$\begin{array}{ll} \text{(SimInit)} & \mathit{InitPsi} \Rightarrow \mathit{InitPhi} \\ \text{(SimStep)} & [N]_w \Rightarrow [M]_v \end{array}$$

which are both easily shown by action reasoning. SimInit is thus shown by simply expanding the definitions of *InitPhi* and *InitPsi*, and SimStep similarly, where we just have to take each sub-action into consideration.

Lemma SimInit
Assume *InitPsi* **Prove** *InitPhi*

Proof
Expand *InitPhi* in **Goal**
Expand *InitPsi* in **Hyp**
Qed

Lemma SimStep
Assume [*N*]_*w* **Prove** [*M*]_*v*

Proof

<1>1 **Case** *alpha1*
Expand *alpha1* in **Hyp**
Expand *v* in **Goal**
Qed

<1>2 **Case** *beta1*
Expand *beta1* in **Hyp**
Expand *M*, *M1* in **Goal**
Qed

<1>3 **Case** *gamma1*
Expand *gamma1* in **Hyp**
Expand *v* in **Goal**
Qed

<1>4 **Case** *alpha2*
Expand *alpha2* in **Hyp**
Expand *v* in **Goal**
Qed

<1>5 **Case** *beta2*
Expand *beta2* in **Hyp**
Expand *M*, *M2* in **Goal**
Qed

<1>6 **Case** *gamma2*
Expand *gamma2* in **Hyp**
Expand *v* in **Goal**
Qed

<1>7 **Case Unchanged**(*w*)
Expand *w* in **Hyp**

Expand v in **Goal**
Qed

By-Cases
Expand N, *N1*, *N2* in **Hyp**
Qed

From these lemmas we get the Simulation lemma by some temporal reasoning. Using lemma SimInit we get *InitPhi*, and from the rule *ImplBox1:*

$$\frac{\models F \Longrightarrow G}{\models \Box(F) \Longrightarrow \Box(G)}$$

and lemma SimStep we get $\Box[M]_v$.

Lemma Simulation
Psi `=>` *InitPhi* `/\ [][`M`]_`v
Proof
By-Implication
Expand *Psi* in **Hyp**
UseTempFact **Lemma**_SimInit
Instantiate *ImplBox1* with temp_F `<- ([`N`]_`w`),`
temp_G `<- ([`M`]_`v`)`
UseTempFact *ImplBox1*
Qed

9.5 The fairness proof

We now move on to the difficult part of the refinement proof, showing that *Psi* satisfies the fairness constraints of *Phi*. We want to show two symmetric lemmas:

(FairnessM1)	$Psi \Rightarrow \mathrm{SF}_v(M1)$
(FairnessM2)	$Psi \Rightarrow \mathrm{SF}_v(M2)$

(As in the TLA report, we show strong fairness although weak fairness is all we need). Here we will only show the proof of FairnessM1, the two proofs being symmetric copies. For this we use the TLP variant of the SF2 rule:

$$\frac{\begin{array}{l}\models Assump \Longrightarrow (\langle N \wedge B\rangle_f \Longrightarrow \langle M\rangle_g)\\ \models Assump \Longrightarrow ((P \wedge P' \wedge \langle N \wedge A\rangle_f) \Longrightarrow B)\\ \models Assump \Longrightarrow ((P \wedge Enabled(\langle M\rangle_g)) \Longrightarrow Enabled(\langle A\rangle_f))\\ \models (\Box(Assump) \wedge \Box([N \wedge \neg(B)]_f) \wedge \mathrm{SF}_f(A) \wedge \Box(F)) \Longrightarrow \Diamond(\Box(P))\\ \models Phi \Longrightarrow \Box(Assump)\\ \models Phi \Longrightarrow (\Box([N]_f) \wedge \mathrm{SF}_f(A) \wedge \Box(F))\end{array}}{\models Phi \Longrightarrow \mathrm{SF}_g(M)}$$

The instantiations that we will use are

$$\begin{array}{lcl} \mathit{Assump} & \leftarrow & \mathit{AStepAssump} == \mathit{TPsi} \wedge \mathit{TPsi}' \wedge \mathit{IPsi} \wedge \mathit{IPsi}' \\ \\ N & \leftarrow & N \\ A & \leftarrow & N1 \\ B & \leftarrow & \mathit{beta1} \\ f & \leftarrow & w \\ P & \leftarrow & \mathit{pc1} = b \\ F & \leftarrow & \mathrm{SF}_w(N2) \\ \mathit{Phi} & \leftarrow & \mathit{Psi} \\ \\ M & \leftarrow & M1 \\ g & \leftarrow & v \end{array}$$

and to show FairnessM1 we thus first need to show the premises

$$\begin{array}{ll} \text{(M1SF2i)} & \mathit{AStepAssump} \Rightarrow (\langle N \wedge \mathit{beta1} \rangle_w \Rightarrow \langle M1 \rangle_v) \\ \text{(M1SF2ii)} & \mathit{AStepAssump} \Rightarrow (\mathit{pc1} = b \wedge (\mathit{pc1} = b)' \wedge \langle N \wedge N1 \rangle_w \Rightarrow \mathit{beta1}) \\ \text{(M1SF2iii)} & \mathit{AStepAssump} \Rightarrow (\mathit{pc1} = b \wedge \mathit{Enabled}\ \langle M1 \rangle_v \Rightarrow \mathit{Enabled}\ \langle N1 \rangle_w) \\ \text{(M1SF2iv)} & (\Box \mathit{AStepAssump} \wedge \Box[N \wedge \neg \mathit{beta1}]_w \wedge \mathrm{SF}_w(N1) \wedge \Box \mathrm{SF}_w(N2)) \\ & \Rightarrow \Diamond\Box(\mathit{pc1} = b) \\ \text{(M1SF2v)} & \mathit{Psi} \Rightarrow \Box \mathit{AStepAssump} \\ \text{(M1SF2vi)} & \mathit{Psi} \Rightarrow \Box N w \wedge \mathrm{SF}_w(N1)) \wedge \Box \mathrm{SF}_w(N2)) \end{array}$$

These may be categorized as follows:

M1SF2i: action reasoning, a simple simulation step;

M1SF2ii: action reasoning, trivially deduced from the contradiction of $\mathit{pc1} = b$, $(\mathit{pc1} = b)'$, and $N1$;

M1SF2iii: action reasoning, trivially deduced from $\mathit{pc1} = b$ and $\mathit{Enabled}\ \langle N1 \rangle_w$;

M1SF2iv: temporal reasoning, the complicated step, following from multiple use of the SF1 rule;

M1SF2v: temporal reasoning, follows from the theorems TPsi and IPsi; and

M1SF2vi: temporal reasoning, follows from the definitions, adding a $\Box$ to $\mathrm{SF}_w(N2)$.

We begin with the three action reasoning premises, which should not need any further explanation.

`fairness-m1.tlp`:

Name FairnessM1

`%-` **Use** simulation
`%-` **Include** methods

Actions

AStepAssump == *TPsi* /\ *TPsi'* /\ *IPsi* /\ *IPsi'*

Lemma M1SF2i
Assume *AStepAssump*, <*N* /\ *beta1*>_*w* **Prove** <*M1*>_*v*
Proof
Expand *TPsi*, *beta1*, *w* in **Hyp**
Expand *M1*, *v* in **Goal**
Expand *AStepAssump*, *TPsi* in **Hyp**
IncrementNotEqual on *x*
Qed

Lemma M1SF2ii
Assume *AStepAssump*, *pc1* = *b* /\ (*pc1* = *b*)' /\ <*N* /\ *N1*>_*w* **Prove** *beta1*
Proof
Expand *N1*, *alpha1*, *beta1*, *gamma1* in **Hyp**
Qed

Lemma M1SF2iii
Assume *AStepAssump*, *pc1* = *b* /\ **Enabled**<*M1*>_*v* **Prove Enabled**<*N1*>_*w*
Proof
Expand *Nab** in **Goal**
Qed

Now to the fourth premise. We want to prove

$$\begin{array}{ll} \text{(M1SF2iv)} & \wedge\ \Box AStepAssump \\ & \wedge\ \Box[N \wedge \neg beta1]_w \\ & \wedge\ \mathrm{SF}_w(N1) \\ & \wedge\ \Box\mathrm{SF}_w(N2) \\ & \Rightarrow \Diamond\Box(pc1 = b) \end{array}$$

which can be shown from two lemmas:

$$\begin{array}{ll} \text{(M1SF2iv1)} & \wedge\ \Box AStepAssump \\ & \wedge\ \Box[N \wedge \neg beta1]_w \\ & \wedge\ \mathrm{SF}_w(N1) \\ & \wedge\ \Box\mathrm{SF}_w(N2) \\ & \Rightarrow \Diamond(pc1 = b) \end{array}$$

$$\begin{array}{ll} \text{(M1SF2iv2)} & \wedge\ \Box AStepAssump \\ & \wedge\ \Box[N \wedge \neg beta1]_w \\ & \wedge\ \mathrm{SF}_w(N1) \\ & \wedge\ \Box\mathrm{SF}_w(N2) \\ & \Rightarrow \Box(pc1 = b \Rightarrow \Box(pc1 = b)) \end{array}$$

The first lemma is proved from the temporal substep

$$\begin{array}{l} \wedge\ \Box AStepAssump \\ \wedge\ \Box[N \wedge \neg beta1]_w \\ \wedge\ \mathrm{SF}_w(N1) \\ \wedge\ \Box\mathrm{SF}_w(N2) \\ \Rightarrow (pc1 = a \vee pc1 = b \vee pc1 = g) \rightsquigarrow pc1 = b \end{array}$$

which is itself done by the help of two lemmas:

$$\begin{array}{ll}
\text{(M1SF2iv1a)} & \wedge\ \Box \mathit{AStepAssump} \\
& \wedge\ \Box[N \wedge \neg \mathit{beta1}]_w \\
& \wedge\ \mathrm{SF}_w(\mathit{N1}) \\
& \wedge\ \Box\mathrm{SF}_w(\mathit{N2}) \\
& \Rightarrow \mathit{pc1} = g \leadsto \mathit{pc1} = a \\
\\
\text{(M1SF2iv1b)} & \wedge\ \Box \mathit{AStepAssump} \\
& \wedge\ \Box[N \wedge \neg \mathit{beta1}]_w \\
& \wedge\ \mathrm{SF}_w(\mathit{N1}) \\
& \wedge\ \Box\mathrm{SF}_w(\mathit{N2}) \\
& \Rightarrow \mathit{pc1} = a \leadsto \mathit{pc1} = b
\end{array}$$

These can both be done by the SF1 rule in the form

$$\frac{\begin{array}{l}
\models \mathit{Assump} \Longrightarrow ((P \wedge [N]_f) \Longrightarrow (P' \vee Q')) \\
\models \mathit{Assump} \Longrightarrow ((P \wedge \langle N \wedge A\rangle_f) \Longrightarrow Q') \\
\models (\Box(\mathit{Assump}) \wedge \Box(P) \wedge \Box([N]_f) \wedge \Box(F)) \Longrightarrow \Diamond(\mathit{Enabled}(\langle A\rangle_f))
\end{array}}{\models (\Box(\mathit{Assump}) \wedge \Box([N]_f) \wedge \mathrm{SF}_f(A) \wedge \Box(F)) \Longrightarrow (P \leadsto Q)}$$

(The first could in fact be done a little easier with the WF1 rule, since $\mathrm{SF}_f(A) \Rightarrow \mathrm{WF}_f(A)$ for all f and A.) In the first lemma we instantiate

$$\begin{array}{lcl}
\mathit{Assump} & \leftarrow & \mathit{AStepAssump} \\
\\
P & \leftarrow & \mathit{pc1} = g \\
Q & \leftarrow & \mathit{pc1} = g \\
N & \leftarrow & N \wedge \neg \mathit{beta1} \\
f & \leftarrow & w \\
F & \leftarrow & \mathrm{SF}_w(\mathit{N2}) \\
A & \leftarrow & \mathit{N1}
\end{array}$$

and we thus have to prove the premises:

$$\begin{array}{ll}
\text{(M1SF1ai)} & \mathit{AStepAssump} \\
& \Rightarrow \mathit{pc1} = g \wedge [N \wedge \neg \mathit{beta1}]_w \\
& \quad \Rightarrow (\mathit{pc1} = g)' \vee (\mathit{pc1} = a)' \\
\\
\text{(M1SF1aii)} & \mathit{AStepAssump} \\
& \Rightarrow \mathit{pc1} = g \wedge \langle N \wedge \neg \mathit{beta1} \wedge \mathit{N1}\rangle_w \\
& \quad \Rightarrow (\mathit{pc1} = a)' \\
\\
\text{(M1SF1aiii)} & \wedge\ \Box \mathit{AStepAssump} \\
& \wedge\ \Box(\mathit{pc1} = g) \\
& \wedge\ \Box[N \wedge \neg \mathit{beta1}]_w \\
& \wedge\ \Box\mathrm{SF}_w(\mathit{N2}) \\
& \Rightarrow \Diamond \mathit{Enabled}\ \langle \mathit{N1}\rangle_w
\end{array}$$

We continue with the proofs of these three premises. It should be obvious that these are very simple. The first is a trivial proof that follows from considering each of the possible actions of $[N \wedge \neg beta1]_w$. In the second we are only looking at the actions of *N1*, and from the fact that *pc1* is equal to g we know that we are performing a *gamma1* step. Showing $\Diamond Enabled\ \langle N1 \rangle_w$ in the last step is trivial as $pc1 = g$ directly satisfies $Enabled\ \langle N1 \rangle_w$, we only have to 'remove a box and insert a diamond'.

Lemma M1SF1ai
Assume *AStepAssump*, *pc1* = *g* /\ [*N* /\ ~*beta1*]_*w* **Prove** (*pc1* = *g*)' \/ (*pc1* = *a*)'
Proof

<1>1 **Case** *N* /\ ~*beta1*

<2>1 **Case** *alpha1*
Expand *alpha1* in **Hyp**
Qed

<2>2 **Case** *gamma1*
Expand *gamma1* in **Hyp**
Qed

<2>3 **Case** *alpha2*
Expand *alpha2* in **Hyp**
Qed

<2>4 **Case** *beta2*
Expand *beta2* in **Hyp**
Qed

<2>5 **Case** *gamma2*
Expand *gamma2* in **Hyp**
Qed

By-Cases
Expand *N*, *N1*, *N2* in **Hyp**
Qed

<1>2 **Case Unchanged**(*w*)
Expand *w* in **Hyp**
Qed

By-Cases
Qed

Lemma M1SF1aii
Assume *AStepAssump*, *pc1* = *g* /\ <*N* /\ ~*beta1* /\ *N1*>_*w* **Prove** (*pc1* = *a*)'
Proof
Expand *N1*, *alpha1*, *gamma1* in **Hyp**
Qed

Lemma M1SF1aiii
/\ []*AStepAssump*
/\ [](*pc1* = *g*)
/\ [][*N* /\ ~*beta1*]_*w*

/\ []**SF**(*w*, *N2*)
=> <>**Enabled**<*N1*>_*w*

Proof

<1>1 **Assume** *pc1* = *g* **Prove Enabled**<*N1*>_*w*
Expand *Nab** in **Goal**
Qed

By-Implication
Apply *BoxElim1* to **Hyp**
UseTempFact **Step**1 Apply *DmdIntro1* to **Step**1
Qed

Now we are able to show the first of the two lemmas for proving the first lemma of the fourth premise of the SF2 rule, by simply applying the SF1 rule.

Lemma M1SF2iv1a
/\ []*AStepAssump*
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N1*)
/\ []**SF**(*w*, *N2*)
=> *pc1* = *g* ~> *pc1* = *a*

Proof
SF1 with pred_P <- (*pc1* = *g*),
pred_Q <- (*pc1* = *a*),
act_N <- (*N* /\ ~*beta1*),
sf_f <- *w*,
temp_F <- (SF(*w*, *N2*)),
act_A <- *N1*,
act_Assump <- *AStepAssump*

Qed

We then prove the second lemma similarly, by using the SF1 rule with the instantiations

$$
\begin{array}{lcl}
\mathit{Assump} & \leftarrow & \mathit{AStepAssump} \\
 & & \\
P & \leftarrow & \mathit{pc1} = a \\
Q & \leftarrow & \mathit{pc1} = b \\
N & \leftarrow & N \wedge \neg \mathit{beta1} \\
f & \leftarrow & w \\
F & \leftarrow & \mathrm{SF}_w(\mathit{N2}) \\
A & \leftarrow & \mathit{N1}
\end{array}
$$

by which we need to show the premises:

(M1SF1bi)
$$\begin{array}{l} AStepAssump \\ \Rightarrow pc1 = a \wedge [N \wedge \neg beta1]_w \\ \quad \Rightarrow (pc1 = a)' \vee (pc1 = b)' \end{array}$$

(M1SF1bii)
$$\begin{array}{l} AStepAssump \\ \Rightarrow pc1 = a \wedge \langle N \wedge \neg beta1 \wedge N1 \rangle_w \\ \quad \Rightarrow (pc1 = b)' \end{array}$$

(M1SF1biii)
$$\begin{array}{l} \wedge \Box AStepAssump \\ \wedge \Box(pc1 = a) \\ \wedge \Box[N \wedge \neg beta1]_w \\ \wedge \Box \mathrm{SF}_w(N2) \\ \Rightarrow \Diamond Enabled\ \langle N1 \rangle_w \end{array}$$

Of course, this time the proof is not as trivial as before, as we can't as easily derive $\Diamond Enabled\ \langle N1 \rangle_w$ from the fact that *pc1* is equal to a – process 2 may have taken the semaphore, whereby we have to wait for it to finish *beta2* and *beta3* before $\langle N1 \rangle_w$ is again enabled. This is where we will need the strong fairness part $\mathrm{SF}_w(N2)$. But first we perform the proofs of the two first premises, which are just as simple as before.

Lemma M1SF1bi
Assume *AStepAssump*, *pc1* = *a* /\ [*N* /\ ~*beta1*]_*w* **Prove** (*pc1* = *a*)' \/ (*pc1* = *b*)'
Proof
<1>1 **Case** *N* /\ ~*beta1*
<2>1 **Case** *alpha1*
Expand *alpha1* in **Hyp**
Qed
<2>2 **Case** *gamma1*
Expand *gamma1* in **Hyp**
Qed
<2>3 **Case** *alpha2*
Expand *alpha2* in **Hyp**
Qed
<2>4 **Case** *beta2*
Expand *beta2* in **Hyp**
Qed
<2>5 **Case** *gamma2*
Expand *gamma2* in **Hyp**
Qed
By-Cases
Expand *N*, *N1*, *N2* in **Hyp**
Qed
<1>2 **Case Unchanged**(*w*)
Expand *w* in **Hyp**

Qed

By-Cases
Qed

Lemma M1SF1bii
Assume *AStepAssump*, *pc1* = *a* /\ <*N* /\ ~*beta1* /\ *N1*>_*w* **Prove** (*pc1* = *b*)'
Proof
Expand *N1*, *alpha1*, *gamma1* in **Hyp**
Qed

Now the third premise. To prove this we need to show that we eventually get into a state in which *pc1* and *pc2* are both equal to a. Using the invariant, and the assumption that *pc1* is equal to a, this means proving that *pc2* will eventually change to a from any of its two other values, namely b and g. I.e. we first want to show the two substeps

$$\begin{array}{ll} (\text{<1>1}): & \land \Box \mathit{AStepAssump} \\ & \land \Box(\mathit{pc1} = a) \\ & \land \Box[N \land \neg \mathit{beta1}]_w \\ & \land \Box \mathrm{SF}_w(\mathit{N2}) \\ & \Rightarrow (\mathit{pc2} = b) \leadsto (\mathit{pc2} = g) \end{array}$$

$$\begin{array}{ll} (\text{<1>2}): & \land \Box \mathit{AStepAssump} \\ & \land \Box(\mathit{pc1} = a) \\ & \land \Box[N \land \neg \mathit{beta1}]_w \\ & \land \Box \mathrm{SF}_w(\mathit{N2}) \\ & \Rightarrow (\mathit{pc2} = g) \leadsto (\mathit{pc2} = a) \end{array}$$

from which we get, by simple application of the Grønning lattice rules and using $\Box \mathit{TPsi}$ (stating that *pc2* is always either a, b, or g), the step

$$\begin{array}{ll} (\text{<1>3}): & \land \Box \mathit{AStepAssump} \\ & \land \Box(\mathit{pc1} = a) \\ & \land \Box[N \land \neg \mathit{beta1}]_w \\ & \land \Box \mathrm{SF}_w(\mathit{N2}) \\ & \Rightarrow \Diamond(\mathit{pc2} = a) \end{array}$$

Then, from this step and a simple invariance,

$$\begin{array}{ll} (\text{<1>4}): & \land \Box \mathit{AStepAssump} \\ & \land \Box(\mathit{pc1} = a) \\ & \land \Box[N \land \neg \mathit{beta1}]_w \\ & \land \Box \mathrm{SF}_w(\mathit{N2}) \\ & \Rightarrow \Box(\mathit{pc2} = a \Rightarrow \mathit{Enabled}\ \langle \mathit{N1} \rangle_w) \end{array}$$

we can deduce lemma M1SF1biii by simple temporal reasoning.

We proceed by presenting the proofs of steps <1>1 and <1>2 above. This again is two

applications of the SF1 rule. In the first case, we use the instantiations

$$\begin{array}{lcl} Assump & \leftarrow & AStepAssump \\ \\ P & \leftarrow & pc2 = b \\ Q & \leftarrow & pc2 = g \\ N & \leftarrow & N \wedge \neg beta1 \\ f & \leftarrow & w \\ F & \leftarrow & pc1 = a \\ A & \leftarrow & N2 \end{array}$$

and in the other, the instantiations

$$\begin{array}{lcl} Assump & \leftarrow & AStepAssump \\ \\ P & \leftarrow & pc2 = g \\ Q & \leftarrow & pc2 = a \\ N & \leftarrow & N \wedge \neg beta1 \\ f & \leftarrow & w \\ F & \leftarrow & pc1 = a \\ A & \leftarrow & N2 \end{array}$$

(the instantiations of F are in fact redundant, and could as well be just *true*), so that we now, finally, should prove the premises

(M1SF1ci)
$$\begin{array}{l} AStepAssump \\ \Rightarrow pc2 = b \wedge [N \wedge \neg beta1]_w \\ \quad \Rightarrow (pc2 = b)' \vee (pc2 = g)' \end{array}$$

(M1SF1cii)
$$\begin{array}{l} AStepAssump \\ \Rightarrow pc2 = b \wedge \langle N \wedge \neg beta1 \wedge N1 \rangle_w \\ \quad \Rightarrow (pc2 = g)' \end{array}$$

(M1SF1ciii)
$$\begin{array}{l} \wedge \Box AStepAssump \\ \wedge \Box(pc2 = b) \\ \wedge \Box[N \wedge \neg beta1]_w \\ \wedge \Box(pc1 = a) \\ \Rightarrow \Diamond Enabled\ \langle N2 \rangle_w \end{array}$$

and

(M1SF1di) $AStepAssump$
$\Rightarrow pc2 = g \land [N \land \neg beta1]_w$
$\quad \Rightarrow (pc2 = g)' \lor (pc2 = a)'$

(M1SF1dii) $AStepAssump$
$\Rightarrow pc2 = g \land \langle N \land \neg beta1 \land N1 \rangle_w$
$\quad \Rightarrow (pc2 = a)'$

(M1SF1diii) $\land \Box AStepAssump$
$\land \Box(pc2 = g)$
$\land \Box[N \land \neg beta1]_w$
$\land \Box(pc1 = a)$
$\Rightarrow \Diamond Enabled\ \langle N2 \rangle_w$

We continue with the proofs of these premises, all being straightforward.

Lemma M1SF1ci
Assume *AStepAssump*, *pc2* = *b* /\ [*N* /\ ~*beta1*]_*w* \
Prove (*pc2* = *b*)' \/ (*pc2* = *g*)'
Proof
<1>1 **Case** *N* /\ ~*beta1*
<2>1 **Case** *alpha1*
Expand *alpha1* in **Hyp**
Qed
<2>2 **Case** *gamma1*
Expand *gamma1* in **Hyp**
Qed
<2>3 **Case** *alpha2*
Expand *alpha2* in **Hyp**
Qed
<2>4 **Case** *beta2*
Expand *beta2* in **Hyp**
Qed
<2>5 **Case** *gamma2*
Expand *gamma2* in **Hyp**
Qed
By-Cases
Expand *N*, *N1*, *N2* in **Hyp**
Qed
<1>2 **Case Unchanged**(*w*)
Expand *w* in **Hyp**
Qed
By-Cases
Qed

Lemma M1SF1cii
Assume *AStepAssump*, *pc2* = *b* /\ <*N* /\ ~*beta1* /\ *N2*>_*w* \
Prove (*pc2* = *g*)'
Proof
Expand *N2*, *alpha2*, *beta2*, *gamma2* in **Hyp**
Qed

Lemma M1SF1ciii
/\ []*AStepAssump*
/\ [](*pc2* = *b*)
/\ [][*N* /\ ~*beta1*]_*w*
/\ [](*pc1* = *a*)
=> <>**Enabled**<*N2*>_*w*
Proof

<1>1 **Assume** *pc2* = *b* **Prove Enabled**<*N2*>_*w*
Expand *Nab** in **Goal**
Qed

By-Implication
Apply *BoxElim1* to **Hyp**
UseTempFact **Step**1
Apply *DmdIntro1* to **Step**1
Qed

Lemma M1SF1di
Assume *AStepAssump*, *pc2* = *g* /\ [*N* /\ ~*beta1*]_*w* \
Prove (*pc2* = *g*)' \/ (*pc2* = *a*)'
Proof

<1>1 **Case** *N* /\ ~*beta1*

<2>1 **Case** *alpha1*
Expand *alpha1* in **Hyp**
Qed

<2>2 **Case** *gamma1*
Expand *gamma1* in **Hyp**
Qed

<2>3 **Case** *alpha2*
Expand *alpha2* in **Hyp**
Qed

<2>4 **Case** *beta2*
Expand *beta2* in **Hyp**
Qed

<2>5 **Case** *gamma2*
Expand *gamma2* in **Hyp**
Qed

By-Cases

Expand *N*, *N1*, *N2* in **Hyp**
Qed

<1>2 **Case Unchanged**(*w*)
Expand *w* in **Hyp**
Qed

By-Cases
Qed

Lemma M1SF1dii
Assume *AStepAssump*, *pc2* = *g* /\ <*N* /\ ~*beta1* /\ *N2*>_*w* \
Prove (*pc2* = *a*)'
Proof
Expand *N2*, *alpha2*, *beta2*, *gamma2* in **Hyp**
Qed

Lemma M1SF1diii
/\ []*AStepAssump*
/\ [](*pc2* = *g*)
/\ [][*N* /\ ~*beta1*]_*w*
/\ [](*pc1* = *a*)
=> <>**Enabled**<*N2*>_*w*
Proof

<1>1 **Assume** *pc2* = *g* **Prove Enabled**<*N2*>_*w*
Expand *Nab** in **Goal**
Qed

By-Implication
Apply *BoxElim1* to **Hyp**
UseTempFact **Step**1
Apply *DmdIntro1* to **Step**1
Qed

We are finally able to continue with the proof of the third premise, which we began on page 118.

The two first steps just need the SF1 rule to be instantiated, as we already have all the premises. The third step requires some more advanced temporal reasoning than what we have seen before. We first deduce that *any* value of *pc2* should lead to *a*. This is done with the so called Grønning lattice rules, displayed on page 193. The way it works is shown in comments after the lines where it happens in the proof below – we refer to the the three predicates $pc2 = a$, $pc2 = b$, and $pc2 = g$ simply as A, B, and G. The fourth step and the conclusion is just ordinary, simple temporal reasoning.

Lemma M1SF1biii
/\ []*AStepAssump*
/\ [](*pc1* = *a*)
/\ [][*N* /\ ~*beta1*]_*w*
/\ []**SF**(*w*, *N2*)

=> <>**Enabled**<*N1*>_*w*

Proof

<1>1 /\ []*AStepAssump*
/\ [](*pc1* = *a*)
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N2*)
=> (*pc2* = *b*) ~> (*pc2* = *g*)

SF1 with pred_P <- (*pc2* = *b*),
pred_Q <- (*pc2* = *g*),
act_N <- (*N* /\ ~*beta1*),
sf_f <- *w*,
temp_F <- (*pc1* = *a*),
act_A <- *N2*,
act_Assump <- *AStepAssump*

Qed

<1>2 /\ []*AStepAssump*
/\ [](*pc1* = *a*)
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N2*)
=> (*pc2* = *g*) ~> (*pc2* = *a*)

SF1 with pred_P <- (*pc2* = *g*),
pred_Q <- (*pc2* = *a*),
act_N <- (*N* /\ ~*beta1*),
sf_f <- *w*,
temp_F <- (*pc1* = *a*),
act_A <- *N2*,
act_Assump <- *AStepAssump*

Qed

<1>3 /\ []*AStepAssump*
/\ [](*pc1* = *a*)
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N2*)
=> <>(*pc2* = *a*)

<2>1 /\ []*AStepAssump*
/\ [](*pc1* = *a*)
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N2*)
=> (*pc2* = *a* \/ *pc2* = *b* \/ *pc2* = *g*) ~> *pc2* = *a*

By-Implication
UseTempFact **Step**<1>1, **Step**<1>2 % B ~> G, G ~> A
UseLatticeRules % B ~> A, G ~> A, A ~> A
LatticeDisjunctionIntr (*pc2* = *a*) (*pc2* = *b*) (*pc2* = *a*) % (A \/ B) ~> A
LatticeDisjunctionIntr (*pc2* = *a* \/ *pc2* = *b*)
(*pc2* = *g*) (*pc2* = *a*) % (A \/ B \/ G) ~> A

Qed

By-Implication

UseTempFact **Step**1 % ABG ~> A
Normalize **Step**1 with *Leadsto* % [](ABG => <>A)
Apply *BoxElim1* to **Step**1 % ABG => <>A
Expand *AStepAssump*, *TPsi* in **Hyp** % []ABG
Apply *BoxElim1* to **Hyp** % ABG -- and we get <>A
Qed

<1>4 /\ []*AStepAssump*
/\ [](*pc1* = *a*)
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N2*)
=> [](*pc2* = *a* => **Enabled**<*N1*>_*w*)

<2>1 **Assume** *AStepAssump* /\ *pc1* = *a*, *pc2* = *a* **Prove** **Enabled**<*N1*>_*w*
Expand *AStepAssump*, *IPsi* in **Hyp**
Expand *Nab** in **Goal**
Qed

By-Implication

Instantiate *ImplBox1* with temp_F <- (*AStepAssump* /\ *pc1* = *a*),
temp_G <- (*pc2* = *a* => **Enabled**<*N1*>_*w*)
Activate *AndBox*

UseTempFact *ImplBox1*
Qed

By-Implication

Apply *BoxElim1* to **Hyp** % **SF**(*w*, *N2*)
UseTempFact **Step**3, **Step**4 % <>A, [](A => *NabN1*)
Apply *ImplDmd2* to **Step**4 % [](<>A => <>*NabN1*)
Apply *BoxElim1* to **Step**4 % <>A => <>*NabN1*
Qed

We thus have the third premise, and just have to apply the SF1 rule to get the second lemma.

Lemma M1SF2iv1b
/\ []*AStepAssump*
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N1*)
/\ []**SF**(*w*, *N2*)
=> *pc1* = *a* ~> *pc1* = *b*

Proof

SF1 with pred_P <- (*pc1* = *a*),
pred_Q <- (*pc1* = *b*),
act_N <- (*N* /\ ~*beta1*),
sf_f <- *w*,
temp_F <- (SF(*w*, *N2*)),

act_A <- *N1*,
act_Assump <- *AStepAssump*

Qed

Now let's recall. We are back to the proof of the fourth premise of the SF2 rule; the proof that was begun on page 113. The first lemma of this proof, M1SF2iv1, needed two sub-lemmas to be shown, the two lemmas that we have just finished proving. Again we apply some use of the Grønning lattice rules to reach the conclusion.

Lemma M1SF2iv1
/\ []*AStepAssump*
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N1*)
/\ []**SF**(*w*, *N2*)
=> <>(*pc1* = *b*)

Proof

<1>1 /\ []*AStepAssump*
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N1*)
/\ []**SF**(*w*, *N2*)
=> (*pc1* = *a* \/ *pc1* = *b* \/ *pc1* = *g*) ~> (*pc1* = *b*)

By-Implication
UseTempFact **Lemma**_M1SF2iv1a, **Lemma**_M1SF2iv1b % G ~> A, A ~> B
UseLatticeRules % G ~> B, A ~> B, B ~> B
LatticeDisjunctionIntr (*pc1* = *a*) (*pc1* = *b*) (*pc1* = *b*) % (A \/ B) ~> B
LatticeDisjunctionIntr (*pc1* = *a* \/ *pc1* = *b*)
(*pc1* = *g*) (*pc1* = *b*) % (A \/ B \/ G) ~> B

Qed

By-Implication
UseTempFact **Step**1 % ABG ~> B
Normalize **Step**1 with *Leadsto* % [](ABG => <>B)
Apply *BoxElim1* to **Step**1 % ABG => <>B
Expand *AStepAssump*, *TPsi* in **Hyp** % []ABG
Apply *BoxElim1* to **Hyp** % ABG -- and we get <>B

Qed

The second lemma of the fourth premise is essentially an easy invariant proof, which can be done in a single action reasoning step – the value of *pc1* never changes from *b* as long as we don't execute *beta1* steps. To this we apply some simple temporal reasoning to get the final 'box'.

Lemma M1SF2iv2
/\ []*AStepAssump*
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N1*)
/\ []**SF**(*w*, *N2*)
=> [](*pc1* = *b* => [](*pc1* = *b*))

Proof

<1>1 /\ []*AStepAssump*
/\ [][*N* /\ ~*beta1*]_*w*
/\ **SF**(*w*, *N1*)
/\ []**SF**(*w*, *N2*)
=> (*pc1* = *b* => [](*pc1* = *b*))

<2>1 **Assume** *pc1* = *b*, [*N* /\ ~*beta1*]_*w* **Prove** (*pc1* = *b*)'

<3>1 **Case** *N* /\ ~*beta1*

<4>1 **Case** *alpha1*
Expand *alpha1* in **Hyp**
Qed

<4>2 **Case** *gamma1*
Expand *gamma1* in **Hyp**
Qed

<4>3 **Case** *alpha2*
Expand *alpha2* in **Hyp**
Qed

<4>4 **Case** *beta2*
Expand *beta2* in **Hyp**
Qed

<4>5 **Case** *gamma2*
Expand *gamma2* in **Hyp**
Qed

By-Cases
Expand *N*, *N1*, *N2* in **Hyp**
Qed

<3>2 **Case Unchanged**(*w*)
Expand *w* in **Hyp**
Qed

By-Cases
Qed

<2>2 *pc1* = *b* /\ [][*N* /\ ~*beta1*]_*w* => [](*pc1* = *b*)
INV1 with pred_I <- (*pc1* = *b*), act_N <- (*N* /\ ~*beta1*), sf_f <- *w*
Qed

By-Implication
UseTempFact **Step**2
Qed

Instantiate *ImplBox1* with temp_F <- ([]*AStepAssump* /\ [][*N* /\ ~*beta1*]_*w* /\ **SF**(*w*, *N1*) /\ []**SF**(*w*, *N2*)),
temp_G <- (*pc1* = *b* => [](*pc1* = *b*))
Activate *AndBox*, *BoxElim2*, *BoxSF*
Qed

That's it – we now have all we need to prove the fourth premise of the SF2 rule.

Lemma M1SF2iv
  /\ []*AStepAssump*
  /\ [][*N* /\ ~*beta1*]_*w*
  /\ **SF**(*w*, *N1*)
  /\ []**SF**(*w*, *N2*)
  => <>[](*pc1* = *b*)
Proof
By-Implication
UseTempFact **Lemma**_M1SF2iv1, **Lemma**_M1SF2iv2 % <>B, [](B => []B)
Apply *ImplDmd2* to **Lemma**_M1SF2iv2 % [](<>B => <>[]B)
Apply *BoxElim1* to **Lemma**_M1SF2iv2 % <>B => <>[]B
Qed

The fifth premise of the SF2 rule follows trivially from the theorems TPsi and IPsi, and the sixth from the definition of *Psi*.

Lemma M1SF2v
  Psi => []*AStepAssump*
Proof
  <1>1 *Psi* => [](*IPsi* /\ *IPsi'* /\ *TPsi* /\ *TPsi'*)
  By-Implication
  UseTempFact **Theorem**_IPsi, **Theorem**_TPsi
  Apply *BoxPrime* to **Theorem**_IPsi, **Theorem**_TPsi
  Activate *AndBox*
  Qed
Expand *AStepAssump* in **Goal**
Qed

Lemma M1SF2vi
  Psi => [][*N*]_*w* /\ **SF**(*w*, *N1*) /\ []**SF**(*w*, *N2*)
Proof
By-Implication
Expand *Psi* in **Hyp**
Normalize **Goal** with *BoxSF*
Qed

We thus have all the premises of the SF2 rule, and only need to instantiate this as proposed, to get the first of the two symmetric fairness lemmas.

Lemma FairnessM1
  Psi => **SF**(*v*, *M1*)
Proof
SF2 with act_N <- *N*,
         act_A <- *N1*,
         act_B <- *beta1*,

sf_f <- w,
pred_P <- ($pc1$ = b),
temp_F <- **SF**(w, $N2$),
temp_Phi <- Psi,
act_M <- $M1$,
sf_g <- v,
act_Assump <- $AStepAssump$

Qed

9.6 The refinement proof

The proof of the second fairness lemma can be constructed from the first one by simply exchanging 1's and 2's. After this has been done, very little remains to conclude with the fairness part of the refinement proof, that Psi implies $\mathrm{WF}_v(M1)$ and $\mathrm{WF}_v(M2)$. As the fairness lemmas we have shown are too strong, showing strong fairness where we only need weak fairness, we use a temporal rule, *SFImplWF* (which is derivable from the definitions of weak and strong fairness), and the fairness lemmas to produce a weaker fairness corollary.

`refinement.tlp`:

Name Refinement

`%-` **Use** fairness-m2
`%-` **Include** methods

Lemma Fairness
  Psi => **WF**(v, $M1$) /\ **WF**(v, $M2$)
Proof
By-Implication
UseTempFact **Lemma**_FairnessM1,
  Lemma_FairnessM2 % **SF**(v, $M1$) /\ **SF**(v, $M2$)
Apply *SFImplWF* to **Lemma**_FairnessM1,
  Lemma_FairnessM2 % **WF**(v, $M1$) /\ **WF**(v, $M2$)
Qed

Finally, we join the results and prove the refinement theorem.

Theorem Refinement
  Psi => Phi
Proof
By-Implication
Expand Phi in **Goal**
UseTempFact **Lemma**_Simulation, **Lemma**_Fairness
Qed

10

The Spanning-Tree Example

The Spanning-Tree Example is based on an early example by Leslie Lamport on how to use TLA for reasoning about liveness and refinement. A first version using TLP appeared at the Computer-Aided Verification workshop in 1992 [12]. The purpose of this example is to show how TLP deals with a more complicated proof, and especially one in which the final argument is one of induction, based on the lattice rule of TLA. This correctness proof is probably the most complex in structure that the author has yet performed with TLP. It took more than a week to create in its last form, in which a number of lemmas where added as the structure of the proof grew deeper. With the aid of the TLP front-end, however, it was always possible to concentrate on the difficult parts, assuming the correctness of more trivial facts, so that the general outline of the proof never was lost in the mind its creator. The proof is presented and explained in detail in the next sections, and especially the induction part is worthy of a closer examination.

The basis of the example is a simple algorithm that, given a finite connected graph and a root, constructs a spanning tree. For each node n, the algorithm computes the distance $d[n]$ from n to the root and, if n is not the root, its father $f[n]$ in the spanning tree.

When the algorithm is expressed formally, d and f are variables whose values are functions with domain equal to the set of nodes. We will use the following notation. The expression $\lambda x \in S : e(x)$ denotes a function f whose domain is S, such that $f[x]$ equals $e(x)$ for all x in S. If f is a function, then $f[s := v]$ is the function that is the same as f except with $f[s] = v$. This is defined formally as follows, where $dom\, f$ denotes the domain of f, and $\triangleq$ means *equals by definition*.

$$f[s := v] \triangleq \lambda x \in dom\, f : \textbf{if } x = s \textbf{ then } v \textbf{ else } f[x]$$

(Thus, $s \notin dom\, f$ implies $f[s := v] = f$.) If f is a function and T a set, then $f[s :\in T]$ is the set of all functions $f[s := v]$ with $v \in T$. Finally, $[S \to T]$ denotes the set of all functions f with domain S such that $f[x] \in T$ for all $x \in S$.

We now describe the Spanning-Tree algorithm. Initially, $d[n]$ equals 0 for the root and equals ∞ for all other nodes. For each node n, there is a process that repeatedly executes *improvement steps* that choose a neighbour m with $d[m] + 1 < d[n]$, decrease $d[n]$, and set $f[n]$ to m. The improvement step could simply decrease $d[n]$ to $d[m]+1$, but for reasons that are irrelevant to this discussion, we consider a more general algorithm in which $d[n]$ is set to a nondeterministically chosen number between its old value and $d[m]$. The algorithm terminates when no more improvement steps are possible.

$$
\begin{array}{lcl}
\mathit{Init}_\Pi & \triangleq & \land\ d = \lambda n \in \mathit{Node} : \textbf{if}\ n = \mathit{Root}\ \textbf{then}\ 0\ \textbf{else}\ \infty \\
& & \land\ f \in [\mathit{Node} \to \mathit{Node}] \\
\mathcal{N}_{\Pi 2}(n, m) & \triangleq & \land\ d[m] \neq \infty \\
& & \land\ d' \in d[n\ :\in\ [d[m] + 1,\ d[n])] \\
& & \land\ f' = f[n\ :=\ m] \\
\mathcal{N}_\Pi & \triangleq & \exists n \in \mathit{Node} : \exists m \in \mathit{Nbrs}(n) : \mathcal{N}_{\Pi 2}(n, m) \\
v_\Pi & \triangleq & (d, f) \\
\Pi & \triangleq & \mathit{Init}_\Pi\ \land\ \Box[\mathcal{N}_\Pi]_{v_\Pi}\ \land\ \mathrm{WF}_{v_\Pi}(\mathcal{N}_\Pi)
\end{array}
$$

Figure 10.1: The Spanning-Tree algorithm.

The TLA formula Π describing this algorithm is defined in figure 10.1, where *Node* is the set of nodes, *Root* is the root, $\mathit{Nbrs}(n)$ is the set of neighbours of node n in the graph, and $[a,\ b)$ is the set of natural numbers c such that $a \leq c < b$.

The initial condition is described by the predicate Init_Π. It asserts that $d[n]$ has the appropriate value (0 or ∞) and that $f[n]$ is a node, for each $n \in \mathit{Node}$.

Action $\mathcal{N}_{\Pi 2}(n, m)$ describes an improvement step, in which $d[n]$ is decreased and $f[n]$ set equal to m. However, it does not assert that m is a neighbour of n. The action is enabled only if $d[m] + 1 < d[n]$. (In this formula, d and f are flexible variables, while m and n are rigid variables.)

Action $\mathcal{N}_\Pi$ is the disjunction of the actions $\mathcal{N}_{\Pi 2}(n, m)$ for every node n and neighbour m of n. It is the next-state relation of the algorithm, describing how the variables d and f may change. We define v_Π to be the pair (d, f) of variables, and Π to be the canonical formula describing the algorithm. The weak fairness condition $\mathrm{WF}_{v_\Pi}(\mathcal{N}_\Pi)$ asserts that $\mathcal{N}_\Pi$ steps are eventually taken as long as they remain possible – that is, as long as the action $\mathcal{N}_\Pi$ remains enabled. Concurrency is represented by the nondeterministic interleaving of the different processes' (atomic) improvement steps.

The correctness property to be proved is that, for every node n, the values of $d[n]$ and $f[n]$ eventually become and remain correct. Letting $\mathit{Dist}(n, m)$ denote the distance in the graph between nodes n and m, the correctness of these values is expressed by the predicate Done_Π, defined to equal

$$
\begin{array}{lll}
\forall\, n \in \mathit{Node} : & \land\ d[n] = \mathit{Dist}(\mathit{Root}, n) & \\
& \land\ 0 < d[n] < \infty\ \Rightarrow & \land\ f[n] \in \mathit{Nbrs}(n) \\
& & \land\ \mathit{Dist}(\mathit{Root}, f[n]) = \mathit{Dist}(\mathit{Root}, n) - 1
\end{array}
$$

(If the graph is not connected, then for every node n not in the root's connected component, *Done* asserts only that $d[n] = \infty$.) The assertion that Done_Π eventually becomes and remains true is expressed by the TLA formula $\Diamond\Box \mathit{Done}_\Pi$. Correctness of the algorithm is expressed by the formula $\Pi \Rightarrow \Diamond\Box \mathit{Done}_\Pi$, which asserts that $\Diamond\Box \mathit{Done}_\Pi$ holds for every behaviour satisfying Π.

10.1 The TLP specification

We now turn to the TLP specification. The basis is again a file containing declarations of most of the variables and operators that we are going to need.

`declarations.tlp`:

`%-Use` frame

Name Declarations

Values

0, 1, *infty*
Nat, *NatInf*

Operators

`+, -`	**: Val, Val -> Val**
`<<, <<=`	**: Val, Val -> Bool**
min	**: Val -> Val**
delete	**: Val, Val -> Val**
insert	**: Val, Val -> Val**
dom	**: Val -> Val**
openInter	**: Val, Val -> Val**
subSet	**: Val, Val -> Bool**

Constants

Node	**: Val**
Root	**: Val**
NbrRel(*n*, *m*)	**: Bool**
Nbrs(*n*)	**: Val**
Dist(*n*, *m*)	**: Val**

Rigid Variables

n, *m*, *p*, *q*, *k*, *l* **: Val**

The value *infty* stands for infinity, and *NatInf* is the set of natural numbers extended with *infty*. `<<=` is the operator less than or equal to, defined on *Nat* as well as *NatInf*. The operator *min* is assumed to be the operator that given a nonempty set of natural numbers returns the minimum. *delete* and *insert* are operators for inserting and deleting an element of a set. *dom* returns the domain of a function, *subSet* is the subset relation, and *openInter* creates the open interval between two *NatInf*s, as used above.

The constant parameters of the example describe the graph under consideration. We thus have the set of nodes *Node*, a *Root* node, and the set of edges described by a neighbour relation *NbrRel*. The two functions *Nbrs* and *Dist* describe the neighbours of a node and the shortest distance between any two nodes; these are derivable from the other parameters.

Before going to the specification of the algorithm itself, we want to make some assumptions about the parameters about which we are reasoning. In the file `assump.tlp` we first of all give a set of rules for reasoning about arithmetic, which are all defined directly in LP code. The only interesting rule should be the one giving a meaning to the *min* operator:

set name *NatsMin*
assert
  when *subSet*(i_S, *Nat*),
  in(i_u, i_S)
  yield *in*(*min*(i_S), i_S) /\ (*min*(i_S) <<= i_u)
  . .

This states that whenever there is an element in the set i_S, then this must be no less than the minimum, which is also in the set (given that the set is a subset of the natural numbers). The minimum operator and this assumption about it is of course essential in the proof of correctness that we are going to make. Note that we give no other definition of *min*; the assumption is going to be all we know about it.

The rest of the file contain assumptions about the parameters describing the graph. The division of these into *Assumptions* and *Facts* is insignificant; it is a remnant from earlier versions in which what were described as facts were derived from more basic assumptions.

`assump.tlp`:

Name Assump

%-Use declarations

⋮ (*LP rules for reasoning about arithmetic*)

Predicates

Assump1 == /\ *Root* in *Node*
  /\ **Forall** *n* in *Node* :
    Forall *m* in *Node* :
      NbrRel(*n*, *m*) = *NbrRel*(*m*, *n*) [* *assump12*(*n*) *]
    [* *assump11* *]

Assump2 == **Forall** *n* in *Node* :
  Nbrs(*n*) = {*m* in *Node* : *NbrRel*(*n*, *m*) [* *assump22*(*n*) *]}
  [* *assump21* *]

Assump3 == **Forall** *n* in *Node* :
  Forall *m* in *Node* :
    Dist(*n*, *m*) in *NatInf* [* *assump32*(*n*) *]
  [* *assump31* *]

Fact1 == /\ *Dist*(*Root*, *Root*) = 0
  /\ **Forall** *n* in *delete*(*Root*, *Node*) :
    0 << *Dist*(*Root*, *n*) [* *fact1* *]

Fact2 == **Forall** *n* in *Node* :
  Forall *m* in *Nbrs*(*n*) :
    Dist(*Root*, *n*) <<= (*Dist*(*Root*, *m*) + 1) [* *fact22*(*n*) *]
  [* *fact21* *]

Fact3 == **Forall** *n* in *delete*(*Root*, *Node*) :
  (*Dist*(*Root*, *n*) << *infty*) =>
    Exists *m* in *Nbrs*(*n*) :

```
                    Dist(Root, n) = (Dist(Root, m) + 1) [* fact32(n) *]
                  [* fact31 *]
  Assump  == /\ Assump1
             /\ Assump2
             /\ Assump3
             /\ Fact1
             /\ Fact2
             /\ Fact3

Act
  Do "assert Assump"
End
```

Assump1 states that the root is indeed a node, and that the neighbour relation is symmetric. *Assump2* defines the function *Nbrs* as the one returning the set of neighbours of any node it is applied to. *Assump3* states that the distance between any two nodes is either a natural number or infinity. *Fact1* further more states that the distance from the root to itself is 0, and that the distance to any other node is strictly greater. *Fact2* states that the distance to any node is always *at most* the distance to any of its neighbours plus one, and *Fact3* that when a node is connected to the root, then there is at least one neighbour so that the distance to the node is equal to the distance to the neighbour plus one.

These assumptions together describe what we need to know to reason about the graph, and to be able to speak about correctness of our computation of a spanning tree.

We are now ready to specify the algorithm:

`pi.tlp`:

```
Name Pi

%-Use declarations assump

Variables
  d, f          : Val

Statefunction
  vPi           == (d, f)

Predicate
  InitPi        == /\ d = Lambda n in Node :
                           If n = Root then 0 else infty [* dist *]
                   /\ f in [Node -> Node]

Actions
  NPi2(n, m) == /\ d[m] ~= infty
                /\ d' in d[n :in openInter(d[m] + 1, d[n])]
                /\ f' = f[n := m]
  NPi1(n)    == Exists m in Nbrs(n) :
                  NPi2(n, m) [* n2(n) *]
  NPi        == Exists n in Node :
                  NPi1(n) [* n1 *]
```

Temporal
Pi == *InitPi* /\ [][*NPi*]_*vPi* /\ **WF**(*vPi*, *NPi*)

The TLP specification is almost identical to the one in figure 10.1, the only difference being that we split the action definition in three, just to make reference to each part easier.

10.2 The safety proof

As usual, the first step in reasoning about the algorithm will be to prove an invariant. The appropriate invariant Inv_Π for our algorithm is the following.

$$\begin{array}{l} \wedge\ d \in [Node \to \mathsf{Nat} \cup \{\infty\}] \\ \wedge\ f \in [Node \to Node] \\ \wedge\ d[Root] = 0 \\ \wedge\ \forall\, n \in delete(Root, Node) : d[n] < \infty \ \Rightarrow\ \begin{array}{l} \wedge\ Dist(Root, n) \leq d[n] \\ \wedge\ f[n] \in Nbrs(n) \\ \wedge\ d[f[n]] < d[n] \end{array} \end{array}$$

We choose to do the proof in two parts, proving the invariance of the two first conjuncts first, so that we can use this as a fact in the second part, where we prove invariance of the rest. In TLP, we call the two first two conjuncts together *TPi*, and the other two *IPi*.

Before we begin the proof of the invariance of *TPi*, we will show a small predicate lemma, 'Neighbours'. This simply states that whenever m is the neighbour of the node n, then m is also a node. This follows directly from assumption 2, but will be useful in the proofs to come.

`tpi.tlp`:

Name TPi

%-Use pi
%-Include methods
%-Include quantmethods

Lemma Neighbours
Assume *n* in *Node*, *m* in *Nbrs*(*n*) **Prove** *m* in *Node*
Proof
Use *Assump*, *Assump2*
UseForall on *assump21* with (*n*)
UseSetof set *Node* func *assump22*[*n*]
Qed

We then continue with the proof that Π implies always *TPi*, a traditional invariance proof that differs from the ones of the Increment Example mostly by the complexity of the specification. To show e.g. that d is in the function space from *Node* to *NatInf*, we assume that n is a node and prove that $d[n]$ is then in *NatInf*. In the initial case, we furthermore know that d has domain *Node*, and in the action-step case that the domain

is not changed. The rest is just application of the rules for reasoning about functions (see page 188).

The use of the *UseExists* rule may need some explanation. When showing that *TPi* and *NPi* implies *TPi'*, we assume that the action *NPi2*(n, m) is performed for some node n and its neighbour m. The assumptions are made in the beginning of step <3>1 and step <4>1. We conclude that the goal is valid on the assumption *NPi* by twice using *UseExists*, the TLP rule for elimination of existential quantification, which looks like

$$\frac{\begin{array}{l}\exists(S, f)\\ (\mathsf{x} \in S \;\&\; f(\mathsf{x})) \Rightarrow B\end{array}}{B}$$

Application of the rule is handled by the method UseExists, which takes one parameter, the name of the *quantifier function*, which is the name given inside the `[*...*]` construct after the first use of the quantified formula. In one of the cases this function takes a parameter, which should then be correctly instantiated as shown at the end of step <3>1.

`tpi.tlp`:

Predicate
TPi == /\ *d* in [*Node* -> *NatInf*]
/\ *f* in [*Node* -> *Node*]

Theorem TPi
Pi => [] *TPi*
Proof
<1>1 **Assume** *InitPi* **Prove** *TPi*
Activate *TPi*
Expand *InitPi* in **Hyp**
<2>1 *d* in [*Node* -> *NatInf*]
<3>1 **Assume** *n* in *Node* **Prove** *d*[*n*] in *NatInf*
ApplyLambda vbl *n* set *Node*
<4>1 **Case** *n* = *Root*
Qed
<4>2 **Case** *n* ~= *Root*
Qed
By-Cases
Qed
ProveFuncSpaceLambda func *dist* domain *Node* codomain *NatInf*
Qed
Qed
<1>2 **Assume** *TPi*, [*NPi*]_*vPi* **Prove** *TPi'*
Activate *TPi*
<2>1 **Case** *NPi*
Expand *NPi* in **Hyp**

```
<3>1 Assume n in Node /\ NPi1(n)
Expand NPi1 in Hyp
  <4>1 Assume m in Nbrs(n) /\ NPi2(n, m)
  Expand NPi2 in Hyp
    <5>1 m in Node
    Instantiate Lemma_Neighbours with v_n_ <- (n), v_m_ <- (m)
    Qed
    <5>2 d' in [Node -> NatInf]
    ProveFuncSpaceUpdateIn orig d domain Node codomain NatInf
      <6>1 Assume k in openInter(d[m] + 1, d[n]) Prove k in NatInf
      Instantiate UseFuncSpace with a_f <- d
      Instantiate NatsAddSub with i_u <- (d[m]), i_v <- 1
      Qed
    Qed
    <5>3 f' in [Node -> Node]
    ProveFuncSpaceUpdateEq orig f domain Node codomain Node
    Qed
  Qed
  UseExists on n2[n]
  Qed
UseExists on n1
Qed
<2>2 Case Unchanged(vPi)
Expand vPi in Hyp
Qed
By-Cases
Qed
<1>3 TPi /\ [][NPi]_vPi => []TPi
INV1 with pred_I <- TPi, act_N <- NPi, sf_f <- vPi
Qed
By-Implication
Expand Pi in Hyp
UseTempFact Step1, Step3
Qed
```

We now turn to the proof of the remaining part of the invariant, which is more interesting. We want to show that, at any point in the execution of Π, $d[Root]$ is equal to 0, and for all other nodes n, if $d[n]$ is not infinity, then three statements are valid: $d[n]$ is greater than the correct, shortest distance from the root to n, $f[n]$, the father node (so far), is one of n's neighbours, and $d[f[n]]$, the computed distance to that node (so far) is smaller than the one of n. Expressed in the TLP language, this looks as follows:

`ipi.tlp`:

Name IPi

%-Use tpi
%-Include methods
%-Include quantmethods
%-Include factmethods

Predicate

```
IPi        == /\ IPi1
              /\ IPi2
IPi1       == d[Root] = 0
IPi2       == Forall n in delete(Root, Node) : IPi2x(n) [* ipi2 *]
IPi2x(n)   == d[n] << infty
              => /\ IPi21(n)
                 /\ IPi22(n)
                 /\ IPi23(n)
IPi21(n)   == Dist(Root, n) <<= d[n]
IPi22(n)   == f[n] in Nbrs(n)
IPi23(n)   == d[f[n]] << d[n]
```

Before we begin with this proof, some considerations: We saw in the *TPi* proof that to do the non-stuttering part we needed a couple of rather trivial facts, just to use the rules in the assumptions file. E.g. to show that d' was a $Node \to NatInf$, we needed to know that $d[m]+1$ was a *NatInf*, and this could only be shown, using the *UseFuncSpace* rule, if we knew that m was a node – which we had to deduce from the fact that m was a neighbour of n, n being a node, etc.

In the following proofs, we will constantly be needing similar facts, mostly about 'types' of the different expressions. To make the reasoning as smooth as possible, we therefore first of all prove a couple of lemmas, that produce many of these facts once and for all.

The first lemma, 'GeneralTypes', assumes that we know that *TPi* is satisfied, and that n and m is an arbitrary pair of neighbours. From these assumptions it concludes with a conjunction of type facts about d, *Dist*, the root, and n and m. The proofs are very simple and left uncommented.

Lemma GeneralTypes

```
Assume TPi, n in Node, m in Nbrs(n)
Prove  /\ m in Node
       /\ Root in Node
       /\ d[n] in NatInf
       /\ d[m] in NatInf
       /\ d[Root] in NatInf
       /\ dom(d) = Node
       /\ dom(f) = Node
       /\ Dist(Root, n) in NatInf
       /\ Dist(Root, m) in NatInf
```

Proof

Use *Assump*

Activate *TPi*

<1>1 m in *Node*
Instantiate **Lemma**_Neighbours with v_n_ <- (n), v_m_ <- (m)
Qed

<1>2 *Root* in *Node*
Activate *Assump1*
Qed

<1>3 /\ d[n] in *NatInf*
/\ d[m] in *NatInf*
/\ d[*Root*] in *NatInf*
UseFuncSpace func d vbl n
UseFuncSpace func d vbl m
UseFuncSpace func d vbl *Root*
Qed

<1>4 /\ *dom*(d) = *Node*
/\ *dom*(f) = *Node*
UseDom d
UseDom f
Qed

<1>5 *Dist*(*Root*, n) in *NatInf*
UseAssump3 from *Root* to n
Qed

<1>6 *Dist*(*Root*, m) in *NatInf*
UseAssump3 from *Root* to m
Qed

Qed

The next, 'StepTypes', provides three additional 'type' facts about d in the case where we are executing the action *NPi2* on n and m, and where we assume that *IPi* is satisfied. The facts are based on *TPi*, *IPi*, and one of the rules about natural numbers. As a start, we make use of the 'GeneralTypes' lemma, by instantiating it with our newly chosen n and m.

Lemma StepTypes
Assume *TPi*, *TPi'*, *IPi*, n in *Node*, m in *Nbrs*(n), *NPi2*(n, m)
Prove /\ d[m] in *Nat*
/\ (d[m] + 1) in *Nat*
/\ d'[*Root*] in *NatInf*
Proof
Use *Assump*
Instantiate **Lemma**_GeneralTypes with v_n_ <- (n), v_m_ <- (m)
Activate *TPi*, *IPi*, *NPi2*

<1>1 d[m] in *Nat*
Qed

<1>2 (d[m] + 1) in *Nat*

Instantiate *NatsAddSub* with i_u <- (*d*[*m*]), i_v <- 1
Qed

<1>3 *d'*[*Root*] in *NatInf*
UseFuncSpace func *d'* vbl *Root*
Qed

Qed

Finally, 'StepFacts', provides an array of facts about the values of our variables in the non-stuttering step case. These facts are again based on *TPi* and *IPi*, as well as rules about natural numbers, functions, and sets. They are all quite trivial, but proving them here once and for all will save a lot of work later. Some explanation of the proofs:

<1>2: We assume that n is the root and thus have that $d'[Root]$ is in the open interval from $d[m]+1$ to $d[Root]$. Activating *IPi1* tells us that $d[Root] = 0$. *UseOpenInterval* gets applied automatically as we know that $d[m] + 1$ is a natural number, and we thus get the contradiction that $d'[Root]$ is a *Nat* smaller than 0.

<1>3: We assume that n is the same as m, and from *NPi2* get that $d[n] + 1 \leq d'[n]$ and $d'[n] < d[n]$. From the *NatsAddSubOne* rule we get that $d[n]$ is strictly smaller than $d'[n]$, and thus by transitivity (*NatsLess*) than itself. By non-reflexivity (also *NatsLess*) this is a contradiction.

<1>4: We know that m is a node, and that $d[m]$ is a *Nat*. But we have to take into consideration that m might be the root, so that we get two different cases. The easy case, m is the root: We use *Fact1* to show that $Dist(Root, Root)$ is 0, and automatically get one of the *Nats01* rules applied.

The harder case, m is not the root: We should use *IPi21* on m. First we need to know that m is in $delete(Root, Node)$, which is done by the use of one of the *Set* rules. Then we apply *UseForall*. Now, to use *IPi21* on m, we just need to show that $d[m]$ is smaller than infinity. As we know that it's in *Nat*, we just apply the right rule.

<1>5: We know that $Dist(Root, m)$ is a *NatInf* – the problem is to show that it's not infinity. But as we have proved that it's less than or equal to $d[m]$, we just have to assume the opposite, and we will get a contradiction from the *NatsInftyLess* rule.

<1>7: We know that $n \neq m$, so this is just applying the *UpdateIn* and *UpdateEq* rules on the new values of d and f.

Lemma StepFacts
Assume *TPi*, *TPi'*, *IPi*, n in *Node*, m in *Nbrs*(n), *NPi2*(n, m)
Prove /\ *d'*[n] in *openInter*(*d*[m] + 1, *d*[n])
/\ $n \sim= Root$
/\ $n \sim= m$
/\ *Dist*(*Root*, m) <<= *d*[m]
/\ *Dist*(*Root*, m) in *Nat*
/\ *f'*[n] = m
/\ **Unchanged**(*d*[m])

/\ **Unchanged**(f[m])

Proof

Use *Assump*

Instantiate **Lemma**_GeneralTypes,
Lemma_StepTypes with v_n_ <- (n), v_m_ <- (m)

Activate *TPi*, *IPi*, *NPi2*

<1>1 d'[n] in *openInter*(d[m] + 1, d[n])
UpdateIn vbl n updated d
Qed

<1>2 n ~= *Root*
By-Contradiction
Activate *IPi1*
Instantiate *NatsLess* with i_u <- (d'[*Root*]), i_v <- 0
Qed

<1>3 n ~= m
By-Contradiction
Instantiate *NatsAddSubOne* with i_u <- (d[n]), i_v <- (d'[n])
Instantiate *NatsLess* with i_u <- (d[n])
Qed

<1>4 *Dist*(*Root*, m) <<= d[m]

<2>1 **Case** m = *Root*
Activate *Fact1*
Qed

<2>2 **Case** m ~= *Root*
Activate *IPi*, *IPi2*, *IPi2x*, *IPi21*
Activate *Set*
UseForall on *ipi2* with m
Instantiate *NatsInftyLess* with i_u <- (d[m])
Qed

By-Cases
Qed

<1>5 *Dist*(*Root*, m) in *Nat*

<2>1 *Dist*(*Root*, m) ~= *infty*
By-Contradiction
Instantiate *NatsInftyLess* with i_u <- (d[m])
Qed

Qed

<1>6 f'[n] = m
UpdateEq vbl n updated f
Qed

<1>7 **Unchanged**(d[m]) /\ **Unchanged**(f[m])
UpdateIn vbl m updated d
UpdateEq vbl m updated f

Qed

Qed

Now we are ready to do the *IPi* proof, which again is a traditional invariance proof. *TPi* has been added to the assumptions as we have already proved $\Box\, TPi$ as a consequence of *Pi*. The soundness of the assumption is established in the concluding temporal reasoning, where we refer to theorem *TPi*.

In the step where we consider the initial state, as well as in the action step, we prove *IPi1* and *IPi2* separately by first looking at the root and then at all other nodes. *IPi2* is proved by showing $IPi2x(n)$ for an arbitrary node n to which we apply the *ProveForall* rule.

In the initial step, the two proofs are both quite easy. We prove that $d[Root] = 0$ by using the value of d defined within *InitPi*. We only need to explicitly tell LP to rewrite the application of d to *Root*, for which it will need to know that the *Root* is a Node – a fact that is part of *Assump1*. The proof of $IPi2x(n)$ is almost as easy, as $d[n]$ is equal to infinity, so that the implication is vacuously true. We prove this again by rewriting the application of d to n. This time we use the axioms about set constructions to show that n, being a member of $delete(Root, Node)$, is also a member of *Node*. We use the *NatsInftyLess* rule to show that *infty* is not less than itself.

The action step is much harder. In the non-stuttering case, we first select a pair of neighbours, n and m, so that we reason about $NPi2(n, m)$. We instantiate the lemmas that we created for the purpose, and then are ready to do the two proofs. In <5>1 we know that n is not the root, and thus that $d[Root]$ hasn't changed and is therefore still equal to 0 – we get that by applying *UpdateIn* (this works as we know from the lemmas that d has domain *Node* and that *Root* is a *Node*). Then comes the hardest part in step <5>2. For any node other than the root we have to show that if its distance after the action is performed is different from infinity, then the three facts described above all hold. As usual we begin by choosing a node (as n is already used as the node on which we perform the action, we now use p as the selected, arbitrary node). Now we have to consider two different situations, either p is the same node as n, or it isn't. We will explain in detail how these are done. First, $p = n$:

<8>1: We want to show that $Dist(Root, p)$, the true minimal distance from the root to p, is less than or equal to $d'[p]$. Now, p is the same as n, so $d'[p]$ is in the open interval from $d[m] + 1$ to $d[p]$. We know as a fact (*Fact2*) that $Dist(Root, p)$ is less than or equal to $Dist(Root, m) + 1$, and as *IPi* is assumed to be satisfied for d, we know from a lemma that $Dist(Root, m)$ is less than or equal to $d[m]$. Together, these facts give us:

$$Dist(Root, p) \leq Dist(Root, m) + 1 \leq d[m] + 1 \leq d'[p]$$

We show each of the three $\leq$'s in a step of its own, and use transitivity to derive the goal. To do this, the only remaining thing we have to show is that $Dist(Root, m)+1$ is natural number.

<8>2: As the lemma 'StepFacts' tells us that $f'[n]$ is equal to m, p is the same as n, and m is chosen as a neighbour of n, this is trivial.

<8>3: We want to show that $d'[f'[p]]$ is smaller than $d'[p]$. This is just showing $d'[m] < d'[p]$, which is the same as $d[m] < d'[p]$, all by our lemmas. But we know that $d[m]+1 \leq d[n]$, so this can be handled by one of our 'Nats' rules, *NatsAddSubOne*.

The case where p is not equal to n is not really as hard as the first one, but as we don't know as much about p from the lemmas, we have to begin with some local lemmas. Steps <8>1–<8>3 here thus just produce some simple facts about p, $d[p]$, and $f[p]$. Then *IPi21′* and *IPi22′* follow quite easily, as we are able to use the facts of *IPi* (by applying the *USeForall* rule). This time, showing that $d'[f'[p]]$ is smaller than $d'[p]$ is the hard part. Since we don't know which node $f[p]$ is, we can't directly state that d of this is unchanged (it isn't necessarily). The interesting cases are whether $f[p]$ is the node n, that we perform the action on, or not. In the case with $f[p] = n$ we already know from *IPi*, that $d[n]$ is smaller than $d[p]$, and by *NPi2* that $d'[n]$ is smaller than $d[n]$. So this is just the transitivity rule. In the case with $f[p] \neq n$ we know from *NPi2* that $d[f[p]]$ is unchanged, so that this becomes trivial.

Remains only the stuttering step. As one would imagine, this is trivial; we only have to unfold definitions and use universal quantification elimination and subsequent introduction to get *IPi2′* from *IPi2*. The rest of the proof is ordinary temporal reasoning, much like in the *TPi* proof.

Theorem IPi
Pi => []*IPi*
Proof
<1>1 **Assume** *TPi*, *InitPi* **Prove** *IPi*
Activate *IPi*, *TPi*
Use *Assump*
Expand *InitPi* in **Hyp**
<2>1 *IPi1*
Expand *IPi1* in **Goal**
ApplyLambda vbl *Root* set *Node*
NonImmunize *Assump1*
Qed
<2>2 *IPi2*
Expand *IPi2* in **Goal**
<3>1 **Assume** *n* in *delete*(*Root*, *Node*) **Prove** *IPi2x*(*n*)
Expand *IPi2x* in **Goal**
<4>1 ~(*d*[*n*] << *infty*)
ApplyLambda vbl *n* set *Node*
Activate *Set*
Instantiate *NatsInftyLess* with i_u <- *infty*
Qed
Qed
ProveForall on *ipi2*
Qed
Qed

<1>2 **Assume** *TPi*, *TPi'*, *IPi*, [*NPi*]_*vPi* **Prove** *IPi'*
Activate *IPi*, *TPi*
Use *Assump*

<2>1 **Case** *NPi*
Expand *NPi* in **Hyp**

<3>1 **Assume** *n* in *Node* /\ *NPi1*(*n*)
Expand *NPi1* in **Hyp**

<4>1 **Assume** *m* in *Nbrs*(*n*) /\ *NPi2*(*n*, *m*)
Instantiate **Lemma**_GeneralTypes,
Lemma_StepTypes,
Lemma_StepFacts with v_n_ <- (*n*), v_m_ <- (*m*)
Expand *NPi2* in **Hyp**

<5>1 *IPi1'*
Activate *IPi1*
UpdateIn vbl *Root* updated *d*
Qed

<5>2 *IPi2'*
Activate *IPi2*

<6>1 **Assume** *p* in *delete*(*Root*, *Node*) **Prove** *IPi2x'*(*p*)
Activate *IPi2x'*
By-Implication

<7>1 **Case** *p* = *n*

<8>1 *IPi21'*(*p*)
Activate *IPi21'*

<9>1 *Dist*(*Root*, *p*) <<= (*Dist*(*Root*, *m*) + 1)
UseFact2 node *p* neighbour *m*
Qed

<9>2 (*Dist*(*Root*, *m*) + 1) <<= (*d*[*m*] + 1)
Instantiate *NatsAddSubOne*.1 with i_u <- (*Dist*(*Root*, *m*)),
i_v <- (*d*[*m*])
Instantiate *NatsAddSubOne*.2 with i_u <- (*Dist*(*Root*, *m*)),
i_v <- (*d*[*m*] + 1)
Qed

<9>3 (*d*[*m*] + 1) <<= *d'*[*p*]
Qed

Instantiate *NatsAddSub* with i_u <- (*Dist*(*Root*, *m*)), i_v <- 1
Instantiate *NatsLess* with i_u <- (*Dist*(*Root*, *p*))
Qed

<8>2 *IPi22'*(*p*)
Activate *IPi22'*
Qed

<8>3 *IPi23'*(*p*)

Activate *IPi23'*
Instantiate *NatsAddSubOne* with i_u <- (*d*[*m*]), i_v <- (*d'*[*p*])
Qed

Qed

<7>2 **Case** *p* ~= *n*

<8>1 *p* in *Node*
Activate *Set*
Qed

<8>2 *d*[*p*] in *NatInf* /\ *f*[*p*] in *Node*
UseFuncSpace func *d* vbl *p*
UseFuncSpace func *f* vbl *p*
Qed

<8>3 **Unchanged**(*d*[*p*]) /\ **Unchanged**(*f*[*p*])
UpdateIn vbl *p* updated *d*
UpdateEq vbl *p* updated *f*
Qed

UseForall on *ipi2* with *p*

<8>4 *IPi21'*(*p*)
Activate *IPi21'*
Qed

<8>5 *IPi22'*(*p*)
Activate *IPi22'*
Qed

<8>6 *IPi23'*(*p*)
Activate *IPi23'*

<9>1 **Case** *f*[*p*] = *n*
Instantiate *Natsless* with i_u <- (*d'*[*n*])
Qed

<9>2 **Case** *f*[*p*] ~= *n*
UpdateIn vbl (*f*[*p*]) updated *d*
Qed

By-Cases
Qed

Qed

By-Cases
Qed

ProveForall on *ipi2'*
Qed

Qed

UseExists on *n2*[*n*]
Qed

UseExists on *n1*
Qed

<2>2 **Case Unchanged**(*vPi*)
Expand *vPi* in **Hyp**

<3>1 *IPi1'*
Activate *IPi1'*
Qed

<3>2 *IPi2'*
Activate *IPi2'*

<4>1 **Assume** *p* in *delete*(*Root*, *Node*) **Prove** *IPi2x'*(*p*)
Activate *IPi2x*, *IPi21*, *IPi22*, *IPi23*
UseForall on *ipi2* with *p*
Qed

ProveForall on *ipi2'*
Qed

Qed

By-Cases
Qed

<1>3 []*TPi* /\ *IPi* /\ [][*NPi*]_*vPi* => []*IPi*
INV1 with pred_I <- *IPi*, act_N <- *NPi*, sf_f <- *vPi*
Qed

By-Implication
UseTempFact **Theorem**_TPi
Expand *Pi* in **Hyp**
Instantiate *BoxElim1* with temp_F <- *TPi*
UseTempFact **Step**1, **Step**3
Qed

We now want to go on with the proof of correctness, showing $\Pi \Rightarrow \Diamond\Box Done_\Pi$. As we here want to reason about liveness properties, we have to compute the enabled predicates of the actions used. As explained before, these are assumptions on which the proofs rely. The enabled predicates in this case are easy to compute, and only the enabled predicate for $\mathcal{N}_{\Pi 2}(n, m)$ may cause some deeper thoughts. As *Enabled* $\langle\mathcal{N}_{\Pi 2}(n, m)\rangle_{v_\Pi}$ should only be true if an $\mathcal{N}_{\Pi 2}(n, m)$ step can be executed which *changes* v_Π, it may not be enough to ensure the 'precondition'. In this case, however, we see that any execution of the action makes d change its value on n, so that the enabled predicate becomes what we intuitively would think.

donepi.tlp:

Name DonePi

%-**Use** ipi
%-**Include** methods
%-**Include** quantmethods
%-**Include** factmethods

Predicates

Enabled<*NPi2*(*n*, *m*)>_*vPi* == /\ *d*[*m*] ~= *infty*
/\ **Exists** *x* in *Nat* :
x in *openInter*(*d*[*m*] + 1, *d*[*n*])
[* *inint*(*n*, *m*) *]

Enabled<*NPi1*(*n*)>_*vPi* == **Exists** *m* in *Nbrs*(*n*) :
Enabled<*NPi2*(*n*, *m*)>_*vPi*
[* *enabnpi1*(*n*) *]

Enabled<*NPi*>_*vPi* == **Exists** *n* in *Node* :
Enabled<*NPi1*(*n*)>_*vPi*
[* *enabnpi* *]

We then continue with the definitions of Inv_Π and $Done_\Pi$ in TLP.

Predicates

InvPi == *TPi* /\ *IPi*

DonePi == **Forall** *n* in *Node* : *DonePi1*(*n*) /\ *DonePi2*(*n*)
[* *donepi* *]

DonePi1(*n*) == *d*[*n*] = *Dist*(*Root*, *n*)

DonePi2(*n*) == /\ 0 << *d*[*n*]
/\ *d*[*n*] << *infty*
=> /\ *f*[*n*] in *Nbrs*(*n*)
/\ *Dist*(*Root*, *f*[*n*]) = *Dist*(*Root*, *n*) - 1

We still haven't proved $\Pi \Rightarrow \Box Inv_\Pi$, so we will do that here. This is just a combination of the two theorems *TPi* and *IPi*.

Lemma InvPi
Pi => []*InvPi*
Proof
By-Implication
UseTempFact **Theorem**_IPi, **Theorem**_TPi
Activate *AndBox*
Expand *InvPi* in **Goal**
Qed

We begin with the easiest part of the correctness theorem. This is the part that states (informally) that if we eventually reach $Done_\Pi$, then $Done_\Pi$ will always be true from then on. This is a rather trivial proof, as it can easily be shown to be the case that no non-stuttering action can be performed from a state in which $Done_\Pi$ is true, and thus the variables never change. The proof can be done in a number of ways, but we here choose to use the traditional invariant proof approach, by looking at the different cases of non-stuttering and stuttering steps. (There is no 'initial state' step, as we prove the invariance only from a state in which $Done_\Pi$ has become true.)

The non-stuttering case, which is the only interesting one, should of course lead to a contradiction. This is accomplished in <4>1: We know from *DonePi* that $d[m] = Dist(Root, m)$ and that $d[n] = Dist(Root, n)$. We know from *Fact2* that $Dist(Root, n) \leq$

$Dist(Root, m) + 1$. Finally, we know from *NPi2* that $d[m] + 1 \leq d'[n] < d[n]$, so that $d[n]$ should be smaller than or equal to $d'[n]$. But this is a contradiction, and we are done with the case.

At the end we do some temporal reasoning, applying some of the simple temporal logic rules to derive the goal.

Theorem AlwaysDonePi
Pi => [](*DonePi* => []*DonePi*)
Proof
<1>1 **Assume** *InvPi*, *InvPi'*, *DonePi*, [*NPi*]_*vPi* **Prove** *DonePi'*
Activate *InvPi*, *TPi*, *IPi*, *DonePi*
Use *Assump*
<2>1 **Case** *NPi*
Expand *NPi* in **Hyp**
<3>1 **Assume** n in *Node* /\ *NPi1*(n)
Expand *NPi1* in **Hyp**
<4>1 **Assume** m in *Nbrs*(n) /\ *NPi2*(n, m)
Instantiate **Lemma**_GeneralTypes,
Lemma_StepTypes,
Lemma_StepFacts with v_n_ <- (n), v_m_ <- (m)
Expand *NPi2* in **Hyp**
Activate *DonePi1*
<5>1 d[m] = *Dist*(*Root*, m) /\ d[n] = *Dist*(*Root*, n)
UseForall on *donepi* with m
UseForall on *donepi* with n
Qed
<5>2 *Dist*(*Root*, n) <<= (*Dist*(*Root*, m) + 1)
UseFact2 node n neighbour m
Qed
<5>3 d[m] + 1 <<= d'[n] /\ d'[n] << d[n]
Qed
<5>4 d[n] <<= d'[n]
Instantiate *NatsLess* with i_u <- (d[n])
Qed
Instantiate *NatsLess* with i_u <- (d'[n])
Qed
UseExists on *n2*[n]
Qed
UseExists on *n1*
Qed
<2>2 **Case Unchanged**(*vPi*)
Expand *vPi* in **Hyp**
<3>1 **Assume** p in *Node* **Prove** *DonePi1'*(p) /\ *DonePi2'*(p)
Activate *DonePi1*, *DonePi2*

UseForall on *donepi* with *p*
Qed

ProveForall on *donepi'*
Qed

By-Cases
Qed

<1>2 []*InvPi* /\ [][*NPi*]_*vPi* => (*DonePi* => []*DonePi*)
INV1 with pred_I <- *DonePi*,
act_N <- *NPi*,
sf_f <- *vPi*,
pred_Assump <- *InvPi*
By-Implication
By-Implication
UseTempFact *INV1*
Qed

<1>3 []*InvPi* /\ [][*NPi*]_*vPi* => [](*DonePi* => []*DonePi*)
Instantiate *ImplBox1* with temp_F <- ([]*InvPi* /\ [][*NPi*]_*vPi*),
temp_G <- (*DonePi* => []*DonePi*)
Activate *AndBox*, *BoxElim2*
Qed

By-Implication
UseTempFact **Step**3, **Lemma**_InvPi
Expand *Pi* in **Hyp**
Qed

10.3 The liveness proof

The part of the correctness proof that states that we eventually reach $Done_\Pi$ is clearly harder than any of the proofs we have done so far. We here have to use the TLA Lattice rule (see figure 3.2, page 26), showing that executing Π means that we constantly move downwards in a well-founded set, as long as we haven't reached $Done_\Pi$.

We will need first to define a suitable well-founded order. Currently, we don't want to reason about well-foundedness in TLP – although it is indeed possible inside LP – but will just assume that the order defined is well-founded by asserting this as yet another assumption. The set that we want to order is the set of functions from the set of nodes to the natural numbers extended with infinity, and the order that we have chosen may be described as follows. The function c is smaller than the function d *iff* there exists a node n so that c[n] is strictly smaller than d[n], and for all other nodes m, c[m] is not greater than d[m]. That this order is well-founded should be a trivial observation.

The specification of the order in TLP looks as follows, where *Wforddef* is the definition that we assert as being valid. (The reason that we do this in such a cryptic way, is that there are no ways of defining operators, or even constant expressions in TLP. This should change in the future, but has been given rather low priority compared to the other parts of the system.)

Rigid Variables
cv, *dv* : **Val**

Operators
-< : **Val**, **Val** -> **Bool**

Predicates
Wforddef == *cv* -< *dv* =
Exists *n* in *Node* :
cv[*n*] << *dv*[*n*] /\
Forall *m* in *delete*(*n*, *Node*) :
cv[*m*] <<= *dv*[*m*] [* *wforddef1*(*cv*, *dv*, *n*) *]
[* *wforddef2*(*cv*, *dv*) *]

Act
Do "set name *AssumeWforddef*
set activity on
set immunity off
assert *Wforddef*"
NonImmunize *Wforddef*
Activate *Wforddef*
End

Temp
Do "set name *Wforder*
declare op *Wforder* : -> Any
assert *Wforder* @ int_c @ int_d -> int_d -< int_c
assert *Wellfounded*(*funcspace*(*Node*, *NatInf*), *Wforder*)"
End

For the action reasoning environment, we simply make the definition active, while for the temporal reasoning environment, in which we are going to apply the Lattice rule, we assert that the order is well-founded. Again, the complicated procedures ought to be changed into a more automated style in future versions of the TLP system.

Now we make some definitions that will be useful in the proof. *dDone* is the constant function that d should be equal to, when we have reached $Done_\Pi$. *NotDoneSet* is a state function which we will need in our final lattice proof – see page 155. H is the 'H' of the Lattice rule; $H(\mathsf{c})$ simply means that c is the distance function d, and that this is still greater than the 'done function', *dDone*. Finally, *LatPrem* and *LatCons* are the existentially quantified formulas that we will use in the premise and the consequence of the Lattice rule; we make these definitions mostly to save us from writing the formulas throughout the proof.

Statefunctions
dDone == **Lambda** *n* in *Node* : *Dist*(*Root*, *n*) [* *ddone* *]
NotDoneSet == {*x* in *Nat* :
Exists *p* in *Node* : *x* = *dDone*[*p*] /\ *dDone*[*p*] << *d*[*p*]
[* *pnotdone*(*x*) *]
[* *notdoneset* *]}

Predicates

$H(cv)$ == $cv = d$ /\ $dDone$ -< cv

$LatPrem(cv)$ == **Exists** dv in [$Node$ -> $NatInf$] :
(dv -< cv) /\ $H(dv)$ [* $latprem(cv)$ *]

$LatCons$ == **Exists** cv in [$Node$ -> $NatInf$] :
$H(cv)$ [* $latcons$ *]

Now, we are going to show that we constantly move downwards with respect to the ordering we specified, as long as we haven't reached $Done_\Pi$. This means that we are going to reason from an assumption that $Done_\Pi$ is not satisfied. If $Done_\Pi$ is not satisfied, this means that there is some node n, for which either *DonePi1* or *DonePi2* is not satisfied. This seems to produce a number of different cases, that we each time have to take into account. Fortunately, it is not really so, as $DonePi2(n)$ is a consequence of our assumptions, Inv_Π and $DonePi1(n)$. If $Done_\Pi$ is not satisfied, we therefore know that there is an n so that $DonePi1(n)$ is false, which is thus the only case we have to consider.

We could have shown a lemma that would let us use this fact throughout the final proofs, but we chose another strategy, in which we simply replace $Done_\Pi$ by the simpler formula *DoneOne*, which is the universal quantification over just *DonePi1*. To show that this is sufficient to prove eventually $Done_\Pi$, we need a lemma. The only complicated part here is the case <6>2, where we have to prove $Dist(Root, n) - 1 = Dist(Root, f[n])$ from the assumption that $d[n]$ is a number between 0 and infinity, n being a node different from the root. This involves a lot of tedious arithmetic, but shouldn't otherwise be difficult.

Predicates
DoneOne == **Forall** n in *Node* : *DonePi1*(n) [* *done1* *]
DoneAll == *DoneOne* => *DonePi*

Lemma DoneOneDoneAll
Pi => []*DoneAll*
Proof
<1>1 **Assume** *InvPi* **Prove** *DoneAll*
Expand *DoneAll*, *DoneOne* in **Goal**
<2>1 **Assume** *InvPi*, **Forall**(*Node*, *done1*) **Prove** *DonePi*
Activate *InvPi*, *TPi*, *IPi*, *DonePi*, *DonePi1*, *DonePi2*
Use *Assump*
<3>1 **Assume** n in *Node* **Prove** *DonePi1*(n) /\ *DonePi2*(n)
<4>1 *DonePi1*(n)
UseForall on *done1* with n
Qed
<4>2 *DonePi2*(n)
<5>1 **Case** $0 << d[n]$ /\ $d[n] << infty$
<6>1 **Case** $n = Root$
Use *Fact1*
Instantiate *NatsLess* with i_u <- 0, i_v <- ($d[n]$)
Qed
<6>2 **Case** $n \sim= Root$

Activate *Set*, *IPi2**
UseForall on *ipi2* with n
Instantiate **Lemma**_GeneralTypes with v_n_ <- (n),
v_m_ <- ($f[n]$)

<7>1 *DonePi1*($f[n]$)
UseForall on *done1* with ($f[n]$)
Qed

<7>2 *Dist*(*Root*, n) <<= (*Dist*(*Root*, $f[n]$) + 1)
UseFact2 node n neighbour ($f[n]$)
Qed

<7>3 *Dist*(*Root*, $f[n]$) ~= *infty*
By-Contradiction
Instantiate *NatsInftyLess* with i_u <- (*Dist*(*Root*, n))
Qed

<7>4 *Dist*(*Root*, n) = (*Dist*(*Root*, $f[n]$) + 1)
By-Contradiction
Instantiate *NatsAddSubOne*.1 with i_u <- (*Dist*(*Root*, n)),
i_v <- (*Dist*(*Root*, $f[n]$))
Instantiate *NatsAddSubOne*.2 with i_u <- (*Dist*(*Root*, $f[n]$)),
i_v <- (*Dist*(*Root*, n))
Instantiate *NatsLess*.3 with i_u <- (*Dist*(*Root*, $f[n]$) +1)
Instantiate *NatsAddSub* with i_u <- (*Dist*(*Root*, $f[n]$)),
i_v <- 1
Qed

Instantiate *NatsAddSubOne*.3 with i_u <- (*Dist*(*Root*, n)),
i_v <- (*Dist*(*Root*, $f[n]$))
Instantiate *NatsLess* with i_u <- (*Dist*(*Root*, n))
Instantiate *NatsLess* with i_v <- (*Dist*(*Root*, n))
Qed

By-Cases
Qed

By-Cases
Qed

Qed

ProveForall on *donepi*
Qed

Qed

<1>2 []*InvPi* => []*DoneAll*
Instantiate *ImplBox1* with temp_F <- *InvPi*, temp_G <- *DoneAll*
Qed

By-Implication
UseTempFact **Lemma**_InvPi, **Step**2
Qed

The predicate $H(\mathsf{c})$ states that c is equal to d and that d is greater than *dDone*. The reason that d must be greater than *dDone* is that we are going to show that $H(\mathsf{c})$ leads to either *DoneOne* or *LatPrem*(c). To do this, we need to use the WF1 rule, which means that $H(\mathsf{c})$ has to imply that $\mathcal{N}_\Pi$ is enabled – which it wouldn't be, if d was equal to *dDone*. (Another way would of course be to show $d = dDone \Rightarrow DoneOne$, and then to show $H(\mathsf{c}) \leadsto DoneOne \vee LatPrem(\mathsf{c})$ under the assumption $d \neq dDone$, where $H(\mathsf{c})$ would have to be just $\mathsf{c} = d$, but this all winds up to be the same thing.)

The way we chose to do the proof, it is useful first to show the lemma 'DoneOneOrH', which states that as long as *DoneOne* is not satisfied, $H(d)$ must be. $H(d)$, of course, just states that d is greater than *dDone*.

In step <3>1 of this proof we use the rule *UseNotForall* (see page 188) to show that $\neg DoneOne$ implies that here exists an n, such that $\neg DonePi1(n)$. This rule is a supplement to the traditional introduction and elimination rules, and can easily be derived from these.

Predicates
DoneOneOrH == ~*DoneOne* => *H*(*d*)

Lemma DoneOneOrH
Pi => []*DoneOneOrH*
Proof
<1>1 **Assume** *InvPi* **Prove** *DoneOneOrH*
Expand *DoneOneOrH* in **Goal**
<2>1 **Assume** *InvPi*, ~*DoneOne* **Prove** *H*(*d*)
Activate *H*, *InvPi*, *TPi*, *IPi*, *DoneOne*, *DonePi1*
Use *Assump*
Activate *Fact1*, *Assump1*
<3>1 **Exists** *n* in *Node* : ~*DonePi1*(*n*) [* *notdone1* *]
UseNotForall on *done1 and notdone1* set *Node*
Qed
<3>2 **Assume** *p* in *Node* /\ ~*DonePi1*(*p*)
Activate *dDone*, *Fact1*
ApplyLambda vbl *p* set *Node*
Activate *IPi**, *TPi*
<4>1 *dDone*[*p*] << *d*[*p*]
UseForall on *ipi2* with *p*
Activate *Set*
<5>1 *p* ~= *Root*
By-Contradiction
Qed
<5>2 **Case** *d*[*p*] = *infty*
UseAssump3 from *Root* to *p*
Instantiate *NatsInftyLess* with i_u <- (*dDone*[*p*])
Qed
<5>3 **Case** *d*[*p*] ~= *infty*

UseFuncSpace func (*d*) vbl *p*
Instantiate *NatsInftyLess* with i_u <- (*d*[*p*])
Qed

By-Cases
Qed

<4>2 **Forall**(*delete*(*p*, *Node*), *wforddef1*(*dDone*, *d*, *p*))

<5>1 **Assume** *q* in *delete*(*p*, *Node*) **Prove** *dDone*[*q*] <<= *d*[*q*]
ApplyLambda vbl *q* set *Node*

<6>1 **Case** *q* ~= *Root*
Activate *Set*
UseForall on *ipi2* with *q*

<7>1 **Case** *d*[*q*] = *infty*
UseAssump3 from *Root* to *q*
Instantiate *NatsInftyLess* with i_u <- (*dDone*[*q*])

<8>1 **Case** *dDone*[*q*] = *infty*
Qed

<8>2 **Case** *dDone*[*q*] ~= *infty*
Qed

By-Cases
Qed

<7>2 **Case** *d*[*q*] ~= *infty*
UseFuncSpace func (*d*) vbl *q*
Instantiate *NatsInftyLess* with i_u <- (*d*[*q*])
Qed

By-Cases
Qed

By-Cases
Qed

ProveForall on *wforddef1*[*dDone*, *d*, *p*]
Qed

ProveExists on *wforddef2*[*dDone*, *d*] with *p*
Qed

UseExists on *notdone1*
Qed

Qed

<1>2 []*InvPi* => []*DoneOneOrH*
Instantiate *ImplBox1* with temp_F <- *InvPi*, temp_G <- *DoneOneOrH*
Qed

By-Implication
UseTempFact **Lemma**_InvPi, **Step**2
Qed

We are now almost ready to begin the final lattice proof. A simple lemma will let us use some of the important invariants that we have proved so far.

Predicates
EDoneAssump == *InvPi* /\ *DoneOneOrH*

Lemma EDoneAssump
Pi => []*EDoneAssump*
Proof
By-Implication
UseTempFact **Lemma_**InvPi**, Lemma_**DoneOneOrH
Activate *AndBox*
Expand *EDoneAssump* in **Goal**
Qed

The lattice proof follows on pages 156–163. It works as follows:

<1>1: First of all we show that initially we are either done (a possible situation, where the root is not connected to any nodes) or there is a c so that $H(\mathsf{c})$. This is clearly a consequence of the 'DoneOneOrH' lemma.

<1>2: Then we use the Lattice rule to show that if such a c exists, then we will eventually be done. The Lattice rule in TLP is exactly the one we know from the TLA report, although technicalities and our encoding in LP make it look a bit different:

$$\frac{\begin{array}{l}\models (F \wedge \mathsf{c} \in S) \Longrightarrow (H(\mathsf{c}) \leadsto (G \vee \exists(S, f(\mathsf{c}))))\\ f(\mathsf{c})(\mathsf{d}) = (s(\mathsf{c})(\mathsf{d}) \wedge H(\mathsf{d}))\\ g(\mathsf{c}) = H(\mathsf{c})\\ \mathit{Wellfounded}(S, s)\end{array}}{\models F \Longrightarrow \exists(S, g) \leadsto G}$$

The instantiations that we will use are

$$\begin{array}{lcl} F & \leftarrow & \Pi \\ G & \leftarrow & \mathit{DoneOne} \\ S & \leftarrow & [\mathit{Node} \rightarrow \mathit{NatInf}] \\ H & \leftarrow & H \\ f & \leftarrow & \mathit{latprem} \\ g & \leftarrow & \mathit{latcons} \\ s & \leftarrow & \mathit{wforder} \end{array}$$

so that we may prove that Π implies $\exists([\mathit{Node} \rightarrow \mathit{NatInf}], \mathit{wforder}) \leadsto \mathit{DoneOne}$ from the fact that Π and $\mathsf{c} \in [\mathit{Node} \rightarrow \mathit{NatInf}]$ implies $H(\mathsf{c}) \leadsto (\mathit{DoneOne} \vee \exists([\mathit{Node} \rightarrow \mathit{NatInf}], \mathit{latprem}(\mathsf{c}))$.

<1>3: We finally conclude that Π implies that being initially in a state that satisfies either *DoneOne* or $H(\mathsf{c})$ for some c, means that we will eventually reach *DoneOne*. This is simple reasoning by using step <1>2 and some of the Grønning lattice rules.

The interesting reasoning is done in <1>2, where we prove the premise of the instantiated lattice rule. If we knew that every step was an $\mathcal{N}_\Pi$ step, this wouldn't be hard, as we can easily show that $\mathcal{N}_\Pi$ takes us from a d to a strictly smaller d' (which is the essence of the premise). But as we allow stuttering, we have to involve the WF1 rule, and show that not only is $\mathcal{N}_\Pi$ the needed 'downwards' action, but it is also constantly enabled, as long as we haven't reached *DoneOne*. The latter is probably the hardest proof of this example, and will need a thorough examination. But first let us see how the WF1 rule is used. It looks as follows:

$$\frac{\begin{array}{l}\models Assump \implies ((P \wedge [N]_f) \implies (P' \vee Q')) \\ \models Assump \implies ((P \wedge \langle N \wedge A\rangle_f) \implies Q') \\ \models Assump \implies (P \implies Enabled(\langle A\rangle_f))\end{array}}{\models (\Box(Assump) \wedge \Box([N]_f) \wedge \mathrm{WF}_f(A)) \implies (P \rightsquigarrow Q)}$$

We use the instantiations

$$\begin{array}{lcl} Assump & \leftarrow & EDoneAssump \wedge EDoneAssump' \\ N & \leftarrow & \mathcal{N}_\Pi \\ f & \leftarrow & v_\Pi \\ A & \leftarrow & \mathcal{N}_\Pi \\ P & \leftarrow & H(cv) \\ Q & \leftarrow & DoneOne \vee LatPrem(cv) \end{array}$$

where cv is the function that we have assumed to be an element of $[Node \rightarrow NatInf]$.

<3>1 (under <1>2, page 157) is just a simple lemma, that we need in the following steps. <3>2 is a substantial part of the proof, being essentially the proof where we neglect the stuttering possibility, just showing that $\mathcal{N}_\Pi$ is an action that takes us 'downwards'. This is used in both of the two first premises of the WF1 rule. The proof is just showing that d' is smaller than d and that $H(d)'$ is satisfied, when we execute $\mathcal{N}_\Pi$. Finally, <3>3, <3>4, and <3>5 are the three premises. <3>3 and <3>4 by now follow trivially from <3>2, although with some simple reasoning about the stuttering case in <3>3.

<3>5 is the step that we want to spend some time on. Here we show that $\mathcal{N}_\Pi$ is enabled, based on the single assumption that $H(cv)$ is satisfied – i.e. that d is greater than *dDone*. $\mathcal{N}_\Pi$ is enabled, informally, whenever there is a pair of neighbours, so that the computed distance of one differs from the distance of the other by more than one. The problem lies in finding 'the right pair' of neighbours, for which we know that this is the case.

Recall the state function *NotDoneSet*, that we defined on page 149. This is defined as the set of natural numbers, each being the true distance from the root to some node, to which we still haven't found the shortest path. We know that, if d is greater than *dDone*, then this set is not empty. If it's not empty, then it has a minimal element (that's one of our assumption about subsets of the natural numbers, in this case). Now, if there is a minimum, there must be at least one node, for which the distance is equal to this minimum. Let's call that node p. From the assumptions about the graph, we know that as p's distance from the root is not infinity, it has at least one neighbour, q, whose distance is exactly one smaller. But as the distance of q is smaller than the minimum of *NotDoneSet*, q can not be one of the nodes for which we still haven't computed the

correct distance, hence $d[q]$ must be equal to $Dist(Root, q)$. So, what we have is

$$d[q] + 1 = Dist(Root, q) + 1 = Dist(Root, p) < d[p]$$

and thus we know that $\mathcal{N}_{\Pi 2}(p, q)$ is enabled. The steps are explained below:

<4>1 (of <3>5, page 158)**:** We show that *NotDoneSet* is not empty (it contains an element). This is not too hard, as there must be one p for which $d[p]$ is smaller than $dDone[p]$, according to the definition of the ordering.

<4>2: We conclude that the minimum thus exists, and is smaller than any other element, showing first that *NotDoneSet* is a subset of *Nat* and then using the *NatsMin* rule.

<4>3: We show that there is thus a node, whose distance equals the minimum.

<4>4: Assuming that p is this node, we show our goal, by:

<5>1: showing that if q is a node for which we haven't computed the right distance, then its distance must be greater than that of p;

<5>2-4: showing that p cannot be the root (we know the distance of the root from the start), and that $dDone[p]$ is a natural number;

<5>5: and finally, using the fact that there is thus a neighbour q so that $Dist(Root, q)+1 = Dist(Root, p)$, to show that $\mathcal{N}_{\Pi 2}(p, q)$, and therefore also $\mathcal{N}_{\Pi}$, are enabled. A number of substeps are needed to derive all the facts we need about d and q, after which the conclusion, in step <6>4 is trivial.

```
Theorem EventuallyDoneOne
  Pi => <>DoneOne
Proof
  <1>1 Pi => DoneOne \/ LatCons
    <2>1 Assume EDoneAssump Prove DoneOne \/ LatCons
      <3>1 Case ~DoneOne
      Expand LatCons in Goal
      Expand EDoneAssump in Hyp<2>
        <4>1 H(d)
        Expand DoneOneOrH in Hyp<2>
        Qed
        <4>2 d in [Node -> NatInf]
        Expand InvPi, TPi in Hyp<2>
        Qed
      ProveExists on latcons with d
      Qed
    By-Cases
    Qed
  By-Implication
```

UseTempFact **Lemma**_EDoneAssump, **Step**1
Instantiate *BoxElim1* with temp_F <- *EDoneAssump*
Qed

<1>2 *Pi* => (*LatCons* ~> *DoneOne*)

<2>1 (*Pi* /\ *cv* in [*Node* -> *NatInf*]) =>
(*H*(*cv*) ~> (*DoneOne* \/ *LatPrem*(*cv*)))

<3>1 *dDone* = *dDone'*
Expand *dDone* in **Goal**
ProveLambda on *ddone ddone' Node*
Qed

<3>2 **Assume** *EDoneAssump* /\ *EDoneAssump'*, *H*(*cv*) /\ *NPi*
Prove (*DoneOne* \/ *LatPrem*(*cv*))'
Expand *EDoneAssump*, *InvPi*, *H*, *NPi* in **Hyp**

<4>1 **Assume** *n* in *Node* /\ *NPi1*(*n*)
Expand *NPi1* in **Hyp**

<5>1 **Assume** *m* in *Nbrs*(*n*) /\ *NPi2*(*n*, *m*)
Instantiate **Lemma**_GeneralTypes,
Lemma_StepTypes,
Lemma_StepFacts with v_n_ <- (*n*), v_m_ <- (*m*)
Expand *NPi2* in **Hyp**

<6>1 **Case** ~*DoneOne'*
Expand *LatPrem* in **Goal**

<7>1 *d'* -< *d*

<8>1 *d'*[*n*] << *d*[*n*]
Qed

<8>2 **Forall**(*delete*(*n*, *Node*), *wforddef1*(*d'*, *d*, *n*))

<9>1 **Assume** *p* in *delete*(*n*, *Node*) **Prove** *d'*[*p*] <<= *d*[*p*]
Activate *Set*
UpdateIn vbl *p* updated *d*
Qed

ProveForall on *wforddef1*[*d'*, *d*, *n*]
Qed

ProveExists on *wforddef2*[*d'*, *d*] with *n*
Qed

<7>2 *H*(*d*)'
Expand *DoneOneOrH* in **Hyp**<3>
Qed

<7>3 *d'* in [*Node* -> *NatInf*]
Expand *TPi* in **Hyp**<3>
Qed

ProveExists on *latprem'*[*d*] with *d'*
Qed

By-Cases
Qed

UseExists on *n2*[*n*]
Qed

UseExists on *n1*
Qed

<3>3 **Assume** *EDoneAssump* /\ *EDoneAssump*′, *H*(*cv*) /\ [*NPi*]_*vPi*
Prove *H*(*cv*)′ \/ (*DoneOne* \/ *LatPrem*(*cv*))′

<4>1 **Case Unchanged**(*vPi*)
Expand *vPi* in **Hyp**
Activate **Step**<3>1
Activate *H*
Qed

<4>2 **Case** *NPi*
Instantiate **Step**<3>2 with v_cv_ <- (*cv*)
Qed

By-Cases
Qed

<3>4 **Assume** *EDoneAssump* /\ *EDoneAssump*′,
H(*cv*) /\ <*NPi* /\ *NPi*>_*vPi*
Prove (*DoneOne* \/ *LatPrem*(*cv*))′
Instantiate **Step**<3>2 with v_cv_ <- (*cv*)
Qed

<3>5 **Assume** *EDoneAssump* /\ *EDoneAssump*′, *H*(*cv*)
Prove **Enabled**<*NPi*>_*vPi*
Use *Assump*

<4>1 **Exists** *k* in *Nat* : *k* in *NotDoneSet* [* *notempty* *]

<5>1 *dDone* -< *d*
Expand *H* in **Hyp**<3>
Qed

<5>2 **Assume** *p* in *Node* /\ *dDone*[*p*] << *d*[*p*] /\
Forall(*delete*(*p*, *Node*), *wforddef1*(*dDone*, *d*, *p*))
Activate *dDone*
ApplyLambda vbl *p* set *Node*

<6>1 *dDone*[*p*] in *Nat*

<7>1 *dDone*[*p*] ~= *infty*
By-Contradiction
Expand *EDoneAssump*, *InvPi*, *TPi* in **Hyp**<3>
UseFuncSpace func *d* vbl *p*
Instantiate *NatsInftyLess*.2 with i_u <- (*d*[*p*])
Qed

Activate *Assump1*

UseAssump3 from *Root* to *p*
Qed

<6>2 *dDone*[*p*] in *NotDoneSet*

<7>1 **Exists**(*Node*, *pnotdone*(*dDone*[*p*]))
ProveExists on *pnotdone*[*dDone*[*p*]] with *p*
Qed

ProveSetof *elem* (*dDone*[*p*]) set *Nat* func *notdoneset*
Expand *NotDoneSet* in **Goal**
Qed

ProveExists on *notempty* with (*dDone*[*p*])
Qed

UseExists on *wforddef2*[*dDone*, *d*]
Qed

<4>2 /\ *min*(*NotDoneSet*) in *NotDoneSet*
/\ **Forall** *k* in *NotDoneSet* : *min*(*NotDoneSet*) << *k*
[* *minisleast* *]

<5>1 *subSet*(*NotDoneSet*, *Nat*)

<6>1 **Assume** *k* in *NotDoneSet* **Prove** *k* in *Nat*
UseSetof set *Nat* func *notdoneset*
Expand *NotDoneSet* in **Hyp**
Qed

Instantiate *ProveSubSet* with a_S <- *NotDoneSet*, a_T <- *Nat*
Qed

<5>2 **Assume** *k* in *Nat* /\ *k* in *NotDoneSet*
NatsMin set *NotDoneSet elem k*

<6>1 **Assume** *l* in *NotDoneSet* **Prove** *min*(*NotDoneSet*) << *l*
NatsMin set *NotDoneSet elem l*
Qed

ProveForall on *minisleast*
Qed

UseExists on *notempty*
Qed

<4>3 **Exists**(*Node*, *pnotdone*(*min*(*NotDoneSet*)))
Activate *NotDoneSet*
UseSetof set *Nat* func *notdoneset*
Qed

<4>4 **Assume** *p* in *Node* /\ *dDone*[*p*] = *min*(*NotDoneSet*) /\
dDone[*p*] << *d*[*p*]
Use *Assump*
Expand *EDoneAssump*, *InvPi* in **Hyp**<3>
Activate *dDone*
ApplyLambda vbl *p* set *Node*

<5>1 **Forall** q in *Node* :
$dDone[q] << d[q] => dDone[p] <<= dDone[q]$
[* *pleast*(p) *]

<6>1 **Assume** q in *Node*, $dDone[q] << d[q]$
Prove $dDone[p] <<= dDone[q]$

<7>1 $dDone[q]$ in *NotDoneSet*
ApplyLambda vbl q set *Node*

<8>1 $dDone[q]$ in *Nat*

<9>1 $dDone[q] \sim= infty$
By-Contradiction
Expand *TPi* in **Hyp**<3>
UseFuncSpace func d vbl q
Instantiate *NatsInftyLess*.2 with i_u <- ($d[q]$)
Qed

Activate *Assump1*
UseAssump3 *from Root* to q
Qed

<8>2 **Exists**(*Node*, *pnotdone*($dDone[q]$))
ProveExists on *pnotdone*[$dDone[q]$] with q
Qed

ProveSetof *elem* ($dDone[q]$) set *Nat* func *notdoneset*
Expand *NotDoneSet* in **Goal**
Qed

UseForall on *minisleast* with ($dDone[q]$)
Qed

ProveForall on *pleast*[p]
Qed

<5>2 $p \sim= Root$
By-Contradiction
Expand *IPi** in **Hyp**<3>
Activate *Fact1*
Instantiate *NatsLess*.3 with i_u <- 0
Qed

<5>3 $dDone[p]$ in *NatInf*
Activate *Assump1*
UseAssump3 *from Root* to n
Qed

<5>4 $dDone[p] \sim= infty$
By-Contradiction
Expand *TPi* in **Hyp**<3>
UseFuncSpace func d vbl p
Instantiate *NatsInftyLess*.2 with i_u <- ($d[p]$)
Qed

<5>5 **Assume** q in $Nbrs(p)$ /\ $Dist(Root, p) = Dist(Root, q) + 1$
Instantiate **Lemma**_GeneralTypes with v_n_ <- (p), v_m_ <- (q)
ApplyLambda vbl q set *Node*
Activate *Nab**

<6>1 $dDone[q] \sim= infty$
By-Contradiction
Instantiate *NatsInftyAdd* with i_u <- 1
Qed

<6>2 $dDone[q] << dDone[p]$
Instantiate *NatsAddSubOne* with i_u <- ($dDone[q]$),
i_v <- ($dDone[p]$)
Qed

<6>3 $dDone[q] = d[q]$

<7>1 $\sim(dDone[q] << d[q])$
Instantiate *NatsLess*.3 with i_u <- ($dDone[q]$),
i_v <- ($dDone[p]$)
UseForall on *pleast*[p] with q
Qed

<7>2 $dDone[q] <<= d[q]$
Expand *InvPi* in **Hyp**<3>
Activate *Set*, *IPi**
UseForall on *ipi2* with q

<8>1 **Case** $d[q] = infty$
Instantiate *NatsInftyLess*.1 with i_u <- ($dDone[q]$)
Qed

<8>2 **Case** $d[q] \sim= infty$

<9>1 **Case** $q = Root$
Activate *Fact1*
Qed

<9>2 **Case** $q \sim= Root$
Instantiate *NatsInftyLess*.1 with i_u <- ($d[q]$)
Qed

By-Cases
Qed

By-Cases
Qed

Qed

<6>4 p in *Node* /\ **Enabled**<$NPi1(p)$>_vPi

<7>1 q in $Nbrs(p)$ /\ **Enabled**<$NPi2(p, q)$>_vPi

<8>1 /\ $dDone[p]$ in *Nat*
/\ $dDone[p]$ in $openInter(d[q] + 1, d[p])$
Instantiate *ProveOpenInterval* with i_u <- ($dDone[p]$),

```
                                                    i_v <- (d[q] + 1),
                                                    i_w <- (d[p])
                    Qed
                  ProveExists on inint[p, q] with (dDone[p])
                  Qed
                ProveExists on enabnpi1[p] with q
                Qed
              ProveExists on enabnpi with p
              Qed
            Activate Fact3, Set
            UseForall on fact31 with p
            Instantiate NatsInftyLess.1 with i_u <- (dDone[p])
            UseExists on fact32[p]
            Qed
          UseExists on pnotdone[min(NotDoneSet)]
          Qed
          <3>6 []EDoneAssump /\ [][NPi]_vPi /\ WF(vPi, NPi) =>
                 (H(cv) ~> (DoneOne \/ LatPrem(cv)))
          WF1 with pred_P <- (H(cv)),
                   pred_Q <- (DoneOne \/ LatPrem(cv)),
                   act_N <- Npi,
                   sf_f <- vPi,
                   act_A <- NPi,
                   act_Assump <- (EDoneAssump /\ EDoneAssump')
          By-Implication
          Instantiate PrimeBox with pred_P <- EDoneAssump
          UseTempFact WF1, PrimeBox
          Qed
        By-Implication
        UseTempFact Lemma_EDoneAssump, Step6
        Expand Pi in Hyp
        Qed
        Lattice with temp_F <- Pi,
                     temp_G <- DoneOne,
                     set_s <- ([Node -> NatInf]),
                     temp_H <- H,
                     fnc_f <- f_latprem,
                     fnc_g <- f_latcons,
                     ord_s <- wforder
      Expand LatPrem in Step1
      Expand LatCons in Goal
      Activate FunctionDefinitions*, wforder
      Qed
      <1>3 Pi => (DoneOne \/ LatCons) ~> DoneOne
```

By-Implication % DoneOne \/ LatCons
UseTempFact **Step**<1>2 % LatCons ~> DoneOne
UseLatticeRules % DoneOne ~> DoneOne
LatticeDisjunctionIntr *DoneOne LatCons DoneOne*
% (DoneOne \/ LatCons) ~> DoneOne
Qed

By-Implication
UseTempFact **Step**1, **Step**3
Expand *Leadsto* in **Step**3
Apply *BoxElim1* to **Step**3
Qed

10.4 The correctness proof

We are done with the liveness part of the correctness proof, and only have to combine the results to prove the final correctness theorem, 'DonePi'. The proof depends on the lemma 'DoneOneDoneAll' (page 150) and the two theorems 'AlwaysDonePi' (page 147) and 'EventuallyDoneOne' (page 156). The temporal reasoning needed to combine these has been spelled out into four steps, which should make it easy to understand.

Theorem DonePi
Pi => <>[]*DonePi*
Proof
<1>1 *Pi* => [](*DoneOne* => *DonePi*)
By-Implication
UseTempFact **Lemma**_DoneOneDoneAll
Activate *DoneAll*
Qed
<1>2 *Pi* => [](<>*DoneOne* => <>*DonePi*)
By-Implication
UseTempFact **Step**<1>1
Apply *ImplDmd2* to **Step**<1>1
Qed
<1>3 *Pi* => <>*DonePi*
By-Implication
UseTempFact **Theorem**_EventuallyDoneOne, **Step**<1>2
Apply *BoxElim1* to **Step**<1>2
Qed
<1>4 *Pi* => (<>*DonePi* => <>[]*DonePi*)
By-Implication
UseTempFact **Theorem**_AlwaysDonePi
Apply *ImplDmd2* to **Theorem**_AlwaysDonePi
Apply *BoxElim1* to **Theorem**_AlwaysDonePi
Qed

By-Implication

UseTempFact **Step**3, **Step**4
Qed

11

The Multiplier

The Multiplier example was created by Bob Kurshan and Leslie Lamport as part of an effort showing how theorem proving can be combined with automatic model checking, verifying systems that neither method would be capable of alone. The example and the proofs that have been done were presented in the CAV paper "Verification of a Multiplier: 64 Bits and Beyond" [18]. It is included here to show how TLP may be used in a wider context, reasoning about a realistic system. As the separate details of the proof do not present much new compared to what we have shown in the two previous chapters, we will here just give an overview what was done, with a note on what we learned from using TLP.

We want to reason about a piece of hardware for multiplying two $k \cdot 2^m$-bit numbers. The multiplier is constructed from k-bit multipliers by recursively applying a method for implementing a $2N$-bit multiplier with four N-bit multipliers. The k-bit multipliers constitute complex algorithms that would be hard to verify by theorem proving. Using a k no greater than 8, they may however be verified by model checking. Here we thus only have to consider the verification of the combination of four N-bit multipliers to give a $2N$-bit multiplier, from which we will have verified a $k \cdot 2^m$-bit multiplier for any m.

As the two previous examples have already served to show how different kinds of proofs may be done in TLP, we will not include this example in its entirety. Instead, we will here just show how the multipliers may be specified in TLP, and explain the steps of the proofs that have been performed. The complete text of the example, structured and commented so that it should be suitable for investigation, may be found among the examples contained in the TLP distribution.

11.1 The TLP specification

The specification of an N-bit multiplier may be written as below. a and x are the two (constant) numbers being multiplied, and *out* is the variable that should contain the resulting value when the multiplication has been finished. We use the constant NN for N, as LP does not differentiate between upper- and lower-case letters, and n is used elsewhere as a variable. The execution of the multiplier is synchronized with its environment using a two-phase handshaking protocol on the one-bit variables *sig* and *ack*. The environment is free to change the inputs whenever $sig = ack$; it complements *sig* when the output is ready. The multiplier can change *out* when $sig \neq ack$; it complements *sig* when ready.

Initially, *sig* and *ack* are both 0 and the multiplier is ready to receive input.

MultiplyNN(a, x) simply denotes the $2 * NN$-bit vector obtained by multiplying a and x, so that the action *Finish* delivers the right result in *out*. *Multiply*, *BitVector*, and all variables and constants are declared elsewhere, and the operators are given a meaning through assumption predicates; we will not go into detail with those here. The full multiplier is denoted by M and its environment by E.

Name Mult

%-Use frame multvars

Predicates

MInit == /\ *out* in *BitVector*(2 * *NN*)
/\ *ack* = 0

EInit == /\ *a* in *BitVector*(*NN*) /\ *x* in *BitVector*(*NN*)
/\ *sig* = 0

Actions

Think == /\ *sig* ~= *ack*
/\ *ack'* = *ack*
/\ *out'* in *BitVector*(2 * *NN*)

Finish == /\ *sig* ~= *ack*
/\ *ack'* = 1 - *ack*
/\ *out'* = *Multiply*(*NN*, *a*, *x*)

MNext == (*Think* \/ *Finish*)

ENext == /\ *sig* = *ack*
/\ *sig'* = 0 \/ *sig'* = 1
/\ *a'* in *BitVector*(*NN*) /\ *x'* in *BitVector*(*NN*)

Temporal

M == /\ *MInit*
/\ [][*MNext*]_(*out*, *ack*)
/\ **WF**((*out*, *ack*), *Finish*)

E == /\ *EInit*
/\ [][*ENext*]_(*a*, *x*, *sig*)

To make a $2N$-bit multiplier, we make four copies of the N-bit multiplier: LL, LH, HL, and HH, modified so that they work on aL and xL, aL and xH, aH and xL, and aH and xH respectively, with the interface variables modified to be *ackLL*, *ackLH*, *ackHL*, or *ackHH*, and likewise for *out* (we need only one, general *sig*, however). aL, aH, xL, and xH are defined as the lower and higher half of the $2N$-bit vectors a and x, as follows:

Statefunctions

aH == **Lambda** *i* in *LessThan*(*NN*) : *a*[*i* + *NN*] [* *aH* *]
aL == **Lambda** *i* in *LessThan*(*NN*) : *a*[*i*] [* *aL* *]
xH == **Lambda** *i* in *LessThan*(*NN*) : *x*[*i* + *NN*] [* *xH* *]
xL == **Lambda** *i* in *LessThan*(*NN*) : *x*[*i*] [* *xL* *]

Note that we could have used a parameterized specification of the multiplier, so that we could just refer to the multipliers as e.g. $M(L, H)$, L, and H being constant values. This

makes our work more structured, as we don't have to repeat invariant and fairness proofs (we would use general rigid variables instead of L and H), and is generally considered the right way to do this.[1] A parameterized version with some of the initial proofs is supplied with the TLP distribution. The parameter mechanism however tend to make the specifications and proofs somewhat harder to read, and may cause the Larch Prover unnecessary efficiency problems, which is why we here have chosen to use four similar-looking copies instead.

With these definitions we now know that we may compute the product of a and x from aL, xL, aH and xH as

$$a \cdot x = 2^{2N} \cdot (aH \cdot xH) + 2^N \cdot ((aH \cdot xL) + (aL \cdot xH)) + (aL \cdot xL)$$

The four parenthesized products in this expression are of course just the results from executing the four N-bit multipliers, so assuming that we can handle the remaining additions and shifting operations, we may specify the $2N$-bit multiplier as below.

Name DblMult

%-Use frame bitvectors multvars ll lh hl hh dbl

Predicates

```
Adder1   == out in BitVector(4 * NN)
Adder2   == IVal(4 * NN, out) =
                ((((2 ^ (2 * NN)) * IVal(2 * NN, outHH)) +
                  ((2 ^ NN) * (IVal(2 * NN, outLH) + IVal(2 * NN, outHL))) +
                  IVal(2 * NN, outLL)) |
                 (2 ^ (4 * NN)))
AdderP   == Adder1 /\ Adder2
```

Statefunctions

```
acks     == (ack, ackLL, ackLH, ackHL, ackHH)
outs     == (out, outLL, outLH, outHL, outHH)
axsig    == (a, x, sig)
vars     == (acks, outs, axsig)
```

Predicates

```
AckInit  == ack = 0
AckGuard == /\ ackLL = sig
            /\ ackLH = sig
            /\ ackHL = sig
            /\ ackHH = sig
```

Actions

```
AckNext  == ack' = If AckGuard then sig else (1 - sig)
```

Temporal

```
ExtAck   == AckInit /\ [][AckNext]_acks /\ WF(acks, AckNext)
Adder    == []AdderP
DM       == LLM /\ LHM /\ HLM /\ HHM /\ Adder /\ ExtAck
DE       == DblE
```

[1] Even better would be to use a module system as in TLA$^+$.

ExtAck is a temporal formula specifying a simple component that computes the *ack* variable of the $2N$-bit multiplier from its *sig* variable and each of the *ack* variables of the internal multipliers.

The *Adder2* predicate specifies the relation between the $2N$-bit multiplier output *out* and the outputs of the four internal multipliers, according to the expression above. For simplicity, we assume that we have a combinational adder *Adder* (rather than a separate, sequential component) that satisfies $\Box AdderP$, maintaining this relation throughout the execution of the multiplier.

The $2N$-bit multiplier may now be specified by DM, with its environment DE being just the $2N$-bit instantiation $DblE$ of the environment E of the Mult module.

11.2 The proofs

Kurshan and Lamport describe in their paper how to prove that the recursively defined $k \cdot 2^m$-bit multiplier satisfies its high-level specification, using model checking as an induction basis and theorem proving (through the Decomposition Theorem [18]) to show the induction step. The hardest part of the induction step is of course to show that the low-level specification of the $2N$-bit multiplier (composed of the four high-level N-bit multiplier specifications) implements the high-level $2N$-bit multiplier specification. This is in TLP to show the validity of the temporal formula

$$DE \wedge DM \Rightarrow DblM$$

where $DblM$ is the formula M of the $2N$-bit instantiation of Mult. The proof of this formula has been done in TLP, based on a set of assumptions on the used data types with addition and multiplication. In addition to these assumptions, the proof uses a few derived facts, that TLP does not yet seem strong enough to handle in a feasible manner. The strongest of these not-proven facts asserts that when each of the N-bit multiplier outputs equals the product of their inputs, and $AdderP$ holds, then *out* equals the product of a and x – in LP:

assert
  when $AdderP$,
    $outLL = Multiply(NN, aL, xL)$,
    $outLH = Multiply(NN, aL, xH)$,
    $outHL = Multiply(NN, aH, xL)$,
    $outHH = Multiply(NN, aH, xH)$
  yield $out = Multiply(2 \cdot NN, a, x)$

With a stronger engine for arithmetic reasoning it should be unnecessary to make such assertions; for now we have to accept that these are assumptions on which the soundness of the proof rely.

The proofs do not differ much in character from the ones shown in detail in chapters 9 and 10. Like the Increment Example, they consist of some invariant proofs, a simulation proof, and a fairness proof. The number of invariants that have been proved for the Multiplier is a bit bigger than that of the Increment proofs, but that taken into account, the simulation proof is easily done in just about 100 lines.

The fairness proof is also quite similar in structure and complexity to the ones of the Increment Example. It consists of a single application of the WF2 rule, of which only the

fourth premise presents any difficulties. This is verified from four complicated lemmas, identical modulo the referenced 'L's and 'H's. Each of these lemmas is proved in four steps, of which the second is an application of the WF1 rule. The fourth step, combining the first three to conclude the lemma, is the only one that we will take a closer look at here, however. In the first three steps we have shown (with logical components renamed and somewhat simplified for brevity):

$$\Box Assumptions \Rightarrow \Box(\neg(\neg A \wedge B))$$
$$\Box Assumptions \Rightarrow \Box((\neg A \wedge \neg B) \leadsto (A \wedge \neg B))$$
$$\Box Assumptions \Rightarrow \Box((A \wedge \neg B) \Rightarrow \Box(A \wedge \neg B))$$

From this we would like to arrive at the conclusion, $\Box Assumptions \Rightarrow \Diamond\Box A$. This may indeed be done by application of simple temporal reasoning. Experience tells us, however, that it often produces great difficulties to do such a proof by manual application of temporal rules. This is a clear case where we may take advantage of the LTL checker, creating a simple lemma that may be checked in a matter of seconds. This lemma is shown below exactly as it appears in the Multiplier fairness proof.

Rigid Variables
  A, B : **Bool**

Lemma WF2ivLattice
  /\ [](~(~A /\ B))
  /\ (~A /\ ~B) ~> (A /\ ~B)
  /\ []((A /\ ~B) => [](A /\ ~B))
  => <>[] A
Proof
By-LTL
Qed

An interesting point about the Multiplier Example is that doing the proofs with TLP actually lead to the finding of critical bugs in the first version of the specifications. *ExtAck* was originally defined as

ExtAck == [](*ack* = **If** *AckGuard* **then** *sig* **else** (1 - *sig*))

and the step *ENext* of E (and thus *DblE*) was guarded by the assumption that *ack* and *out* were unchanged. With this specification, $DM \wedge DE$ allowed behaviours in which both *ack* and *out* changed in a single step, while *DblM* in the first specification clearly disallowed such behaviours. This problem wasn't found until I started on the simulation proof, where I at one point needed to show that *sig* was unchanged in a step where *ack* had been changed – which I obviously couldn't. After a closer look it proved to be impossible to write a purely combinational specification for *ExtAck*, and so Kurshan and Lamport changed the specifications to the current setting.

Conclusions

12

Conclusions

Researchers have been working with the scientific aspects of verifying sequential and concurrent algorithms for more than twenty years. When Leslie Lamport in the fall of 1990 requested the author's assistance in investigating the feasibility of mechanical reasoning in the Temporal Logic of Actions, it was based on a view that most of the problems that remain to be solved are in the realm of engineering, rather than science. The basic principles of formal specification and verification have been known for many years, but there is a growing need for computer-assisted tools that will make reasoning about real systems practical.

With the aim of contributing to this area, Leslie Lamport, Peter Grønning, and the author in 1991 began working on an encoding of TLA for the Larch Prover, a tool that seemed to suit our purposes very well, providing us with an efficient automatic rewriting system and allowing a very straightforward implementation of TLA based on the proof rules. With little work, we were able to produce mechanically verifiable proofs of refinements, containing both safety and liveness reasoning.

In a short time we found that the encoding we had chosen covering the complete logic was not especially well-suited when it came to reasoning about ordinary first-order logic. Realizing that most of the reasoning performed in any of our proofs was indeed based on first-order logic, this lead to the refinement and split-up of the original encoding into two separate encodings, for taking care of what we referred to as *action* and *temporal* reasoning respectively. This novel approach, based on a simple observation, made many of our previous proofs shrink by a factor of three or more.

As our aim was to make reasoning practically available, the complexity of the machine verifiable proofs that we had written was still a major problem. Not only were the proofs necessarily more detailed than their respective hand proofs, but they also had to be presented in a form specific to the two different Larch Prover encodings. We solved the latter part of the problem by introducing a tool for automatically translating specifications and proofs written in a high level language into their encodings. This made it possible for us to write specifications in pure TLA, and to write human readable proofs that were much closer to the hand proofs we were used to, and furthermore made it unnecessary to deal with more than one instance of each, as the translator was capable of producing different encodings of the same specifications and proofs, automatically splitting the proofs into parts to be verified in the different encodings.

With the introduction of separate encodings, a high-level language, and a translator for connecting the two, the content of the specifications and proofs had become simpler

and more directly related to what they where describing. The process of verifying, and not less significantly, developing proofs for evolving specifications, had not been made any simpler, however. A major contribution making incremental proof-development manageable was the author's introduction of an interactive front-end. The front-end, built on top of the customizable text-editor GNU Emacs, was made first of all to take care of all communication between the user, the translator, and the verification back-end, the Larch Prover. But with the front-end it was furthermore made possible for the user to develop proofs interactively in a top-down or bottom-up fashion, dealing with each step at a time, and adding steps and instructions whenever needed without having to worry about technical issues regarding re-translation and verification of the specific proof sections and dealing with the separate encodings.

With the improved convenience of handling different encodings and reasoning techniques, achieved by the introduction of the front-end, it was a straightforward task to add new back-ends for dealing with specific proof tasks. The author experimented in this area by introducing an encoding supporting automated verification of simple temporal logic tautologies by David Long's linear temporal logic-checker.

12.1 The results

The thesis has as its main point shown that it can be made feasible to perform rigorous reasoning about non-trivial systems and have the results mechanically verified. It has been shown how it may be useful to experiment with shallow implementations, where full safety of the system is sacrificed to gain flexibility. By depending on an integrated system with much of its internal structure embedded in the interface, we were able to find ways to reduce the complexity in the chain of reasoning. Although the safety of any proof verified in TLP has to rely on a more vulnerable system, the practical achievements may make it worth the cost.

It has been shown how the problem of dealing with the perplexity of encodings may be solved by introducing a translator and an interactive front-end for dealing with the technical details of verification. The importance of these tools should not be underestimated; experiments doing e.g. the Increment Example of chapter 3.4 without the aid of the tools (i.e. writing the encoded specifications and the low-level proofs manually and performing some sort of manually directed bottom-up proof construction) have shown to be a terrible and barely successful experience, in spite of its relative simplicity.

Most of all, the integration of different tools and proof techniques together with a single language for specifications and proofs was shown as a big step towards practical manageability of mechanical reasoning.

GNU Emacs was shown to be an excellent platform for the construction of the interactive front-end. Besides being a very advanced text editor with possibility of syntax directed editing on many levels, Emacs provides an extension language which makes it easy to implement a front-end with very high functionality. Using Emacs as our front-end is advantageous, in that it offers the same specialized environment for editing TLP specifications and proofs when in the middle of a verification session, as is available at other times. The TLP front-end built on top of Emacs uses different windows for displaying the source files, messages from the front-end about the status of verification sessions including syntax errors and verification problems, and output from the verification back-ends. The

possibility for the user to apply his own standard setup and private extensions, makes his work with TLP as flexible as we could ever have made it in a dedicated front-end, built from scratch.

The TLP system has shown its qualities mostly through practical use. Although the number of case studies that the author knows of is not yet very large, the different examples that have been tried give an idea of the wide range of problems in which TLP could be successfully applied.

The Spanning-Tree Example described in chapter 10 thus serves to show how reasoning about the correctness, and most significantly, the liveness of a non-finite state system may be handled by application of the TLA rule for well-founded induction. The Multiplier example of chapter 11, fully described by Kurshan and Lamport [18], gives a good presentation of how model checking and theorem proving may be combined in the verification of realistic systems, something that Kurshan and Lamport as well as others have been promoting as a future direction of research in the area of mechanical reasoning.

One of the most interesting examples of the use of TLP has been one by Stephan Merz, described in the recent article by Lamport and Merz "Specifying and Verifying Fault-Tolerant Systems"[25]. The article presents specifications for a well-known solution to the Byzantine generals problem and delivers a rigorous, hierarchically structured proof, of which most steps has been verified with TLP.

Peter Grønning has at the Danish Technical University been using TLP in the verification of a coherent cache, an example that has yet to be published.

12.2 Open problems and future work

As was mentioned in the first parts of the thesis, there are still many fragments of the TLP systems that are missing or which should be improved. One of the most important ones that comes into mind is the ability to either verify or get automatically generated enabled predicates. We may use TLP without this ability, as computing enabled predicates manually seldom constitutes a very hard task. The main aim of TLP being the practical availability of mechanical reasoning, we don't see the resulting lack of safety to be a major problem. The correctness of the enabled predicates will merely be assumptions on the correctness of proofs depending on them.

It is clear that one of the big hurdles in any mechanical proof is concerned with reasoning about data-types. It would be nice to have e.g. a set of arithmetical decision procedures for solving some of the big problems we have had reasoning about seemingly extremely simple arithmetical conjectures. With the integration capabilities and flexibility of TLP, adding such a tool as a new back-end should be a simple exercise. In this connection, it would also be very useful to add a tool for handling reasoning about the well-foundedness of partial orders.

Refinement mappings have been handled in different ways through the development of TLP. The current implementation lets you define refinement mappings on all variables, but reasoning is made unnecessarily difficult especially when it comes to variables which are not part of the mappings, as the translator currently is unaware of which variables it should map, the solution being to use identity mappings for all other variables. A better scheme has been designed but never implemented.

The front-end today works mostly in one direction: it applies a translator to the

high-level language specifications and proofs, and executes the encodings through the back-ends, but does little parsing of the resulting output from the verification tool. A full, two-way front-end should also be able to interpret the encoded parts of the output from the back-end and translate the information about conjectures and hypotheses back to the high-level language.

The front-end could also be improved with respect to editing features. The syntax directed capabilities existing today present only few examples of what could be done; an intelligent routine for generation of different kinds of proofs would for example be a future possibility. A 'folding' mode for hiding parts of proofs that are outside the current scope could be very useful, as well as a more flexible 'make' system, for doing various sorts of batch verification.

Appendices

Appendix A

Syntax of the TLP language

A.1 Reserved words

Act	**Exists**	**Qed**
Action[s]	**False**	**Rigid**
Array	**Forall**	**SF**
Assume	**Goal**	**Statefunction**[s]
Bar	**Hyp**	**Step**
Bool	**If**	**Temp**
Bottom	**in**	**Temporal**
By-Cases	**Include**	**then**
By-Implication	**is**	**Theorem**
By-Contradiction	**Lemma**	**Transitionfunction**[s]
By-LTL	**Lambda**	**True**
Case	**Method**[s]	**Unchanged**
Constant[s]	**Name**	**Use**
ContraHyp	**Operator**[s]	**Val**
Do	**Predicate**[s]	**Value**[s]
else	**Proof**	**Variable**[s]
Enabled	**Prove**	**WF**
End		

A.2 Reserved symbols

`=`	`~=`	`\/`	`/\`	`~`	`=>`	`<=>`	`==`
`[]`	`<>`	`~>`	`'`	`(`	`)`	`[`	`]`
`<`	`>`	`{`	`}`	`,`	`@`	`_`	`.`
`:`	`|`	`:=`	`:in`	`->`	`<-`	`%`	`%-`
`**`	`(*`	`*)`	`[*`	`*]`	#[*alphanum*]		

A.3 Grammar

In the following $n^{*\mathtt{nl}}$ means a possibly empty list of n's, all ended by a newline; $n^{+\mathtt{comma}}$ means a nonempty list of n's separated by '`,`'s; and $n^{+\mathtt{spc}}$ means a nonempty list of n's

separated by blanks. {...|...} denotes alternatives and [...] denotes optional parts.

Newlines in grammar-rules indicate where newlines are needed. Optional newlines are allowed between definitions, proofs, etc., and inside formulas and lists after any infix operator and the separators '`,`' and '`*`'. A newline character can be put *anywhere* between two grammatical objects, when escaped by a preceding backslash '\'.

Two sorts of comments are allowed, which are not described in the grammar below (as they are not considered as grammatical objects):

- *endline comments* begin with a '`%`' and end with a newline. They are regarded just as newline characters and can appear in the same positions as these.
- *parenthesized comments* begin with a '`(*`' and end with a '`*)`', always containing a balanced amount of these two symbols. They are regarded as space, and can appear between any two grammatical objects.

tlp-script ::= *item*$^{*\mathrm{nl}}$

item ::= *name*
| *directive*
| *declaration*
| *definition*
| *proof*
| *methods-definition*
| *method-group*

name ::= **Name** *ident*

directive ::= `%-` **Use** *file-name*$^{+\mathrm{spc}}$
| `%-` **Bottom**
| `%-` **Include** *file-name*$^{+\mathrm{spc}}$

declaration ::= **Values**
value-dec$^{*\mathrm{nl}}$
| **Constants**
const-dec$^{*\mathrm{nl}}$
| **Operators**
operator-dec$^{*\mathrm{nl}}$
| **Rigid Variables**
rigvar-dec$^{*\mathrm{nl}}$
| **Variables**
var-dec$^{*\mathrm{nl}}$

const-dec ::= (*ident*[*param*])$^{+\mathrm{comma}}$ **:** {**Val** | **Bool**}

value-dec ::= (*ident*[*param*])$^{+\mathrm{comma}}$

operator-dec ::= {*ident* | *infix-ident*}$^{+\mathrm{comma}}$ **:** {**Val** | **Bool**}$^{+\mathrm{comma}}$ `->` {**Val** | **Bool**}

rigvar-dec ::= *ident*$^{+\text{comma}}$ **:** {**Val** | **Bool**}

var-dec ::= *ident*$^{+\text{comma}}$ **:** {**Val** | **Bool**}

definition ::= **Statefunctions**
def$^{*\text{nl}}$
| **Predicates**
def$^{*\text{nl}}$
| **Transitionfunctions**
def$^{*\text{nl}}$
| **Actions**
def$^{*\text{nl}}$
| **Temporal**
def$^{*\text{nl}}$

def ::= *def-name* == *indent-expr*

proof ::= {**Theorem** | **Lemma**} *ident*
[**] *goal*
Proof
proof-item$^{*\text{nl}}$
Qed

goal ::= *indent-expr*
| **Assume** *expr*$^{+\text{comma}}$ **Prove** *indent-expr*

proof-item ::= *method-group*
| *subproof*
| *method-use*
| **By-Cases**
| **By-Implication**
| **By-Contradiction**
| **By-LTL**

subproof ::= *label* [**] *subgoal*
proof-item$^{*\text{nl}}$
Qed

subgoal ::= *indent-expr*
| **Assume** *expr*$^{+\text{comma}}$ [**Prove** *indent-expr*]
| **Case** *indent-expr*

methods-definition ::= **Methods**
method-def$^{*\text{nl}}$

method-def ::= *ident method-defpar*$^{+\text{spc}}$ **is**
method-defuse$^{*\text{nl}}$

End

method-defuse ::= {**Do** | *ident*} *method-defusepar*$^{+\mathrm{spc}}$

method-group ::= {**Act** | **Temp**}
method-use$^{*\mathrm{nl}}$
End

method-use ::= {**Do** | *ident*} *method-usepar*$^{+\mathrm{spc}}$

indent-expr ::= *expr*
| *expr*
=> *indent-expr*
| *expr*
<=> *indent-expr*
| *expr*
~> *indent-expr*
| *pre-op-expr*$^{*\mathrm{nl}}$

pre-op-expr ::= /\ {*expr* | *pre-op-expr*}
| \/ {*expr* | *pre-op-expr*}
| => {*expr* | *pre-op-expr*}
| <=> {*expr* | *pre-op-expr*}

expr ::= (*expr*$^{+\mathrm{comma}}$)
| *expr* /\ *expr*
| *expr* \/ *expr*
| ~*expr*
| *expr* => *expr*
| *expr* <=> *expr*
| [] *expr*
| <> *expr*
| *expr* ~>*expr*
| **WF**(*expr*, *expr*)
| **SF**(*expr*, *expr*)
| [*expr*]_*expr*
| <*expr*>_*expr*
| **Unchanged**(*expr*$^{+\mathrm{comma}}$)
| **Enabled**<*ident*[*param*]>_*ident*[*param*]
| **Bar**(*expr*$^{+\mathrm{comma}}$)
| *expr* = *expr*
| *expr* ~=*expr*
| *expr* **in** *expr*
| **If** *expr* **then** *expr* **else** *expr*
| **Lambda** *ident* **in** *expr* : *expr* [*ident*[*param*]:{**Val** | **Bool**}*]
| [*expr* -> *expr*]
| **Forall** *ident* [**in** *expr*] : *expr* [*ident*[*param*]*]

| **Forall** (*[expr,] ident[param]*)
| **Exists** *ident* [**in** *expr*] : *expr* [***ident[param]****]
| **Exists** ([*expr,] ident[param]*)
| {*ident* **in** *expr* : *expr* [***ident[param]****]}
| {*expr* | *ident* **in** *expr* [***ident[param]****]}
| *expr infix-ident expr*
| *expr* @ *expr*
| *expr* [*expr*]
| *expr* [*expr* := *expr*]
| *expr* [*expr* :**in** *expr*]
| *expr'*
| *ident*['][(*expr*$^{+\text{comma}}$)]
| **True**
| **False**

param ::= (*ident*$^{+\text{comma}}$)

ident ::= {A–Z | a–z | 0–9}$^{+}$

infix-ident ::= {! | # | $ | & | * | + | . | / | < | > | = | @ | \ | ^ | | | ~ | -}$^{+}$
| *ident*

def-name ::= *ident[param]*
| **Enabled<***ident[param]***>_***ident[param]*
| **Bar(***ident[param]***)**

method-defpar ::= *ident*
| {#n | #nl | #t | #f | #fl | #il}

method-usepar ::= *method-idpar*$^{+\text{comma}}$
| *method-inst*$^{+\text{comma}}$
| (*expr*)
| "*string*"

method-idpar ::= *method-ident*['][[*expr*$^{+\text{comma}}$]]
method-ident ::= {A–Z | a–z | 0–9}{A–Z | a–z | 0–9 | _ | *}$^{+}$[*method-idlabel*]

method-idlabel ::= <{0–9}$^{+}$>{0–9}*[.{0–9}$^{+}$[:{0–9}$^{+}$]]

method-inst ::= *method-ident* <- (*expr*)
| *method-ident* <- *method-idpar*

string ::= {Any ASCII-character other than "}$^{+}$

method-defusepar ::= *method-xidpar*$^{+\text{comma}}$
| *method-xinst*$^{+\text{comma}}$
| (*expr*)

| "*string*"

method-xidpar ::= *method-xident*['][[*expr*$^{+\text{comma}}$]]
| #{0–9}$^{+}$

method-xident ::= *method-ident*
| #{0–9}$^{+}$

method-xinst ::= *method-xident* <- (*expr*)
| *method-xident* <- *method-xidpar*

file-name ::= {A–Z | a–z | 0–9 | _ | . | , | : | / | ~ | ! |
? | @ | # | $ | ^ | + | = | -}$^{+}$

label ::= <{0–9}$^{+}$>{0–9}$^{+}$

Appendix B

Logical basis for reasoning with the Larch Prover

In this appendix we present the rules that are used for reasoning in the proofs we have shown. This is the rules that appear in the files `quant_act.lp`, `base_temp.lp`, `temporal_temp.lp`, and `tla_temp.lp`, that are part of the TLP distribution.

To improve readability, the rules have been typeset using different fonts and mathematical symbols where appropriate. The conversion from pure LP code is given below:

Deduction rules: A rule of the form **when** $F_1 \ldots F_n$ **yield** G is written as

$$\frac{\begin{array}{c} F_1 \\ \vdots \\ F_n \end{array}}{G}$$

If any of the variables in the premises $F_1 \ldots F_n$ are universally quantified by the LP construct (**forall** x), these are written in a sans serif font: x. In the temporal rules we simply delete the behaviour variable σ in hypotheses where it is universally quantified, as well as in in the consequence G, if it is free (not bound by the premises). The rule

$$\begin{array}{ll} \textbf{when} & (\textbf{forall}\ \sigma) \\ & \sigma \models F \\ \textbf{yield} & \sigma \models G \end{array}$$

is thus simply written as

$$\frac{\models F}{\models G}$$

TLA and LP boolean operators: The TLA boolean operators used in the temporal environment are written as *true*, *false*, $\neg$, $\vee$, $\wedge$, $\Longrightarrow$, $\Longleftrightarrow$, and $==$, while the built-in LP operators are written as true, false, not, |, & , $\Rightarrow$, $\Leftrightarrow$, and =.

Operators: Many of the prefix operators declared in LP have been replaced by mathematical (sometimes infix or mixfix) symbols, with the following conventions:

LP:	typeset as:
for_all	$\forall$
exists	$\exists$
lambda	λ
$f\,.\,x$	$f(x)$
$f\,@\,x$	$f(x)$
$in(x, S)$	$x \in S$
$funcSpace(S, T)$	$[S \to T]$
$updateEq(f, n, e)$	$F[n := e]$
$updateIn(f, n, e)$	$F[n :\in e]$
$subSet(S, T)$	$S \subset T$
Box	$\Box$
Dmd	$\Diamond$
$BAct(A, f)$	$[A]_f$
$DAct(A, f)$	$\langle A \rangle_f$
$WF(f, A)$	$\mathrm{WF}_f(A)$
$SF(f, A)$	$\mathrm{SF}_f(A)$
$Prime(P)$	P'

Variable names: All variable names used in the rules have a prefix such as '*i_*' or '*pred_*' that is used for preventing name clashes with the user defined operators and variables, and which in some cases as well specify what kind of expression the variable is expected to be instantiated by. The prefixes would be distracting here and has thus been removed. (When you use e.g. the TLA rules by executing one of the syntactic macros inside Emacs, or use methods such as *ProveForall* you will automatically get the correct prefixes.)

B.1 Rules for action reasoning

B.1.1 Quantification

ProveForall ($\forall I$)*:*

$$\frac{f(\mathsf{x})}{\forall(f)}$$

ProveForall ($\forall I$)*:*

$$\frac{\mathsf{x} \in S \Rightarrow f(\mathsf{x})}{\forall(S, f)}$$

UseForall ($\forall E$)*:*

$$\frac{\forall(f)}{f(x)}$$

UseForall ($\forall E$)*:*

$$\frac{\begin{array}{l}\forall(S, f) \\ x \in S\end{array}}{f(x)}$$

ProveExists ($\exists I$)*:*

$$\frac{f(x)}{\exists(f)}$$

ProveExists ($\exists I$)*:*

$$\frac{\begin{array}{l}f(x) \\ x \in S\end{array}}{\exists(S, f)}$$

UseExists ($\exists E$)*:*

$$\frac{\begin{array}{l}\exists(f) \\ f(\mathsf{x}) \Rightarrow B\end{array}}{B}$$

UseExists ($\exists E$)*:*

$$\frac{\begin{array}{l}\exists(S, f) \\ (\mathsf{x} \in S \ \&\ f(\mathsf{x})) \Rightarrow B\end{array}}{B}$$

ProveNotForall:

$$\frac{\begin{array}{l}\exists(S, g) \\ g(\mathsf{x}) = \mathrm{not}(f(\mathsf{x}))\end{array}}{\mathrm{not}(\forall(S, f))}$$

UseNotForall:

$$\frac{\begin{array}{l}\text{not}(\forall(S, g)) \\ g(\mathsf{x}) = \text{not}(f(\mathsf{x}))\end{array}}{\exists(S, f)}$$

ProveNotExists:

$$\frac{\begin{array}{l}\forall(S, g) \\ g(\mathsf{x}) = \text{not}(f(\mathsf{x}))\end{array}}{\text{not}(\exists(S, f))}$$

UseNotExists:

$$\frac{\begin{array}{l}\text{not}(\exists(S, g)) \\ g(\mathsf{x}) = \text{not}(f(\mathsf{x}))\end{array}}{\forall(S, f)}$$

B.1.2 Functions

UseLambda:

$$\frac{\begin{array}{l}\lambda(S, f) = \lambda(S, g) \\ x \in S\end{array}}{f(x) = g(x)}$$

ProveLambda:

$$\frac{\mathsf{x} \in S \Rightarrow f(\mathsf{x}) = g(\mathsf{x})}{\lambda(S, f) = \lambda(S, g)}$$

Apply:

$$\frac{x \in S}{\lambda(S, f)(x) \quad \rightarrow \quad f(x)}$$

UseFuncSpace:

$$\frac{\begin{array}{l}f \in [S \rightarrow T] \\ x \in S\end{array}}{f(x) \in T}$$

ProveFuncSpace:

$$\frac{\begin{array}{l}dom(f) = S \\ x \in S \Rightarrow f(x) \in T\end{array}}{f \in [S \rightarrow T]}$$

ProveFuncSpaceLambda:

$$\frac{\mathsf{x} \in S \Rightarrow \lambda(S, f)(\mathsf{x}) \in T}{\lambda(S, f) \in [S \rightarrow T]}$$

ProveFuncSpaceUpdateEq:

$$\frac{\begin{array}{l} f \in [S \to T] \\ n \in S \\ v \in T \end{array}}{f[n := v] \in [S \to T]}$$

ProveFuncSpaceUpdateIn:

$$\frac{\begin{array}{l} f \in [S \to T] \\ g \in f[n :\in U] \\ n \in S \\ \mathsf{x} \in U \Rightarrow \mathsf{x} \in T \end{array}}{g \in [S \to T]}$$

UseDom:

$$dom(\lambda(S, f)) \quad \to \quad S$$

UseDom:

$$\frac{f \in [S \to T]}{dom(f) \quad \to \quad S}$$

UpdateEq:

$$\frac{n \in dom(f)}{f[n := e](n) \quad \to \quad e}$$

UpdateEq:

$$\frac{\begin{array}{l} n \in dom(f) \\ (m = n) == false \end{array}}{f[m := e](n) \quad \to \quad f(n)}$$

UpdateIn:

$$\frac{\begin{array}{l} n \in dom(f) \\ g \in f[n :\in S] \end{array}}{g(n) \in S}$$

UpdateIn:

$$\frac{\begin{array}{l} n \in dom(f) \\ g \in f[m :\in S] \\ (n = m) == false \end{array}}{g(n) \quad \to \quad f(n)}$$

B.1.3 Sets

UseSetof:

$$\frac{x \in setof(S, f)}{x \in S \;\&\; f(x)}$$

ProveSetof:

$$\frac{f(x)}{x \in S \Rightarrow x \in setof(S, f)}$$

UseSetexp:

$$\frac{\begin{array}{l} x \in setexp(S, f) \\ (y \in S \;\&\; x = f(y)) \Rightarrow b \end{array}}{b}$$

ProveSetexp:

$$\frac{\begin{array}{l} x = f(y) \\ y \in S \end{array}}{x \in setexp(S, f)}$$

UseSubSet:

$$\frac{\begin{array}{l} S \subset T \\ x \in S \end{array}}{x \in T}$$

ProveSubSet:

$$\frac{\mathsf{x} \in S \Rightarrow \mathsf{x} \in T}{S \subset T}$$

SetEq:

$$\frac{\begin{array}{l} S \subset T \\ T \subset S \end{array}}{S = T}$$

B.2 Rules for temporal reasoning

B.2.1 Normalization, introduction, and elimination rules

Boolean operators

Active rules:
TrueNorm:

$$\sigma \models \mathit{true}$$

FalseNorm:

$$\mathrm{not}(\sigma \models \mathit{false})$$

OrNorm:

$$\sigma \models F \vee G \quad \rightarrow \quad (\sigma \models F) \mid (\sigma \models G)$$

AndNorm:

$$\sigma \models F \wedge G \quad \rightarrow \quad (\sigma \models F) \,\&\, (\sigma \models G)$$

ImplNorm:

$$\sigma \models F \Longrightarrow G \quad \rightarrow \quad (\sigma \models F) \Rightarrow (\sigma \models G)$$

BiImplNorm:

$$\sigma \models F \Longleftrightarrow G \quad \rightarrow \quad (\sigma \models F) = (\sigma \models G)$$

Passive rules:
NegElim:

$$\frac{\sigma \models \neg(F)}{\mathrm{not}(\sigma \models F)}$$

NegIntro:

$$\frac{\mathrm{not}(\sigma \models F)}{\sigma \models \neg(F)}$$

Always

BoxIntro:

$$\frac{\models F}{\models \Box(F)}$$

BoxElim1:

$$\frac{\sigma \models \Box(F)}{\sigma \models F}$$

BoxElim2:

$$\sigma \models \Box(\Box(F)) \quad \rightarrow \quad \sigma \models \Box(F)$$

ImplBox1:

$$\frac{\models F \Longrightarrow G}{\models \Box(F) \Longrightarrow \Box(G)}$$

ImplBox2:

$$\frac{\sigma \models \Box(F \Longrightarrow G)}{\sigma \models \Box(F) \Longrightarrow \Box(G)}$$

EqBox1:

$$\frac{\models F \Longleftrightarrow G}{\models \Box(F) \Longleftrightarrow \Box(G)}$$

EqBox2:

$$\frac{\sigma \models \Box(F \Longleftrightarrow G)}{\sigma \models \Box(F) \Longleftrightarrow \Box(G)}$$

Eventually

DmdIntro1:

$$\frac{\sigma \models F}{\sigma \models \Diamond(F)}$$

DmdIntro2:

$$\frac{\sigma \models F \Longrightarrow G}{\sigma \models F \Longrightarrow \Diamond(G)}$$

ImplDmd1:

$$\frac{\models F \Longrightarrow G}{\models \Diamond(F) \Longrightarrow \Diamond(G)}$$

ImplDmd2:

$$\frac{\sigma \models \Box(F \Longrightarrow G)}{\sigma \models \Box(\Diamond(F) \Longrightarrow \Diamond(G))}$$

ImplDmd3:

$$\frac{\models \Box(F) \Longrightarrow (G \Longrightarrow H)}{\models \Box(F) \Longrightarrow (\Diamond(G) \Longrightarrow \Diamond(H))}$$

EqDmd1:

$$\frac{\models F \Longleftrightarrow G}{\models \Diamond(F) \Longleftrightarrow \Diamond(G)}$$

EqDmd2:

$$\frac{\sigma \models \Box(F \Longleftrightarrow G)}{\sigma \models \Box(\Diamond(F) \Longleftrightarrow \Diamond(G))}$$

Leadsto

Leadsto:

$$F \leadsto G \quad \rightarrow \quad \Box(F \Longrightarrow \Diamond(G))$$

B.2.2 Distributive laws

OrDmd:

$$\Diamond(F \vee G) \quad \rightarrow \quad \Diamond(F) \vee \Diamond(G)$$

AndBox:

$$\Box(F \wedge G) \quad \rightarrow \quad \Box(F) \wedge \Box(G)$$

AndDmdBox:

$$\Diamond(\Box(F)) \wedge \Diamond(\Box(G)) \quad \rightarrow \quad \Diamond(\Box(F \wedge G))$$

B.2.3 The Lattice rule

Lattice:

$$\frac{\begin{array}{l} \models (F \wedge \mathsf{c} \in S) \Longrightarrow (H(\mathsf{c}) \leadsto (G \vee \exists(S, f(\mathsf{c})))) \\ f(\mathsf{c})(\mathsf{d}) = (s(\mathsf{c})(\mathsf{d}) \wedge H(\mathsf{d})) \\ g(\mathsf{c}) = H(\mathsf{c}) \\ \mathit{Wellfounded}(S, s) \end{array}}{\models F \Longrightarrow \exists(S, g) \leadsto G}$$

B.2.4 Grønning lattice rules

LatticeReflexivity:

$$\sigma \models A \leadsto A$$

LatticeDisjunctionElim:

$$\frac{\sigma \models (A \vee B) \leadsto C}{\sigma \models A \leadsto C}$$

LatticeDisjunctionIntr:

$$\frac{\begin{array}{c} \sigma \models A \leadsto C \\ \sigma \models B \leadsto C \end{array}}{\sigma \models (A \vee B) \leadsto C}$$

LatticeDiamond:

$$\frac{\begin{array}{l} \sigma \models B \leadsto D \\ \sigma \models A \leadsto (B \vee C) \\ \sigma \models C \leadsto D \end{array}}{\sigma \models A \leadsto D}$$

LatticeTriangle:

$$\frac{\sigma \models B \leadsto D \quad \sigma \models A \leadsto (B \vee D)}{\sigma \models A \leadsto D}$$

LatticeTransitivity:

$$\frac{\sigma \models A \leadsto B \quad \sigma \models B \leadsto C}{\sigma \models A \leadsto C}$$

B.2.5 Syntactic definitions

BAct:

$$[A]_f \quad \rightarrow \quad A \vee \mathit{Unchanged}(f)$$

DAct:

$$\langle A \rangle_f \quad \rightarrow \quad A \wedge \neg(\mathit{Unchanged}(f))$$

Unchanged:

$$\mathit{Unchanged}(f) \quad \rightarrow \quad f = f'$$

WeakFairness:

$$\mathrm{WF}_f(A) \quad \rightarrow \quad \Box(\Diamond(\langle A \rangle_f)) \vee \Box(\Diamond(\neg(\mathit{Enabled}(\langle A \rangle_f))))$$

StrongFairness:

$$\mathrm{SF}_f(A) \quad \rightarrow \quad \Box(\Diamond(\langle A \rangle_f)) \vee \Diamond(\Box(\neg(\mathit{Enabled}(\langle A \rangle_f))))$$

B.2.6 TLA rules

Basic

TLA1:

$$\sigma \models \Box(P) \Longleftrightarrow (P \wedge \Box([P \Longrightarrow P']_P)$$

TLA2:

$$\frac{\models (P \wedge [A]_f) \Longrightarrow (Q \wedge [B]_g)}{\models (\Box(P) \wedge \Box([A]_f)) \Longrightarrow (\Box(Q) \wedge \Box([B]_g))}$$

INV

INV1.1:

$$\frac{\models I \Longrightarrow ([N]_f \Longrightarrow I')}{\models (I \wedge \Box([N]_f)) \Longrightarrow \Box(I)}$$

INV1.2:

$$\frac{\models Assump \Longrightarrow (I \Longrightarrow ([N]_f \Longrightarrow I'))}{\models (\Box(Assump) \wedge I \wedge \Box([N]_f)) \Longrightarrow \Box(I)}$$

INV1.3:

$$\frac{\models Assump \Longrightarrow (Assump' \Longrightarrow (I \Longrightarrow ([N]_f \Longrightarrow I')))}{\models (\Box(Assump) \wedge I \wedge \Box([N]_f)) \Longrightarrow \Box(I)}$$

INV2:

$$\sigma \models \Box(I) \Longrightarrow (\Box([N]_f) \Longleftrightarrow \Box([N \wedge I \wedge I']_f))$$

WF

WF1:

$$\frac{\begin{array}{c}\models Assump \Longrightarrow ((P \wedge [N]_f) \Longrightarrow (P' \vee Q')) \\ \models Assump \Longrightarrow ((P \wedge \langle N \wedge A\rangle_f) \Longrightarrow Q') \\ \models Assump \Longrightarrow (P \Longrightarrow Enabled(\langle A\rangle_f))\end{array}}{\models (\Box(Assump) \wedge \Box([N]_f) \wedge \mathrm{WF}_f(A)) \Longrightarrow (P \rightsquigarrow Q)}$$

WF2:

$$\frac{\begin{array}{l}\models Assump \Longrightarrow (\langle N \wedge B\rangle_f \Longrightarrow \langle M\rangle_g) \\ \models Assump \Longrightarrow ((P \wedge P' \wedge \langle N \wedge A\rangle_f) \Longrightarrow B) \\ \models Assump \Longrightarrow ((P \wedge Enabled(\langle M\rangle_g)) \Longrightarrow Enabled(\langle A\rangle_f)) \\ \models (\Box(Assump) \wedge \Box([N \wedge \neg(B)]_f) \wedge \mathrm{WF}_f(A) \wedge \Box(F)) \Longrightarrow \Diamond(\Box(P)) \\ \models Phi \Longrightarrow \Box(Assump) \\ \models Phi \Longrightarrow (\Box([N]_f) \wedge \mathrm{WF}_f(A) \wedge \Box(F))\end{array}}{\models Phi \Longrightarrow \mathrm{WF}_g(M)}$$

SF

SF1:

$$\frac{\begin{array}{l}\models Assump \Longrightarrow ((P \wedge [N]_f) \Longrightarrow (P' \vee Q')) \\ \models Assump \Longrightarrow ((P \wedge \langle N \wedge A\rangle_f) \Longrightarrow Q') \\ \models (\Box(Assump) \wedge \Box(P) \wedge \Box([N]_f) \wedge \Box(F)) \Longrightarrow \Diamond(Enabled(\langle A\rangle_f))\end{array}}{\models (\Box(Assump) \wedge \Box([N]_f) \wedge \mathrm{SF}_f(A) \wedge \Box(F)) \Longrightarrow (P \rightsquigarrow Q)}$$

SF2:

$$\frac{\begin{array}{l}\models \mathit{Assump} \implies (\langle N \land B\rangle_f \implies \langle M\rangle_g)\\ \models \mathit{Assump} \implies ((P \land P' \land \langle N \land A\rangle_f) \implies B)\\ \models \mathit{Assump} \implies ((P \land \mathit{Enabled}(\langle M\rangle_g)) \implies \mathit{Enabled}(\langle A\rangle_f))\\ \models (\Box(\mathit{Assump}) \land \Box([N \land \neg(B)]_f) \land \mathrm{SF}_f(A) \land \Box(F)) \implies \Diamond(\Box(P))\\ \models \mathit{Phi} \implies \Box(\mathit{Assump})\\ \models \mathit{Phi} \implies (\Box([N]_f) \land \mathrm{SF}_f(A) \land \Box(F))\end{array}}{\models \mathit{Phi} \implies \mathrm{SF}_g(M)}$$

Additional

BoxSF:

$$\sigma \models \mathrm{SF}_f(N) \iff \Box(\mathrm{SF}_f(N))$$

BoxWF:

$$\sigma \models \mathrm{WF}_f(N) \iff \Box(\mathrm{WF}_f(N))$$

SFImplWF:

$$\frac{\sigma \models \mathrm{SF}_f(N)}{\sigma \models \mathrm{WF}_f(N)}$$

PrimeBox:

$$\sigma \models \Box(P \land P') \iff \Box(P)$$

BoxPrime:

$$\frac{\sigma \models \Box(P)}{\sigma \models \Box(P \land P')}$$

Appendix C

Standard methods

Methods from `base/methods.tlp` and `base/quantmethods.tlp` for reasoning with the Larch Prover.

C.1 General methods

Methods

Expand `#nl` in `#nl` **is**
  Do normalize `#2` with `#1`
End

Rewrite `#nl` with `#nl` **is**
  Do normalize `#1` with `#2`
End

Instantiate `#nl` with `#il` **is**
  Do instantiate `#2` in `#1`
End

Crit `#nl` with `#nl` **is**
  Do "critical-pairs" `#1` with `#2`
End

Apply `#nl` to `#nl` **is**
  Do apply `#1` to `#2`
End

Activate `#nl` **is**
  Do make active `#1`
End

Passify `#nl` **is**
  Do make passive `#1`
End

Immunize `#nl` **is**
  Do make immune `#1`
End

NonImmunize `#nl` **is**

Do make nonimmune #1
End

Use #nl **is**
NonImmunize #1
Activate #1
End

OrderedPair on #nl **is**
Do normalize #1 with *OrderedPair*
End

C.2 Methods for temporal reasoning

Methods

INV1 with #il **is**
Instantiate *INV1* with #1
End

INV2 with #il **is**
Instantiate *INV2* with #1
End

WF1 with #il **is**
Instantiate *WF1* with #1
End

WF2 with #il **is**
Instantiate *WF2* with #1
End

SF1 with #il **is**
Instantiate *SF1* with #1
End

SF2 with #il **is**
Instantiate *SF2* with #1
End

Lattice with #il **is**
Instantiate *Lattice* with #1
End

UseTempFact #nl **is**
Passify #1
Instantiate #1 with σ <- σc
End

LatticeDisjunctionIntr #t #t #t **is**
Instantiate *LatticeDisjunctionIntr* with *A* <- #1,
B <- #2,
C <- #3
End

LatticeReflexivity #t **is**
  Instantiate *LatticeReflexivity* with A <- #1
End

UseLatticeRules **is**
  Activate *LatticeReflexivity*,
    LatticeDisjunctionElim,
    LatticeDiamond,
    LatticeTriangle,
    LatticeTransitivity
End

C.3 Methods for dealing with quantification, sets, and functions

Methods

ProveForall on #f **is**
  Instantiate *ProveForall* with f <- #1
End

UseForall on #f with #t **is**
  Instantiate *UseForall* with f <- #1, x <- #2
End

ProveExists on #f with #t **is**
  Instantiate *ProveExists* with f <- #1, x <- #2
End

UseExists on #f **is**
  Instantiate *UseExists* with f <- #1
End

UseNotExiststst on #f and #f set #t **is**
  Instantiate *UseNotExists* with g <- #1, f <- #2, S <- #3
End

UseNotForall on #f and #f set #t **is**
  Instantiate *UseNotForall* with g <- #1, f <- #2, S <- #3
End

ProveLambda on #f #f #t **is**
  Instantiate *ProveLambda* with f <- #1, g <- #2, S <- #3
End

UseLambda on #f #f #t #t **is**
  Instantiate *UseLambda* with f <- #1, g <- #2, x <- #3, S <- #4
End

ApplyLambda vbl #t set #t **is**
  Instantiate *Apply* with x <- #1, S <- #2
End

ApplyLambdaFunc vbl #t set #t func #f **is**

Instantiate *Apply* with x <- #1, S <- #2, f <- #3
End

UpdateIn vbl #t updated #t **is**
Instantiate *UpdateIn* with n <- #1, f <- #2
End

UpdateEq vbl #t updated #t **is**
Instantiate *UpdateEq* with n <- #1, f <- #2
End

UseDom #t **is**
Instantiate *UseDom* with f <- #1
End

UseFuncSpace func #t vbl #t **is**
Instantiate *UseFuncSpace* with f <- #1, x <- #2
End

ProveFuncSpace func #t domain #t codomain #t **is**
Instantiate *ProveFuncSpace* with f <- #1, S <- #2, T <- #3
End

ProveFuncSpaceLambda func #f domain #t codomain #t **is**
Instantiate *ProveFuncSpaceLambda* with S <- #2, f <- #1, T <- #3
End

ProveFuncSpaceUpdateIn orig #t domain #t codomain #t **is**
Instantiate *ProveFuncSpaceUpdateIn* with f <- #1, S <- #2, T <- #3
End

ProveFuncSpaceUpdateEq orig #t domain #t codomain #t **is**
Instantiate *ProveFuncSpaceUpdateEq* with f <- #1, S <- #2, T <- #3
End

UseSetof set #t func #f **is**
Instantiate *UseSetof* with S <- #1, f <- #2
End

ProveSetof elem #t set #t func #f **is**
Instantiate *ProveSetof* with f <- #3, x <- #1, S <- #2
End

Appendix D

TLP-mode commands

tlp-mode ()	Major mode for editing TLP specifications. tlp-mode runs the hook tlp-mode-hook without any arguments.
tlp-make-tlp-frame ()	Create a frame for working with TLP.
tlp-set-tlp-colors ()	Set frame colours that goes well with the colours of a hilit TLP-buffer.
tlp-clean-buffer ()	Remove the correction-overlay. In Emacs-18, removes the TLP arrow marker.
tlp-reconfigure-windows (&optional make)	Reset the window and buffer setup. This is good if things should start to look a little strange. MAKE optionally specifies to display the make buffer instead of the edit buffer.
tlp-find-master ()	Go back to the TLP master file. This selects the tlp-master-buffer, if it exists, in the current window.
tlp-beginning-of-step ()	Move to the beginning of the current step. Moves to the previous or encapsulating one if at the beginning of this one. The beginning is defined as the beginning of the line containing the "Theorem" or label.
tlp-end-of-outer-step ()	Move to the beginning of the last line of the current step. As tlp-end-of-step, but takes you to the end of the encapsulating step if such exists.
tlp-end-of-step ()	Move to the beginning of the first line after the current step. Moves to the next substep of the current step if such exists. The end of the current step is defined as the end of the line containing the "Qed" of the current step.

tlp-mark-step ()	Set the region to the current proof step. This is defined as the region including the current Qed construct and its corresponding label.
tlp-indent-line ()	Indent the current line as TLP code. This is still a rather naive implementation, and not all interesting cases are covered. Most notably, nothing will be done about multi-line formulas.
tlp-step-indent ()	Re-indent the current proof step. This is defined as the region including the current Qed construct and its corresponding label.
tlp-fill-paragraph (&optional justify)	Fill paragraph at or after point. Prefix arg means justify as well. Like fill-paragraph, but handles TLP endline comments. If any part of the current line is a comment, fills the comment or the paragraph that point is in, preserving the comment's indentation.
tlp-insert-dmacro (name)	Insert the dmacro NAME. Just like insert-dmacro, but will highlight the inserted text, if highlighting is turned on. It prompts for NAME. When called from Lisp programs, NAME is a string; if NAME is not a valid dmacro in the current buffer, then NAME itself is inserted.
tlp-command ()	Query the user for a command to run. Default is dependent on the current state of the verification system.
tlp-make-but-dont (file)	Display in the *TLP make* window, what the verify command will do.
tlp-translate ()	Translate the current buffer. Works similar to tlp-verify, but only runs the translator on the current buffer.

tlp-verify (&optional no-ask sorts)	Verify the proofs of the current buffer. The proofs are verified with respect to the current proof sort (can be changed with tlp-toggle-sort). All this function really does is to set the make hooks appropriately and then start make up in the background. The buffer should not be read-only. Be aware that it will cause problems to restart verification during verification – Emacs can only handle one session at a time. The functions used for callback is tlp-verify-done and tlp-verify-error. Any previously set values of the make hooks is saved and restored when verification completes. Verification will make buffer read-only and will reset the overlay arrow, tlp-faulty-file, tlp-source-buffer and tlp-correction-region. The optional argument NO-ASK is obsolete, the user is never asked for confirmation. Optional second argument SORTS is a list of the proof sorts with respect to which the proof should be verified, instead of just the current.
tlp-verify-all (&optional no-ask)	Extensively verify the proofs of the current buffer. As tlp-verify, but checks the proof with respect to each proof sort, one after one, in the order determined by tlp-proof-sorts.
tlp-quit ()	Quit the current TLP session.
tlp-barring (&optional value)	Generate barred definitions. If VALUE is "+" or "−", the generating of barred versions of variables etc. is set accordingly. Otherwise its value is toggled.
tlp-immunizing (&optional value)	Automatically immunize theorems. If VALUE is "+" or "−", automatic immunizing and passification of theorems is set accordingly. Otherwise its value is toggled.
tlp-instantiating (&optional value)	Do automatic instantiation of quantifier functions. If VALUE is "+" or "−", automatic instantiation of quantification functions is set accordingly. Otherwise its value is toggled.

tlp-proof-order (&optional value)	Change proof order. If VALUE is "+", verification will be done in pre-order, if it is "−", verification will be done in post-order. In all other cases, or without argument, the current proof-order is toggled.
tlp-registering (&optional value)	Generate registering hints. If VALUE is "+" or "−", the generating of registering hints is set accordingly. Otherwise its value is toggled.
tlp-toggle-mode (&optional sort)	Toggle the current verification mode. If SORT is any of the accepted proof sorts in the list tlp-proof-sorts, the proof sort is changed accordingly. Otherwise, and with no argument, the proof sort is changed to the next sort (circularly) of that list.
tlp-commit-correction ()	Commit the correction. This inserts the new lines from the edit buffer, or replaces the failing step in the source buffer with the corrected step from the edit buffer and marks this as unmodified.
tlp-execute-correction ()	Continue verification of the TLP specification. Verification is resumed with the corrected proof-step as starting point. If the edit buffer contains uncommitted changes, the user is prompted for committal.
tlp-find-next-error ()	Locate the next syntax error. (Uses tlp-find-error-tlp or tlp-find-error depending on whether you are in verification mode.)
tlp-widen-correction ()	Move the scope of the correction up one level. I.e. include the enclosing proof-step. This will always replace the current contents of the correction buffer, meaning that any uncommitted changes will be lost. An error is issued if on the top level.
tlp-widen-for-lp-syntax-error ()	Move the scope of the correction up one level. Used in the case of a step producing an unsuspected LP error. The part of the step from tlp-lp-syntax-error to (but not including) the Qed is put into the proof correction buffer.

tlp-widen-for-missing-qed ()	Move the scope of the correction up one level. Used in the case of a step having been verified before the Qed (i.e. an LP missing []). The part of the step from tlp-missing-qed to (but not including) the Qed is put into the proof correction buffer.
tlp-quit-correction ()	Quit the current correction. This kills the edit buffer, maintaining all committed changes. The correction overlay (or synchronization arrow) is not deleted, since the synchronization point is still valid.
tlp-dis-all ()	Display all facts of the proofs done yet.
tlp-dis-facts ()	Display all current facts. Displays all existing (TLP) theorems and lemmas as well as all facts and hypotheses of the current proof.
tlp-dis-goal ()	Display the current goal.
tlp-dis-hyp ()	Display the current hypotheses.
tlp-dis-lemma ()	Display a lemma.
tlp-dis-proof ()	Display the current status of the proof.
tlp-dis-theorem ()	Display a theorem.
tlp-display ()	Display some facts.
tlp-lp-exe (coms)	Execute the commands COMS in the LP buffer. COMS is either a string or a list of strings to be executed as LP commands. Each command must be a string to be executed. Non-strings are silently skipped. LP is run asynchronously, making a call-back whenever the LP is ready for the next command. Commands are buffered in the variable 'tlp-lp-com-buffer' and will be automatically removed, unless an error occurs. In that case, or when then the buffer runs empty, the value of 'tlp-callback-function' is called. Please refer to 'tlp-lp-filter' for more information on the callback mechanism. Note that if there is a running LP session, this will be used rather than starting a new. If you want to ensure a clean session, use tlp-lp-kill or tlp-lp-start with a non-nil argument, before calling tlp-lp-exe.

tlp-ltl-exe (coms)

Execute the commands COMS in the LTL buffer.

Just as tlp-lp-exe for the LTL back-end.

tlp-find-error ()

Locate a TLP error.

If tlp-faulty-file is non-nil, this function will find the corresponding TLP file in a buffer (if it exists), and try to locate the error.
If the error is a TLP error, subsequent calls to tlp-find-error will move to the next error. When being on the last error, the next call to tlp-find-error will not move point, but the following call will move point to the first error.
If the error was an LP error, the error message is returned, if found. This is found in the '`.lpscr`' file, and is the rest of the line following the '%%ERROR:' marker. The cursor is placed at the label of the current step, which is found by executing a "display conjecture" in LP. Note: In the special case where the error is due to an error in a 'pure' LP file, an LP file written by the user, the position close to the error, as specified by LP, will be marked, and no further action will be done. To proceed, one will have to quit verification.
If the error was an LTL error, the error message is returned, as found in the LTL buffer and the cursor is positioned at the right label in the proof.
The error is marked with an overlay arrow, unless tlp-no-arrow is non-nil. Note that the arrow covers the first two characters of the line. This function can be put into tlp-make-error-hook to find errors when make fails.

tlp-lp-correct-error ()

Interact with user in recovering from an LP verification error.
Assumes that point is located at the end of the relevant proof-step.

tlp-lp-kill ()

Kill the current LP session if any such exist.
The associated buffer is not changed.

tlp-ltl-correct-error ()

Interact with user in recovering from an LTL verification error.
Assumes that point is located at the beginning of the relevant proof-step.

tlp-ltl-kill ()

Kill the current LTL session if any such exist.
The associated buffer is not changed.

Appendix E

Installation

The pre-compiled versions of the TLP tools currently run on UNIX platforms of two kinds only, namely *Digital DECstations (MIPS)* with *Ultrix* and *Sun SPARCs* with *Solaris (SunOS)*. The translator itself can be compiled for other architectures, and the front-end will run without problems on all platforms that support GNU Emacs. To use TLP on other systems than the two mentioned, you will however also have to compile a modified version of the Larch Prover from MIT. Furthermore, the linear temporal logic checker that is supported in the current system will only be available for the mentioned systems.

The TLP tools, setup files, and documentation, can all be obtained by FTP from

`ftp.daimi.aau.dk (130.225.16.162)`

from the directory

`pub/tlp`

Alternatively, the same archive may be reached from the TLP WWW page at:

`http://www.daimi.aau.dk:/~urban/tlp/tlp.html`

The files are all compressed by the GNU compression tool, gzip, which you will need to uncompress them; gzip is available on many FTP servers around the world, usually at the same place that you get GNU Emacs from.

The directory contains the following files, where you should substitute ⟨*release*⟩ by the latest TLP release number:

`tlp-`⟨*release*⟩`.tar.gz`	Archive containing code and setup-files for the TLP system, documentation, and a number of examples.
`tlp-`⟨*release*⟩`.dec-ultrix.gz`	Compiled version of the TLP translator for DECstations running Ultrix.
`tlp-`⟨*release*⟩`.sparc-solaris.gz`	The same for Sun SPARCs running Solaris.
`lp2.4xl.dec-ultrix.gz`	Compiled version of the Larch Prover, slightly modified for interactive use in TLP, for DECstations.

`lp2.4xl.sparc-solaris.gz`	The same for SPARCs.
`bddt.dec-ultrix.gz`	Compiled version of the LTL checker, for DECstations.
`bddt.sparc-solaris.gz`	Compiled version of the LTL checker, for SPARCs.
`translator-`⟨*release*⟩`.src.tar.gz`	The sources for the TLP translator, written in New Jersey Standard ML.

In the following, substitute either `dec-ultrix` or `sparc-solaris` for ⟨*platform*⟩, depending on which platform you want to run the system on.

To get to use the TLP system, you should get the archive, `tlp-`⟨*release*⟩`.tar.gz`, and one of each of the binaries for the translator, LP, and the LTL checker. You only need to get the translator source if you think that you would like to do modifications or want to compile the translator yourself on some other architecture.

E.1 The Larch Prover

To do verification, you will need to install the Larch Prover, version 2 on your system. To get the interactive features of the front-end working properly (notably the widening of proof scopes), you will need the modified version LP 2.4xl that has been made available in the FTP directory.

Documentation for the Larch Prover can be obtained from

```
larch.lcs.mit.edu (18.26.0.95)
```

in the directory `pub/Larch`.

Note that if you have a non-modified version of LP 2, you will still be able to do verification correctly – the mentioned modifications concerns only the interactive feature described as widening of proof scope. The experimental versions LP 3.0 and 3.1 include some syntactical changes, however, and will not work correctly.

E.2 The linear temporal logic checker

If you also want to be able to do automatic checking of linear temporal logic formulas, you should get one of the 'bddt'-files for your respective architecture. The LTL checker *can only* be made available for DECstations and SPARCS.

E.3 Emacs

To run the interactive front-end, from which all reasoning is supposed to be done, you will need a version of GNU Emacs. Any release later than 19.22 should work without problems. TLP may also work with XEmacs (former Lucid Emacs) and Emacs 18 but these are not currently being supported. The latest release of GNU Emacs should be available from FTP servers all around the world.

E.4 How to install

When you have got the files you should do the following operations.

1. In a shell, run the command

   `gzip -cd tlp-`⟨*release*⟩`.tar.gz | (cd` ⟨*dir*⟩`; tar xvf -)`

   This will create a subdirectory `tlp-`⟨*release*⟩ in the existing directory ⟨*dir*⟩ containing the following items:

`README`	Instructions for installation of TLP.
`CHANGES`	List of recent changes.
`COPYING`	The GNU General Public License, for all the Emacs code.
`base`	Directory containing different initialization and setup files:
`bool_act.lp`	Basic boolean reasoning.
`base_act.lp`	Basic setup for action reasoning.
`quant_act.lp`	Setup for reasoning about quantifiers.
`base_temp.lp`	Basic setup for temporal reasoning.
`temporal_temp.lp`	Basic temporal rules.
`tla_temp.lp`	TLA rules.
`methods.tlp`	Basic methods for reasoning with LP.
`quantmethods.tlp`	Methods for reasoning about quantifiers.
`bin`	Directory containing the executables:
`tlp`	Shell script for running the translator.
`emacs`	Directory containing the GNU Emacs lisp files:
`tlp-mode.el`	The Emacs major mode.
`tlp-19.el`	Special features for GNU Emacs 19.
`tlp-dmacro.el`	Special features using the Dmacro package.
`tlp-local.el`	Local initialization of mode-specific variables.
`dm-c.el`	The Dmacro package for GNU Emacs 18 and 19, by Wayne Mesard, version 2.1.
`dm-compat.el`	
`dmacro-bld.el`	
`dmacro-sv.el`	
`dmacro.el`	
`examples`	Directory containing tlp examples.

	`byzantine`	The Byzantine Generals, by Stephan Merz.
	`inc94`	The Increment Example from the TLP report, described in the thesis.
	`mult94`	The Multiplier example, from the CAV-93 article by Bob Kurshan and Leslie Lamport [18].
	`mult-par`	Part of the multiplier example, using parameterized definitions.
	`prawitz`	An example illustrating how to reason with quantification.
	`span94`	Revised version of the Spanning-Tree Example from the CAV-92 article by Engberg, Grønning, and Lamport [12], with complete correctness proof, as described in the thesis.
`ltl`		Directory containing setup files for the LTL checker.
	`bdd-foreign.t`	BDD library interface.
	`weekaux.t`	BDD library interface.
	`model.t`	BDD library interface.
	`boolean-fml.t`	Kripke structure routines.
	`fml.t`	LTL model checker.
	`plaisted.t`	Some benchmarks, used for testing.
	`build-ltl`	Source file for compiling the LTL checker.
	`test-ltl`	Source file for testing the LTL checker.

2. Do the following in a shell:

   `gzip -d *.`⟨*platform*⟩`.gz`

   and move the files to ⟨*dir*⟩`/tlp-`⟨*release*⟩`/bin`.

3. Edit the file `emacs/tlp-local.el` to fit your local preferences. This is the place where all user options available to the front-end is declared and initialized. You will probably need to change the path specifications indicating where your TLP binaries have been put. A 'feature' section contains variables that determine which special features should be used. If your Emacs or your work station does not support some of these features, they will automatically be turned off; you should thus only turn features off manually if you really don't want them. The rest of the user options contain variables that you might want to change when you have learned how to use TLP, and want something to act differently. You don't have to go through these items at this point.

4. Copy all the the `*.el` files from the `emacs` directory to somewhere in the load-path of your Emacs, or better: insert ⟨*dir*⟩`/tlp-`⟨*release*⟩`/emacs` in your load-path. You might already have Dmacro on your system, in which case you don't need to get this.

5. Insert the following lines into your `~/.emacs` file

```
;;; TLP

(setq auto-mode-alist (append
                       (list
                        '("\\.tlp$" . tlp-mode)
                        '("_act\\.lp$" . tlp-mode)
                        '("_temp\\.lp$" . tlp-mode)
                        '("_ltl\\.t$" . tlp-mode))
                       auto-mode-alist))
(autoload 'tlp-mode "tlp-mode"
          "Major mode for editing and verifying TLP specifications"
          'interactive)
```

6. Set the environment variable TLPHOME to ⟨*dir*⟩`/tlp-`⟨*release*⟩, i.e. in your `~/.login` file, put something like

   `setenv TLPHOME` ⟨*dir*⟩`/tlp-`⟨*release*⟩

   This needs to be done before you startup the Emacs session in which you do your verification.

7. If you want to use the LTL checker, go to the ltl directory and run the command:

   ⟨*dir*⟩`/tlp-`⟨*release*⟩`/bin/bddt.`⟨*platform*⟩ `< build-ltl`

   This should create compiled images for future use. You may also want to test the system by saying:

   ⟨*dir*⟩`/tlp-`⟨*release*⟩`/bin/bddt.`⟨*platform*⟩ `< test-ltl`

8. If you haven't done this already, you may uncompress and print out the thesis "Reasoning in the Temporal Logic of Actions – the design and implementation of an interactive computer system" describing the TLP system. The latest version can be found in ⟨*dir*⟩`/tlp-`⟨*release*⟩`/thesis`. Device-independent (dvi) and PostScript (ps) versions are available.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Andrew Appel, James S. Mattson, and David R. Tarditi. *A lexical analyzer for Standard ML*. Department of Computer Science, Princeton University, Princeton, NJ 08544 and School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1994. Version 1.5.0.

[3] AT&T Bell Laboratories. *Standard ML of New Yersey, User's Guide*, February 1993. Version 0.93.

[4] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.

[5] Robert S. Boyer and J Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

[6] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbool*. Academic Press, New York, 1988.

[7] Ching-Tsun Chou. Predicates, temporal logic, and simulations. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 310–323, Berlin, 1993. Springer-Verlag.

[8] Ching-Tsun Chou. A sequent formulation of a logic of predicates in HOL. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume A-20 of *IFIP Transactions*, pages 71–80, North-Holland, 1993.

[9] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 347–374, Berlin, Heidelberg, New York, June 1993. Springer-Verlag. Proceedings of a REX School/Symposium.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–248, 1986.

[11] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[12] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55, Berlin, 1992. Springer-Verlag.

[13] Stephen J. Garland and John V. Guttag. An overview of LP, the Larch prover. In N. Dershowitz, editor, *Proceedings Proc. 3rd Intl. Conf. Rewriting Techniques and Applications*, volume 355 of *Lecture Notes on Computer Science*, pages 137–151. Springer-Verlag, April 1989.

[14] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center, December 1991.

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

[16] Mike Gordon. HOL – a proof generating system for higher-order logic. Technical Report 103, Computer Laboratory, University of Cambridge, Cambridge, 1987.

[17] Sara Kalvala. A formulation of TLA in Isabelle. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 214–228, Aspen Grove, UT, USA, September 1995. Springer-Verlag.

[18] R. P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Proceedings of the Fifth International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, Berlin, 1993. Springer-Verlag.

[19] Leslie Lamport. 'Sometime' is sometimes 'not never': A tutorial on the temporal logic of programs. In *Proceedings of the Seventh Annual Symposium on Principles of Programming Languages*, pages 174–185. ACM SIGACT-SIGPLAN, January 1980.

[20] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, December 1991.

[21] Leslie Lamport. How to write a long formula. Research Report 118, Digital Equipment Corporation, Systems Research Center, December 1993. To appear in *FACJ 6*(5) (September/October 1994).

[22] Leslie Lamport. How to write a proof. Research Report 94, Digital Equipment Corporation, Systems Research Center, February 1993. To appear in *American Mathematical Monthly*.

[23] Leslie Lamport. Hybrid systems in TLA$^+$. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102, Berlin, Heidelberg, 1993. Springer-Verlag.

[24] Leslie Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 347–374, Berlin, Heidelberg, New York, June 1993. Springer-Verlag. Proceedings of a REX School/Symposium.

[25] Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, September 1994.

[26] Peter A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, 3(1), January 1988.

[27] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.

[28] Lawrence C. Paulson. The foundation of a generic theorem prover. Technical Report 130, Computer Laboratory, University of Cambridge, Cambridge, 1988.

[29] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, Computer Laboratory, University of Cambridge, Cambridge, 1990.

[30] Dag Prawitz. *Natural Deduction, a Proof-Theoretical Study*, volume 3 of *Acta Universitatis Stockholmiensis, Stockholm studies in Philosophy*. Almqvist & Wiksell, Stockholm, 1965.

[31] Dana Scott, David Bostock, Graeme Forbes, Daniel Isaacson, and Gören Sundholm. Notes on the formalization of logic. Study Aids Monograph no. 2 and 3, Sub-faculty of Philosophy, Oxford, July 1981.

[32] Richard Stallman. *GNU Emacs Manual*. The Free Software Foundation, Inc., 675 Massachusetts Avenue, Cambridge, MA 02139 USA, tenth edition, July 1994. Updated for Emacs version 19.28.

[33] David R. Tarditi and Andrew Appel. *ML-Yacc User's Manual*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 and Department of Computer Science, Princeton University, Princeton, NJ 08544, May 1994. Version 2.2.

[34] Joakim von Wright. Mechanising the temporal logic of actions in HOL. In *Proceedings of the HOL Tutorial and Workshop*. ACM, August 1991.

[35] Joakim von Wright and Thomas Långbacka. Using a theorem prover for reasoning about concurrent algorithms. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 56–68, Berlin, 1992. Springer-Verlag.

Index

Recent Publications in the BRICS Dissertation Series

DS-96-3 **Lars Arge.** ***Efficient External-Memory Data Structures and Applications.*** **August 1996. Ph.D. thesis. xii+169 pp.**

DS-96-2 **Allan Cheng.** ***Reasoning About Concurrent Computational Systems.*** **August 1996. Ph.D. thesis. xiv+229 pp.**

DS-96-1 **Urban Engberg.** ***Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system.*** **August 1996. Ph.D. thesis. xvi+222 pp.**